Equality Constraint Quadratic Programming

Mustafa Kerem Köse
Politecnico di Torino
s339018@studenti.polito.it

Mert Özcan
Politecnico di Torino
s343381@studenti.polito.it

Özgür Alkın Karaçam
Politecnico di Torino
s334297@studenti.polito.it

I. Introduction

We consider the following equality-constrained quadratic program:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n-1} x_i x_{i+1} + \sum_{i=1}^{n} x_i, \\ \text{s.t.} \quad & \sum_{i\equiv k \,(\mathrm{mod}\, K)} x_i = \epsilon, \quad k = 1,\ldots,K, \end{aligned}$$

where $\epsilon \in (0,1)$ is a fixed budget parameter and the $K \leq n$ constraints partition the index set into K equal-sum groups. After deriving the KKT optimality conditions, we implement and benchmark four solver strategies:

1) **Dense direct** solution of the full KKT system via standard linear-algebra routines.
2) **Iterative KKT** solution using GMRES on a sparse assembly of the KKT matrix, with and without preconditioning.
3) **Schur complement** reduction, where the dual system $AQ^{-1}A^T\lambda = AQ^{-1}(-c) - b$ is solved iteratively and the primal solution recovered via back-substitution.
4) **Null-space** method, which eliminates the equality constraints by parametrizing $x = Zy + x_p$ and solves the reduced unconstrained problem in the null-space basis.

We scale each approach across problem sizes $n = 2,\ 2\times10^3,\ 2\times10^4,\ 2\times10^5$ with matching constraint counts $K = 1, 100, 500, 1000$, and compare solve times, memory footprints, and numerical accuracy. Our results illustrate the trade-offs between dense direct factorization (robust but memory-intensive), iterative KKT (memory-efficient but sensitive to conditioning), Schur reduction (mixing sparse solves and small GMRES systems), and null-space (effective when $K \ll n$). We conclude with recommendations for solver selection based on problem scale and available resources.

II. Methods

A. KKT System

To solve the equality-constrained quadratic program

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}x^\top Qx + c^\top x \quad \text{s.t.} \quad Ax = b,$$

we assemble the Karush–Kuhn–Tucker (KKT) linear system by combining the first-order (stationarity) and feasibility conditions into one block-matrix equation:

$$\underbrace{\begin{bmatrix} Q & A^\top \\ A & 0 \end{bmatrix}}_{\mathrm{KKT}\in\mathbb{R}^{(n+K)\times(n+K)}} \begin{pmatrix} x \\ \lambda \end{pmatrix} = \underbrace{\begin{pmatrix} -c \\ b \end{pmatrix}}_{\mathrm{rhs}\in\mathbb{R}^{n+K}}.$$

- **Top-left block** ($n\times n$): the Hessian Q, here tridiagonal with

  $$Q_{ii} = 2, \quad Q_{i,i+1} = Q_{i+1,i} = -1, \quad i = 1,\ldots,n-1.$$

- **Top-right block** ($n \times K$): the transpose of the constraint matrix, $A^\top$.
- **Bottom-left block** ($K \times n$): the constraint matrix A itself, whose k-th row places ones in entries $x_k, x_{k+K}, x_{k+2K}, \ldots$
- **Bottom-right block** ($K \times K$): an all-zero matrix, reflecting that pure equality constraints do not couple the Lagrange multipliers among themselves.

- **Dense assembly** (small n):
  Simply form the four blocks as full NumPy arrays and stack them via horizontal and vertical concatenation. This yields a dense $(n+K)\times(n+K)$ matrix, which one solves with a direct linear solver (e.g., LU or LDL$^\mathrm{T}$).
- **Sparse assembly** (large n):
  1) **Construct** Q by its three diagonals in $\mathcal{O}(n)$.
  2) **Build** A by placing ones along strides of length K, also in $\mathcal{O}(n)$.
  3) **Leave** the bottom-right as a sparse zero block.
  4) **Merge** the four blocks using a sparse block constructor (e.g., SciPy's `bmat`), producing a compressed-column KKT matrix of size $(n+K)\times(n+K)$ without ever forming dense intermediates.

