Adapted Interior-Point Methods for Sparse Model Predictive Control

Final Project

Luke Nuculaj
APM 5334: Applied Numerical Methods - Matrix Methods
Instructor: Dr. Giselle Saylor
April 25, 2025

Oakland UNIVERSITY

Contents

1 Introduction

1.1 Abstract

In the domain of modern control techniques, model predictive control (MPC) is among the most widely adopted control techniques for its anticipatory nature and its ability to account for system constraints [1, 2]. However, the computational overhead imposed by MPC prevents its easy integration in most high-speed control applications running on low-speed edge devices [3]. Interestingly, many MPC problems are almost exclusively sparse [4]. Hence, the following work investigates various techniques which exploit the sparsity of linear systems that arise from the interior-point method as applied to quadratic programs for model predictive control. All corresponding MATLAB code is contained in the Appendix for reference.

1.2 Outline

Section 2 introduces some background of model predictive control and how the output reference tracking problem can be developed. The primal-dual system is then derived from the KKT conditions, from which the basic interior point algorithm that determines the optimal control expenditure for the reference tracking problem. The sparsity of the matrices that emerge in this algorithm is discussed, and the problem is formally stated. Section 3 investigates how Gaussian elimination and backward substitution can be applied both naively and intelligently to the primal-dual system. A short discussion on the omission of LU factorization and the impracticality of the COO format for Gaussian elimination is included here. Section 4 investigates the eigenvalue distributions and properties of the linear system to confirm that Gauss-Seidel and successive over-relaxation are appropriate iterative techniques to apply to the problem. Their implementations are explained, and some quantitative results regarding the "optimal" selection of relaxation parameter ω are discussed. Furthermore, symmetrization and the imposition of positive definiteness to the primal-dual system via a doubly augmented formulation are explored, and the conjugate gradient method is applied to the modified system. Section 5 discusses matrix inversion via the Schur complement, a remarkably effective technique that reduces the problem of inverting a large matrix to that of inverting a smaller matrix. To invert the smaller matrix, LU factorization and quasi-Newton methods are explored and compared with one another. All results are discussed in Section 6, comparing the techniques as applied to the primal-dual and the symmetric positive definite (SPD) systems on the basis of execution time, memory requirements, and scalability. Finally, conclusions and future work are discussed in Section 7.

1.3 Nomenclature

In mathematical contexts, matrices are denoted by capitalized letters of the Latin alphabet (e.g. A, B), vectors are denoted by lowercase Latin letters (e.g. x, y), and scalars are denoted by lowercase Greek letters (e.g. α, β, etc.).

2 Problem Formulation

2.1 Model Predictive Control

The problem centers around the finite-time optimal control problem (OCP) formulation for MPC

$$
\begin{aligned}
\min_{x,u} \quad & \ell_N(x_N) + \sum_{k=0}^{N-1} \ell_k(x_k, u_k) && \text{(1a)}\\
\text{s.t.} \quad & x_0 = p && \text{(1b)}\\
& x_{k+1} = A_k x_k + B_k u_k + f_k && \text{(1c)}\\
& F_k x_k + G_k u_k \leq c_k && \text{(1d)}\\
& \quad k = 0, \dots, N-1 && \text{(1e)}\\
& F_N x_N \leq c_N && \text{(1f)}
\end{aligned}
$$

where N is the prediction horizon, $p \in \mathbb{R}^{n_x}$ is the current state vector, $u_k \in \mathbb{R}^{n_u}$ is the vector of control variables at prediction time k for n_x. The function $\ell_k : \mathbb{R}^{n_x \times n_u} \to \mathbb{R}$ is the stage cost, and $\ell_k : \mathbb{R}^{n_x} \to \mathbb{R}$ is the terminal cost – as we will see shortly, we will opt for quadratic functions here because of their favorable convex properties. (1c) describes the discrete-time dynamics of the system. (1d) and (1f) account for any variety of constraints – whether mandatory or preferential – that we wish to impose on the system. In Layman's terms, given our current state p, (1) provides an optimal control sequence that minimizes the objective while abiding by all of the constraints.

2.2 Reference Tracking

For the sake of a functioning example, let's consider an output reference tracking example. We define the output $y_k := Cx_k$, costs $\ell_N(y_N) := \frac{1}{2}(y_N - y_{ref,N})^\top Q_N(y_N - y_{ref,N})$ and $\ell_k(y_k, u_k) := \frac{1}{2}(y_k - y_{ref,k})^\top Q_k(y_k - y_{ref,k}) + \frac{1}{2}u_k^\top R_k u_k$. Say we also impose box constraints on the states x_k and controls u_k, then we get the following OCP

$$
\begin{aligned}
\min_{x,y,u} \quad & \frac{1}{2}(y_N - y_{\text{ref},N})^\top Q_N(y_N - y_{\text{ref},N}) \\
& + \frac{1}{2}\sum_{k=0}^{N-1} \left[(y_k - y_{\text{ref},k})^\top Q_k(y_k - y_{\text{ref},k}) + u_k^\top R_k u_k \right] && \text{(2a)}\\
\text{s.t.} \quad & x_0 = p && \text{(2b)}\\
& x_{k+1} = A_k x_k + B_k u_k + f_k && \text{(2c)}\\
& y_k = Cx_k && \text{(2d)}
\end{aligned}
$$

$$x_{\min,k} \le x_k \le x_{\max,k} \tag{2e}$$

$$u_{\min,k} \le u_k \le u_{\max,k} \tag{2f}$$

$$k = 0, \dots, N-1 \tag{2g}$$

$$y_N = Cx_N \tag{2h}$$

$$x_{\min,N} \le x_N \le x_{\max,N} \tag{2i}$$

With some manipulations, we can reframe (2) as the following:

$$\min_{\mathbf{u}} \quad \frac{1}{2}\mathbf{u}^\top H\mathbf{u} + f^\top \mathbf{u} \tag{3a}$$

$$\text{s.t.} \quad A_I\mathbf{u} \le b_I \tag{3b}$$

where $u := \begin{bmatrix} u_0^\top & u_1^\top & \dots & u_{N-1}^\top \end{bmatrix}^\top$, $H \in \mathbb{S}_{++}^{n_u N} := M_{CAB}^\top M_Q M_{CAB} + M_R$ is the Hessian matrix, $f \in \mathbb{R}^{n_u N} := (p^\top M_{CA}^\top - y_{ref}^\top) M_Q M_{CAB}$, $A_I \in \mathbb{R}^{m \times n_u N} := \begin{bmatrix} M_{AB}^\top & -M_{AB}^\top & I_{n_u N} & -I_{n_u N} \end{bmatrix}^\top$, $b_I \in \mathbb{R}^m := \begin{bmatrix} (x_{max} - M_{Ak}p)^\top & (-x_{min} + M_{Ak}p)^\top & u_{max}^\top & u_{min}^\top \end{bmatrix}^\top$, and

$$M_{AB} \in \mathbb{R}^{n_x N \times n_u N} := \begin{bmatrix} B & 0 & 0 & \cdots & 0 \\ AB & B & 0 & \cdots & 0 \\ A^2B & AB & B & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A^{N-1}B & A^{N-2}B & A^{N-3} & \cdots & B \end{bmatrix} \tag{4a}$$

$$M_{CAB} \in \mathbb{R}^{N \times n_u N} := \begin{bmatrix} CB & 0 & 0 & \cdots & 0 \\ CAB & CB & 0 & \cdots & 0 \\ CA^2B & CAB & CB & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ CA^{N-1}B & CA^{N-2}B & CA^{N-3} & \cdots & CB \end{bmatrix} \tag{4b}$$

$$M_{Ak} \in \mathbb{R}^{n_x N \times n_x} := \begin{bmatrix} A \\ A^2 \\ \vdots \\ A^{N-1} \\ A^N \end{bmatrix}, \quad M_{CAk} \in \mathbb{R}^{N \times n_x} := \begin{bmatrix} CA \\ CA^2 \\ \vdots \\ CA^{N-1} \\ CA^N \end{bmatrix}, \tag{4c}$$

and for our weighting matrices, we have

$$M_Q \in \mathbb{R}^{n_x N \times n_x N} := \begin{bmatrix} Q_1 & 0 & \cdots & 0 \\ 0 & Q_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & Q_N \end{bmatrix}, \quad M_R \in \mathbb{R}^{n_u N \times n_u N} := \begin{bmatrix} R_0 & 0 & \cdots & 0 \\ 0 & R_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & R_{N-1} \end{bmatrix}. \tag{5}$$

Our goal is to solve the QP (3), which will provide the optimal control sequence for the output reference tracking problem.

2.3 Primal-Dual System

Among the various methods available to solve the QP (3), we select the interior point method, which considers a relaxed form of the OCP

$$\min_{u,s} \quad \frac{1}{2}u^\top H u + f^\top u - \mu \sum_{i=1}^{n_s} \log s_i \tag{6a}$$

$$\text{s.t.} \quad A_I u - b_I + s = 0 \tag{6b}$$

$$s \geq 0 \tag{6c}$$

where $s \in \mathbb{R}^{n_s}$ and $\mu > 0$ is a scaling parameter. As we reduce the value of μ, the barrier functions reward slacks that are closer to 0 (by adding a value $\to -\infty$ to the objective as $s \to 0$), incrementally pushing us towards the optimal solution. For (6), optimality is achieved when the Karush-Kuhn-Tucker (KKT) conditions are satisfied

$$Hu + f - A_I^\top z = 0 \tag{7a}$$

$$A_I u - b_I + s = 0 \tag{7b}$$

$$Sz - \mu e = 0, \tag{7c}$$

where $z \in \mathbb{R}^m$ is the vector of Lagrange multipliers, $e \in \mathbb{R}^{n_s} := \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}^\top$, and $S \in \mathbb{R}^{n_s \times n_s} := \operatorname{diag}(s)$ [5]. We recognize that (7) is a nonlinear system of equations. A basic interior-point algorithm applies Newton's method to (7) to get a search direction p

$$\begin{bmatrix} H & 0 & -A_I^\top \\ A_I & I & 0 \\ 0 & Z & S \end{bmatrix} \begin{bmatrix} p_x \\ p_z \\ p_s \end{bmatrix} = \begin{bmatrix} Hu + f - A_I^\top z \\ A_I u - b_I + s \\ Sz - \mu e \end{bmatrix}, \tag{8}$$

where $Z \in \mathbb{R}^{m \times m} := \operatorname{diag}(z)$. With each successive solution to (8), μ is decreased until a specific convergence criterion is met.

2.4 Basic Interior Point Algorithm

For clarity, we outline these steps in Algorithm 1, where

$$E(x, s, z; \mu) := \max\left\{ \left\|Hu + f - A_I^\top z\right\|,\ \left\|A_I u - b_I + s\right\|,\ \left\|Sz - \mu e\right\| \right\}$$

The objective of this project is centered around solving the linear system (8) as efficiently as possible. The next section discusses the sparsity of this linear system and how this fact grants us an opportunity to heavily optimize our interior-point algorithm – an itemized list of project objectives will be provided thereafter.

