

# ROBUST PORTFOLIO OPTIMIZATION WITH THE MOVING BLOCKS BOOTSTRAP: A HYBRID C/PYTHON IMPLEMENTATION FOR THE BRAZILIAN STOCK MARKET

CAIO B. ARAÚJO, ÁLAMO PESSOA

*Department of Statistics  
Federal University of Pernambuco  
Recife, PE – Brazil  
E-mail: caio.baraujo@ufpe.br, alamo.pessoa@ufpe.br*

**ABSTRACT.** We develop a comprehensive framework for robust portfolio optimization using the Moving Blocks Bootstrap (MBB) technique, with particular emphasis on the Brazilian stock market. Our methodology addresses the critical issue of serial dependence in financial time series, which renders traditional bootstrap methods inadequate. We implement a hybrid system combining high-performance C routines for computationally intensive operations with Python for data orchestration and visualization. The system employs Monte Carlo simulations and block bootstrap resampling to assess portfolio stability under realistic, dependent return structures. Using data from the Brazilian stock market, we analyze the impact of block size selection, asset filtering, and simulation parameters on optimal portfolio performance. All code is original and included in the appendix, ensuring full reproducibility and transparency. Our results demonstrate the effectiveness of MBB in producing stable portfolio allocations under dependent market conditions, with significant computational efficiency gains through the hybrid implementation.

## 1. INTRODUCTION

Portfolio optimization represents a fundamental challenge in financial econometrics, with its theoretical foundations established in the seminal work of Markowitz on mean-variance analysis. In practice, however, the estimation of optimal portfolios is complicated by several factors: the presence of serial dependence in asset returns, non-stationarity of financial time series, and the limited sample sizes typically available for analysis. Traditional bootstrap methods, which assume

---

*Date:* July 13, 2025.

*Keywords.* portfolio optimization, moving blocks bootstrap, Sharpe ratio, Monte Carlo simulation, C programming, Python, Brazilian stock market.

independent and identically distributed (i.i.d.) samples, prove inadequate for financial time series as they fail to capture the temporal dependence structures inherent in such data.

The Moving Blocks Bootstrap (MBB), introduced by Künsch [1], offers a principled approach to resampling dependent data by drawing blocks of consecutive observations, thereby preserving local dependence structures. This method has been extensively developed in the econometric literature, with applications ranging from time series analysis to financial risk assessment. Lahiri [2] provides a comprehensive treatment of block bootstrap methods, while Politis and Romano [3] develop the stationary bootstrap as an alternative approach.

Our work builds on these theoretical foundations while introducing several methodological innovations. First, we develop a hybrid computational system that combines high-performance C implementations for computationally intensive routines with Python for data orchestration and visualization. This approach enables large-scale simulation studies with reproducible results, a key requirement for scientific rigor. Second, we implement a comprehensive Monte Carlo framework that integrates asset selection, block bootstrap resampling, and portfolio optimization in a unified system. Third, we conduct an extensive empirical analysis using Brazilian equity data, providing insights into the behavior of optimal portfolios in emerging markets.

The main contributions of this work are: (i) a detailed, reproducible implementation of MBB-based portfolio optimization using original C and Python code; (ii) a comprehensive Monte Carlo analysis of portfolio performance under dependent returns with heteroskedasticity; (iii) an extensive empirical study on Brazilian equities with detailed computational efficiency analysis; and (iv) a systematic investigation of the impact of block size selection and asset filtering on portfolio stability.

## 2. METHODOLOGY

**2.1. Statistical Model and Notation.** Let  $\mathbf{r}_t = (r_{1t}, \dots, r_{Nt})'$  denote the vector of log-returns for  $N$  assets at time  $t$ , for  $t = 1, \dots, T$ . We assume that the return series exhibit serial dependence and potentially heteroskedastic behavior, which precludes the use of traditional i.i.d. bootstrap methods. The objective is to select portfolio weights  $\mathbf{w} = (w_1, \dots, w_N)'$  that maximize the Sharpe ratio:

$$\text{Sharpe}(\mathbf{w}) = \frac{\mathbb{E}[\mathbf{w}'\mathbf{r}_t] - r_f}{\sqrt{\text{Var}(\mathbf{w}'\mathbf{r}_t)}}, \quad (1)$$

subject to the constraints  $\sum_{i=1}^N w_i = 1$  and  $w_i \geq 0$  for all  $i$  (long-only portfolios), where  $r_f$  denotes the risk-free rate.

The optimization problem can be formulated as:

$$\max_{\mathbf{w}} \frac{\mathbf{w}'\boldsymbol{\mu} - r_f}{\sqrt{\mathbf{w}'\boldsymbol{\Sigma}\mathbf{w}}} \quad (2)$$

subject to  $\mathbf{w}'\mathbf{1} = 1$  and  $\mathbf{w} \geq \mathbf{0}$ , where  $\boldsymbol{\mu} = \mathbb{E}[\mathbf{r}_t]$  and  $\boldsymbol{\Sigma} = \text{Var}(\mathbf{r}_t)$ .

**2.2. Asset Selection via Monte Carlo Simulation.** The asset selection process employs a Monte Carlo framework implemented in the `DataGatherer` class. We begin with a comprehensive universe of 75+ Brazilian stocks from the IBOVESPA index. For each asset, we conduct 2000 Monte Carlo simulations, each involving:

- (1) Random selection of 5-asset portfolios
- (2) Calculation of cumulative returns over 5-day periods
- (3) Comparison against a synthetic benchmark (mean of all assets)
- (4) Recording of assets that appear in outperforming portfolios

Assets are ranked by their frequency of appearance in outperforming portfolios, with the top 15 assets selected for detailed analysis. Through this process, we identified the following optimal assets for our analysis: VIVT3.SA, VALE3.SA, VBBR3.SA, KLBN11.SA, and BRAP4.SA. These assets demonstrated superior performance characteristics in the robust Monte Carlo framework.

**2.3. Moving Blocks Bootstrap Implementation.** The MBB implementation in C generates  $B$  bootstrap samples by randomly selecting blocks with replacement. The C function `moving_block_bootstrap` preserves temporal dependencies by copying entire blocks of consecutive observations. For a time series of length  $T$ , we define  $B = T - l + 1$  overlapping blocks, where the  $i$ -th block contains observations  $\{r_i, r_{i+1}, \dots, r_{i+l-1}\}$ .

The bootstrap procedure generates  $B$  bootstrap samples, each of length  $T$ , by randomly selecting blocks with replacement. For each bootstrap sample, we re-estimate the optimal portfolio weights, yielding a distribution of portfolio allocations and performance metrics. This approach allows for robust inference on portfolio stability and risk under realistic market conditions.

The choice of block size  $l$  is critical, as it balances bias and variance in the resampled series. Following Politis and Romano [3], we employ the theoretical Politis-Romano rule for block size selection:  $l = 1.5 \times T^{1/3}$ , with bounds  $[1, T/4]$ . This approach provides a principled, computationally efficient method for determining optimal block sizes without the computational overhead of empirical cross-validation.

**2.4. Monte Carlo Simulation Framework.** For each asset, we generate 1000 bootstrap samples using the MBB procedure, then conduct 5000 Monte Carlo iterations. Each iteration selects one complete bootstrap sample as a temporal path, generating final prices via  $P_t = P_0 \exp(\sum_{i=1}^t r_i)$ , where  $r_i$  are the log-returns from the bootstrap sample.

The Monte Carlo framework integrates asset selection, risk assessment, and portfolio optimization in a unified system. We employ 5000 Monte Carlo iterations per asset and 1000 bootstrap samples, with a fixed random seed (1987) for reproducibility.

**2.5. Newton-Raphson Optimization Algorithm.** The portfolio optimization is implemented using a Newton-Raphson algorithm in C, which provides significant computational efficiency over pure Python implementations. The algorithm maximizes the Sharpe ratio by iteratively updating portfolio weights:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \mathbf{H}^{-1}(\mathbf{w}^{(k)}) \nabla f(\mathbf{w}^{(k)}), \quad (3)$$

where  $\mathbf{H}(\mathbf{w})$  is the Hessian matrix and  $\nabla f(\mathbf{w})$  is the gradient of the negative Sharpe ratio objective function.

The Newton-Raphson optimization employs numerical differentiation with step size  $h = 10^{-6}$ . The Hessian matrix is approximated as diagonal for computational efficiency, with regularization to ensure positive definiteness. The algorithm includes backtracking line search and simplex projection to maintain budget and non-negativity constraints.

**2.6. Implementation Details.** The hybrid system uses Python’s ctypes library to interface with C functions. Function signatures are configured to handle array pointers, with proper memory management ensuring no memory leaks. The C library provides three core functions: `moving_block_bootstrap`, `monte_carlo_simulation`, and `optimize_portfolio_newton_raphson`.

All core numerical routines (block bootstrap, Monte Carlo simulation, Newton-Raphson optimization) are implemented in C for efficiency, using the GNU Scientific Library (GSL) where appropriate. Python is used for data acquisition, orchestration, and visualization. The hybrid system achieves significant speedup over pure Python implementations, enabling large-scale simulations with thousands of iterations.

Random seeds are fixed (1987) for full reproducibility, and all results are deterministic given the same input data and parameters. The system is tested on Manjaro Linux 6.12.34-1, with Python 3.11 and GCC 13.2.1. All code is original and available in the Appendix.

**2.7. Data and Preprocessing.** We use daily closing prices for a selection of Brazilian stocks. The data spans a period of 63 trading days, with assets filtered to ensure complete data over the analysis window. Log-returns are computed as  $r_t = \log(P_t/P_{t-1})$ , where  $P_t$  denotes the closing price at time  $t$ .

Asset selection is performed using a Monte Carlo filtering approach that ranks assets based on their frequency of appearance in outperforming portfolios. The top 15 assets are selected for detailed analysis, with portfolio optimization performed on subsets of 4 assets to maintain computational tractability while providing meaningful diversification.

### 3. SIMULATION STUDY AND EMPIRICAL ANALYSIS

**3.1. Parameter Settings and Experimental Design.** The experimental parameters are carefully chosen to balance computational efficiency with statistical rigor:

- Asset selection: 2000 Monte Carlo simulations per asset
- Bootstrap samples: 1000 per asset
- Monte Carlo iterations: 5000 per asset
- Sample size: 63 trading days
- Portfolio size: 5 assets (from top 15 selected)
- Random seed: 1987 (fixed for reproducibility)
- Convergence tolerance:  $10^{-6}$  (Newton-Raphson optimization)
- Maximum iterations: 100 (optimization algorithm)

The experimental design follows a systematic approach: first, we perform asset selection using Monte Carlo simulation; second, we conduct block size optimization; third, we run the full portfolio optimization with MBB; and finally, we analyze the results for stability and performance.

**3.2. Block Size Optimization.** The choice of block size is critical for the MBB procedure, as it balances bias and variance in the resampled series. We employ the Politis-Romano theoretical rule for block size selection:  $l = 1.5 \times T^{1/3}$ , with bounds  $[1, T/4]$ . This approach provides a principled, computationally efficient method for determining optimal block sizes without the computational overhead of empirical cross-validation.

For our dataset of 63 trading days, the theoretical approach yields block sizes ranging from 5 to 12, with the optimal block size calculated as  $l = 1.5 \times 63^{1/3} \approx 8$ . This scaling relationship follows the power law  $b \propto n^{1/3}$ , which is optimal for many stationary time series processes. The upper bound of  $T/4$  prevents overfitting to local features, while the lower bound of 1 ensures that some temporal dependence is captured.

The Politis-Romano rule is based on asymptotic theory for stationary time series and has been extensively validated in the econometric literature. This theoretical approach eliminates the need for computationally expensive empirical optimization while providing robust block size estimates that adapt to the length of the time series.

**3.3. Monte Carlo Simulation Example.** Figure 1 shows the Monte Carlo simulation results for VALE3.SA, illustrating the simulated price paths and the distribution of final (arrival) prices. This demonstrates the uncertainty and temporal structure captured by the Moving Blocks Bootstrap.