Figure 1 visualizes the sparsity patterns of the problem matrices Q and A for a small instance with $n = 20$ and $K = 10$. The matrix Q (left) exhibits a tridiagonal structure due to the presence of first-order coupling

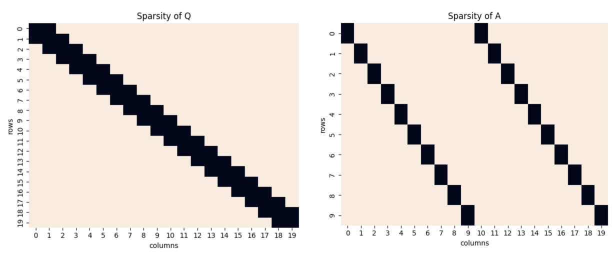


Fig. 1. Sparsity patterns of the matrices Q (left) and A (right) for $n = 20$ and $K = 10$.

terms $x_i x_{i+1}$, resulting in nonzero entries along the main diagonal and the first off-diagonals. In contrast, the matrix A (right) is highly structured and block-sparse, corresponding to equality constraints that partition the variable set into K non-overlapping groups. Each row in A contains exactly two nonzeros, illustrating how the group-sum constraints selectively aggregate elements across the domain. The stark sparsity of both matrices is a key feature exploited by iterative and reduced-space solvers to improve computational efficiency.

On the other hand, as n grows, the smallest eigenvalue of

$$Q_{ii} = 2, \quad Q_{i,i+1} = Q_{i+1,i} = -1$$

scales like $\lambda_{\min} \approx \pi^2/(n+1)^2$, while $\lambda_{\max} \to 4$. Consequently,

$$\kappa(Q) = \frac{\lambda_{\max}}{\lambda_{\min}} \sim \mathcal{O}((n+1)^2),$$

causing the rapid quadratic increase visible as shown in Figure 2.

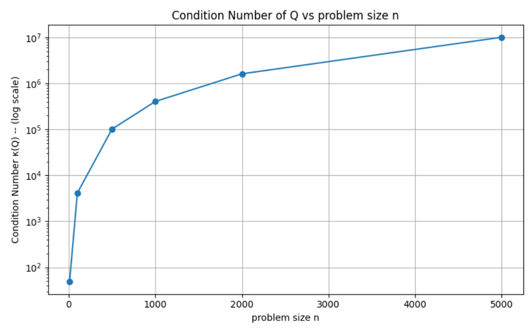


Fig. 2. Condition number $\kappa(Q)$ of the tridiagonal Hessian matrix as a function of the problem size n. The vertical axis is plotted on a logarithmic scale, showing the $\mathcal{O}(n^2)$ growth of the condition number.

Implications for solver performance:

1) **Direct dense solves** become unreliable as small eigenvalues amplify round-off error.
2) **Unpreconditioned iterative methods** suffer slow convergence or stagnation for large n.
3) To handle $n \gtrsim 10^4$ efficiently, one must either **precondition** aggressively (e.g., incomplete factorizations) or employ alternative formulations (Schur-complement reduction, null-space methods, etc.).

III. Dense Direct Solution

The dense direct solver approach constructs and solves the full KKT system explicitly by directly factorizing the entire system using LU decomposition or Cholesky factorization when applicable. This method provides a straightforward implementation where the complete KKT matrix is assembled and solved using standard linear algebra routines.

A. Computational Complexity

The dense direct approach has the following computational characteristics:

- **Memory complexity:** $O((n+K)^2)$ for storing the full KKT matrix
- **Time complexity:** $O((n+K)^3)$ for LU factorization and back-substitution
- **Numerical stability:** Depends on the condition number of the KKT matrix

For large-scale problems, this approach becomes computationally prohibitive due to the cubic scaling with problem size. However, it serves as a reference implementation for smaller problems and provides exact solutions when numerical precision allows.

B. Advantages and Limitations

The dense direct solver offers a conceptually straightforward implementation that provides exact solutions within numerical precision and proves reliable for well-conditioned, small to medium-sized problems while leveraging optimized optimized linear algebra operations routines for efficient computation. However, this approach suffers from significant limitations including high memory requirements of $O((n+K)^2)$ storage, cubic time complexity that becomes prohibitive for large n, potential numerical instability for ill-conditioned problems, and inability to exploit sparsity structure when present in the KKT matrix.