Algorithm 1 Basic Interior-Point Algorithm

```
1: Input: $x^{(0)}, s^{(0)} > 0$
2: Output: $x^*, s^*, z^*$
3:
4: Set $\mu^{(0)} > 0, \sigma \in (0,1), k \leftarrow 0$
5: while stopping criteria is not met do
6:     while $E(x^{(k)}, s^{(k)}, z^{(k)}; \mu^{(k)}) > \mu^{(k)}$ do
7:         $p \leftarrow$ Solve the linear system (8).
8:         $\alpha \leftarrow$ Determined by fraction-to-boundary rule.
9:         $x^{(k+1)} \leftarrow x^{(k)} + \alpha \cdot p_x$
10:        $s^{(k+1)} \leftarrow s^{(k)} + \alpha \cdot p_s$
11:        $z^{(k+1)} \leftarrow z^{(k)} + \alpha \cdot p_z$
12:        $k \leftarrow k + 1$
13:    end while
14:    $\mu^{(k+1)} \leftarrow \sigma \cdot \mu^{(k)}$
15: end while
```

2.5 Sparsity of Matrices

To assess how much room there is for us to improve, identifying the level of sparsity present in $G \in \mathbb{R}^{(n_uN+2m)\times(n_uN+2m)} := \nabla F(x,s,z)$, where $F(x,s,z) = 0$ is the nonlinear system described in (7). We can identify the guaranteed zeros in our system, which lie in the zero blocks as well as the off-diagonal elements of I, S, and Z. From this, we have $n_uN \cdot m + n_uN \cdot m + m \cdot m + (m^2 - m) + (m^2 - m) + (m^2 - m) = 4m^2 + (2n_uN - 3)m$ zeros. G contains $4m^2 + (4n_uN)m + n_u^2N^2$ total elements. Without any prior knowledge of the system dynamics or constraints, we can guarantee $100 \cdot \frac{(2n_uN+3)m+n_u^2N^2}{4m^2+(4n_uN)m+n_u^2N^2}\%$ sparsity of G. However, a quick look at the constituent matrices that comprise H and A_I in our output reference tracking example reveals that the level of sparsity is indeed much higher. Furthermore, let's consider the simple linear time-invariant case of a vehicle driving in a straight line, accounting for aerodynamic drag forces. We define $x_k := \begin{bmatrix} d_k & v_k \end{bmatrix}^\top$ and $u_k := a_k$, where d_k is the position, v_k is the velocity, and a_k is the acceleration of the vehicle at time step k. We construct the discrete-time system $x_{k+1} = Ax_k + Bu_k$, where

$$A = \begin{bmatrix} 1 & T_s \\ 0 & 1 - \frac{c}{m}T_s \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ \frac{T_s}{m} \end{bmatrix},$$

T_s is the sampling time in seconds, m is the mass of the vehicle in kilograms, and c is a drag-coefficient. We are limited to $n_x = 2$ and $n_u = 1$, but we have discretion over the value we set our prediction horizon N to. N too small results in a myopic, short-sighted control scheme, and N too large gives diminishing returns at the expense of poorly-scaling computational complexity.

The sparsity of G is visualized in Fig. 1, where it is evident that for any value of N, the linear system (8) is almost completely sparse. Furthermore, we can also see that G has a predictable structure, which opens up the possibility for efficient techniques to accelerate the computation of a search direction in the interior point method.

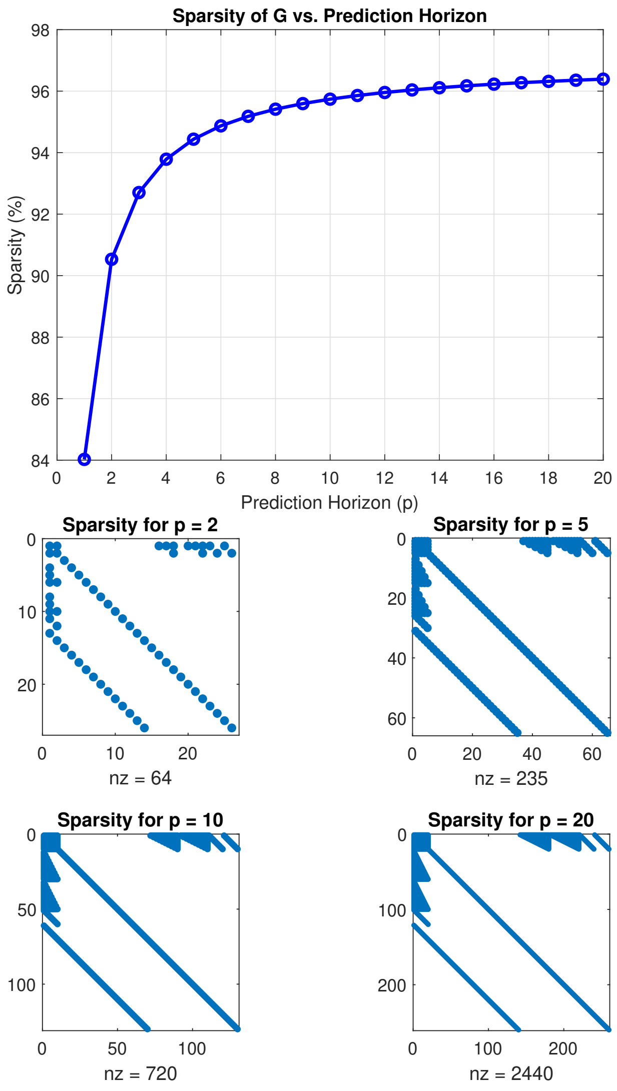

Figure 1: Sparsity of the output reference tracking problem for various prediction horizons N.

3 Direct Methods

In this section, we discuss Gaussian elimination and backward substitution algorithms as direct methods applied to solving (8). We intentionally omit factorization techniques (e.g., LU) from this section as Algorithm 1 indicates that Z and S change – and consequently, the linear system transforms – between iterations. Thus, factorization techniques offer little computational advantage over repeated Gaussian elimination in this case. For proper background, we refer the reader to Section 8 for full listings of the standard Gaussian elimination and backward substitution algorithms.

3.1 Gaussian Elimination

By observing that the linear system at hand is $\in \mathbb{R}^{(2m+N)\times(2m+N)}$, we can estimate that a naive Gaussian elimination algorithm will require $\approx \frac{2}{3}(2m+N)^3 = \mathbf{\frac{16}{3}m^3 + 8m^2N + 4mN^2 + \frac{2}{3}N^3}$ flops, which scales poorly even for a "small" prediction horizon and constraint number.

Recall the sparse structure of our linear system, which can be re-imagined with tiling as shown in Fig. 2. This interpretation of the linear system's block structure will come in useful

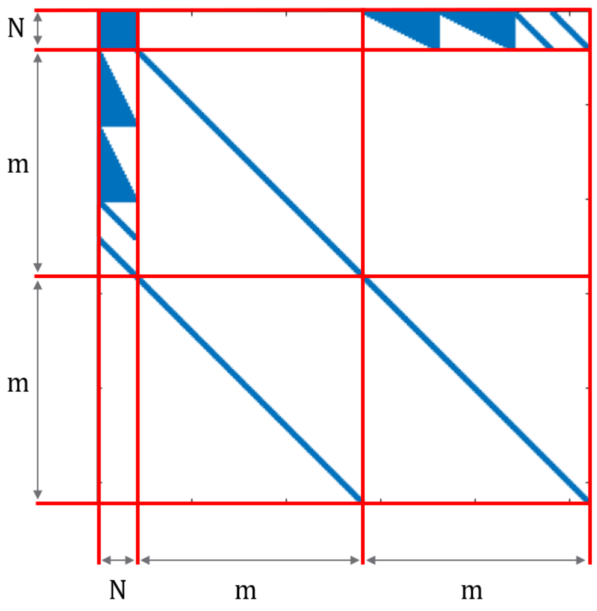

Figure 2: The tiled approach for the KKT linear system.

not only for Gaussian elimination, but the rest of the methods discussed in this work. Specifically, Gaussian elimination for this matrix can be divided up into phases wherein the block sparsity is exploited. Consider the illustrations in Fig. 3, which shows that the standard Gaussian elimination can be broken down into block-wise phases.

From here on, we will treat the block matrix as a grid and B_{ij} will denote the block matrix in the ith row and jth column of the grid. We observe that the first phase eliminates the lower

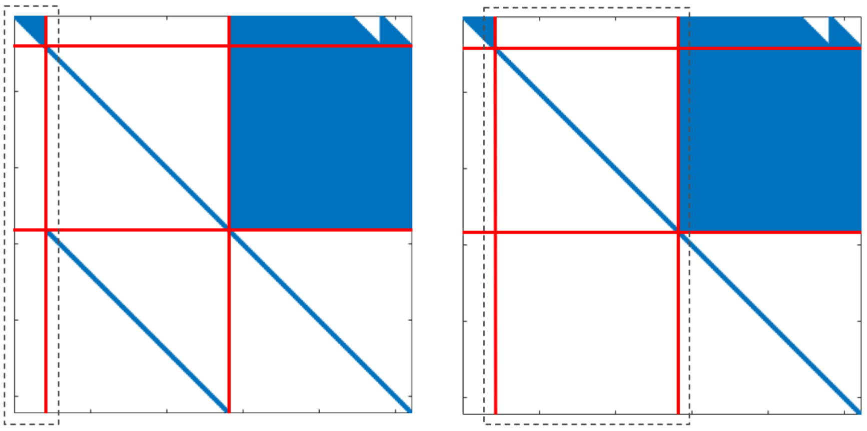

Figure 3: Block-wise Gaussian elimination – Phases 1 (**left**) and 2 (**right**).

half of B_{11}, all of B_{21}, and leaves B_{31} unaffected since it is entirely zeros. Next, the second phase eliminates B_{32}, which is a diagonal matrix by default. Note that only these two phases are required for Gaussian elimination since B_{33} is already a diagonal matrix that coincides with the principal diagonal of the main matrix. Clearly, we can exploit the sparsity of the system to speed up the standard Gaussian elimination algorithm – this optimized algorithm is listed in Algorithm 2.

Algorithm 2 Optimized Gaussian Elimination Algorithm

```
1:  Input: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $TOL \in \mathbb{R}$
2:  Output: $U \in \mathbb{R}^{n \times n}$, $c \in \mathbb{R}^n$
3:
4:  $U \leftarrow A$, $c \leftarrow b$
5:  for $j = 1 : N + m$ do
6:    if $j \leq N$ then
7:      for $i = j + 1 : N + m$ do
8:        $m \leftarrow \frac{u_{ij}}{u_{jj}}$
9:        for $k = j : n$ do
10:         $u_{ik} \leftarrow u_{ik} - m \cdot u_{jk}$
11:       end for
12:       $c_i \leftarrow c_i - m \cdot c_j$
13:     end for
14:   else if $j > N$ then
15:     $m \leftarrow u_{j+m,j}$
16:     $u_{j+m,j} = 0$
17:     $c_{j+m} = c_{j+m} - m$
18:   end if
19: end for
```

Lines 7 through 13 are responsible for Phase 1, which performs Gaussian elimination on only the rows that make up B_{11} and B_{21}. Note that Lines 9 and 11 can be further optimized by using index k to skip over elements in columns $N+1$ through $N+m$. Lines 14 through 18 carry out Phase 2, which only sees one subtraction per iteration. Note that we need not multiply m by anything since B_{22} is an identity matrix, so multiplication by 1 in this case would be redundant. Using flop-counting methods we learned in class, Phase 1 requires $4m^2N+3mN^2+\frac{2}{3}N^3+2mN-\frac{1}{2}N^2-\frac{7}{6}N+2m$ flops, and Phase 2 requires only m flops. Thus, the total flops required for Algorithm 2 is $\mathbf{4m^2N + 3mN^2 + \frac{2}{3}N^3 + 2mN - \frac{1}{2}N^2 - \frac{13}{6}N + m}$ flops. We note the removal of the m^3 term, which is important considering that in most practical scenarios, m is several times larger than N. For perspective, if we consider the case where $N = 5$ and $m = 30$, the naive Gaussian elimination requires $\approx 183,083$ flops, whereas the optimized version only takes $\approx 20,641$ flops.