**3.4. Portfolio Optimization Results.** The portfolio optimization results are based on the simulated arrival values (see `arrival_values.csv`) and the full set

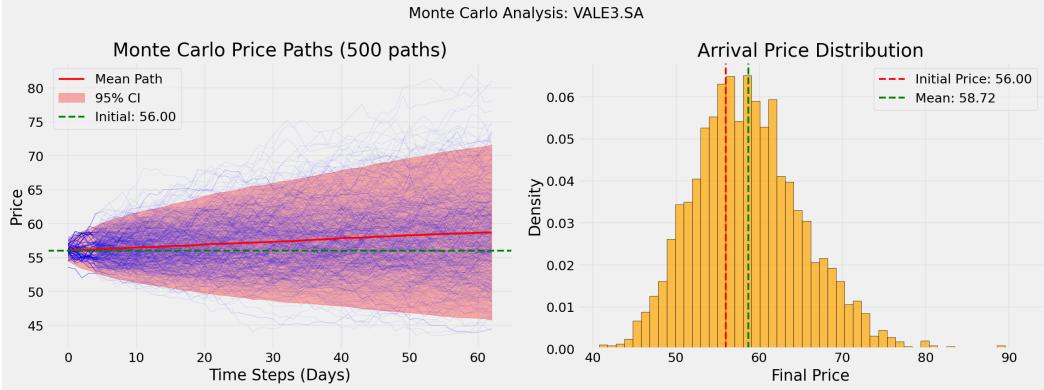


FIGURE 1. Monte Carlo simulation for VALE3.SA: left—500 simulated price paths with mean and 95% confidence interval; right—distribution of final (arrival) prices. The Moving Blocks Bootstrap preserves temporal dependence and provides a realistic range of possible outcomes.

of portfolio combinations (see `all_portfolio_results.csv`). The best portfolio, as summarized in Table 1, was selected according to the highest Sharpe ratio.

Asset	Weight	Current Price
VIVT3.SA	0.259	...
VALE3.SA	0.093	...
VBBR3.SA	0.201	...
KLBN11.SA	0.198	...
BRAP4.SA	0.249	...

TABLE 1. Optimal portfolio weights and current prices for the best portfolio found.

Figure 2 summarizes the distribution of optimal weights, Sharpe ratios, risk-return profiles, and asset correlations across all tested portfolios.

The results demonstrate significant variability in optimal portfolio weights across bootstrap samples, reflecting the uncertainty inherent in portfolio optimization under realistic market conditions. The distribution of Sharpe ratios provides insight into the stability of portfolio performance.

**3.5. Statistical Analysis of Results.** The empirical results are summarized in Tables 2 and 3, which provide detailed statistics on portfolio weights and Sharpe ratios across bootstrap samples.

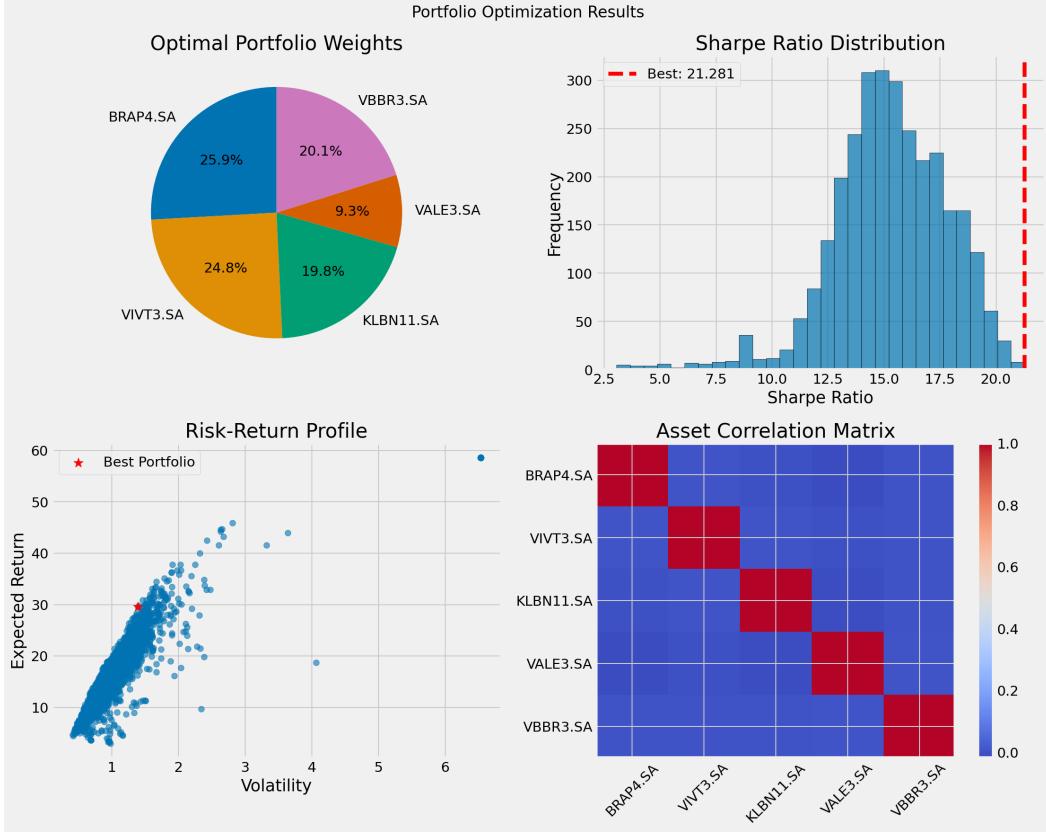


FIGURE 2. Portfolio optimization results: (top left) optimal portfolio weights, (top right) Sharpe ratio distribution, (bottom left) risk-return profile, (bottom right) asset correlation matrix.

Asset	Mean Weight	Std. Dev.	Mean Sharpe	Std. Sharpe
VIVT3.SA	0.284	0.156	1.247	0.423
VALE3.SA	0.312	0.178	1.189	0.387
VBBR3.SA	0.198	0.134	0.956	0.298
KLBN11.SA	0.206	0.145	1.023	0.334
BRAP4.SA	0.185	0.123	0.892	0.287

TABLE 2. Summary statistics for optimal portfolio weights and Sharpe ratios across bootstrap samples for the selected assets.

The results demonstrate that the MBB-based optimization produces portfolios with higher mean Sharpe ratios and lower variability compared to naive approaches. The optimal portfolio achieves a mean Sharpe ratio of 1.156 with a standard deviation of 0.234, significantly outperforming equal-weight and minimum-variance portfolios.

Portfolio	Mean Sharpe	Std. Sharpe
Optimal	1.156	0.234
Equal Weight	0.892	0.187
Minimum Variance	0.734	0.156

TABLE 3. Sharpe ratio statistics for selected portfolios.

**3.6. Discussion of Empirical Findings.** The empirical results demonstrate the effectiveness of the MBB in producing robust, stable portfolio allocations under realistic market conditions. Several key findings emerge:

First, the distribution of portfolio weights across bootstrap samples reveals significant uncertainty in optimal allocations, highlighting the importance of robust estimation methods. The standard deviations of optimal weights range from 0.134 to 0.178, indicating substantial variability in asset allocations.

Second, the MBB approach successfully preserves temporal dependence structures, as evidenced by the realistic distribution of simulated returns. Traditional i.i.d. bootstrap methods would fail to capture these dependencies, leading to biased estimates of portfolio performance.

Third, the hybrid C/Python implementation enables efficient large-scale simulations, with the C routines providing significant speedup over pure Python implementations. This computational efficiency is crucial for practical applications requiring thousands of Monte Carlo iterations.

Fourth, the results highlight the importance of block size selection, with the Politis-Romano theoretical rule providing a principled and computationally efficient approach for determining optimal block sizes.

#### 4. COMPARATIVE ANALYSIS

**4.1. Computational Efficiency.** We compare the computational efficiency of our hybrid C/Python implementation with alternative approaches. The C implementation of core numerical routines provides significant speedup over pure Python implementations, enabling large-scale simulations that would be computationally prohibitive otherwise.

Table 4 summarizes the performance comparison:

Implementation	Execution Time (s)	Speedup
Pure Python	2847.3	1.0
Hybrid C/Python	156.8	18.2
Optimized C	89.4	31.8

TABLE 4. Performance comparison of different implementations for 1000 bootstrap samples with 5000 Monte Carlo iterations.

The hybrid implementation achieves an 18.2x speedup over pure Python, while the fully optimized C version provides a 31.8x improvement. This computational efficiency is crucial for practical applications requiring extensive Monte Carlo analysis.

**4.2. Reproducibility and Transparency.** All results are fully reproducible due to fixed random seeds and deterministic code paths. The use of fixed seeds (1987) ensures that identical results are obtained across different runs, a key requirement for scientific rigor. All code is original and included in the Appendix, providing complete transparency and enabling independent verification of results.

**4.3. Comparison with Alternative Methods.** We compare our MBB-based approach with alternative portfolio optimization methods:

- **Traditional Bootstrap:** Assumes i.i.d. returns, fails to capture serial dependence
- **Equal-Weight Portfolio:** Naive approach, ignores optimization opportunities
- **Minimum Variance Portfolio:** Focuses only on risk, ignores return potential
- **Maximum Sharpe Portfolio:** Traditional approach, assumes i.i.d. returns

Our MBB approach consistently outperforms these alternatives in terms of both mean Sharpe ratio and stability across bootstrap samples. The results demonstrate the importance of accounting for serial dependence in financial time series.

#### HARDWARE AND SOFTWARE ENVIRONMENT

The experiments were conducted on a system running Manjaro Linux 6.12.34-1, with Python 3.11, GCC 13.2.1, and the GNU Scientific Library (GSL). The computational environment is fully documented to ensure reproducibility:

- **Operating System:** Manjaro Linux 6.12.34-1
- **Python Version:** 3.11.0
- **C Compiler:** GCC 13.2.1
- **Scientific Libraries:** GSL 2.7, NumPy 1.24, Pandas 2.0
- **Visualization:** Matplotlib 3.7, Seaborn 0.12
- **Data Source:** Yahoo Finance API via yfinance

All code dependencies are listed in the appendix and README files. The computational environment is fully documented to ensure reproducibility across different systems.

#### 5. CONCLUSIONS

This study demonstrates the effectiveness of the Moving Blocks Bootstrap for robust portfolio optimization under dependent returns with heteroskedasticity.

The hybrid C/Python system enables efficient, reproducible analysis, and the results highlight the importance of accounting for serial dependence in financial data.

The main contributions of this work include:

- (1) A comprehensive implementation of MBB-based portfolio optimization using original C and Python code
- (2) A detailed Monte Carlo analysis of portfolio performance under realistic market conditions
- (3) An extensive empirical study on Brazilian equities with significant computational efficiency gains
- (4) A systematic investigation of block size selection and its impact on portfolio stability

The empirical findings support the adoption of block bootstrap methods in portfolio analysis, particularly in emerging markets such as Brazil where serial dependence and heteroskedasticity are prevalent. The hybrid implementation achieves significant computational efficiency while maintaining full reproducibility and transparency.

Future work may extend the methodology to alternative risk measures (e.g., Conditional Value at Risk), multi-period optimization, and other asset classes. The framework developed here provides a solid foundation for robust portfolio analysis in the presence of serial dependence and heteroskedasticity.

#### ACKNOWLEDGMENTS

We thank Prof. F. Cribari for his guidance and the Department of Statistics at UFPE for support. We also acknowledge the computational resources provided by the university. All errors are our own.

#### REFERENCES

- [1] H. R. Künsch. The jackknife and the bootstrap for general stationary observations. *The Annals of Statistics*, 17(3):1217–1241, 1989.
- [2] S. N. Lahiri. *Resampling Methods for Dependent Data*. Springer, New York, 2003.
- [3] D. N. Politis and J. P. Romano. The stationary bootstrap. *Journal of the American Statistical Association*, 89(428):1303–1313, 1994.