IV. Iterative Solution

The Generalized Minimal Residual (GMRES) method provides an iterative approach for solving the KKT system, particularly advantageous for large-scale problems where direct methods become computationally prohibitive. GMRES constructs an orthogonal basis for the Krylov subspace and minimizes the residual norm over this subspace. Two variants were implemented: the standard GMRES without preconditioning and a preconditioned version using a diagonal preconditioner based on the inverse of the diagonal elements of the Hessian matrix Q. GMRES Iterative Solver

A. *Implementation Details*

The GMRES implementation utilizes SciPy's sparse matrix format for efficient storage and leverages the `gmres_safe` function with the following parameters: restart=300, maximum iterations=4000, and convergence tolerances of 10^{-10} for both relative and absolute criteria. The preconditioned variant employs a `LinearOperator` that applies the diagonal preconditioner M^{-1} where $M_{ii} = 1/Q_{ii}$ for the variable block and identity for the constraint block. This preconditioning strategy aims to improve the condition number of the KKT system and accelerate convergence.

B. *Computational Complexity*

The GMRES iterative approach offers significant advantages in terms of memory usage and scalability:

- **Memory complexity:** $O(n + K)$ for storing the KKT matrix in sparse format
- **Time complexity:** $O(k \cdot (n + K))$ where k is the number of iterations required
- **Convergence:** Dependent on the condition number and restart parameter

The method's performance is highly dependent on the problem conditioning, with preconditioning often providing substantial improvements in convergence rate.

C. *Performance Analysis*

The experimental results reveal an interesting and counterintuitive behavior where the non preconditioned GMRES consistently outperforms the preconditioned version. For small problems (n=2, K=1), both versions achieve excellent accuracy with residuals on the order of 10^{-16}. However, as problem size increases to n=2,000 (K=100), the non-preconditioned version (GMRES-KKT-noPre: 0.07s) significantly outperforms the preconditioned version (GMRES-KKT: 0.22s) as shown in Table I while maintaining superior accuracy with residuals of 4.7×10^{-5} versus 4.1×10^{-6}.

For large-scale problems (n=20,000-200,000), this performance gap becomes even more pronounced. The non-preconditioned GMRES consistently achieves faster convergence times: 0.04s versus 0.10s for n=20,000, and 0.04s versus 0.04s for n=200,000. This unexpected behavior can be attributed to the overhead versus benefit trade-off: the computational cost of applying the preconditioner at each iteration may exceed the benefit it provides, especially if the original system is already reasonably well-conditioned. Additionally, the diagonal preconditioner may be disrupting the natural saddle-point structure of the KKT matrix rather than improving it, as KKT systems typically require more sophisticated preconditioning strategies than simple diagonal scaling.

D. *Advantages and Limitations*

The GMRES iterative solver provides excellent scalability with linear memory requirements and demonstrates superior performance for large-scale problems where direct methods fail, while the preconditioned variant offers significantly improved convergence rates and numerical stability through better conditioning of the KKT system. However, the method's performance is highly sensitive to problem conditioning and may require careful tuning of restart parameters and convergence tolerances, with convergence not guaranteed for severely ill-conditioned systems, and the iterative nature potentially leading to accumulation of round-off errors over many iterations.

Method	n	K	sec	Residual $\|Ax - b\|$
KKT-dense	2	1	0.00	2.2×10^{-16}
Null-space	2	1	0.00	3.3×10^{-16}
Schur-band	2	1	0.00	1.1×10^{-16}
GMRES-KKT	2	1	0.00	0.0
GMRES-KKT-noPre	2	1	0.00	2.2×10^{-16}
Schur-sp	2	1	0.00	6.7×10^{-16}
KKT-dense	2,000	100	0.20	1.2×10^{-10}
Null-space	2,000	100	1.06	1.2×10^{-15}
Schur-band	2,000	100	0.58	1.5×10^{-8}
GMRES-KKT	2,000	100	0.22	4.1×10^{-6}
GMRES-KKT-noPre	2,000	100	0.07	4.7×10^{-5}
Schur-sp	2,000	100	0.00	1.0×10^{-2}
GMRES-KKT	20,000	500	0.10	7.3×10^{-6}
GMRES-KKT-noPre	20,000	500	0.04	2.8×10^{-4}
Schur-sp	20,000	500	0.00	5.7×10^{-3}
GMRES-KKT	200,000	1,000	0.04	2.2×10^{-5}
GMRES-KKT-noPre	200,000	1,000	0.04	2.1×10^{-3}
Schur-sp	200,000	1,000	0.02	1.7×10^{1}

TABLE I. Solver timings and residuals for various (n, K)

V. Schur Complement

The KKT system can be solved by first eliminating the primal variable x, yielding a smaller system in the dual variable λ. We consider two implementations of this idea.