3.2 Backward Substitution

As for backward substitution, the naive approach takes $\mathbf{4m^2 + 4mN + N^2 + 2m + N - 1}$ flops; we will use this as a reference point to gauge how much we can improve. Recall that the

Algorithm 3 Optimized Backward Substitution Algorithm

```
1: Input: $U \in \mathbb{R}^{n\times n}$, $c \in \mathbb{R}^n$
2: Output: $x \in \mathbb{R}^n$
3:
4: % Phase 1
5: for $i = N + 2m : -1 : N + m + 1$ do
6:     $x_i \leftarrow \frac{c_i}{u_{ii}}$
7: end for
8:
9: % Phase 2
10: for $i = N + m : -1 : N + 1$ do
11:     $s \leftarrow 0$
12:     for $j = N + m + 1 : N + 2m$ do
13:         $s \leftarrow s + u_{ij} \cdot x_j$
14:     end for
15:     $x_i \leftarrow c_i - s$
16: end for
17:
18: % Phase 3
19: for $i = N : -1 : 1$ do
20:     $s \leftarrow 0$
21:     for $j = i + 1 : N$ do
22:         $s \leftarrow s + u_{ij} \cdot x_j$
23:     end for
24:     for $j = N + m + 1 : N + 2m$ do
25:         $s \leftarrow s + u_{ij} \cdot x_j$
26:     end for
27:     $x_i \leftarrow \frac{c_i - s}{u_{ii}}$
28: end for
```

U matrix (the right subfigure in Fig. 3) is the matrix with which we perform the backward substitution, and clearly we can exploit the sparse structure here as well. For example, B_{33} is a diagonal matrix, so the solution vector along these rows is computed by a simple division for each element (in a similar way to GE, let's call this "Phase 1"). We also see that B_{22} is a diagonal matrix, but must be careful since it lies adjacent to a dense matrix B_{23} (Phase 2). Lastly, B_{12} is empty, so we only need to accumulate terms along grid columns 1 and 3 in the final phase (Phase 3). For completeness, we list these steps above in Algorithm 3. Using the same techniques as in the previous subsection, we find that Algorithm 3 requires $\mathbf{N^2 + 2mN + N}$ flops, which omits the dominating $4m^2$ terms from the naive approach. Using $N = 5$ and $m = 30$, the naive backward substitution algorithm takes $4,289$ flops, while the optimized algorithm takes only 330 flops.

Remark. *We remark that an approach tailored to a sparse storage solution (e.g., coordinate (COO) format for sparse matrices) has been tried and is highly impractical, notably because we cannot permute our rows in such a way to completely prevent fill-in [6]. This fact necessitates a linear search embedded within several loops to ensure that the location of a fill-in is indeed zero, which drives up the time complexity dramatically. Furthermore, Gaussian elimination techniques that use partial pivoting are also difficult since the swapping rows may result in an unpredictable sparsity structure that is difficult to conveniently divide into a grid and solve. So, the optimized Gaussian elimination and backward substitution here do cut back on the number of flops, but the difficulties with the COO-format and partial pivoting make for an ultimately inefficient implementation in terms of memory and numerical stability.*

4 Iterative Methods

This section explores iterative methods, an alternative to solving the linear system directly. We briefly recall how to construct an iterative method to solve a linear system $Ax = b$:

$$x^{(k+1)} = Hx^{(k)} + c,$$

where we can express our matrix as the difference of two matrices $A = M - N$, $H := M^{-1}N$, and $c := M^{-1}b$. Specifically, we set out to find whether iterative methods are appropriate ways to solve our linear system, and if so, how to optimize them to exploit sparsity. Afterwards, we revisit the theory to show how to induce both symmetry and positive definiteness onto the system such that the conjugate gradient method can be applied. Finally, we look into how to adapt the conjugate gradient method when the linear system uses the COO sparse storage format.

4.1 Gauss-Seidel

Recall the Gauss-Seidel iteration $(D - L)x^{(k+1)} = Ux^{(k)} + b$, where $D \in \mathbb{R}^{(N+2m)\times(N+2m)}$ is a square matrix whose diagonal is the diagonal of the KKT matrix, $U \in \mathbb{R}^{(N+2m)\times(N+2m)}$ is the negative upper triangular portion excluding the main diagonal, and $D \in \mathbb{R}^{(N+2m)\times(N+2m)}$ is the negative lower triangular portion excluding the main diagonal. The iteration matrix $H = (D - L)^{-1}U$, and recall that the iterations corresponding to this matrix converge for any initial guess $x^{(0)}$ if and only if the spectral radius $\rho(H) < 1$. Some preliminary computations

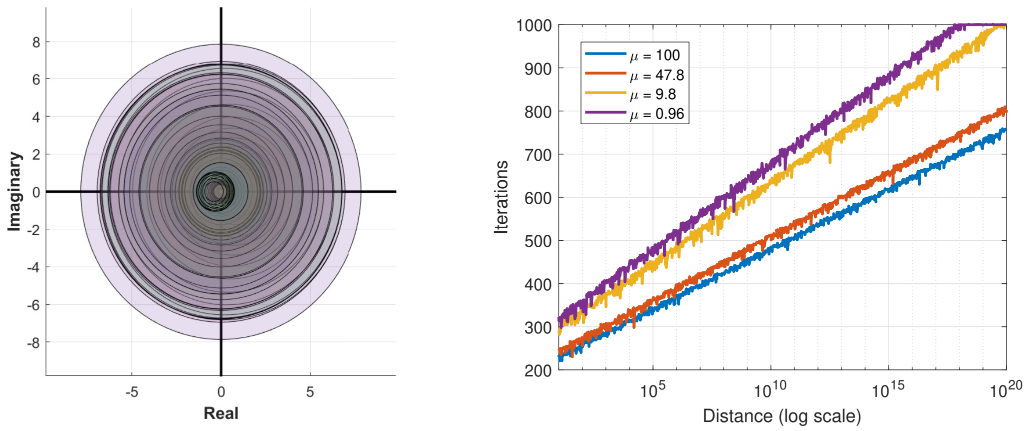

Figure 4: Gauss-Seidel Gershgorin circles (**left**) and iterations to converge with varying distances of the initial guess from the solution (**right**).

reveal that the spectral radius of this linear system falls comfortably beneath 1 (at about ≈ 0.92 for $\mu = 100$), and Fig. 4 reinforces this notion with a plot of the Gershgorin circles for the Gauss-Seidel iteration matrix from a random point in the IP algorithm's execution, along with how the number of iterations to converge is impacted by how far away the initial guess is from the solution. We see that the Gauss-Seidel algorithm takes longer to converge the less relaxed (i.e., the smaller the value μ) the system is and if we are generally start further from the solution. Hence, we may select any initial guess and expect to converge by the necessary and sufficient condition. Later we will see how to transform the KKT system in such a way that we can reduce the spectral radius for Gauss-Seidel – and for that matter, SOR – even further.

The question of exploiting the Gauss-Seidel iterations for sparsity starts with examining the structure of matrices $D-L$ and U, which is shown below in Fig. 5. Which will help us simplify

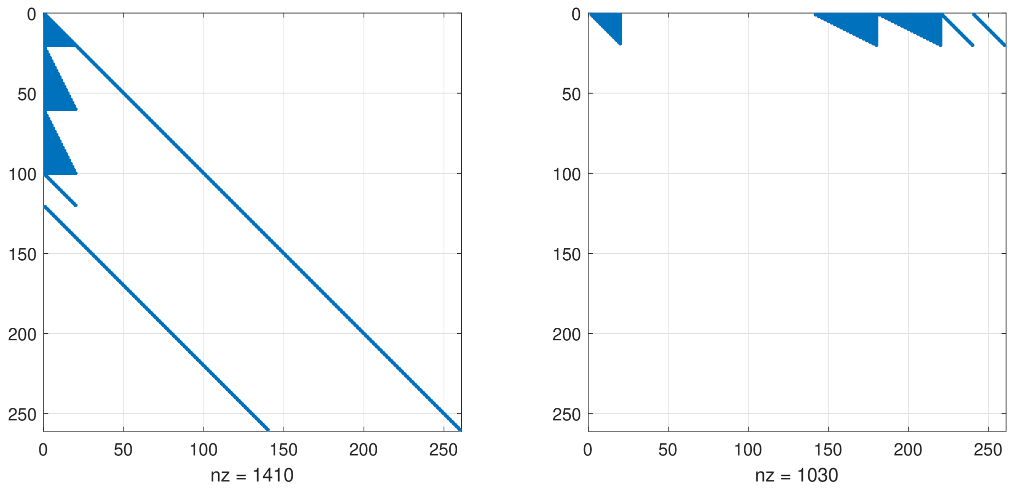

Figure 5: $D-L$ (**left**) and U (**right**).

the elementwise Gauss-Seidel iterations

$$x_i^{(k+1)} = \frac{1}{d_{ii}}\left(\sum_{j=1}^{i-1} l_{ij}x_j^{(k+1)} + \sum_{j=i+1}^{n} u_{ij}x_j^{(k)} + b_i\right)$$

into three phases:

$$x_i^{(k+1)} = \frac{1}{d_{ii}}\left(\sum_{j=1}^{i-1} l_{ij}x_j^{(k+1)} + \sum_{j=i+1}^{N} u_{ij}x_j^{(k)} + \sum_{j=N+m+1}^{N+2m} u_{ij}x_j^{(k)} + b_i\right), \quad 1 \le i \le N \tag{9a}$$

$$x_i^{(k+1)} = \sum_{j=1}^{N} l_{ij}x_j^{(k+1)} + b_i, \quad N+1 \le i \le N+m \tag{9b}$$

$$x_i^{(k+1)} = \frac{1}{d_{ii}}\left(l_{i,i-m}x_{i-m}^{(k+1)} + b_i\right), \quad N+m+1 \le i \le N+2m. \tag{9c}$$

Phase 1 (9a) is the closest out of all the phases to the original Gauss-Seidel iteration, except we account for the emptiness of B_{12} in U, so we skip over these columns. We acknowledge that for this particular example, there are plenty of zeros inside of B_{21} of $D-L$ and B_{13} of U, but generally we cannot make any assumptions about the structure or level of sparsity of these blocks. The structure of these blocks is entirely dependent on the constraints, which are situation-dependent, so we treat them as dense. Phase 2 (9b) considers only the elements in B_{21} of $D-L$ since further columns (excluding the main diagonal) contain only zeros. Note that we do not divide by d_{ii} since it is always equal to 1 here, so the division by 1 is redundant. Also, there are only zeros in these rows of the U matrix, so elements coming from this matrix play no role in computing the next iterate $x_i^{(k+1)}$. The same is true for Phase 3 (9c), which takes advantage of there being only two non-zero entries per row and uses only these terms from $D-L$ to compute the next iterate.

Because the exact number of iterations for Gauss-Seidel – and for that matter, any iterative method discussed in this work – is unknown apriori, it is not appropriate to assign a "complete" flop count to describe how much work it takes to converge to the solution within a predefined tolerance. Instead, we consider flops per iteration as an appropriate metric for iterative methods. That said, the number of flops required per iteration of Gauss-Seidel is $2(N+2m)^2 - N - 2m = \mathbf{8m^2 + 8mN + 2N^2 - 2m - N}$ – there are $2(N+2m)-1$ flops per row, and $N+2m$ rows. Compared to our optimized approach, (9a) requires $2N^2 + 2mN - N$ flops, (9b) requires $2mN$ flops, and (9c) requires $3m$ flops, totaling $\mathbf{2N^2 + 4mN + 3m - N}$ flops per iteration of the optimized Gauss-Seidel method. Similar to other methods, the removal of the dominating m^2 term is the most important change to note.

Because Gauss-Seidel is a specific case of SOR, we will spend less time in the next subsection 4.2 discussing the rationale behind the equations, placing more focus on a pseudocode description from which Gauss-Seidel can be derived ($\omega = 1$).