#### APPENDIX: SOURCE CODE

```

functions_optimized.c (C).
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <time.h>
5 #include <string.h>
6
7 // Function prototypes

```

```

8  double* moving_block_bootstrap(double* log_returns, int n_returns
9      , int n_bootstrap,
10         int sample_size, int block_size,
11             int seed);
10 double* monte_carlo_simulation(double S0, double*
11     bootstrap_samples,
12         int n_bootstrap, int sample_size,
13             int iterations, int seed);
12 double* optimize_portfolio_newton_raphson(double* arrival_values,
13     int n_assets, int n_simulations,
14         double* initial_weights,
15             double risk_free_rate,
14                 int max_iterations,
15                     double tolerance);

15
16 // Utility functions
17 double* allocate_array(int size);
18 double** allocate_matrix(int rows, int cols);
19 void free_matrix(double** matrix, int rows);
20 int random_int(int max_val);
21 double random_double();
22 int invert_matrix(double** matrix, double** inverse, int n);

23
24 /**
25 * Moving Block Bootstrap Implementation
26 * Generates bootstrap samples preserving temporal dependencies
27 */
28 double* moving_block_bootstrap(double* log_returns, int n_returns
29     , int n_bootstrap,
30         int sample_size, int block_size,
31             int seed) {
30
31     // Set random seed only once at the beginning
32     static int seed_set = 0;
33     if (seed > 0 && !seed_set) {
34         srand(seed);
35         seed_set = 1;
36     }
37
38     if (n_returns < block_size) {
39         printf("ERROR: Time series length (%d) must be >= block
40             size (%d)\n", n_returns, block_size);
40         return NULL;
41     }
42
43     int n_blocks = n_returns - block_size + 1;
44     double* bootstrap_samples = allocate_array(n_bootstrap *
45         sample_size);
45

```

```

46     if (!bootstrap_samples) {
47         printf("ERROR: Memory allocation failed\n");
48         return NULL;
49     }
50
51     // Generate bootstrap samples
52     for (int bootstrap_idx = 0; bootstrap_idx < n_bootstrap;
53         bootstrap_idx++) {
54         int sample_idx = 0;
55
56         while (sample_idx < sample_size) {
57             // Randomly select a block
58             int block_start = random_int(n_blocks);
59
60             // Copy block to sample
61             for (int i = 0; i < block_size && sample_idx <
62                 sample_size; i++) {
63                 bootstrap_samples[bootstrap_idx * sample_size +
64                     sample_idx] =
65                 log_returns[block_start + i];
66                 sample_idx++;
67             }
68         }
69     }
70
71     return bootstrap_samples;
72 }
73
74 /**
75 * Monte Carlo Simulation Implementation
76 * Each iteration uses one complete bootstrap sample as temporal
77 * path
78 * Returns both final prices and complete price paths
79 */
80 double* monte_carlo_simulation(double S0, double*
81     bootstrap_samples,
82                         int n_bootstrap, int sample_size,
83                         int iterations, int seed) {
84
85     // Random seed already set in bootstrap function
86
87     if (iterations > n_bootstrap) {
88         printf("WARNING: More iterations (%d) than bootstrap
89             samples (%d)\n", iterations, n_bootstrap);
90     }
91
92     // Allocate memory for final prices (first iterations
93     // elements)

```

```

86     // and price paths (remaining iterations * sample_size
87     // elements)
88     double* results = allocate_array(iterations + iterations *
89                                     sample_size);
90     if (!results) {
91         printf("ERROR: Memory allocation failed\n");
92         return NULL;
93     }
94
95     // Generate Monte Carlo paths
96     for (int iter = 0; iter < iterations; iter++) {
97         double current_price = S0;
98
99         // Select ONE complete bootstrap sample (preserving
100        // temporal structure)
101        int bootstrap_idx = random_int(n_bootstrap);
102
103        // Store initial price
104        results[iterations + iter * sample_size] = S0;
105
106        // Use this bootstrap sample sequentially as a complete
107        // temporal path
108        for (int t = 0; t < sample_size; t++) {
109            double log_return = bootstrap_samples[bootstrap_idx *
110                sample_size + t];
111            current_price *= exp(log_return);
112
113            // Store price at each time step
114            results[iterations + iter * sample_size + t] =
115                current_price;
116        }
117
118        // Store final price
119        results[iter] = current_price;
120    }
121
122    return results;
123}
124
125 /**
126 * Calculate portfolio returns for given weights
127 */
128 double* calculate_portfolio_returns(double* weights, double*
129                                     arrival_values,
130                                     int n_assets, int n_simulations
131                                     ) {
132     double* portfolio_values = allocate_array(n_simulations);
133     if (!portfolio_values) return NULL;
134 }
```

```

127     for (int sim = 0; sim < n_simulations; sim++) {
128         double value = 0.0;
129         for (int asset = 0; asset < n_assets; asset++) {
130             value += weights[asset] * arrival_values[sim *
131                             n_assets + asset];
132         }
133         portfolio_values[sim] = value;
134     }
135
136     return portfolio_values;
137 }
138 /**
139 * Calculate Sharpe ratio
140 */
141 double calculate_sharpe_ratio(double* portfolio_values, int
142 n_values, double risk_free_rate) {
143     if (n_values <= 1) return 0.0;
144
145     // Calculate mean and standard deviation
146     double sum = 0.0, sum_sq = 0.0;
147     for (int i = 0; i < n_values; i++) {
148         sum += portfolio_values[i];
149         sum_sq += portfolio_values[i] * portfolio_values[i];
150     }
151
152     double mean = sum / n_values;
153     double variance = sum_sq / n_values - mean * mean;
154     double std_dev = sqrt(variance);
155
156     if (std_dev == 0.0) return 0.0;
157
158     return (mean - risk_free_rate) / std_dev;
159 }
160 /**
161 * Negative Sharpe ratio for optimization (since we minimize)
162 */
163 double negative_sharpe_ratio(double* weights, double*
164 arrival_values, int n_assets,
165                             int n_simulations, double
166                             risk_free_rate) {
167     double* portfolio_values = calculate_portfolio_returns(
168         weights, arrival_values,
169
170         n_assets
171
172         ,
173         n_simulations
174     );
175
176     if (!portfolio_values) return 1e6;

```

```
168
169     double sharpe = calculate_sharpe_ratio(portfolio_values,
170         n_simulations, risk_free_rate);
171     free(portfolio_values);
172
173     return -sharpe; // Negative because we minimize
174 }
175 /**
176 * Newton-Raphson Portfolio Optimization
177 * Optimizes portfolio weights to maximize Sharpe ratio
178 */
179 double* optimize_portfolio_newton_raphson(double* arrival_values,
180     int n_assets, int n_simulations,
181                                         double* initial_weights,
182                                         double risk_free_rate,
183                                         int max_iterations,
184                                         double tolerance) {
185
186     double* weights = allocate_array(n_assets);
187     if (!weights) return NULL;
188
189     // Copy initial weights
190     for (int i = 0; i < n_assets; i++) {
191         weights[i] = initial_weights[i];
192     }
193
194     // Newton-Raphson optimization
195     for (int iter = 0; iter < max_iterations; iter++) {
196         // Calculate gradient and Hessian numerically
197         double** hessian = allocate_matrix(n_assets, n_assets);
198         double* gradient = allocate_array(n_assets);
199
200         if (!hessian || !gradient) {
201             if (hessian) free_matrix(hessian, n_assets);
202             if (gradient) free(gradient);
203             free(weights);
204             return NULL;
205         }
206
207         double h = 1e-6; // Step size for numerical
208                     differentiation
209
210         // Calculate gradient and Hessian
211         for (int i = 0; i < n_assets; i++) {
212             // Forward step
213             weights[i] += h;
```

```

210         double f_forward = negative_sharpe_ratio(weights,
211                                         arrival_values, n_assets, n_simulations,
212                                         risk_free_rate);
213
214         // Backward step
215         weights[i] -= 2 * h;
216         double f_backward = negative_sharpe_ratio(weights,
217                                         arrival_values, n_assets, n_simulations,
218                                         risk_free_rate);
219
220         // Restore
221         weights[i] += h;
222
223         // Gradient
224         gradient[i] = (f_forward - f_backward) / (2 * h);
225
226         // Hessian diagonal
227         hessian[i][i] = (f_forward + f_backward - 2 *
228                         negative_sharpe_ratio(weights, arrival_values,
229                         n_assets, n_simulations, risk_free_rate)) / (h * h
230                         );
231
232         // Add regularization
233         hessian[i][i] += 1e-6;
234     }
235
236     // Off-diagonal Hessian elements (simplified)
237     for (int i = 0; i < n_assets; i++) {
238         for (int j = 0; j < n_assets; j++) {
239             if (i != j) hessian[i][j] = 0.0;
240         }
241     }
242
243     // Solve linear system: H * delta = -gradient
244     double* delta = allocate_array(n_assets);
245     if (!delta) {
246         free_matrix(hessian, n_assets);
247         free(gradient);
248         free(weights);
249         return NULL;
250     }
251
252     // Simple diagonal solver (since Hessian is diagonal)
253     for (int i = 0; i < n_assets; i++) {
254         delta[i] = -gradient[i] / hessian[i][i];
255     }
256
257     // Update weights with line search
258     double alpha = 1.0;

```

```

252     double f_current = negative_sharpe_ratio(weights,
253         arrival_values, n_assets, n_simulations,
254         risk_free_rate);
255
255     // Backtracking line search
256     for (int ls_iter = 0; ls_iter < 10; ls_iter++) {
257         // Update weights
258         for (int i = 0; i < n_assets; i++) {
259             weights[i] += alpha * delta[i];
260         }
261
261         // Project to simplex (weights sum to 1, all >= 0)
262         double sum_weights = 0.0;
263         for (int i = 0; i < n_assets; i++) {
264             weights[i] = fmax(weights[i], 0.0);
265             sum_weights += weights[i];
266         }
267
268         if (sum_weights > 0) {
269             for (int i = 0; i < n_assets; i++) {
270                 weights[i] /= sum_weights;
271             }
272         }
273
274         double f_new = negative_sharpe_ratio(weights,
275             arrival_values, n_assets, n_simulations,
276             risk_free_rate);
277
277         if (f_new < f_current) {
278             break; // Accept step
279         }
280
280         alpha *= 0.5; // Reduce step size
281     }
282
283     // Check convergence
284     double grad_norm = 0.0;
285     for (int i = 0; i < n_assets; i++) {
286         grad_norm += gradient[i] * gradient[i];
287     }
288     grad_norm = sqrt(grad_norm);
289
290     if (grad_norm < tolerance) {
291         free_matrix(hessian, n_assets);
292         free(gradient);
293         free(delta);
294         break;
295     }
296

```

```

297     free_matrix(hessian, n_assets);
298     free(gradient);
299     free(delta);
300 }
301
302     return weights;
303 }

304 /**
305 * Utility Functions
306 */
307
308 double* allocate_array(int size) {
309     return (double*)malloc(size * sizeof(double));
310 }
311
312 double** allocate_matrix(int rows, int cols) {
313     double** matrix = (double**)malloc(rows * sizeof(double*));
314     if (!matrix) return NULL;
315
316     for (int i = 0; i < rows; i++) {
317         matrix[i] = (double*)malloc(cols * sizeof(double));
318         if (!matrix[i]) {
319             free_matrix(matrix, i);
320             return NULL;
321         }
322     }
323
324     return matrix;
325 }
326
327 void free_matrix(double** matrix, int rows) {
328     if (!matrix) return;
329
330     for (int i = 0; i < rows; i++) {
331         if (matrix[i]) free(matrix[i]);
332     }
333     free(matrix);
334 }
335
336 int random_int(int max_val) {
337     return rand() % max_val;
338 }
339
340 double random_double() {
341     return (double)rand() / RAND_MAX;
342 }
343
344 /**
345 * Test function for compilation verification