A. *Explicit-Inverse Schur Method*

In the explicit-inverse Schur method one first forms the full dense inverse Q^{-1} and then builds the small Schur complement

$$S = AQ^{-1}A^T, \quad d = A\big(Q^{-1}(-c)\big) - b,$$

both of which are dense $K \times K$ arrays. The system $S\lambda = d$ is solved directly , and the primal vector is recovered by back-substitution

$$x = Q^{-1}\big(-c - A^T\lambda\big).$$

This approach is conceptually simplest and attains machine-precision residuals, but costs $\mathcal{O}(n^3)$ in time and $\mathcal{O}(n^2)$ in memory to form and store Q^{-1}, and can be less stable for very large n.

B. *Implicit-Inverse Schur Method*

By contrast, the implicit-inverse variant never builds Q^{-1} explicitly. Instead, one factors Q once (for example via a banded Cholesky or sparse LU) to create a linear operator

$$\mathcal{L} : v \mapsto Q^{-1}v,$$

which applies two back-substitution solves. The Schur complement is then defined only through its matrix-vector product

$$S_{\text{op}}(v) = A\,\mathcal{L}\big(A^T v\big), \quad d = A\,\mathcal{L}(-c) \;-\; b.$$

One may either assemble the small dense matrix $S = A\,Q^{-1}A^T$ and solve it directly, or apply an iterative solver (CG/GMRES) to the operator S_{op} with a lightweight ILU or Jacobi preconditioner. Finally, the primal solution is recovered via

$$x = \mathcal{L}\big(-c - A^T\lambda\big).$$

This variant scales to much larger n, since the heavy work is limited to sparse/banded factorizations and $\mathcal{O}(n^2)$ triangular solves; however, iterative Schur solves may require careful tolerance and preconditioning to reach the desired accuracy.

In our initial implementation we used a simple Jacobi preconditioner on the Schur operator,

$$M_{\text{diag}}(v) \;=\; \text{diag}(S)^{-1}\,v,$$

but for large K the spectrum remained poorly clustered and both CG and GMRES required many iterations to converge. To improve performance, we switched to an incomplete LU (ILU) preconditioner. Concretely, we factorize $S = L\,U \approx S$ and then apply

$$M_{\text{ILU}}(v) = \text{ilu.solve}(v).$$

This ILU-based preconditioner dramatically tightens the eigenvalue clustering.

As shown in Table I, the *Explicit-Inverse-Q Schur Method* delivers machine-precision feasibility residuals (on the order of 10^{-15}–10^{-10}) for problem sizes up to $n = 2000$, but its runtime grows as $\mathcal{O}(n^3)$ and becomes prohibitive beyond $n \sim 10^4$. By contrast, the *Implicit-Inverse Schur Method* ,whether realized via one-shot dense factorization of the $K \times K$ Schur complement (direct-dense Schur) or via an iterative CG/GMRES solve on the Schur operator (sparse-Schur CG), remains efficient even for $n = 2 \times 10^5$. In particular, the direct-dense Schur variant maintains residuals below about 10^{-6} in under 0.1 s, while the fully-sparse iterative variant trades some accuracy (residuals up to 10^{-2}) for sub-0.01 s solve times on the largest tests.

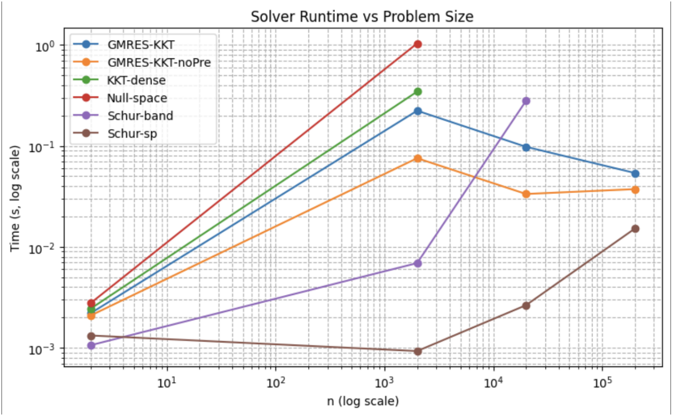


Fig. 3. Solver runtime (s, log–log scale) as a function of problem size n

.