4.2 Successive Over-Relaxation

Recall the iterations for SOR:

$$(D - \omega L)x^{(k+1)} = \left((1-\omega)D + \omega U\right)x^{(k)} + \omega b.$$

In a similar way to Gauss-Seidel, we can express the above in its per-element form

$$x_i^{(k+1)} = \frac{1}{d_{ii}}\left(\omega \sum_{j=1}^{i-1} l_{ij}x_j^{(k+1)} + \omega \sum_{j=i+1}^{N+2m} u_{ij}x_j^{(k)} + (1-\omega)d_{ii}x_i^{(k)} + \omega b_i\right),$$

where we can make like-minded optimizations compared to that of Gauss-Seidel to get the three-phase approach for an optimized SOR algorithm

$$x_i^{(k+1)} = \frac{1}{d_{ii}}\left(\omega \sum_{j=1}^{i-1} l_{ij}x_j^{(k+1)} + \omega \sum_{j=i+1}^{N} u_{ij}x_j^{(k)} + \omega \sum_{j=N+m+1}^{N+2m} u_{ij}x_j^{(k)} + (1-\omega)d_{ii}x_i^{(k)} + \omega b_i\right), \quad 1 \le i \le N \tag{10a}$$

$$x_i^{(k+1)} = \omega \sum_{j=1}^{N} l_{ij}x_j^{(k+1)} + (1-\omega)d_{ii}x_i^{(k)} + \omega b_i, \quad N+1 \le i \le N+m \tag{10b}$$

$$x_i^{(k+1)} = \frac{1}{d_{ii}}\left(\omega l_{i,i-m}x_{i-m}^{(k+1)} + (1-\omega)d_{ii}x_i^{(k)} + \omega b_i\right), \quad N+m+1 \le i \le N+2m. \tag{10c}$$

In the interest of brevity, the rationale for (10a), (10b), and (10c) is omitted as $\boldsymbol{D}$, $\boldsymbol{L}$, and $\boldsymbol{U}$ assume the same structure as in the Gauss-Seidel method – see the previous subsection 4.1 and Fig. 5 for a proper discussion of structural patterns contained within these matrices. Instead, we offer a pseudocode listing in Algorithm 4.

Algorithm 4 Optimized Successive Over-Relaxation Algorithm

```
1: Input: $A \in \mathbb{R}^{(N+2m)\times(N+2m)}$, $b \in \mathbb{R}^{N+2m}$, $x^{(0)} \in \mathbb{R}^{N+2m}$, $\omega \in \mathbb{R}$
2: Output: $x \in \mathbb{R}^{N+2m}$
3:
4: $x \leftarrow x^{(0)}$
5: for $i = 1, 2, ...$until convergence do
6:     $y \leftarrow x$
7:
8:     % Phase 1 (10a)
9:     if $i \leq N$ then
10:        sum1 $\leftarrow 0$
11:        for $j = 1 : i - 1$ do
12:            sum1 $\leftarrow$ sum1 $+ a_{ij}x_j$
13:        end for
14:        sum2 $\leftarrow 0$
15:        for $j = [1 : N, (N + m + 1) : (N + 2m)]_{[i+1:end]}$ do
16:            sum2 $\leftarrow$ sum2 $+ a_{ij}x_j$
17:        end for
18:        $x_i = \frac{1}{a_{ii}}(\omega b_i - \omega \cdot \text{sum1} - \omega \cdot \text{sum2} + (1 - \omega)a_{ii}y_i)$
19:
20:    % Phase 2 (10b)
21:    else if $i \leq N + m$ then
22:        sum1 $\leftarrow 0$
23:        for $j = 1 : N$ do
24:            sum1 $\leftarrow$ sum1 $+ a_{ij}x_j$
25:        end for
26:        $x_i = \frac{1}{a_{ii}}(\omega b_i - \omega \cdot \text{sum1} + (1 - \omega)a_{ii}y_i)$
27:
28:    % Phase 3 (10c)
29:    else if $i \leq N + 2m$ then
30:        $x_i = \frac{1}{a_{ii}}(\omega b_i + \omega a_{i,i-m}x_{i-m} + (1 - \omega)a_{ii}y_i)$
31:    end if
32: end for
```

Remark. *We note that while this approach cuts back on the number of flops – and consequently, the execution time – it assumes the full storage of a sparse matrix. Unlike Gaussian elimination, the SOR algorithm can be easily adapted to handle the three input vectors that result from representing the sparse matrix A in COO format. For convenience, we include a listing for the SOR COO in Algorithm 5. We emphasize that all COO-tailored algorithms discussed in this work* ***assume "arow" is sorted!***

Algorithm 5 COO Successive Over-Relaxation Algorithm

```
1: Input: arow $\in \mathbb{R}^\ell$, acol $\in \mathbb{R}^\ell$, aval $\in \mathbb{R}^\ell$, $x^{(0)} \in \mathbb{R}^{N+2m}$, $\omega \in \mathbb{R}$
2: Output: $x \in \mathbb{R}^n$
3:
4: $x \leftarrow x^{(0)}$
5: for $i = 1, 2, ...$until convergence do
6:     $y \leftarrow x$
7:     $c \leftarrow 1$
8:     for $i = 1 : N + 2m$ do
9:         sum1 $\leftarrow 0$, sum2 $\leftarrow 0$
10:        while $c \leq \ell$ and $\text{arow}_c = i$ do
11:            col $\leftarrow \text{acol}_c$, $a \leftarrow \text{aval}_c$
12:            if col $< i$ then
13:                sum1 $\leftarrow$ sum1 $+ a \cdot x_{\text{col}}$
14:            else if col $> i$ then
15:                sum2 $\leftarrow$ sum2 $+ a \cdot y_{\text{col}}$
16:            end if
17:            $c \leftarrow c + 1$
18:        end while
19:        $x_i = \frac{1}{a_{ii}}(\omega b_i - \omega \cdot \text{sum1} - \omega \cdot \text{sum2} + (1 - \omega)a_{ii}y_i)$
20:    end for
21: end for
```

ℓ denotes the number of non-zero elements in the sparse matrix. We use a counter c to keep track of our place in the vector and iterate over all possible row values i (Lines 8-20). For each new row, we reset our accumulators sum1 and sum2, and initiate a while loop that runs so long as we have not reached the end of the vectors and we are indexing elements that occur in row i (Lines 9-10). For each element, we consider its value and its column index, and check if it comes before or after i. If it comes before, then the sum uses coefficients from the lower triangular portion of A, which are multiplied by the corresponding value in the posterior x (Lines 12-13). If it comes after, then we multiply its value by the corresponding index of the prior vector y and add it to our running accumulator sum2 (Line 14-15). Once the while loop terminates, we have reached the end of the row, and use the formula from Algorithm 4 to compute the ith index of x. With the next iteration of the outermost for loop, we replace y with x, and repeat this process.

Remark. *Unlike Algorithm 4, Algorithm 5 extends SOR to maximally exploit sparsity since the sparse matrix is now represented with only three vectors. Furthermore, this approach is more adaptable to various MPC problems. In the pursuit of a more general approach, Algorithm 4 was designed with the assumption that B_{21} of $D - L$ and B_{13} of U are completely dense since their structure is problem-dependent. On the other hand, the COO SOR inherently removes any and all zeros regardless of where they may occur, so if these blocks are sparse in any way, it is already accounted for.*

As we observed for Gauss-Seidel, the iteration matrices for SOR exhibit spectral radii that fall beneath 1 (see Fig. 6). The Gershgorin circles for SOR with $\omega = 0.45$ are less wide, limiting

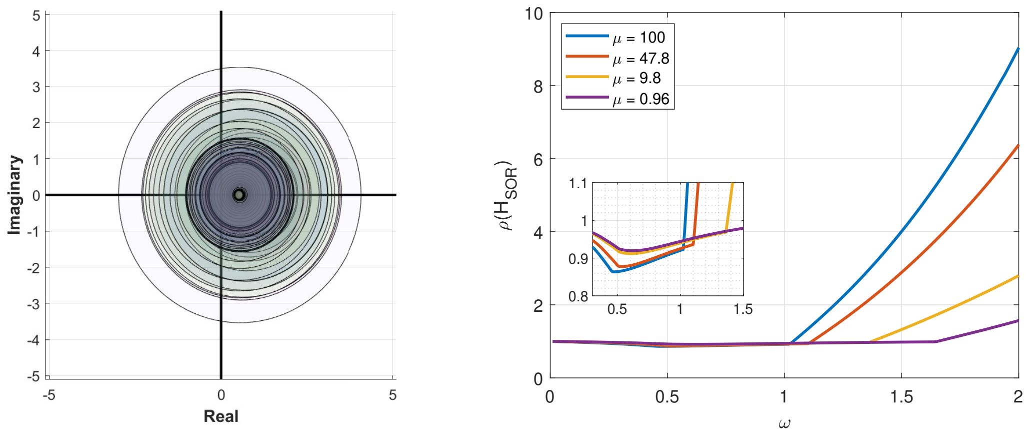

Figure 6: Gershgorin circles for H_{SOR} with $\omega = 0.45$ (**left**) and spectral radii for various ω (**right**).

the possibility of a larger spectral radius. The figure on the right corroborates this claim as we see for $\mu = 100$ the spectral radius for SOR gets as low as 0.86. Thus, even for SOR, we still have a theoretical guarantee of convergence to the solution for any initial guess and an appropriately selected ω – we will see in the next section how to reduce the spectral radius even further.

4.3 Symmetrization of Primal-Dual System

Up to this point, we have been dealing with an asymmetric linear system. We recall a theorem from class which states that if our matrix A is SPD, then SOR with $0 < \omega < 2$ converges for any initial guess. If we can leverage existing techniques to transform our linear system (8) into something SPD, then this will guarantee convergence with the aforementioned iterative methods, eliminating the need to empirically establish a spectral radius as in Fig. 5 and 6.

To start, the primal-dual system (8) is often rewritten in the equivalent form

$$\begin{bmatrix} H & A_I^\top \\ -A_I & \Sigma^{-1} \end{bmatrix} \begin{bmatrix} p_x \\ p_z \end{bmatrix} = -\begin{bmatrix} Hu + f - A_I^\top z \\ A_I u - b_I - \mu Z^{-1} e \end{bmatrix}, \tag{11}$$

where $\Sigma := \frac{1}{\mu} Z^2$ (we refer the reader to [5] for a proper treatment of the primal system for interior point methods). Note the removal of any terms involving S from (11). We acknowledge that the bottom block row of the system can be multiplied by -1 on both sides to make the system symmetric, but preliminary computations reveal that the system is almost never positive definite. Instead, we use the doubly augmented formulation discussed in [7, 8], which guarantees both a symmetric and positive definite matrix for convex problems through the following block-row operation on (11):

$$\begin{bmatrix} Q & -B^\top & r_1 \\ B & D & r_2 \end{bmatrix} \xrightarrow[R_1 \leftarrow R_1 - 2B^\top D^{-1} R_2]{} \begin{bmatrix} Q + 2B^\top D^{-1} B & B^\top & r_1 + 2B^\top D^{-1} r_2 \\ B & D & r_2 \end{bmatrix},$$

where $Q := H$, $B := -A_I$, $D := \Sigma^{-1}$, $r_1 = -Hu - f + A_I^\top z$, and $r_2 := -A_I u + b_I + \mu Z^{-1} e$. Thus, the SPD version of the original interior point problem is

$$\begin{bmatrix} H + 2A_I^\top \Sigma A_I & -A_I^\top \\ -A_I & \Sigma^{-1} \end{bmatrix} \begin{bmatrix} p_x \\ p_z \end{bmatrix} = \begin{bmatrix} r_1 - 2A_I^\top \Sigma r_2 \\ r_2 \end{bmatrix}. \tag{12}$$

With the new linear system (12), we recollect results from the same experiments run in Fig. 6, which are shown below. First, we note that the spectral radius is indeed less than 1 for

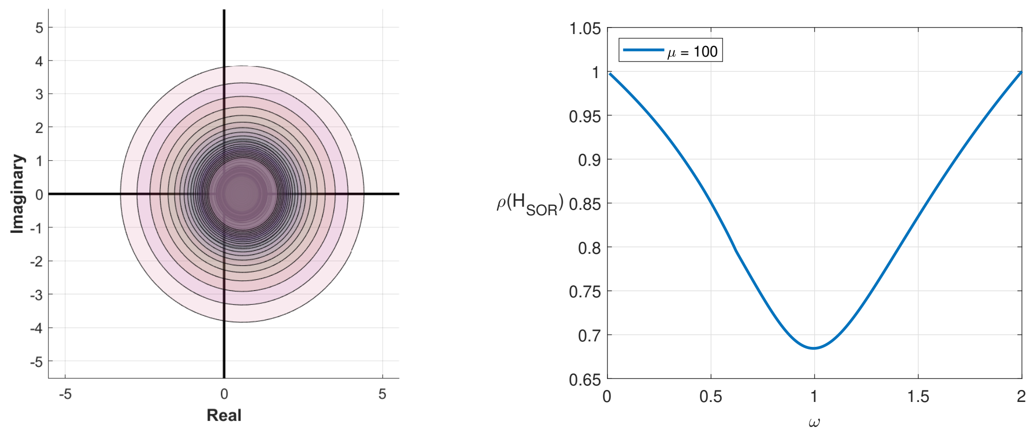

Figure 7: Gershgorin circles for H_{SOR} with $\omega = 0.45$ (**left**) and spectral radii for various ω (**right**).