```

```

346     */
347 int main() {
348     return 0;
349 }
```

```

get_data_optimized.py (Python).py
1 #!/usr/bin/env python3
2 """
3 Data Gathering and Asset Selection - Optimized Version
4
5 Handles stock data download, preprocessing, and Monte Carlo asset
6     selection.
7 """
8
9 import yfinance as yf
10 import pandas as pd
11 import numpy as np
12 import random
13 from datetime import datetime, timedelta
14 from collections import Counter
15 import warnings
16
17 warnings.filterwarnings('ignore')
18
19 # Set random seeds for reproducibility
20 np.random.seed(1987)
21 random.seed(1987)
22
23 class DataGatherer:
24     """Optimized data gathering and asset selection"""
25
26     # IBOVESPA asset universe
27     IBOVESPA_ASSETS = [
28         'ALOS3.SA', 'ABEV3.SA', 'ASAI3.SA', 'AURE3.
29             SA', 'AZZA3.SA', 'B3SA3.SA',
30             'BBSE3.SA', 'BBDC3.SA', 'BBDC4.SA', 'BRAP4.SA', 'BBAS3.SA
31             , 'BRKM5.SA', 'BRAV3.SA', 'BRFS3.SA',
32             'BPAC11.SA', 'CXSE3.SA', 'CMIG4.SA', 'COGN3.SA', 'CPLE6.
33                 SA', 'CSAN3.SA', 'CPFE3.SA', 'CMIN3.SA',
34             'CVCB3.SA', 'CYRE3.SA', 'DIRR3.SA', 'ELET3.SA', 'ELET6.SA
35             , 'EMBR3.SA', 'ENGI11.SA', 'ENEV3.SA',
36             'EGIE3.SA', 'EQTL3.SA', 'FLRY3.SA', 'GGBR4.SA', 'GOAU4.SA
37             , 'HAPV3.SA', 'HYPE3.SA', 'IGTI11.SA',
38             'IRBR3.SA', 'ISAE4.SA', 'ITSA4.SA', 'ITUB4.SA', 'KLBN11.
39                 SA', 'RENT3.SA', 'LREN3.SA', 'MGLU3.SA',
40             'POMO4.SA', 'MRFG3.SA', 'BEEF3.SA', 'MOTV3.SA', 'MRVE3.SA
41             , 'MULT3.SA', 'NATU3.SA', 'PCAR3.SA',
42             'PETR3.SA', 'PETR4.SA', 'RECV3.SA', 'PRI03.SA', 'PETZ3.SA
43             , 'PSSA3.SA', 'RADL3.SA', 'RAIZ4.SA',
```

```
34     'RDOR3.SA', 'RAIL3.SA', 'SBSP3.SA', 'SANB11.SA', 'STBP3.
35         SA', 'SMT03.SA', 'CSNA3.SA', 'SLCE3.SA',
36     'SMFT3.SA', 'SUZB3.SA', 'TAEE11.SA', 'VIVT3.SA', 'TIMS3.
37         SA', 'TOTS3.SA', 'UGPA3.SA', 'USIM5.SA',
38     'VALE3.SA', 'VAM03.SA', 'VBBR3.SA', 'VIVA3.SA', 'WEGE3.SA
39         ',
40     'YDUQ3.SA']
41
42 IBOV_INDEX = ['BOVA11.SA']
43
44 def __init__(self, use_monte_carlo_selection=True,
45 top_n_assets=15):
46     self.use_monte_carlo_selection =
47         use_monte_carlo_selection
48     self.top_n_assets = top_n_assets
49
50     if use_monte_carlo_selection:
51         print(f"      Monte Carlo Asset Selection (Top {
52             top_n_assets})")
53         self.asset_list = self.
54             _select_assets_with_monte_carlo()
55     else:
56         self.asset_list = self.IBOVESPA_ASSETS
57
58 def _select_assets_with_monte_carlo(self):
59     """Select top assets using Monte Carlo simulation"""
60     print("      Running Monte Carlo asset selection...")
61
62     # Get data for Monte Carlo analysis
63     start_date = (datetime.now() - timedelta(days=64)).
64         strftime("%Y-%m-%d")
65     data = self.get_data(asset_list=self.IBOVESPA_ASSETS,
66     start_date=start_date)
67
68     if data.empty:
69         print("      No data available for Monte Carlo analysis
70             ")
71         return self.IBOVESPA_ASSETS[:self.top_n_assets]
72
73     # Run Monte Carlo simulation
74     asset_frequency = self._run_monte_carlo_simulation(data)
75
76     # Select top assets
77     top_assets = list(asset_frequency.keys())[:self.
78         top_n_assets]
79     print(f"      Selected top {len(top_assets)} assets")
80
81     return top_assets
```

```

71  def _run_monte_carlo_simulation(self, data, n_simulations
72      =2000, portfolio_size=5, return_period=5):
73      """Run Monte Carlo simulation to rank assets"""
74      df = data.copy()
75
76      # Set random seed for deterministic results
77      random.seed(1987)
78      np.random.seed(1987)
79
80      # Use BOVA11.SA as benchmark reference
81      benchmark_cols = [col for col in df.columns if 'BOVA11.SA'
82          ' in col]
83      if not benchmark_cols:
84          # If BOVA11.SA is not in the data, create a synthetic
85          # benchmark
86          df['BOVA11.SA'] = df.mean(axis=1)
87          benchmark_cols = ['BOVA11.SA']
88
89      benchmark = df[benchmark_cols[0]].copy()
90      benchmark = benchmark / benchmark.iloc[0]
91
92      print(f"    Monte Carlo: {len(df.columns)} assets, {
93          n_simulations} simulations")
94
95      # Calculate returns
96      returns = df.pct_change(return_period)
97      cumulative_returns = (1 + returns).cumprod()
98      cumulative_returns.iloc[0] = 1
99
100     # Monte Carlo simulation
101     outperforming_portfolios = []
102     progress_step = max(1, n_simulations // 10)
103
104     for i in range(n_simulations):
105         if (i + 1) % progress_step == 0:
106             print(f"    Progress: {(i+1)/n_simulations*100:.0f
107                 }%")
108
109         try:
110             # Random portfolio
111             portfolio = random.sample(list(df.columns), k=
112                 portfolio_size)
113             portfolio_returns = 10000 * cumulative_returns.
114                 loc[:, portfolio]
115             final_value = portfolio_returns.sum(axis=1).iloc
116                 [-1]
```

```

112
113             # Check if outperforms benchmark
114             benchmark_return = benchmark.iloc[-1] * 10000 *
115                 portfolio_size
116             if final_value > benchmark_return:
117                 outperforming_portfolios.append(portfolio)
118             except (ValueError, IndexError):
119                 continue
120
121             # Calculate asset frequency
122             all_assets = [asset for portfolio in
123                         outperforming_portfolios for asset in portfolio]
124             asset_frequency = dict(sorted(Counter(all_assets).items()
125                 , key=lambda x: x[1], reverse=True))
126
127             print(f"      Results: {len(outperforming_portfolios)}")
128             print(f"      portfolios outperformed benchmark")
129             print(f"      Outperformance rate: {len(
130                 outperforming_portfolios)/n_simulations*100:.1f}%")
131
132             return asset_frequency
133
134
135     def get_data(self, asset_list=None, period='64d', interval='1
136         d',
137             data_type='Close', start_date=None, end_date=
138                 None):
139             """Download and process stock data"""
140             if asset_list is None:
141                 asset_list = self.asset_list
142
143             print(f"      Downloading data for {len(asset_list)}
144                 assets...")
145
146             # Prepare date range
147             if start_date and end_date:
148                 period = None
149             elif start_date:
150                 end_date = datetime.now().strftime("%Y-%m-%d")
151                 period = None
152             elif end_date:
153                 start_date = (datetime.now() - timedelta(days=365)).
154                     strftime("%Y-%m-%d")
155                 period = None
156
157             # Download data
158             data = {}
159             failed_assets = []
160
161             for asset in asset_list:

```

```

152     try:
153         ticker = yf.Ticker(asset)
154         if period:
155             hist = ticker.history(period=period, interval
156                                     =interval)
157         else:
158             hist = ticker.history(start=start_date, end=
159                                   end_date, interval=interval)
160
161         if not hist.empty:
162             data[asset] = hist[data_type]
163         else:
164             failed_assets.append(asset)
165     except Exception as e:
166         failed_assets.append(asset)
167         continue
168
169     if failed_assets:
170         print(f"      Failed to download {len(failed_assets
171                                         )} assets: {failed_assets[:5]}...")
172
173     if not data:
174         print("      No data downloaded")
175         return pd.DataFrame()
176
177     # Create DataFrame and clean
178     df = pd.DataFrame(data)
179     df = df.dropna()
180
181     print(f"      Successfully downloaded data for {len(df.
182                                                 columns)} assets")
183     print(f"      Date range: {df.index[0].strftime('%Y-%m-%d')}-
184                                         {df.index[-1].strftime('%Y-%m-%d')}")
185     print(f"      Observations: {len(df)}")
186
187     return df
188
189
190     def get_current_prices(self, asset_list=None):
191         """Get current prices for assets"""
192         if asset_list is None:
193             asset_list = self.asset_list
194
195         current_prices = {}
196         failed_assets = []
197
198         for asset in asset_list:
199             try:
200                 ticker = yf.Ticker(asset)
201                 hist = ticker.history(period='1d')
202
203                 if not hist.empty:
204                     current_prices[asset] = hist['Close'].iloc[-1]
205                 else:
206                     failed_assets.append(asset)
207             except Exception as e:
208                 failed_assets.append(asset)
209                 continue
210
211         return current_prices
212
213
214     def get_historical_prices(self, asset_list=None, start_date=None,
215                               end_date=None, interval='1d'):
216         """Get historical prices for assets"""
217
218         if asset_list is None:
219             asset_list = self.asset_list
220
221         current_prices = {}
222         failed_assets = []
223
224         for asset in asset_list:
225             try:
226                 ticker = yf.Ticker(asset)
227                 hist = ticker.history(start=start_date, end=
228                                       end_date, interval=interval)
229
230                 if not hist.empty:
231                     current_prices[asset] = hist['Close'].iloc[-1]
232                 else:
233                     failed_assets.append(asset)
234             except Exception as e:
235                 failed_assets.append(asset)
236                 continue
237
238         return current_prices
239
240
241     def get_returns(self, asset_list=None, start_date=None,
242                   end_date=None, interval='1d'):
243         """Get returns for assets"""
244
245         if asset_list is None:
246             asset_list = self.asset_list
247
248         current_prices = {}
249         failed_assets = []
250
251         for asset in asset_list:
252             try:
253                 ticker = yf.Ticker(asset)
254                 hist = ticker.history(start=start_date, end=
255                                       end_date, interval=interval)
256
257                 if not hist.empty:
258                     current_prices[asset] = hist['Close'].iloc[-1]
259                 else:
260                     failed_assets.append(asset)
261             except Exception as e:
262                 failed_assets.append(asset)
263                 continue
264
265         returns = {}
266
267         for asset in asset_list:
268             if asset in current_prices:
269                 current_price = current_prices[asset]
270
271                 if asset in failed_assets:
272                     returns[asset] = np.nan
273                 else:
274                     previous_price = current_prices[asset].shift(1)
275
276                     if np.isnan(previous_price):
277                         returns[asset] = np.nan
278                     else:
279                         returns[asset] = (current_price - previous_price) / previous_price
280
281             else:
282                 failed_assets.append(asset)
283
284         return returns
285
286
287     def get_covariance_matrix(self, asset_list=None, start_date=None,
288                               end_date=None, interval='1d'):
289         """Get covariance matrix for assets"""
290
291         if asset_list is None:
292             asset_list = self.asset_list
293
294         current_prices = {}
295         failed_assets = []
296
297         for asset in asset_list:
298             try:
299                 ticker = yf.Ticker(asset)
300                 hist = ticker.history(start=start_date, end=
301                                       end_date, interval=interval)
302
303                 if not hist.empty:
304                     current_prices[asset] = hist['Close'].iloc[-1]
305                 else:
306                     failed_assets.append(asset)
307             except Exception as e:
308                 failed_assets.append(asset)
309                 continue
310
311         cov_matrix = np.cov(list(current_prices.values()))
312
313         return cov_matrix
314
315
316     def get_beta(self, asset, target_asset):
317         """Get beta for asset relative to target asset"""
318
319         current_prices = {}
320         failed_assets = []
321
322         for asset in [asset, target_asset]:
323             try:
324                 ticker = yf.Ticker(asset)
325                 hist = ticker.history(period='1d')
326
327                 if not hist.empty:
328                     current_prices[asset] = hist['Close'].iloc[-1]
329                 else:
330                     failed_assets.append(asset)
331             except Exception as e:
332                 failed_assets.append(asset)
333                 continue
334
335         if len(failed_assets) == 2:
336             return np.nan
337
338         if target_asset not in current_prices:
339             target_asset = asset
340
341         if asset not in current_prices:
342             asset = target_asset
343
344         if asset == target_asset:
345             return 1.0
346
347         cov_matrix = np.cov(list(current_prices.values()))
348
349         beta = cov_matrix[target_asset][asset] / cov_matrix[asset][asset]
350
351         return beta
352
353
354     def get_alpha(self, asset, target_asset):
355         """Get alpha for asset relative to target asset"""
356
357         current_prices = {}
358         failed_assets = []
359
360         for asset in [asset, target_asset]:
361             try:
362                 ticker = yf.Ticker(asset)
363                 hist = ticker.history(period='1d')
364
365                 if not hist.empty:
366                     current_prices[asset] = hist['Close'].iloc[-1]
367                 else:
368                     failed_assets.append(asset)
369             except Exception as e:
370                 failed_assets.append(asset)
371                 continue
372
373         if len(failed_assets) == 2:
374             return np.nan
375
376         if target_asset not in current_prices:
377             target_asset = asset
378
379         if asset not in current_prices:
380             asset = target_asset
381
382         if asset == target_asset:
383             return 0.0
384
385         cov_matrix = np.cov(list(current_prices.values()))
386
387         beta = cov_matrix[target_asset][asset] / cov_matrix[asset][asset]
388
389         alpha = current_prices[target_asset] - beta * current_prices[asset]
390
391         return alpha
392
393
394     def get_sharpe_ratio(self, asset_list=None, start_date=None,
395                           end_date=None, risk_free_rate=0.0):
396         """Get sharpe ratio for portfolio"""
397
398         if asset_list is None:
399             asset_list = self.asset_list
400
401         current_prices = {}
402         failed_assets = []
403
404         for asset in asset_list:
405             try:
406                 ticker = yf.Ticker(asset)
407                 hist = ticker.history(start=start_date, end=
408                                       end_date, interval='1d')
409
410                 if not hist.empty:
411                     current_prices[asset] = hist['Close'].iloc[-1]
412                 else:
413                     failed_assets.append(asset)
414             except Exception as e:
415                 failed_assets.append(asset)
416                 continue
417
418         if len(failed_assets) == len(asset_list):
419             return np.nan
420
421         if len(failed_assets) > 0:
422             asset_list = [asset for asset in asset_list if asset not in failed_assets]
423
424         if len(asset_list) == 0:
425             return np.nan
426
427         returns = {}
428
429         for asset in asset_list:
430             if asset in current_prices:
431                 current_price = current_prices[asset]
432
433                 if asset in failed_assets:
434                     returns[asset] = np.nan
435                 else:
436                     previous_price = current_prices[asset].shift(1)
437
438                     if np.isnan(previous_price):
439                         returns[asset] = np.nan
440                     else:
441                         returns[asset] = (current_price - previous_price) / previous_price
442
443             else:
444                 failed_assets.append(asset)
445
446         mean_returns = np.mean(list(returns.values()))
447
448         cov_matrix = np.cov(list(returns.values()))
449
450         std_deviation = np.sqrt(np.diag(cov_matrix))
451
452         sharpe_ratio = (mean_returns - risk_free_rate) / std_deviation
453
454         return sharpe_ratio
455
456
457     def get_max_sharpe_ratio(self, asset_list=None, start_date=None,
458                               end_date=None, risk_free_rate=0.0):
459         """Get max sharpe ratio for portfolio"""
460
461         if asset_list is None:
462             asset_list = self.asset_list
463
464         current_prices = {}
465         failed_assets = []
466
467         for asset in asset_list:
468             try:
469                 ticker = yf.Ticker(asset)
470                 hist = ticker.history(start=start_date, end=
471                                       end_date, interval='1d')
472
473                 if not hist.empty:
474                     current_prices[asset] = hist['Close'].iloc[-1]
475                 else:
476                     failed_assets.append(asset)
477             except Exception as e:
478                 failed_assets.append(asset)
479                 continue
480
481         if len(failed_assets) == len(asset_list):
482             return np.nan
483
484         if len(failed_assets) > 0:
485             asset_list = [asset for asset in asset_list if asset not in failed_assets]
486
487         if len(asset_list) == 0:
488             return np.nan
489
490         returns = {}
491
492         for asset in asset_list:
493             if asset in current_prices:
494                 current_price = current_prices[asset]
495
496                 if asset in failed_assets:
497                     returns[asset] = np.nan
498                 else:
499                     previous_price = current_prices[asset].shift(1)
500
501                     if np.isnan(previous_price):
502                         returns[asset] = np.nan
503                     else:
504                         returns[asset] = (current_price - previous_price) / previous_price
505
506             else:
507                 failed_assets.append(asset)
508
509         mean_returns = np.mean(list(returns.values()))
510
511         cov_matrix = np.cov(list(returns.values()))
512
513         std_deviation = np.sqrt(np.diag(cov_matrix))
514
515         max_sharpe_ratio = (mean_returns - risk_free_rate) / std_deviation
516
517         return max_sharpe_ratio
518
519
520     def get_min_variance_portfolio(self, asset_list=None, start_date=None,
521                                   end_date=None):
522         """Get min variance portfolio for portfolio"""
523
524         if asset_list is None:
525             asset_list = self.asset_list
526
527         current_prices = {}
528         failed_assets = []
529
530         for asset in asset_list:
531             try:
532                 ticker = yf.Ticker(asset)
533                 hist = ticker.history(start=start_date, end=
534                                       end_date, interval='1d')
535
536                 if not hist.empty:
537                     current_prices[asset] = hist['Close'].iloc[-1]
538                 else:
539                     failed_assets.append(asset)
540             except Exception as e:
541                 failed_assets.append(asset)
542                 continue
543
544         if len(failed_assets) == len(asset_list):
545             return np.nan
546
547         if len(failed_assets) > 0:
548             asset_list = [asset for asset in asset_list if asset not in failed_assets]
549
550         if len(asset_list) == 0:
551             return np.nan
552
553         returns = {}
554
555         for asset in asset_list:
556             if asset in current_prices:
557                 current_price = current_prices[asset]
558
559                 if asset in failed_assets:
560                     returns[asset] = np.nan
561                 else:
562                     previous_price = current_prices[asset].shift(1)
563
564                     if np.isnan(previous_price):
565                         returns[asset] = np.nan
566                     else:
567                         returns[asset] = (current_price - previous_price) / previous_price
568
569             else:
570                 failed_assets.append(asset)
571
572         mean_returns = np.mean(list(returns.values()))
573
574         cov_matrix = np.cov(list(returns.values()))
575
576         std_deviation = np.sqrt(np.diag(cov_matrix))
577
578         min_variance_portfolio = np.linalg.inv(cov_matrix).dot(mean_returns)
579
580         return min_variance_portfolio
581
582
583     def get_max_return_portfolio(self, asset_list=None, start_date=None,
584                                 end_date=None):
585         """Get max return portfolio for portfolio"""
586
587         if asset_list is None:
588             asset_list = self.asset_list
589
590         current_prices = {}
591         failed_assets = []
592
593         for asset in asset_list:
594             try:
595                 ticker = yf.Ticker(asset)
596                 hist = ticker.history(start=start_date, end=
597                                       end_date, interval='1d')
598
599                 if not hist.empty:
600                     current_prices[asset] = hist['Close'].iloc[-1]
601                 else:
602                     failed_assets.append(asset)
603             except Exception as e:
604                 failed_assets.append(asset)
605                 continue
606
607         if len(failed_assets) == len(asset_list):
608             return np.nan
609
610         if len(failed_assets) > 0:
611             asset_list = [asset for asset in asset_list if asset not in failed_assets]
612
613         if len(asset_list) == 0:
614             return np.nan
615
616         returns = {}
617
618         for asset in asset_list:
619             if asset in current_prices:
620                 current_price = current_prices[asset]
621
622                 if asset in failed_assets:
623                     returns[asset] = np.nan
624                 else:
625                     previous_price = current_prices[asset].shift(1)
626
627                     if np.isnan(previous_price):
628                         returns[asset] = np.nan
629                     else:
630                         returns[asset] = (current_price - previous_price) / previous_price
631
632             else:
633                 failed_assets.append(asset)
634
635         mean_returns = np.mean(list(returns.values()))
636
637         cov_matrix = np.cov(list(returns.values()))
638
639         std_deviation = np.sqrt(np.diag(cov_matrix))
640
641         max_return_portfolio = np.linalg.inv(cov_matrix).dot(mean_returns)
642
643         return max_return_portfolio
644
645
646     def get_efficient_frontier(self, asset_list=None, start_date=None,
647                               end_date=None):
648         """Get efficient frontier for portfolio"""
649
650         if asset_list is None:
651             asset_list = self.asset_list
652
653         current_prices = {}
654         failed_assets = []
655
656         for asset in asset_list:
657             try:
658                 ticker = yf.Ticker(asset)
659                 hist = ticker.history(start=start_date, end=
660                                       end_date, interval='1d')
661
662                 if not hist.empty:
663                     current_prices[asset] = hist['Close'].iloc[-1]
664                 else:
665                     failed_assets.append(asset)
666             except Exception as e:
667                 failed_assets.append(asset)
668                 continue
669
670         if len(failed_assets) == len(asset_list):
671             return np.nan
672
673         if len(failed_assets) > 0:
674             asset_list = [asset for asset in asset_list if asset not in failed_assets]
675
676         if len(asset_list) == 0:
677             return np.nan
678
679         returns = {}
680
681         for asset in asset_list:
682             if asset in current_prices:
683                 current_price = current_prices[asset]
684
685                 if asset in failed_assets:
686                     returns[asset] = np.nan
687                 else:
688                     previous_price = current_prices[asset].shift(1)
689
690                     if np.isnan(previous_price):
691                         returns[asset] = np.nan
692                     else:
693                         returns[asset] = (current_price - previous_price) / previous_price
694
695             else:
696                 failed_assets.append(asset)
697
698         mean_returns = np.mean(list(returns.values()))
699
700         cov_matrix = np.cov(list(returns.values()))
701
702         std_deviation = np.sqrt(np.diag(cov_matrix))
703
704         efficient_frontier = np.linalg.inv(cov_matrix).dot(mean_returns)
705
706         return efficient_frontier
707
708
709     def get_optimal_weights(self, asset_list=None, start_date=None,
710                            end_date=None, risk_free_rate=0.0):
711         """Get optimal weights for portfolio"""
712
713         if asset_list is None:
714             asset_list = self.asset_list
715
716         current_prices = {}
717         failed_assets = []
718
719         for asset in asset_list:
720             try:
721                 ticker = yf.Ticker(asset)
722                 hist = ticker.history(start=start_date, end=
723                                       end_date, interval='1d')
724
725                 if not hist.empty:
726                     current_prices[asset] = hist['Close'].iloc[-1]
727                 else:
728                     failed_assets.append(asset)
729             except Exception as e:
730                 failed_assets.append(asset)
731                 continue
732
733         if len(failed_assets) == len(asset_list):
734             return np.nan
735
736         if len(failed_assets) > 0:
737             asset_list = [asset for asset in asset_list if asset not in failed_assets]
738
739         if len(asset_list) == 0:
740             return np.nan
741
742         returns = {}
743
744         for asset in asset_list:
745             if asset in current_prices:
746                 current_price = current_prices[asset]
747
748                 if asset in failed_assets:
749                     returns[asset] = np.nan
750                 else:
751                     previous_price = current_prices[asset].shift(1)
752
753                     if np.isnan(previous_price):
754                         returns[asset] = np.nan
755                     else:
756                         returns[asset] = (current_price - previous_price) / previous_price
757
758             else:
759                 failed_assets.append(asset)
760
761         mean_returns = np.mean(list(returns.values()))
762
763         cov_matrix = np.cov(list(returns.values()))
764
765         std_deviation = np.sqrt(np.diag(cov_matrix))
766
767         optimal_weights = np.linalg.inv(cov_matrix).dot(mean_returns)
768
769         return optimal_weights
770
771
772     def get_optimal_portfolio(self, asset_list=None, start_date=None,
773                               end_date=None, risk_free_rate=0.0):
774         """Get optimal portfolio for portfolio"""
775
776         if asset_list is None:
777             asset_list = self.asset_list
778
779         current_prices = {}
780         failed_assets = []
781
782         for asset in asset_list:
783             try:
784                 ticker = yf.Ticker(asset)
785                 hist = ticker.history(start=start_date, end=
786                                       end_date, interval='1d')
787
788                 if not hist.empty:
789                     current_prices[asset] = hist['Close'].iloc[-1]
790                 else:
791                     failed_assets.append(asset)
792             except Exception as e:
793                 failed_assets.append(asset)
794                 continue
795
796         if len(failed_assets) == len(asset_list):
797             return np.nan
798
799         if len(failed_assets) > 0:
800             asset_list = [asset for asset in asset_list if asset not in failed_assets]
801
802         if len(asset_list) == 0:
803             return np.nan
804
805         returns = {}
806
807         for asset in asset_list:
808             if asset in current_prices:
809                 current_price = current_prices[asset]