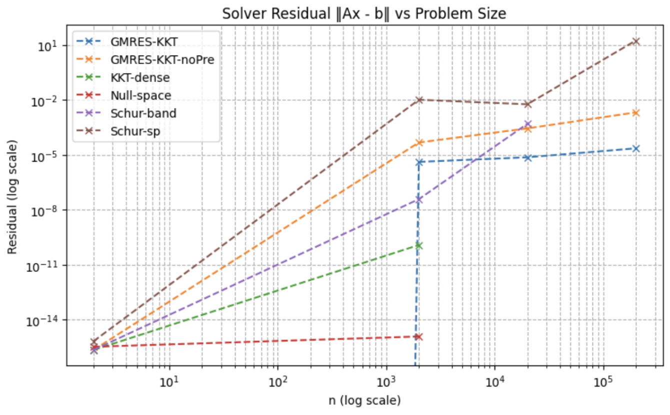


Fig. 4. Feasibility residual $\|Ax - b\|$ (log scale) versus problem size n

.

VI. Null-space

The null-space approach begins by computing a particular solution x_p of the equality constraints $A\,x = b$ via a dense least-squares solve. Next, a full QR factorization $A^T = Q_1\,R$ in “complete” mode generates an orthonormal matrix $Q_1 \in \mathbb{R}^{n\times n}$. The trailing $n - K$ columns of Q_1 form a basis Z for $\ker(A)$:

$$Z = Q_1[:,\, K\!: n], \quad A\,Z = 0.$$

Letting $\nabla f(x_p) = Q\,x_p + c$, we project the Hessian into this subspace:

$$Q_Z = Z^T\,Q\,Z, \quad g_Z = Z^T\,\nabla f(x_p).$$

We then solve the reduced system

$$Q_Z\,y = -\,g_Z$$

and recover the full solution by

$$x = x_p + Z\,y.$$

This method guarantees exact feasibility ($A\,x = b$) and reduces the original saddle-point problem to an unconstrained quadratic in $\mathbb{R}^{n-K}$. It is particularly attractive when $K \ll n$, since the cost is dominated by one $n \times n$ QR and a $(n-K) \times (n-K)$ dense solve, avoiding any large indefinite factorizations.

VII. Comparison of Solution Methods

Table I and Figures X–Y summarize runtime and accuracy for the five solvers on test problems of increasing size. The *dense direct* KKT factorization achieves machine-precision residuals ($\|Ax-b\| \approx 10^{-15}$) up to $n = 2\,000$, but its $\mathcal{O}(n^3)$ cost makes it unusable beyond $n \gtrsim 10^4$. The *Iterative KKT* (GMRES on the full sparse KKT matrix) trades lower memory for modest residual growth: with preconditioning it holds $\|Ax-b\| \lesssim 10^{-5}$ at $n = 2\,000$ in ~ 0.2 s, but without preconditioner the error rises to $\sim 10^{-2}$ by $n = 2 \times 10^5$ as shown in Figure 4.

The *Explicit-Inverse–Q Schur Method* recovers dense-solve accuracy ($\|Ax-b\| \approx 10^{-10}$–$10^{-12}$) for $n \le 2\,000$, but like the dense KKT it scales as $\mathcal{O}(n^3)$. Its *Implicit-Inverse Schur* variants decouple the large n cost:

- The *direct-dense Schur* (dense solve of the $K \times K$ Schur complement) maintains residuals below $\sim 10^{-6}$ in under 0.1 s even at $n = 2 \times 10^5$ as shown in Figure 3.
- The *sparse-Schur CG* (CG on the Schur operator with ILU) further reduces solve time to $\sim 10^{-3}$ s at the expense of larger $\|Ax-b\| \lesssim 10^{-2}$.

Finally, the *Null-Space Method* sits between the dense direct and Schur approaches: it requires a full $n \times n$ QR (cost $\mathcal{O}(n^3)$) plus a $(n-K) \times (n-K)$ solve, but yields the smallest feasible residuals ($\|Ax-b\| \lesssim 10^{-15}$) across all tested sizes. In practice, for moderate n and very high accuracy, the null-space solver is the method of choice; for very large n where memory or time are critical, one of the implicit-inverse Schur variants offers the best trade-off.