$0 < \omega < 2$ per the SPD theorem. Interestingly, the spectral radius curve for H_{SOR} sees its minimum value of ≈ 0.68 at $\omega \approx 1$, suggesting that Gauss-Seidel methods may provide generally faster convergence. We will compare iterations to converge in Section 6 between these methods more thoroughly to indeed see if the smaller spectral radius correlates to faster convergence.

4.4 Conjugate Gradient Method

Now that we have obtained an SPD linear system (12), this provides us the opportunity to apply the conjugate gradient method, which uses A-orthogonal search directions to solve the following optimization problem:

$$x^* = \arg\min_{x \in \mathbb{R}^n} \frac{1}{2} x^\top A x - b^\top x, \tag{13}$$

where the objective is a convex quadratic function, so $x^* = A^{-1}b \Leftrightarrow x^*$ is the global minimizer of the objective. Theoretically, we can guarantee convergence in $N + 2m$ iterations, which makes the conjugate gradient method a desirable choice of algorithm for solving our SPD system.

Remark. *Because we do not disrupt the iterations nor change the steps of the algorithm, we neglect to include a pseudocode listing for the full conjugate gradient method here. Instead, we refer the reader to [5] for a proper treatment of the algorithm in its vanilla and preconditioned forms. For brevity's sake, we will consider only the vanilla conjugate gradient method – preconditioned conjugate gradient will be revisited as a point of future work in Section 7. That being said, we are still interested in optimizing the CG's matrix vector multiplications to accommodate the COO format, as this significantly reduces the storage requirement. The pseudocode listing for a matrix-vector multiply with COO format is included in Algorithm 6.*

Algorithm 6 COO Matrix-Vector Multiplication

```
1:  Input: arow ∈ $\mathbb{R}^\ell$, acol ∈ $\mathbb{R}^\ell$, aval ∈ $\mathbb{R}^\ell$, $v \in \mathbb{R}^{N+2m}$
2:  Output: $y \in \mathbb{R}^{N+2m}$
3:
4:  $c \leftarrow 1$
5:  for $i = 1 : N + 2m$ do
6:      sum ← 0
7:      while $c \leq \ell$ and $\text{arow}_c = i$ do
8:          col ← $\text{acol}_c$, $a \leftarrow \text{aval}_c$
9:          sum ← sum + $a \cdot v_{\text{col}}$
10:         $c \leftarrow c + 1$
11:     end while
12:     $y_i$ ← sum
13: end for
```

The algorithm for COO matrix-vector multiplication is similar in concept to the row-vector dot products from Algorithm 5 where we have an outer for loop that iterates over all possible values of i (Lines 5-13), and for each row we keep a running sum of the elementwise products of the current row with the vector v (Lines 7-10). Once we have reached the end of the current row, we store the sum in y_i (Line 12) and repeat the process until the outer for loop terminates. Again, while Algorithm 6 seems isolated, we do in fact use it to accelerate the matrix vector multiplications typical of the conjugate gradient method and enable us to represent our matrix efficiently as only three vectors.

5 Schur Complement

Up to this point, we have considered direct and iterative methods to solving both the original primal-dual system (8) and the SPD modification (12), none of which have considered the alternative approach – compute or approximate the inverse and solve the system with a matrix-vector multiplication. While there are a plethora of available techniques for us to tackle this, we will consider the Schur complement: a tool that allows us represent the inverse of a block matrix in terms of its constituent blocks. This way, we do not have to compute one large matrix inverse, but instead an inverse for a smaller matrix. Consider the following for a nonsingular block matrix $M \in \mathbb{R}^{(p+q)\times(p+q)}$:

$$M^{-1} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} \tag{14}$$

$$= \begin{bmatrix} (M/D)^{-1} & -(M/D)^{-1}BD^{-1} \\ -D^{-1}C(M/D)^{-1} & D^{-1} + D^{-1}C(M/D)^{-1}BD^{-1} \end{bmatrix} \tag{15}$$

where $A \in R^{p\times p}$, $B \in \mathbb{R}^{p\times q}$, $C \in \mathbb{R}^{q\times p}$, $D \in \mathbb{R}^{q\times q}$, and $M/D := A - BD^{-1}C \in \mathbb{R}^{p\times p}$ where D must be nonsingular. Generally, M/D is a dense matrix, meaning there is little – if any – room to optimize our matrix inversion approaches for sparsity. Generally, we select D that is easily invertible, and if we set $p = N$ and $q = 2m$, we can recall the block structure of the primal-dual system (8) shown in (2) and observe that D is invertible. Specifically, we can perform block Gaussian elimination to find its inverse

$$\begin{bmatrix} I & 0 \\ Z & S \end{bmatrix}\begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} \xrightarrow[R_2 \leftarrow R_2 - ZR_1]{} \begin{bmatrix} I & 0 \\ 0 & S \end{bmatrix}\begin{bmatrix} I & 0 \\ -Z & I \end{bmatrix}$$

$$\begin{bmatrix} I & 0 \\ Z & S \end{bmatrix}\begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} \xrightarrow[R_2 \leftarrow S^{-1}R_2]{} \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}\begin{bmatrix} I & 0 \\ -S^{-1}Z & S^{-1} \end{bmatrix},$$

whose blocks are conveniently the inverses and products of diagonal matrices, which can be heavily optimized since the inverse of a diagonal matrix is just the inverse of the elements on the diagonal, and the product of two diagonal matrices is just a matrix of their diagonals multiplied together. All of this is to say that the inverse of D is cheap to compute – it is the computation of $(M/D)^{-1}$ that is not trivial to compute. This square matrix is $N \times N$, varying only with the size of the prediction horizon. In Layman's terms, a larger prediction horizon grants us more foresight for a well-informed control policy, but the tradeoff is a larger dense matrix to invert for the Schur complement method.

5.1 LU Factorization

The first technique we will employ for inverting (M/D) is the LU factorization method for computing an inverse. As we have mentioned in Section 3, the LU factorization for solving our system offers little benefit over Gaussian elimination since the linear system transforms with each iteration. However, it is favorable when inverting a matrix since we only need to find the LU factorization of (M/D) once and then solve N linear systems – one for each standard basis vector. The solutions to each linear system are the columns of $(M/D)^{-1}$; we briefly present this approach in Algorithm 7. Note that e_i is the ith standard basis vector. We compute the

Algorithm 7 Matrix Inversion via LU Factorization

```
1: Input: $M/D \in \mathbb{R}^{N\times N}$
2: Output: $(M/D)^{-1} \in \mathbb{R}^{N\times N}$
3:
4: $L, U \leftarrow$ LU factorization of $M/D$
5: for $i = 1 : N$ do
6:     $y \leftarrow \text{forwardSub}(L, e_i)$
7:     $(M/D)_i^{-1} \leftarrow \text{backSub}(U, y)$
8: end for
```

LU factorization only once (Line 4), which is an $\approx \frac{2}{3}N^3$ operation, and for each column of $(M/D)^{-1}$ we cycle through a for loop, solving for the inverse via repeated forward and backward substitution. Forward and backward substitution are each $\approx N^2$ operations repeated N, so we incur another $\approx N^3$ flops to find the inverse. This method will serve as more or less a benchmark for the quasi-Newton approach.

5.2 Quasi-Newton

Using Newton-related methods to iteratively solve for the inverse requires selecting a relevant function; we use $F(X) = I - (M/D)X$ to perform the Newton iterations, where $F : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}$. Note that $F(X) = 0 \Leftrightarrow X = (M/D)^{-1}$. Traditional Newton iterations applied to this function would require computing $(M/D)^{-1}$ directly, which is obviously counterproductive to our goal here. Instead, we can use quasi-Newton iterations defined by

$$X^{(k+1)} = 2X^{(k)} - X^{(k)}AX^{(k)};$$

see a more thorough discussion of this method in [9]. Defining the residual at iteration k as $R^{(k)} = I - AX^{(k)}$, it can be shown that the order of convergence of the quasi-Newton method is 2. Nonetheless, we require a good initial guess $X^{(0)}$ to ensure quadratic convergence throughout; we select $X^{(0)} = \frac{A^\top}{|||A|||_1 \cdot |||A|||_\infty}$, which is a typical choice for this method.

6 Results

6.1 Implementation Details

This specific implementation considers the case of trajectory optimization for position control of a vehicle. The simplified linear dynamics of the system are included in Section 2. The simulation sets the mass of the vehicle $m = 1$ kg, the air drag coefficient $c = 5 \cdot 10^{-2}$, the sampling period $T_s = 0.1$ seconds, and attempts to control the position in reference to a predesignated Bezier curve. For the interior point algorithm, $\mu^{(0)} = 100$, $\sigma = 0.5$, and Algorithm 1 terminates is $\mu = 0.1$. The fraction-to-the-boundary rule is used with $\tau = 0.995$ to determine the step size, and these step sizes are multiplied by 50 to accelerate convergence – this figure was arrived at through trial and error. The following timing results were obtained after running the programs in MATLAB 2024b on a Dell laptop with an Intel Core Ultra 7 Processor running at 2.1 GHz with 16 GB of RAM and the laptop charger plugged in. For clarity, the control result is shown in Fig. 8 for prediction horizon $N = 20$ and $m = 120$. The remainder of this

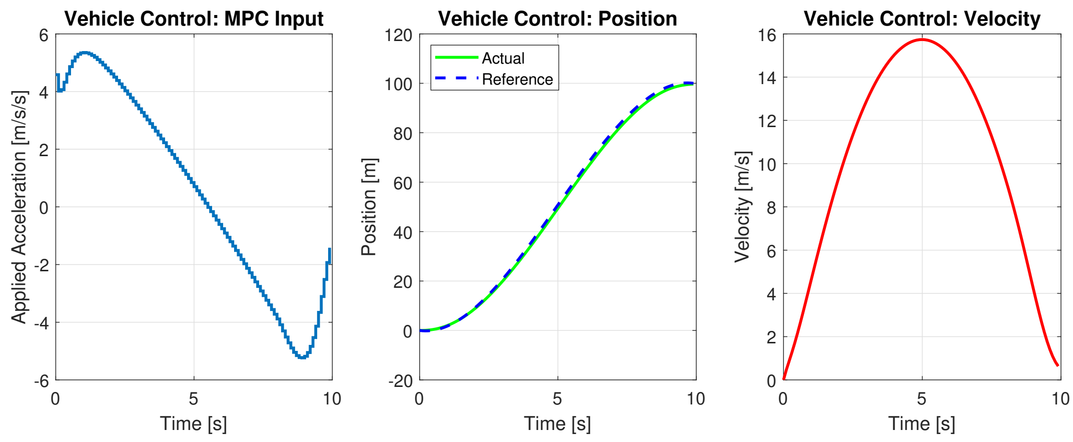

Figure 8: Control expenditure (**left**), position reference tracking (**middle**), and vehicle velocity (**right**).

section focuses less on controller performance and more on the computational performance of the algorithms employed. On that front, tolerance and maximum iterations are fixed at 10^{-9} and 10^3, respectively.