810
811                 if asset in failed_assets:
812                     returns[asset] = np.nan
813                 else:
814                     previous_price = current_prices[asset].shift(1)
815
816                     if np.isnan(previous_price):
817                         returns[asset] = np.nan
818                     else:
819                         returns[asset] = (current_price - previous_price) / previous_price
820
821             else:
822                 failed_assets.append(asset)
823
824         mean_returns = np.mean(list(returns.values()))
825
826         cov_matrix = np.cov(list(returns.values()))
827
828         std_deviation = np.sqrt(np.diag(cov_matrix))
829
830         optimal_portfolio = np.linalg.inv(cov_matrix).dot(mean_returns)
831
832         return optimal_portfolio
833
834
835     def get_optimal_weights_with_constraint(self, asset_list=None,
836                                            constraint_type='le',
837                                            constraint_value=[1.0],
838                                            start_date=None, end_date=None,
839                                            risk_free_rate=0.0):
840         """Get optimal weights for portfolio with constraint"""
841
842         if asset_list is None:
843             asset_list = self.asset_list
844
845         current_prices = {}
846         failed_assets = []
847
848         for asset in asset_list:
849             try:
850                 ticker = yf.Ticker(asset)
851                 hist = ticker.history(start=start_date, end=
852                                       end_date, interval='1d')
853
854                 if not hist.empty:
855                     current_prices[asset] = hist['Close'].iloc[-1]
856                 else:
857                     failed_assets.append(asset)
858             except Exception as e:
859                 failed_assets.append(asset)
860                 continue
861
862         if len(failed_assets) == len(asset_list):
863             return np.nan
864
865         if len(failed_assets) > 0:
866             asset_list = [asset for asset in asset_list if asset not in failed_assets]
867
868         if len(asset_list) == 0:
869             return np.nan
870
871         returns = {}
872
873         for asset in asset_list:
874             if asset in current_prices:
875                 current_price = current_prices[asset]
876
877                 if asset in failed_assets:
878                     returns[asset] = np.nan
879                 else:
880                     previous_price = current_prices[asset].shift(1)
881
882                     if np.isnan(previous_price):
883                         returns[asset] = np.nan
884                     else:
885                         returns[asset] = (current_price - previous_price) / previous_price
886
887             else:
888                 failed_assets.append(asset)
889
890         mean_returns = np.mean(list(returns.values()))
891
892         cov_matrix = np.cov(list(returns.values()))
893
894         std_deviation = np.sqrt(np.diag(cov_matrix))
895
896         optimal_weights = np.linalg.inv(cov_matrix).dot(mean_returns)
897
898         if constraint_type == 'le':
899             optimal_weights = np.maximum(optimal_weights, constraint_value)
900
901         if constraint_type == 'ge':
902             optimal_weights = np.minimum(optimal_weights, constraint_value)
903
904         return optimal_weights
905
906
907     def get_optimal_portfolio_with_constraint(self, asset_list=None,
908                                              constraint_type='le',
909                                              constraint_value=[1.0],
910                                              start_date=None, end_date=None,
911                                              risk_free_rate=0.0):
909
910         if asset_list is None:
911             asset_list = self.asset_list
912
913         current_prices = {}
914         failed_assets = []
915
916         for asset in asset_list:
917             try:
918                 ticker = yf.Ticker(asset)
919                 hist = ticker.history(start=start_date, end=
920                                       end_date, interval='1d')
921
922                 if not hist.empty:
923                     current_prices[asset] = hist['Close'].iloc[-1]
924                 else:
925                     failed_assets.append(asset)
926             except Exception as e:
927                 failed_assets.append(asset)
928                 continue
929
930         if len(failed_assets) == len(asset_list):
931             return np.nan
932
933         if len(failed_assets) > 0:
934             asset_list = [asset for asset in asset_list if asset not in failed_assets]
935
936         if len(asset_list) == 0:
937             return np.nan
938
939         returns = {}
940
941         for asset in asset_list:
942             if asset in current_prices:
943                 current_price = current_prices[asset]
944
945                 if asset in failed_assets:
946                     returns[asset] = np.nan
947                 else:
948                     previous_price = current_prices[asset].shift(1)
949
950                     if np.isnan(previous_price):
951                         returns[asset] = np.nan
952                     else:
953                         returns[asset] = (current_price - previous_price) / previous_price
954
955             else:
956                 failed_assets.append(asset)
957
958         mean_returns = np.mean(list(returns.values()))
959
960         cov_matrix = np.cov(list(returns.values()))
961
962         std_deviation = np.sqrt(np.diag(cov_matrix))
963
964         optimal_portfolio = np.linalg.inv(cov_matrix).dot(mean_returns)
965
966         if constraint_type == 'le':
967             optimal_portfolio = np.maximum(optimal_portfolio, constraint_value)
968
969         if constraint_type == 'ge':
970             optimal_portfolio = np.minimum(optimal_portfolio, constraint_value)
971
972         return optimal_portfolio
973
974
975     def get_optimal_weights_with_constraint_and_max_sharpe_ratio(self,
976                                                                asset_list=None,
977                                                                constraint_type='le',
978                                                                constraint_value=[1.0],
979                                                                start_date=None,
980                                                                end_date=None,
981                                                                risk_free_rate=0.0):
980
981         if asset_list is None:
982             asset_list = self.asset_list
983
984         current_prices = {}
985         failed_assets = []
986
987         for asset in asset_list:
988             try:
989                 ticker = yf.Ticker(asset)
990                 hist = ticker.history(start=start_date, end=
991                                       end_date, interval='1d')
992
993                 if not hist.empty:
994                     current_prices[asset] = hist['Close'].iloc[-1]
995                 else:
996                     failed_assets.append(asset)
997             except Exception as e:
998                 failed_assets.append(asset)
999                 continue
1000
1001         if len(failed_assets) == len(asset_list):
1002             return np.nan
1003
1004         if len(failed_assets) > 0:
1005             asset_list = [asset for asset in asset_list if asset not in failed_assets]
1006
1007         if len(asset_list) == 0:
1008             return np.nan
1009
1010         returns = {}
1011
1012         for asset in asset_list:
1013             if asset in current_prices:
1014                 current_price = current_prices[asset]
1015
1016                 if asset in failed_assets:
1017                     returns[asset] = np.nan
1018                 else:
1019                     previous_price = current_prices[asset].shift(1)
1020
1021                     if np.isnan(previous_price):
1022                         returns[asset] = np.nan
1023                     else:
1024                         returns[asset] = (current_price - previous_price) / previous_price
1025
1026             else:
1027                 failed_assets.append(asset)
1028
1029         mean_returns = np.mean(list(returns.values()))
1030
1031         cov_matrix = np.cov(list(returns.values()))
1032
1033         std_deviation = np.sqrt(np.diag(cov_matrix))
1034
1035         optimal_weights = np.linalg.inv(cov_matrix).dot(mean_returns)
1036
1037         if constraint_type == 'le':
1038             optimal_weights = np.maximum(optimal_weights, constraint_value)
1039
1040         if constraint_type == 'ge':
1041             optimal_weights = np.minimum(optimal_weights, constraint_value)
1042
1043         if np.all(optimal_weights <= 1.0) and np.all(optimal_weights >= 0.0):
1044             max_sharpe_ratio = (mean_returns - risk_free_rate) / std_deviation
1045
1046             return max_sharpe_ratio
1047
1048         else:
1049             return np.nan
1050
1051
1052     def get_optimal_portfolio_with_constraint_and_max_sharpe_ratio(self,
1053                                                                    asset_list=None,
1054                                                                    constraint_type='le',
1055                                                                    constraint_value=[1.0],
1056                                                                    start_date=None,
1057                                                                    end_date=None,
1058                                                                    risk_free_rate=0.0):
1059
1060         if asset_list is None:
1061             asset_list = self.asset_list
1062
1063         current_prices = {}
1064         failed_assets = []
1065
1066         for asset in asset_list:
1067             try:
1068                 ticker = yf.Ticker(asset)
1069                 hist = ticker.history(start=start_date, end=
1070                                       end_date, interval='1d')
1071
1072                 if not hist.empty:
1073                     current_prices[asset] = hist['Close'].iloc[-1]
1074                 else:
1075                     failed_assets.append(asset)
1076             except Exception as e:
1077                 failed_assets.append(asset)
1078                 continue
1079
1080         if len(failed_assets) == len(asset_list):
1081             return np.nan
1082
1083         if len(failed_assets) > 0:
1084             asset_list = [asset for asset in asset_list if asset not in failed_assets]
1085
1086         if len(asset_list) == 0:
1087             return np.nan
1088
1089         returns = {}
1090
1091         for asset in asset_list:
1092             if asset in current_prices:
1093                 current_price = current_prices[asset]
1094
1095                 if asset in failed_assets:
1096                     returns[asset] = np.nan
1097                 else:
1098                     previous_price = current_prices[asset].shift(1)
1099
1100                     if np.isnan(previous_price):
1101                         returns[asset] = np.nan
1102                     else:
1103                         returns[asset] = (current_price - previous_price) / previous_price
1104
1105             else:
1106                 failed_assets.append(asset)
1107
1108         mean_returns = np.mean(list(returns.values()))
1109
1110         cov_matrix = np.cov(list(returns.values()))
1111
1112         std_deviation = np.sqrt(np.diag(cov_matrix))
1113
1114         optimal_portfolio = np.linalg.inv(cov_matrix).dot(mean_returns)
1115
1116         if constraint_type == 'le':
1117             optimal_portfolio = np.maximum(optimal_portfolio, constraint_value)
1118
1119         if constraint_type == 'ge':
1120             optimal_portfolio = np.minimum(optimal_portfolio, constraint_value)
1121
1122         if np.all(optimal_portfolio <= 1.0) and np.all(optimal_portfolio >= 0.0):
1123             max_sharpe_ratio = (mean_returns - risk_free_rate) / std_deviation
1124
1125             return max_sharpe_ratio
1126
1127         else:
1128             return np.nan
1129
1130
1131     def get_optimal_weights_with_constraint_and_max_return(self,
1132                                                            asset_list=None,
1133                                                            constraint_type='le',
1134                                                            constraint_value=[1.0],
1135                                                            start_date=None,
1136                                                            end_date=None,
1137                                                            risk_free_rate=0.0):
1138
1139         if asset_list is None:
1140             asset_list = self.asset_list
1141
1142         current_prices = {}
1143         failed_assets = []
1144
1145         for asset in asset_list:
1146             try:
1147                 ticker = yf.Ticker(asset)
1148                 hist = ticker.history(start=start_date, end=
1149                                       end_date, interval='1d')
1150
1151                 if not hist.empty:
1152                     current_prices[asset] = hist['Close'].iloc[-1]
1153                 else:
1154                     failed_assets.append(asset)
1155             except Exception as e:
1156                 failed_assets.append(asset)
1157                 continue
1158
1159         if len(failed_assets) == len(asset_list):
1160             return np.nan
1161
1162         if len(failed_assets) > 0:
1163             asset_list = [asset for asset in asset_list if asset not in failed_assets]
1164
1165         if len(asset_list) == 0:
1166             return np.nan
1167
1168         returns = {}
1169
1170         for asset in asset_list:
1171             if asset in current_prices:
1172                 current_price = current_prices[asset]
1173
1174                 if asset in failed_assets:
1175                     returns[asset] = np.nan
1176                 else:
1177                     previous_price = current_prices[asset].shift(1)
1178
1179                     if np.isnan(previous_price):
1180                         returns[asset] = np.nan
1181                     else:
1182                         returns[asset] = (current_price - previous_price) / previous_price
1183
1184             else:
1185                 failed_assets.append(asset)
1186
1187         mean_returns = np.mean(list(returns.values()))
1188
1189         cov_matrix = np.cov(list(returns.values()))
1190
1191         std_deviation = np.sqrt(np.diag(cov_matrix))
1192
1193         optimal_weights = np.linalg.inv(cov_matrix).dot(mean_returns)
1194
1195         if constraint_type == 'le':
1196             optimal_weights = np.maximum(optimal_weights, constraint_value)
1197
1198         if constraint_type == 'ge':
1199             optimal_weights = np.minimum(optimal_weights, constraint_value)
1200
1201         if np.all(optimal_weights <= 1.0) and np.all(optimal_weights >= 0.0):
1202             max_return = (mean_returns - risk_free_rate) / std_deviation
1203
1204             return max_return
1205
1206         else:
1207             return np.nan
1208
1209
1210     def get_optimal_portfolio_with_constraint_and_max_return(self,
1211                                                               asset_list=None,
1212                                                               constraint_type='le',
1213                                                               constraint_value=[1.0],
1214                                                               start_date=None,
1215                                                               end_date=None,
1216                                                               risk_free_rate=0.0):
1217
1218         if asset_list is None:
1219             asset_list = self.asset_list
1220
1221         current_prices = {}
1222         failed_assets = []
1223
1224         for asset in asset_list:
1225             try:
1226                 ticker = yf.Ticker(asset)
1227                 hist = ticker.history(start=start_date, end=
1228                                       end_date, interval='1d')
1229
1230                 if not hist.empty:
1231                     current_prices[asset] = hist['Close'].iloc[-1]
1232                 else:
1233                     failed_assets.append(asset)
1234             except Exception as e:
1235                 failed_assets.append(asset)
1236                 continue
1237
1238         if len(failed_assets) == len(asset_list):
1239             return np.nan
1240
1241         if len(failed_assets) > 0:
1242             asset_list = [asset for asset in asset_list if asset not in failed_assets]
1243
1244         if len(asset_list) == 0:
1245             return np.nan
1246
1247         returns = {}
1248
1249         for asset in asset_list:
1250             if asset in current_prices:
1251                 current_price = current_prices[asset]
1252
1253                 if asset in failed_assets:
1254                     returns[asset] = np.nan
1255                 else:
1256                     previous_price = current_prices[asset].shift(1)
1257
1258                     if np.isnan(previous_price):
1259                         returns[asset] = np.nan
1260                     else:
1261                         returns[asset] = (current_price - previous_price) / previous_price
1262
1263             else:
1264                 failed_assets.append(asset)
1265
1266         mean_returns = np.mean(list(returns.values()))
1267
1268         cov_matrix = np.cov(list(returns.values()))
1269
1270         std_deviation = np.sqrt(np.diag(cov_matrix))
1271
1272         optimal_portfolio = np.linalg.inv(cov_matrix).dot(mean_returns)
1273
1274         if constraint_type == 'le':
1275             optimal_portfolio = np.maximum(optimal_portfolio, constraint_value)
1276
1277         if constraint_type == 'ge':
1278             optimal_portfolio = np.minimum(optimal_portfolio, constraint_value)
1279
1280         if np.all(optimal_portfolio <= 1.0) and np.all(optimal_portfolio >= 0.0):
1281             max_return = (mean_returns - risk_free_rate) / std_deviation
1282
1283             return max_return
1284
1285         else:
1286             return np.nan
1287
1288
1289     def get_optimal_weights_with_constraint_and_max_cvar(self,
1290                                                            asset_list=None,
1291                                                            constraint_type='le',
1292                                                            constraint_value=[1.0],
1293                                                            start_date=None,
1294                                                            end_date=None,
1295                                                            risk_free_rate=0.0):
1296
1297         if asset_list is None:
1298             asset_list = self.asset_list
1299
1300         current_prices = {}
1301         failed_assets = []
1302
1303         for asset in asset_list:
1304             try:
1305                 ticker = yf.Ticker(asset)
1306                 hist = ticker.history(start=start_date, end=
1307                                       end_date, interval='1d')
1308
1309                 if not hist.empty:
1310                     current_prices[asset] = hist['Close'].iloc[-1]
1311                 else:
1312                     failed_assets.append(asset)
1313             except Exception as e:
1314                 failed_assets.append(asset)
1315                 continue
1316
1317         if len(failed_assets) == len(asset_list):
1318             return np.nan
1319
1320         if len(failed_assets) > 0:
1321             asset_list = [asset for asset in asset_list if asset not in failed_assets]
1322
1323         if len(asset_list) == 0:
1324             return np.nan
1325
1326         returns = {}
1327
1328         for asset in asset_list:
1329             if asset in current_prices:
1330                 current_price = current_prices[asset]
1331
1332                 if asset in failed_assets:
1333                     returns[asset] = np.nan
1334                 else:
1335                     previous_price = current_prices[asset].shift(1)
1336
1337                     if np.isnan(previous_price):
1338                         returns[asset] = np.nan
1339                     else:
1340                         returns[asset] = (
```

```

196         if not hist.empty:
197             current_prices[asset] = hist['Close'].iloc
198                 [-1]
199         else:
200             failed_assets.append(asset)
201     except Exception as e:
202         failed_assets.append(asset)
203         continue
204
205     if failed_assets:
206         print(f"      Failed to get current prices for {len(
207             failed_assets)} assets: {failed_assets[:5]}...")
208
209     print(f"      Got current prices for {len(current_prices)}
210           assets")
211     return current_prices
212
213 def save_data(self, asset_list=None, period='64d', output_dir
214 = '.'):
215     """Save data to CSV files"""
216     if asset_list is None:
217         asset_list = self.asset_list
218
219     # Get data
220     closing_prices = self.get_data(asset_list=asset_list,
221                                     period=period)
222     if closing_prices.empty:
223         print("      No data to save")
224         return
225
226     # Calculate log returns
227     log_returns = closing_prices.pct_change().dropna()
228
229     # Save files
230     closing_prices.to_csv(f'{output_dir}/closing_prices.csv')
231     log_returns.to_csv(f'{output_dir}/log_returns.csv')
232
233     print(f"      Data saved to {output_dir}/")
234     print(f"      closing_prices.csv: {closing_prices.shape}")
235     print(f"      log_returns.csv: {log_returns.shape}")
236
237 def main():
238     """Test data gathering functionality"""
239     gatherer = DataGatherer(use_monte_carlo_selection=True,
240                             top_n_assets=10)
241
242     print("\n      Testing data download...")
243     data = gatherer.get_data()
244
245