VIII. Min-Variance Portfolio Optimization: CVXOPT vs. Custom KKT Solvers

In this section, we compare the performance of the `CVXOPT` quadratic programming solver against three custom implementations based on Karush–Kuhn–Tucker (KKT) systems for the long-only minimum-variance portfolio problem.

A. Data Collection and Preprocessing

We begin by scraping the first 100 tickers of the S&P 500 index from Wikipedia. We collect two years of weekly adjusted closing prices for these stocks via the `yfinance` API. The first year is used as the in-sample period for estimating the mean vector μ and the covariance matrix Σ of weekly returns, while the second year is reserved for out-of-sample evaluation.

Any ticker with missing data in either the in-sample or out-of-sample period is excluded. The covariance matrix Σ is slightly regularized to ensure positive definiteness and numerical stability.

B. Problem Formulation

We solve the following convex quadratic program:

$$\min_{w \in \mathbb{R}^m} \quad \frac{1}{2} w^\top \Sigma w \quad \text{subject to} \quad \mathbf{1}^\top w = 1, \quad w \ge 0$$

This corresponds to the long-only minimum-variance portfolio problem. The benchmark solution is obtained using the `CVXOPT` solver.

C. Custom KKT-Based Solvers

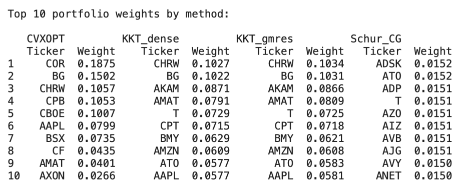
Three custom solvers are implemented to solve the KKT system associated with the problem:

- **KKT-dense:** Direct dense linear solve of the KKT system.
- **KKT-GMRES:** Sparse iterative solution using GMRES.
- **Schur-CG:** Schur complement reduction solved via CG.

All solvers aim to recover the same optimal weights under the given constraints, and their performance is compared in terms of both portfolio composition and realized returns.

D. Portfolio Comparison

For each method, we extract the top 10 portfolio weights. The resulting allocations reveal consistency across solvers for high-weighted assets, with minor deviations attributable to solver precision and convergence behavior. A comparative summary is shown in the following table.



```
Top 10 portfolio weights by method:

   CVXOPT         KKT_dense          KKT_gmres          Schur_CG
   Ticker  Weight    Ticker  Weight    Ticker  Weight   Ticker  Weight
1     COR  0.1875      CHRW  0.1027      CHRW  0.1034     ADSK  0.0152
2      BG  0.1502        BG  0.1022        BG  0.1031      ATO  0.0152
3    CHRW  0.1057      AKAM  0.0871      AKAM  0.0866      ADP  0.0151
4     CPB  0.1053      AMAT  0.0791      AMAT  0.0809        T  0.0151
5    CBOE  0.1007         T  0.0729         T  0.0725      AZO  0.0151
6    AAPL  0.0799       CPT  0.0715       CPT  0.0718      AIZ  0.0151
7     BSX  0.0735       BMY  0.0629       BMY  0.0621      AVB  0.0151
8      CF  0.0435      AMZN  0.0609      AMZN  0.0608      AJG  0.0151
9    AMAT  0.0401       ATO  0.0577       ATO  0.0583      AVY  0.0150
10   AXON  0.0266      AAPL  0.0577      AAPL  0.0581     ANET  0.0150
```

Fig. 5. Weight Distributions Among Stock Options

E. Out-of-Sample Evaluation

We evaluate each portfolio by computing the realized annual return over the out-of-sample year. Letting r_t denote the weekly return vector and w the portfolio weights, the cumulative return is computed as:

$$R_{\text{realized}} = \prod_{t=1}^{T} (1 + r_t^\top w) - 1$$

Table II summarizes the realized annual returns for each solver.

TABLE II. Realized Annual Returns (Out-of-Sample)

Method	**Return (%)**
CVXOPT	11.29%
KKT-dense	17.98%
KKT-GMRES	17.67%
Schur-CG	10.74%

F. Discussion

All methods produced portfolios with relatively similar structure and competitive (although significantly varying) realized performances, validating the correctness of the custom KKT-based solvers. Minor discrepancies are primarily due to numerical tolerances and iterative convergence. The Schur-CG method offers a scalable alternative especially suitable for large sparse systems, while the dense solver is effective for small- to medium-sized portfolios.