6.2 Primal-Dual Results

For $N = 20$ and $m = 120$, Fig. 9 plots the average execution times of various algorithms when used to solve the primal-dual system; the averages are collected over 1000 trials. As per Fig. 6 and the right subplot in Fig. 9, the parameter ω is set to 0.45 since a value in this general range sees the fewest iterations to converge for SOR. Gaussian elimination and SOR appear to require roughly the same amount of time to compute a result (about $1.3 \cdot 10^{-3}$ seconds), whereas the Schur-Newton method is substantially faster at about $7 \cdot 10^{-4}$ seconds on average. Of course, these

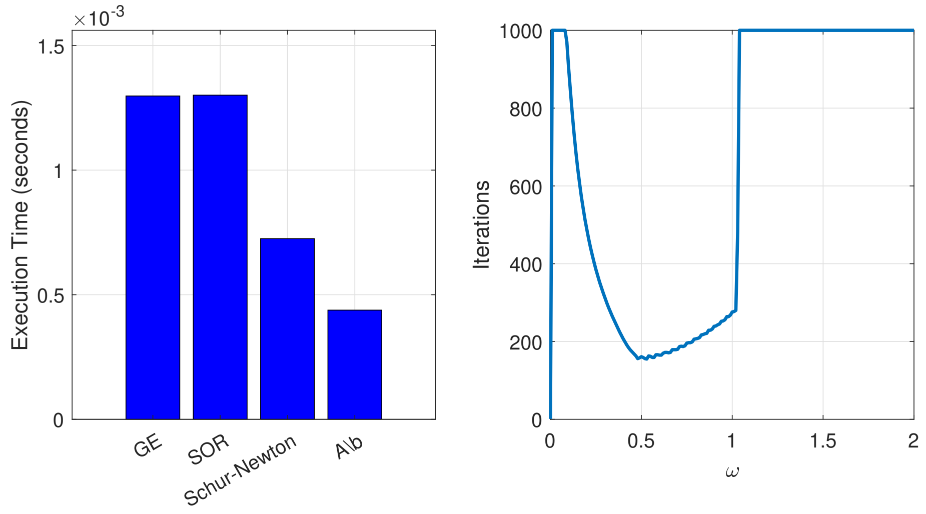

Figure 9: Average execution times of the optimized Gaussian elimination (Alg. 2, 3), SOR with COO format (Alg. 5, $\omega = 0.45$), Schur complement with quasi-Newton extension (Section 5.2), and MATLAB's built-in $A\backslash b$ as applied to the primal-dual system (8).

initial methods are bested by MATLAB's built-in function, which uses techniques unbeknownst to us to solve the linear system in approximately $4.5 \cdot 10^{-4}$ seconds. As a proper reference point, we can observe how these optimized algorithms compare to their unoptimized versions in Fig. 10. Interestingly, Gaussian elimination and SOR exhibit roughly the same execution times before and after optimization, with an average time just shy of $1.6 \cdot 10^{-2}$ seconds. This similarity is likely owed to the fact that the number of iterations to converge for SOR sees $\mathcal{O}(n^2)$ repeated so much that it actually offers only marginal timing gains when compared to Gaussian elimination. As for Schur-Newton, there is not much wiggle room for any optimization since the matrices are dense. Here, the "unoptimized" approach is the LU factorization (Section 5.1), which seems only slightly slower than the quasi-Newton approach (Section 5.2).

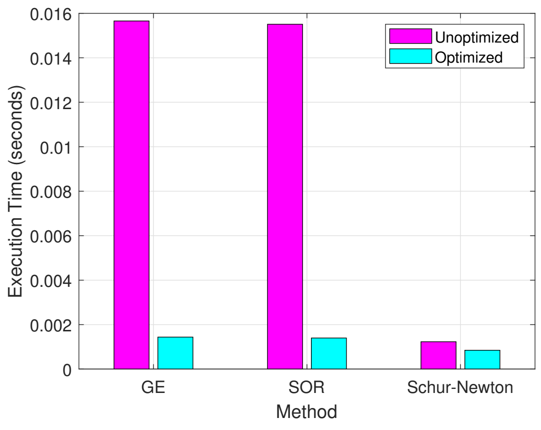

Figure 10: Average execution times of the optimized versus unoptimized algorithms for the primal-dual system.

6.3 SPD Results

We have discussed the benefits of the conversion of the primal-dual system to an SPD system at length in Section 4.3, and those benefits are consistent with the results shown in Fig. 11. Specifically, SOR takes just over $4 \cdot 10^{-4}$ seconds on average to find the solution, whereas the conjugate gradient method is much faster at around $1.5 \cdot 10^{-4}$ seconds. In fact, this is about on par with MATLAB's built-in $A\backslash b$!

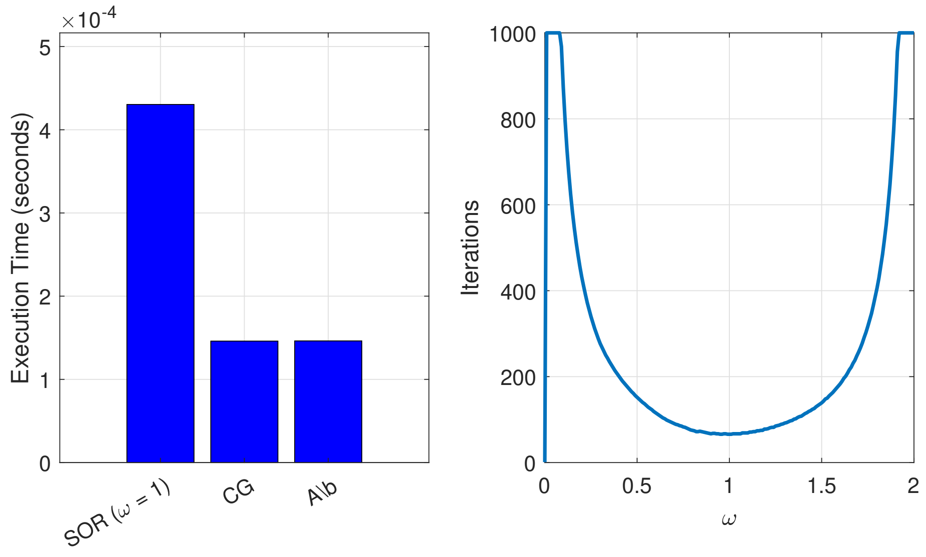

Figure 11: Average execution times of SOR for COO, conjugate gradient method adapted for COO, and MATLAB's built-in $A\backslash b$ for solving the SPD system (12).

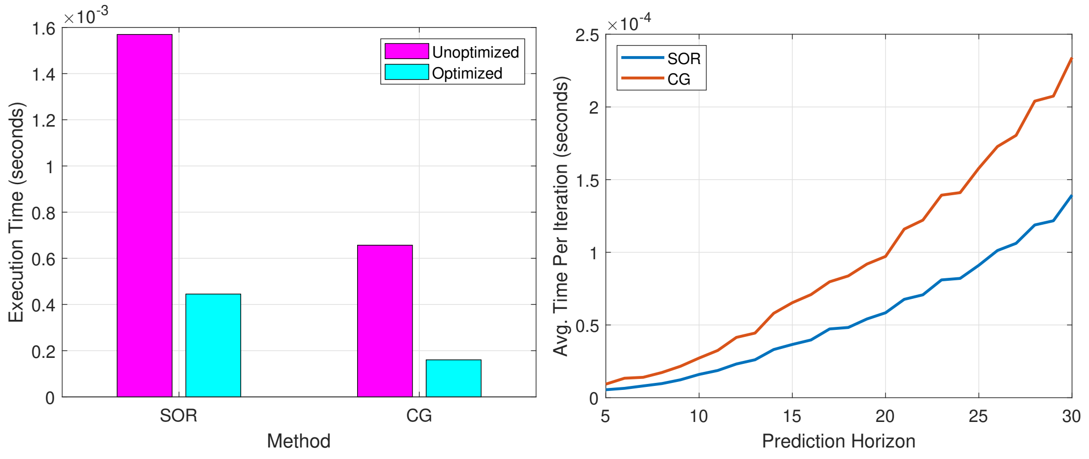

Figure 12: Average execution times of the COO-optimized versus unoptimized algorithms for the SPD system (**left**), average time per iteration of the two algorithms as prediction horizon increases (**right**).

Similar to the primal-dual results, we compare the vanilla and COO-optimized versions SOR and conjugate gradient with one another – the results are plotted in Fig. 12, which sees conjugate gradient requiring less time than SOR in both scenarios to converge to a solution. Specifically, the unoptimized SOR converges in $1.6 \cdot 10^{-3}$ seconds, while conjugate gradient converges in about $6.5 \cdot 10^{-3}$ seconds. It may be easy to assume from this that conjugate gradient is “faster” than SOR, but we emphasize that the average time per iteration of conjugate gradient actually exceeds that of SOR (right subplot in Fig. 12). This indicates that, for this case, conjugate gradient solves the SPD system in fewer iterations than SOR, but requires more time per iteration on average to do so.

7 Conclusion

In summary, we began by investigating Gaussian elimination and backward substitution as a direct method to solve the primal-dual system, adapting the algorithm to handle the sparse structure. In a similar vein, we also explored iterative methods like Gauss-Seidel and SOR, walking through not only how to modify the core algorithms to handle sparse matrices, but also how to adapt them for the COO format, which is a clear advantage over Gaussian elimination. In both theory and practice, these optimizations reduced the total number of computations required for these methods to compute a solution. Furthermore, we used the double augmented method to convert the linear system into an SPD system with reduced dimensions. This technique not only allowed us to achieve a smaller spectral radius for SOR methods, but it also enabled us to apply the conjugate gradient method, the latter of which was the fastest out of all methods discussed in this work. Hence, on the basis of both computation time and storage requirements, **conjugate gradient is the clear winner for solving the linear systems that arise in the interior point algorithm**.

Some points of future work include improving the conjugate gradient method with an appropriate choice of preconditioner. I was able to run the conjugate gradient algorithm with a preconditioner that used Jacobi scaling, but it seemed to converge in slightly more iterations for some reason. I was looking into incomplete Cholesky factorizations as a possible alternative, but ended up running out of time before I could arrive at anything substantial. Still, I am happy that I was able to get a solution to the interior point problem at roughly the same speed as MATLAB’s built-in linear system solver. I would also like to expand further by considering Broyden’s method for approximating the inverse of system matrix, but have difficulty seeing how we could efficiently update a matrix in COO format with arbitrary rank-one updates.

8 Appendix

8.1 Algorithms

Algorithm 8 Gaussian Elimination

```
Input: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$
Output: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$

for $j = 1$ to $n - 1$ do
    for $i = j + 1$ to $n$ do
        $m \leftarrow \frac{a_{ij}}{a_{jj}}$
        for $k = j$ to $n$ do
            $a_{ik} \leftarrow a_{ik} - m \cdot a_{jk}$
        end for
        $b_i \leftarrow b_i - m \cdot b_j$
    end for
end for
```

Algorithm 9 Backward Substitution

```
Input: $U \in \mathbb{R}^{n \times n}$, $c \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$

for $j = 1$ to $n - 1$ do
    for $i = j + 1$ to $n$ do
        $m \leftarrow \frac{a_{ij}}{a_{jj}}$
        for $k = j$ to $n$ do
            $a_{ik} \leftarrow a_{ik} - m \cdot a_{jk}$
        end for
        $b_i \leftarrow b_i - m \cdot b_j$
    end for
end for
```