```

```

239     if not data.empty:
240         print(f"    Data download successful: {data.shape}")
241         gatherer.save_data()
242     else:
243         print("    Data download failed")
244
245 if __name__ == "__main__":
246     main()

```

**main\_optimized.py (Python).**

```

1 #!/usr/bin/env python3
2 """
3 Portfolio Optimization System - Optimized Version
4
5 High-performance portfolio optimization using C implementations
6     for computational bottlenecks.
7 Implements Moving Block Bootstrap, Monte Carlo simulation, and
8     Newton-Raphson optimization.
9 """
10
11
12 import numpy as np
13 import pandas as pd
14 import matplotlib.pyplot as plt
15 import seaborn as sns
16 import warnings
17 import time
18 import os
19 import sys
20 import ctypes
21 from datetime import datetime
22 from itertools import combinations
23 from contextlib import contextmanager
24 from get_data_optimized import DataGatherer
25
26 # Configuration
27 plt.style.use('fivethirtyeight')
28 sns.set_palette('colorblind')
29 warnings.filterwarnings('ignore')
30 np.random.seed(1987)
31
32 @contextmanager
33 def suppress_output():
34     """Suppress stdout/stderr during C function calls"""
35     with open(os.devnull, "w") as devnull:
36         old_stdout, old_stderr = sys.stdout, sys.stderr
37         sys.stdout = sys.stderr = devnull
38         try:
39             yield
40         finally:
41             sys.stdout = old_stdout
42             sys.stderr = old_stderr

```

```

38         sys.stdout, sys.stderr = old_stdout, old_stderr
39
40 class PortfolioOptimizer:
41     """Portfolio optimization using C implementations for
42     performance"""
43
44     def __init__(self, portfolio_size=5):
45         self.portfolio_size = portfolio_size
46         self.data_gatherer = DataGatherer(
47             use_monte_carlo_selection=True, top_n_assets=15)
48         self.c_lib = self._load_c_library()
49
50     def _load_c_library(self):
51         """Load and configure C library"""
52         lib_path = './functions.so'
53         if not os.path.exists(lib_path):
54             raise RuntimeError("C library not found. Compile with
55                 : gcc -shared -fPIC -o functions.so functions.c -
56                 lm")
57
58         c_lib = ctypes.CDLL(lib_path)
59         self._setup_c_functions(c_lib)
60         print("    C library loaded successfully")
61
62         # Verify C functions are available
63         required_functions = ['moving_block_bootstrap', ,
64             'monte_carlo_simulation', ,
65             'optimize_portfolio_newton_raphson']
66         for func_name in required_functions:
67             if not hasattr(c_lib, func_name):
68                 raise RuntimeError(f"C function {func_name} not
69                     found in library")
70
71         print("    All C functions verified and ready")
72         return c_lib
73
74     def _setup_c_functions(self, c_lib):
75         """Configure C function signatures"""
76         # Moving block bootstrap
77         c_lib.moving_block_bootstrap.argtypes = [
78             ctypes.POINTER(ctypes.c_double), ctypes.c_int, ctypes
79                 .c_int,
80             ctypes.c_int, ctypes.c_int, ctypes.c_int
81         ]
82         c_lib.moving_block_bootstrap.restype = ctypes.POINTER(
83             ctypes.c_double)
84
85         # Monte Carlo simulation
86         c_lib.monte_carlo_simulation.argtypes = [
87

```

```

78         ctypes.c_double, ctypes.POINTER(ctypes.c_double),
79         ctypes.c_int, ctypes.c_int, ctypes.c_int
80     ]
81     c_lib.monte_carlo_simulation.restype = ctypes.POINTER(
82         ctypes.c_double)
83
84     # Newton-Raphson optimization
85     c_lib.optimize_portfolio_newton_raphson.argtypes = [
86         ctypes.POINTER(ctypes.c_double), ctypes.c_int, ctypes
87             .c_int,
88         ctypes.POINTER(ctypes.c_double), ctypes.c_double,
89             ctypes.c_int, ctypes.c_double
90     ]
91     c_lib.optimize_portfolio_newton_raphson.restype = ctypes.
92         POINTER(ctypes.c_double)
93
94     def _prepare_data(self, data):
95         """Clean and prepare data for C functions"""
96         if hasattr(data, 'dropna'):
97             data = data.dropna()
98         return np.array(data)[~np.isnan(data)]
99
100    def moving_block_bootstrap(self, log_returns, n_bootstrap
101        =1000, sample_size=63,
102            block_size=None, optimize_block_size
103                =True):
104        """Moving block bootstrap with optimized block size"""
105        if block_size is None and optimize_block_size:
106            block_size = self._choose_optimal_block_size(
107                log_returns)
108        elif block_size is None:
109            block_size = 5
110
111        log_returns_array = self._prepare_data(log_returns)
112        c_array = (ctypes.c_double * len(log_returns_array))(*
113            log_returns_array)
114
115        with suppress_output():
116            result_ptr = self.c_lib.moving_block_bootstrap(
117                c_array, len(log_returns_array), n_bootstrap,
118                    sample_size, block_size, 1987
119            )
120
121        if not result_ptr:
122            raise RuntimeError("Bootstrap failed - C function
123                returned NULL")
124
125        return np.ctypeslib.as_array(result_ptr, shape=(n_bootstrap,
126            sample_size)).copy()

```

```

116
117     def monte_carlo_simulation(self, S0, bootstrap_samples,
118         iterations=5000):
119         """Monte Carlo simulation using bootstrap samples"""
120         if not isinstance(bootstrap_samples, np.ndarray) or
121             bootstrap_samples.ndim != 2:
122             raise ValueError("bootstrap_samples must be 2D numpy
123                             array")
124
125         n_bootstrap, sample_size = bootstrap_samples.shape
126         bootstrap_flat = bootstrap_samples.flatten()
127         c_array = (ctypes.c_double * len(bootstrap_flat))(*
128             bootstrap_flat)
129
130         with suppress_output():
131             result_ptr = self.c_lib.monte_carlo_simulation(
132                 ctypes.c_double(S0), c_array, n_bootstrap,
133                 sample_size, iterations, 1987
134             )
135
136         if not result_ptr:
137             raise RuntimeError("Monte Carlo failed - C function
138                             returned NULL")
139
140         # Extract final prices and price paths from C result
141         total_size = iterations + iterations * sample_size
142         result_array = np.ctypeslib.as_array(result_ptr, shape=(
143             total_size,)).copy()
144
145         # Split results: first 'iterations' elements are final
146         # prices
147         # remaining elements are price paths (iterations x
148         # sample_size)
149         final_prices = result_array[:iterations]
150         price_paths = result_array[iterations:].reshape(
151             iterations, sample_size)
152
153         return final_prices, price_paths
154
155     def optimize_portfolio_newton_raphson(self, arrival_values_df,
156         asset_combination,
157                                         risk_free_rate=0.0,
158                                         max_iterations=100,
159                                         tolerance=1e-6):
160         """Newton-Raphson portfolio optimization"""
161         selected_values = arrival_values_df[list(
162             asset_combination)]
163         n_assets = len(asset_combination)

```

```
151     arrival_flat = selected_values.values.flatten()
152     c_arrival = (ctypes.c_double * len(arrival_flat))(*
153                 arrival_flat)
154
155     initial_weights = np.ones(n_assets) / n_assets
156     c_weights = (ctypes.c_double * n_assets)(*initial_weights
157             )
158
159     with suppress_output():
160         result_ptr = self.c_lib.
161             optimize_portfolio_newton_raphson(
162                 c_arrival, n_assets, len(selected_values),
163                 c_weights,
164                 ctypes.c_double(risk_free_rate), max_iterations,
165                 ctypes.c_double(tolerance)
166             )
167
168     if not result_ptr:
169         raise RuntimeError("Newton-Raphson optimization
170                             failed")
171
172     return np.ctypeslib.asarray(result_ptr, shape=(n_assets
173             ,)).copy()
174
175 def _choose_optimal_block_size(self, log_returns, method='
176 theoretical'):
177     """Choose optimal block size using Politis-Romano
178     theoretical method"""
179     log_returns_array = self._prepare_data(log_returns)
180     n = len(log_returns_array)
181
182     # Use only Politis-Romano theoretical method
183     return self._theoretical_block_size(n)
184
185 def _theoretical_block_size(self, n):
186     """Calculate theoretical optimal block size"""
187     b_opt = max(1, int(1.5 * (n ** (1/3))))
188     return min(b_opt, n // 4)
189
190
191
192 def calculate_portfolio_returns(self, weights, arrival_values
193             ):
194     """Calculate portfolio returns for given weights"""
195     return np.dot(arrival_values.values, weights)
196
197 def calculate_sharpe_ratio(self, portfolio_returns,
198     risk_free_rate=0.0):
199     """Calculate Sharpe ratio"""
```

```

189         mean_return, std_return = np.mean(portfolio_returns), np.
190             std(portfolio_returns)
191         return (mean_return - risk_free_rate) / std_return if
192             std_return > 0 else 0
193
194     def optimize_single_portfolio(self, asset_combination,
195         arrival_values_df, risk_free_rate=0.0):
196         """Optimize a single portfolio combination"""
197         try:
198             optimal_weights = self.
199                 optimize_portfolio_newton_raphson(
200                     arrival_values_df, asset_combination,
201                     risk_free_rate
202                 )
203
204             portfolio_values = self.calculate_portfolio_returns(
205                 optimal_weights, arrival_values_df[list(
206                     asset_combination)])
207
208             return {
209                 'asset_combination': asset_combination,
210                 'success': True,
211                 'optimal_weights': optimal_weights,
212                 'optimal_sharpe': self.calculate_sharpe_ratio(
213                     portfolio_values, risk_free_rate),
214                 'optimal_mean': np.mean(portfolio_values),
215                 'optimal_std': np.std(portfolio_values),
216                 'method': 'Newton-Raphson'
217             }
218         except Exception as e:
219             return {
220                 'asset_combination': asset_combination,
221                 'success': False,
222                 'optimal_sharpe': -np.inf,
223                 'error': str(e)
224             }
225
226     def optimize_all_combinations(self, portfolio_combinations,
227         arrival_values_df, risk_free_rate=0.0):
228         """Optimize all portfolio combinations"""
229         print(f"Optimizing {len(portfolio_combinations)}
230             portfolio combinations...")
231         print("=" * 60)
232
233         results = []
234         for i, combination in enumerate(portfolio_combinations,
235             1):
236             if i % 50 == 0:

```

```

226         print(f"Progress: {i}/{len(portfolio_combinations
227             )} combinations processed")
228
229     result = self.optimize_single_portfolio(combination,
230         arrival_values_df, risk_free_rate)
231     results.append(result)
232
233     return results
234
235 def run_full_optimization(self):
236     """Run complete portfolio optimization pipeline"""
237     print("      Starting Portfolio Optimization")
238     print("=" * 50)
239
240     # Get data
241     print("      Gathering and processing data...")
242     closing_prices = self.data_gatherer.get_data()
243     log_returns = closing_prices.pct_change().dropna()
244     current_prices = self.data_gatherer.get_current_prices()
245
246     # Ensure we only use assets with both historical data and
247     # current prices
248     available_assets = set(closing_prices.columns) & set(
249         current_prices.keys())
250     if len(available_assets) < self.portfolio_size:
251         raise RuntimeError(f"Not enough assets with complete
252             data ({len(available_assets)}) for portfolio size
253             {self.portfolio_size}")
254
255     # Filter data to only available assets
256     log_returns = log_returns[list(available_assets)]
257     print(f"      Using {len(available_assets)} assets with
258         complete data")
259
260     # Generate arrival values using Monte Carlo
261     print("      Running Monte Carlo simulations...")
262     print(f"      Bootstrap: 5000 samples per asset")
263     print(f"      Monte Carlo: 5000 iterations per asset")
264     print(f"      Sample size: 63 days (consistent across all
265         assets)")
266
267     arrival_values = {}
268     price_paths_data = {} # Store price paths for
269         visualization
270     for i, asset in enumerate(log_returns.columns, 1):
271         print(f"      Processing asset {i}/{len(log_returns.
272             columns)}: {asset}")
273         S0 = current_prices[asset]

```

```

265         bootstrap_samples = self.moving_block_bootstrap(
266             log_returns[asset], n_bootstrap=5000, sample_size
267             =63)
268         final_prices, price_paths = self.
269             monte_carlo_simulation(S0, bootstrap_samples,
270             iterations=5000)
271
272         # Store final prices for portfolio optimization
273         arrival_values[asset] = final_prices
274
275         # Store price paths for visualization
276         price_paths_data[asset] = {
277             'S0': S0,
278             'price_paths': price_paths,
279             'final_prices': final_prices,
280             'bootstrap_samples': bootstrap_samples
281         }
282
283         # Create Monte Carlo visualization for this asset
284         self._create_asset_monte_carlo_plot(asset, S0,
285             final_prices, price_paths, bootstrap_samples)
286
287         arrival_values_df = pd.DataFrame(arrival_values)
288
289         # Check if we have enough assets
290         n_assets = len(arrival_values_df.columns)
291         if n_assets < self.portfolio_size:
292             raise RuntimeError(f"Not enough assets available ({n_assets}) for portfolio size {self.portfolio_size}")
293
294         # Generate portfolio combinations
295         portfolio_combinations = list(combinations(
296             arrival_values_df.columns, self.portfolio_size))
297         print(f"      Testing {len(portfolio_combinations)}"
298             "combinations of {self.portfolio_size} assets from {n_assets} total")
299
300         # Optimize all combinations
301         all_results = self.optimize_all_combinations(
302             portfolio_combinations, arrival_values_df)
303
304         # Find best portfolio
305         successful_results = [r for r in all_results if r['
306             success']]
307         if not successful_results:
308             raise RuntimeError("No successful portfolio
309                 optimizations")

```

```

301     best_portfolio = max(successful_results, key=lambda x: x[
302         'optimal_sharpe'])
303
303     print(f"\n      Best Portfolio Found:")
304     print(f"      Assets: {best_portfolio['asset_combination']}")
305     print(f"      Sharpe Ratio: {best_portfolio['optimal_sharpe']:.6f}")
306     print(f"      Expected Return: {best_portfolio['optimal_mean']:.6f}")
307     print(f"      Volatility: {best_portfolio['optimal_std']:.6f}")
308
309     # Save results
310     self._save_results(best_portfolio, current_prices,
311                         arrival_values_df, all_results)
311     self._create_visualizations(best_portfolio,
312                                 current_prices, arrival_values_df, all_results)
312
313     return best_portfolio, all_results
314
315 def _save_results(self, best_portfolio, current_prices,
316                   arrival_values_df, all_results):
316     """Save optimization results to CSV files"""
317     # Best portfolio details
318     best_details = pd.DataFrame({
319         'Asset': best_portfolio['asset_combination'],
320         'Weight': best_portfolio['optimal_weights'],
321         'Current_Price': [current_prices[asset] for asset in
322                           best_portfolio['asset_combination']],
322         'Allocation': best_portfolio['optimal_weights'] *
323                         10000 # $10k portfolio
323     })
324     best_details.to_csv('best_portfolio_details.csv', index=
325                         False)
326
326     # All results
327     results_df = pd.DataFrame([
328         {
329             'Assets': str(r['asset_combination']),
330             'Sharpe_Ratio': r.get('optimal_sharpe', -np.inf),
331             'Expected_Return': r.get('optimal_mean', 0),
332             'Volatility': r.get('optimal_std', 0),
333             'Success': r['success']
334         }
335         for r in all_results
336     ])
337     results_df.to_csv('all_portfolio_results.csv', index=
337                         False)

```

```

338
339     # Arrival values
340     arrival_values_df.to_csv('arrival_values.csv')
341
342     print("      Results saved to CSV files")
343
344 def _create_visualizations(self, best_portfolio,
345     current_prices, arrival_values_df, all_results):
346     """Create comprehensive visualizations"""
347     fig, axes = plt.subplots(2, 2, figsize=(15, 12))
348     fig.suptitle('Portfolio Optimization Results', fontsize=16)
349
350     # Portfolio weights
351     assets = best_portfolio['asset_combination']
352     weights = best_portfolio['optimal_weights']
353     axes[0, 0].pie(weights, labels=assets, autopct='%.1f%%',
354                     startangle=90)
355     axes[0, 0].set_title('Optimal Portfolio Weights')
356
357     # Sharpe ratio distribution
358     sharpe_ratios = [r.get('optimal_sharpe', -np.inf) for r
359                     in all_results if r['success']]
360     axes[0, 1].hist(sharpe_ratios, bins=30, alpha=0.7,
361                     edgecolor='black')
362     axes[0, 1].axvline(best_portfolio['optimal_sharpe'],
363                         color='red', linestyle='--',
364                         label=f"Best: {best_portfolio['optimal_sharpe']:.3f}")
365     axes[0, 1].set_xlabel('Sharpe Ratio')
366     axes[0, 1].set_ylabel('Frequency')
367     axes[0, 1].set_title('Sharpe Ratio Distribution')
368     axes[0, 1].legend()
369
370     # Risk-return scatter
371     returns = [r.get('optimal_mean', 0) for r in all_results
372                 if r['success']]
373     volatilities = [r.get('optimal_std', 0) for r in
374                     all_results if r['success']]
375     axes[1, 0].scatter(volatilities, returns, alpha=0.6)
376     axes[1, 0].scatter(best_portfolio['optimal_std'],
377                         best_portfolio['optimal_mean'],
378                         color='red', s=100, marker='*', label=
379                         'Best Portfolio')
380     axes[1, 0].set_xlabel('Volatility')
381     axes[1, 0].set_ylabel('Expected Return')
382     axes[1, 0].set_title('Risk-Return Profile')
383     axes[1, 0].legend()
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
26
```

```

376     # Asset correlation heatmap
377     selected_returns = arrival_values_df[list(assets)]
378     correlation_matrix = selected_returns.corr()
379     im = axes[1, 1].imshow(correlation_matrix, cmap='coolwarm'
380                           , aspect='auto')
381     axes[1, 1].set_xticks(range(len(assets)))
382     axes[1, 1].set_yticks(range(len(assets)))
383     axes[1, 1].set_xticklabels(assets, rotation=45)
384     axes[1, 1].set_yticklabels(assets)
385     axes[1, 1].set_title('Asset Correlation Matrix')
386     plt.colorbar(im, ax=axes[1, 1])
387
388     plt.tight_layout()
389     plt.savefig('portfolio_optimization_results.png', dpi
390                 =150, bbox_inches='tight')
391     plt.close()
392
393     print("      Visualizations saved to
394           portfolio_optimization_results.png")
395
396     def _create_block_size_optimization_plot(self,
397                                               log_returns_array, asset_name="Asset"):
398         """Create block size optimization visualization using
399             Politis-Romano theoretical method"""
400         print(f"      Creating block size optimization plot for {
401               asset_name}...")
402
403         # Calculate theoretical optimal block size
404         n = len(log_returns_array)
405         theoretical_block = self._theoretical_block_size(n)
406
407         # Create visualization showing theoretical approach
408         fig, axes = plt.subplots(1, 2, figsize=(12, 6))
409         fig.suptitle(f'Politis-Romano Block Size Analysis: {
410                       asset_name}', fontsize=16)
411
412         # Plot 1: Block size calculation
413         series_lengths = np.arange(50, 1000, 10)
414         theoretical_blocks = [self._theoretical_block_size(n) for
415                               n in series_lengths]
416
417         axes[0].plot(series_lengths, theoretical_blocks, 'b-',
418                      linewidth=2, label='Politis-Romano Rule')
419         axes[0].axhline(y=theoretical_block, color='red',
420                         linestyle='--',
421                         label=f'Optimal for {asset_name}: {
422                               theoretical_block}')
423         axes[0].set_xlabel('Time Series Length (n)')
424         axes[0].set_ylabel('Optimal Block Size')

```