8.2 MATLAB Code

```matlab

% =========================================================================
% FILE NAME:    main.m
% AUTHOR:       Luke Nuculaj
% CLASS:        APM 5334 - Applied Numerical Methods
%
% DESCRIPTION:  This script sets up the control problem and all of the
% associated matrices, which are then fed to the interior point functions
% for solving. In the for loop, select between "interiorPoint" (primal-sual
% system) and "interiorPointDense" (SPD system). The results of the
% repeated interior point are plotted, showing the control expenditure, the
% reference tracking performance, and the speed of the vehicle.
% =========================================================================

clear all; close all; clc

selectSPD = 1; % (0) for primal-dual system, (1) for SPD system

%% MPC setup

x = zeros(2,1); % state vector
Ts = 0.1; % sampling period [s]
Tsim = 10; % simulation period [s]
t = 0:Ts:Tsim;

b = 0.05; % drag coefficient
A = [0 1; 0 -b];
B = [0; 1];
C = eye(2);
D = [0; 0];

% discrete time dynamics
Ad = eye(2) + Ts.*A;
Bd = Ts.*B;

nx = 2; % number of states
nu = 1; % number of control inputs
p = 20; % size of prediction horizon
x0_t = [0; 0]; % initial state

xmin = repmat([-100; -30], p, 1);
xmax = repmat([100; 22], p, 1);
umin = repmat([-30], p, 1);
umax = repmat([30], p, 1);
M_ab = zeros(nx*p, nu*p);
```

```
M_ak = zeros(nx*p, nx);
qy_weight = [50; 1];
qu_weight = 1;
Qy = diag(qy_weight);
Qu = diag(qu_weight);
My = diag(repmat(qy_weight, p, 1));
Mu = diag(repmat(qu_weight, p, 1));
Mc = kron(eye(p), C);

% assembling the dense matrices
for i = 1:p
    M_ak((i-1)*nx+1:(i-1)*nx+nx, 1:nx) = C*Ad^i;
    for j = 1:p
        if (j > i)
            M_ab((i-1)*nx+1:(i-1)*nx+nx, (j-1)*nu+1:(j-1)*nu+nu) = zeros(nx,nu);
        else
            M_ab((i-1)*nx+1:(i-1)*nx+nx, (j-1)*nu+1:(j-1)*nu+nu) = C*(Ad^(i-j))*Bd;
        end
    end
end

A_i = [M_ab; -M_ab; eye(nu*p); -eye(nu*p)];
H = (1/2).*((M_ab')*My*M_ab + Mu);

% Reference Bezier curve for position tracking
t_bez = linspace(0, 1, Tsim/Ts);
t_bez = [t_bez repmat(t_bez(end), 1, p)]; % extend ending for prediction horizon
bezier_ref = kron((1-t_bez).^3, 0) + kron(3*(1-t_bez).^2.*t_bez, -5) + kron(3*(1-
    t_bez).*t_bez.^2, 105) + kron(t_bez.^3, 100);

avg_time = 0;

for i = 1:length(t)-1
    i
    pos_ref = bezier_ref(i:i+p-1)';
    y_ref = zeros(nx*p, 1);
    y_ref(1:2:end) = pos_ref;
    f = (M_ab')*My*(M_ak*x0_t - y_ref);
    b_i = [xmax-M_ak*x0_t+y_ref; -xmin+M_ak*x0_t-y_ref; umax; -umin];
    tic
    if selectSPD == 1
        [u,~] = interiorPointDense(H,f,[],[],A_i,b_i,nu,p,1);
    else
        [u,~] = interiorPoint(H,f,[],[],A_i,b_i,nu,p,1);
    end
    avg_time = avg_time + toc/length(t);
    x0_t = Ad*x0_t + Bd*u; % update state with optimal control
```

```matlab
    mpc_states(:, i+1) = x0_t;
    u_mpc(i) = u;
end


%% plot results

figure('Position', [100 100 1200 400]);
subplot(1, 3, 1)
stairs(t(1:end-1), u_mpc, 'LineWidth', 2);
title('Vehicle Control: MPC Input');
xlabel('Time [s]')
ylabel('Applied Acceleration [m/s/s]')
grid on
grid minor
subplot(1, 3, 2)
plot(t(1:end-1), mpc_states(1,1:end-1), 'LineWidth', 2, 'Color', 'g');
hold on
plot(t(1:end-1), bezier_ref(1:length(t)-1), 'b--')
hold off
title('Vehicle Control: Position');
xlabel('Time [s]')
ylabel('Position [m]')
legend('Actual', 'Reference', 'Location', 'northwest')
grid on
grid minor
subplot(1, 3, 3)
plot(t(1:end-1), mpc_states(2,1:end-1), 'LineWidth', 2, 'Color', 'r');
title('Vehicle Control: Velocity');
xlabel('Time [s]')
ylabel('Velocity [m/s]')
grid on
grid minor
```

```matlab

% ========================================================================
% FUNCTION NAME: interiorPoint.m
% AUTHOR:        Luke Nuculaj
% DESCRIPTION:   Takes the matrices for a finite-horizon OCP problem of
% interest as input and performs several interior point iterations,
% tightening the relaxation parameter \mu with each step. Returns the
% approximate optimal control input arrived at.
%
% INPUTS:
%   - H : objective function Hessian matrix
%   - f : objective function gradient
%   - A: equality constraint gradient
%   - b: equality constraint bound
%   - A_i: inequality constraint gradient
%   - b_i: inequality constraint bound
%   - alg: (1) for GE, (2) for SOR, (3) for Schur-Newton, other for A\b
%
% OUTPUTS:
%   - u_mpc: optimal control move
%
% EXIT FLAGS:
%   - exitflag: = 0 means the function was successful
%
% ========================================================================

function [u_mpc, exitflag] = interiorPoint(H, f, A, b, A_i, b_i, n_u, N, alg)

exitflag = 0;

% settings
mu = 100;
sigma = 0.5;
tau = 0.995;
TOL = 1.e-9;
MAXIT = 1000;

% initial guesses for Lagrange multipliers (y, z) and slacks (s)
y = ones(size(A, 1), 1);
z = ones(size(A_i, 1), 1);
Z = diag(z);
m = length(b_i);

% Phase 1 method for generating feasible initial guess
nx = size(H,1); ns = length(b_i);
f_init = [ones(ns, 1); zeros(nx, 1)];
A_init = [-eye(ns) zeros(ns,nx);
    eye(ns) A_i];
b_init = [-5.*ones(ns,1); b_i];
options = optimoptions('linprog', 'Display', 'off');
init_guess = linprog(f_init,A_init,b_init,[],[],[],[],options);
s = init_guess(1:ns);
S = diag(s);
u = init_guess(ns+1:end);

% anonymous functions
c_i = @(x) A_i*x - b_i;

while mu > 0.1

```

```
    H_kkt = [H zeros(size(H,1), size(Z, 2)) A_i';
        A_i eye(size(A_i, 1), size(Z, 2)) zeros(size(A_i,1), size(A_i,1));
        zeros(size(Z,1), size(H,2)) Z S];
    grad_kkt = [H*u-(A_i'*z)+f; c_i(u) + s; S*z - mu.*ones(size(S,1),1)];
    n = length(grad_kkt);

    E = @(u, s, z) max([norm(H*u-(A_i'*z)+f, inf) ...
        norm(diag(s)*z - mu.*ones(length(s),1), inf) ...
        norm(c_i(u) + s,inf)]);
    while (E(u, s, z) > mu)

        if alg == 1 % Gaussian elimination
            [U,c,~] = GaussElim_IP(H_kkt, -grad_kkt, N, m);
            [p,~] = backSubs_IP(U, c, N, m);
        elseif alg == 2 % SOR
            w = 0.45;
            [p,~,~] = SOR_IP(H_kkt, -grad_kkt, zeros(n,1), w, TOL, MAXIT, N, m);
        elseif alg == 3 % Schur-Newton
            inv_H_kkt = schurInverseNewton(H_kkt, N, m, TOL, MAXIT);
            p = -inv_H_kkt*grad_kkt;
        else % MATLAB built-in
            p = -H_kkt\grad_kkt;
        end

        lu = length(u); ls = length(s); lz = length(z);
        p_u = p(1:lu);
        p_s = p(lu+1:lu+ls);
        p_z = p(lu+ls+1:lu+ls+lz);
        alpha_s = LineSearch(s, p_s, tau);
        alpha_z = LineSearch(z, p_z, tau);
        u = u + 50*alpha_s*p_u;
        s = s + 50*alpha_s*p_s;
        z = z + 50*alpha_z*p_z;
        S = diag(s); Z = diag(z);

        H_kkt = [H zeros(size(H,1), size(Z, 2)) A_i';
            A_i eye(size(A_i, 1), size(Z, 2)) zeros(size(A_i,1), size(A_i,1));
            zeros(size(Z,1), size(H,2)) Z S];

        grad_kkt = [H*u-(A_i'*z)+f; c_i(u) + s; S*z - mu.*ones(size(S,1),1)];

    end
    mu = sigma*mu; % tighten mu
end
u_mpc = u(1:n_u);
end
```

```

% ========================================================================
% FUNCTION NAME: interiorPointDense.m
% AUTHOR:        Luke Nuculaj
% DESCRIPTION:   Takes the matrices for a finite-horizon OCP problem of
% interest as input and performs several interior point iterations,
% tightening the relaxation parameter \mu with each step. Returns the
% approximate optimal control input arrived at. Constructs the KKT linear
% system using the double augemented SPD form. User may choose between
% conjugate gradient (default) and SOR.
%
% INPUTS:
%   - H : objective function Hessian matrix
%   - f : objective function gradient
%   - A: equality constraint gradient
%   - b: equality constraint bound
%   - A_i: inequality constraint gradient
%   - b_i: inequality constraint bound
%   - alg: (1) SOR, other for conjugate gradient
%
% OUTPUTS:
%   - u_mpc: optimal control move
%
% EXIT FLAGS:
%   - exitflag: = 0 means the function was successful
%
% ========================================================================

function [u_mpc, exitflag] = interiorPointDense(H, f, A, b, A_i, b_i, n_u, N, alg)

exitflag = 0;

% settings
mu = 100;
sigma = 0.5;
tau = 0.995;
TOL = 1.e-9;
MAXIT = 1000;

% initial guesses for Lagrange multipliers (y, z) and slacks (s)
y = ones(size(A, 1), 1);
z = ones(size(A_i, 1), 1);
Z = diag(z);
m = length(b_i);

% Phase 1 method for generating feasible initial guess
nx = size(H,1); ns = length(b_i);
```

```matlab
f_init = [ones(ns, 1); zeros(nx, 1)];
A_init = [-eye(ns) zeros(ns,nx);
    eye(ns) A_i];
b_init = [-5.*ones(ns,1); b_i];
options = optimoptions('linprog', 'Display', 'off');
init_guess = linprog(f_init,A_init,b_init,[],[],[],[],options);
u = init_guess(ns+1:end);

% anonymous functions
c_i = @(x) A_i*x - b_i;

while mu > 0.1

    Zinv = diag(1 ./ z);
    H_kkt = [H + 2*A_i'*(1/mu)*Z*Z*A_i -A_i'; -A_i mu*Zinv*Zinv];
    grad_kkt = [H*u-(A_i'*z)+f; -c_i(u) - mu*Zinv*ones(m,1)];
    r1 = grad_kkt(1:N); r2 = grad_kkt(N+1:end);
    grad_kkt = [r1 - 2*A_i'*(1/mu)*Z*Z*r2; r2];
    n = length(grad_kkt);

    c_i = @(x) A_i*x - b_i;
    E = @(u, z, Zinv) max([norm(H*u-(A_i'*z)+f, inf) ...
        norm(c_i(u) + mu*Zinv*ones(m,1), inf)]);
    while (E(u, z, Zinv) > mu)

        % convert to COO
        [row, col, v] = find(H_kkt);
        coo = sortrows([row col v], 1);
        row = coo(:,1); col = coo(:,2); v = coo(:,3);

        if alg == 1
            % SOR
            [p,~,~] = SOR_COO(row, col, v, -grad_kkt, zeros(n,1), 1, TOL, MAXIT);
        else
            % conjugate gradient method
            [p,~] = CG_COO(row, col, v, -grad_kkt, [], zeros(n, 1), TOL, MAXIT);
        end

        % line search and step
        lu = length(u); lz = length(z);
        p_u = p(1:lu);
        p_z = p(lu+1:lu+lz);
        alpha_z = LineSearch(z, p_z, tau);
        u = u + 50*alpha_z*p_u;
        z = z + 50*alpha_z*p_z;
        Z = diag(z);
        Zinv = diag(1 ./ z);

        % update linear system
        H_kkt = [H + 2*A_i'*(1/mu)*Z*Z*A_i -A_i'; -A_i mu*Zinv*Zinv];
        grad_kkt = [H*u-(A_i'*z)+f; -c_i(u) - mu*Zinv*ones(m,1)];
        r1 = grad_kkt(1:N); r2 = grad_kkt(N+1:end);
        grad_kkt = [r1 - 2*A_i'*(1/mu)*Z*Z*r2; r2];

    end
    mu = sigma*mu; % tighten mu
end
u_mpc = u(1:n_u);
end
```

```

% ========================================================================
% FUNCTION NAME: GaussElim_IP.m
% AUTHOR:         Luke Nuculaj
% CLASS:          APM 5334 - Applied Numerical Methods
% DESCRIPTION:  This function performs Gaussian elimination on the
% primal-dual system accounting for sparsity.
%
% INPUTS:
%   - A: square nxn matrix
%   - b: nx1 rhs vector
%   - p,q: dimensions
%
% OUTPUTS:
%   - A: upper triangular form of A
%   - b: modified rhs
%
% EXIT FLAGS:
%   - exitflag  = -1 means A is not square,
%               = -2 means b is not a column vector or size not
%                    compatible with A,
%               = 0 means the algorithm was successful.
% ========================================================================

function [A,b,exitflag] = GaussElim_IP(A,b,p,q)

n = size(A,1); % number of rows
m = size(A,2); % number of columns

% Check that A is square
if n ~= m
    exitflag = -1;
    return
end
```

```

% Check b is column vector and compatible with A
if n ~= size(b,1) || size(b,2) ~= 1
    exitflag = -2;
    return
end

for j = 1 : p+q
    if (j <= p)
        for i = j+1:p+q
            m = A(i,j)/A(j,j);
            for k = j : n
                A(i,k) = A(i,k) - m*A(j,k);
            end
            b(i) = b(i) - m*b(j);
        end
    else
        m = A(j+q,j);
        A(j+q,j) = 0;
        b(j+q) = b(j+q) - m;
    end
end
exitflag = 0;
```

```
% ========================================================================
% FUNCTION NAME: backSubs_IP.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  Performs backward substitution optimized for the specific
% sparse structure of the interior point problem.
%
% INPUTS:
%   - U: square nxn matrix
%   - c: vector of n elements
%   - p, q: dimension variables
%
% OUTPUTS:
%   - x: solution to the linear system
%
% EXIT FLAGS: = -2, incompatible U and C
%             = -1, U not square
%             = 0, successful
%
% ========================================================================
function [x,exitflag] = backSubs_IP(U,c,p,q)

n = size(U,1); % number of rows
```

```
m = size(U,2); % number of columns

% Check that U is square
if n ~= m
    exitflag = -1;
    return
end

% Check c is column vector and compatible with U
if n ~= size(c,1) || size(c,2) ~= 1
    exitflag = -2;
    return
end

x = zeros(n,1);

for i = n: -1 : n-q+1
    x(i) = c(i)/U(i,i);
end

for i = n-q : -1 : p+1
    s = 0;
    for j = n-q+1 : n
        s = s + U(i,j)*x(j);
    end
    x(i) = c(i) - s;
end

for i = p : -1 : 1
    s = 0;
    for j = i+1 : p
        s = s + U(i,j)*x(j);
    end
    for j = n-q+1:n
        s = s + U(i,j)*x(j);
    end
    x(i) = (c(i) - s)/U(i,i);
end

exitflag = 0;
```

```
% ========================================================================
% FUNCTION NAME: GaussSeidel_IP.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  This function performs Gauss-Seidel iterations on the
% primal-dual system accounting for sparsity.
```

```
%
% INPUTS:
%    - A: square nxn matrix
%    - b: nx1 rhs vector
%    - x0: initial guess
%    - TOL: tolerance for convergence
%    - MAXIT: maximum number of iterations
%    - p: prediction horizon
%    - m: number of constraints
%
% OUTPUTS:
%    - x: solution to the linear system
%    - k: iterations to converge
%
% EXIT FLAGS:
%    - exitflag  = -1 means algorithm did not converge,
%                = 0 means the algorithm did converge.
% ========================================================================

function [x,k,exitflag] = GaussSeidel_IP(A,b,x0,TOL,MAXIT,p,m)

x = x0(:);
n = length(b);
t = [1:p, p+m+1:p+2*m];

for k = 1 : MAXIT
    y = x; % Save previous iteration

    for i = 1 : n
        if i <= p % Phase 1
            sum1 = 0;
            for j = 1 : i - 1
                sum1 = sum1 + A(i,j)*x(j);
            end
            sum2 = 0;
            for j = t(i+1:end)
                sum2 = sum2 + A(i,j)*y(j);
            end
            x(i) = (b(i) - sum1 - sum2)/A(i,i);
        elseif i <= p+m % Phase 2
            sum1 = 0;
            for j = 1 : p
                sum1 = sum1 + A(i,j)*x(j);
            end
            x(i) = b(i) - sum1;
        else % Phase 3
            x(i) = (b(i) - A(i,i-m)*x(i-m))/A(i,i);
        end
    end

    if norm(y-x,inf) < TOL
        exitflag = 0;
        return
    end

end

exitflag = -1;
```

```

% ==========================================================================
% FUNCTION NAME: SOR_IP.m
% AUTHOR:         Luke Nuculaj
% CLASS:          APM 5334 - Applied Numerical Methods
% DESCRIPTION:  Performs SOR on the primal-dual system where the
% iterations are optimized for the sparsity of the system matrix.
%
% INPUTS:
%    - A: nxn matrix
%    - b: RHS vector
%    - x0: initial guess
%    - omega: relaxation parameter
%    - TOL: tolerance for convergence
%    - MAXIT: maximum number of iterations
%    - p: prediction horizon
%    - m: number of constraints
%
% OUTPUTS:
%    - x: solution to the linear system
%    - k: iterations to converge
%
% EXIT FLAGS: = -1, did not converge
%             = 0, did converge
%
% ==========================================================================

function [x,k,exitflag] = SOR_IP(A,b,x0,omega,TOL,MAXIT,p,m)

x = x0(:);
n = length(b);
t = [1:p, p+m+1:p+2*m];

for k = 1 : MAXIT
    y = x; % Save previous iteration

    for i = 1 : n
        if i <= p % Phase 1
            sum1 = 0;
            for j = 1 : i - 1
                sum1 = sum1 + A(i,j)*x(j);
            end
            sum2 = 0;
            for j = t(i+1:end)
                sum2 = sum2 + A(i,j)*y(j);
            end
            x(i) = (omega*b(i) - omega*sum1 - omega*sum2 + (1-omega)*A(i,i)*y(i))/A(
    i,i);
        elseif i <= p+m % Phase 2
            sum1 = 0;
            for j = 1 : p
                sum1 = sum1 + A(i,j)*x(j);
            end
            x(i) = (omega*b(i) - omega*sum1 + (1-omega)*A(i,i)*y(i))/A(i,i);
        else % Phase 3
            x(i) = (omega*b(i) - omega*A(i,i-m)*x(i-m) + (1-omega)*A(i,i)*y(i))/A(i,
    i);
        end
    end

    if norm(y-x,inf) < TOL
        exitflag = 0;
        return
    end

end

exitflag = -1;
```

```

% =========================================================================
% FUNCTION NAME: SOR_COO.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  Performs SOR where the sparse matrix is represented in COO
% format.
%
% INPUTS:
%   - arow: row index vector
%   - acol: column index vector
%   - aval: value vector
%   - b: RHS vector
%   - x0: initial guess
%   - omega: relaxation parameter
%   - TOL: tolerance for convergence
%   - MAXIT: maximum number of iterations
%
% OUTPUTS:
%   - x: solution to the linear system
%   - k: iterations to converge
%
% EXIT FLAGS: = -1, did not converge
%             = 0, did converge
%
% ========================================================================

function [x,k,exitflag] = SOR_COO(arow,acol,aval,b,x0,omega,TOL,MAXIT)

x = x0(:);
n = length(b);
l = length(aval);

for k = 1 : MAXIT
    y = x; % Save previous iteration
    cnt = 1;
    for i = 1 : n
        sum1 = 0; sum2 = 0; aii = 0;
        while (cnt <= l & arow(cnt) == i)
            col = acol(cnt); a = aval(cnt);
            if col < i
                sum1 = sum1 + a*x(col);
            elseif col > i
                sum2 = sum2 + a*y(col);
            else
                aii = a;
            end
            cnt = cnt + 1;
        end
        x(i) = (omega*b(i) - omega*sum1 - omega*sum2 + (1-omega)*aii*y(i))/aii;
    end

    if norm(y-x,inf) < TOL
        exitflag = 0;
        return
    end

end

exitflag = -1;
```

```

% ==========================================================================
% FUNCTION NAME: schurInverseLU.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  This function computes the inverse of the primal-dual
% system matrix via the Schur complement, computing the smaller inverse via
% LU factorization.
%
% INPUTS:
%   - M: square nxn matrix
%   - p,q: dimensions
%
% OUTPUTS:
%   - Minv: inverse of M
%
% EXIT FLAGS: (none)
% ==========================================================================

function [Minv] = schurInverseLU(M, p, q)

persistent Dinv;
persistent A;
persistent B;
persistent C;

if isempty(A)
    A = M(1:p,1:p);
    B = M(1:p,p+1:end);
    C = M(p+1:end,1:p);
    Dinv = [eye(q) zeros(q); zeros(q, 2*q)];
end

D = M(p+1:end,p+1:end);

Z = D(q+1:end, 1:q); S = D(q+1:end, q+1:end);
diag_sinv = 1 ./ diag(S);
diag_z = diag(Z);

for i = 1:q
    Dinv(i+q,i) = -diag_sinv(i)*diag_z(i);
    Dinv(i+q,i+q) = diag_sinv(i);
end

MsD = A - B*Dinv*C;
MsDinv = mtx_inv_LU(MsD);
T = MsDinv*B*Dinv; U = Dinv*C;
```

```
Minv = [MsDinv -T; -U*MsDinv Dinv+U*T];

end
```

```

% ========================================================================
% FUNCTION NAME: schurInverseNewton.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  This function computes the inverse of the primal-dual
% system matrix via the Schur complement, computing the smaller inverse via
% quasi-Newton iterations.
%
% INPUTS:
%   - M: square nxn matrix
%   - p,q: dimensions
%   - TOL: tolerance for convergence
%   - MAXIT: maximum number of iterations
%
% OUTPUTS:
%   - Minv: inverse of M
%
% EXIT FLAGS: (none)
% ========================================================================

function [Minv] = schurInverseNewton(M, p, q, TOL, MAXIT)

persistent Dinv;
persistent A;
persistent B;
persistent C;

if isempty(A)
    A = M(1:p,1:p);
    B = M(1:p,p+1:end);
    C = M(p+1:end,1:p);
    Dinv = [eye(q) zeros(q); zeros(q, 2*q)];
end

D = M(p+1:end,p+1:end);

Z = D(q+1:end, 1:q); S = D(q+1:end, q+1:end);
diag_sinv = 1 ./ diag(S);
diag_z = diag(Z);

for i = 1:q
    Dinv(i+q,i) = -diag_sinv(i)*diag_z(i);
    Dinv(i+q,i+q) = diag_sinv(i);
end

MsD = A - B*Dinv*C;
MsDinv = mtx_inv_newton(MsD, TOL, MAXIT, MsD'/(norm(MsD,1)*norm(MsD,inf)));
T = MsDinv*B*Dinv; U = Dinv*C;
Minv = [MsDinv -T; -U*MsDinv Dinv+U*T];

end
```

```

% ========================================================================
% FUNCTION NAME: mtx_inv_LU.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  Computes the inverse of the provided matrix using
% the LU factorization and forward + backward substitution on
% consecutive linear systems constructed with the standard basis
% vectors.
%
% INPUTS:
%   - A: square nxn matrix
%
% OUTPUTS:
%   - x: computed inverse
%
% EXIT FLAGS: (none)
%
% ========================================================================

function [x] = mtx_inv_LU(A)

n = size(A, 1);
[L,U] = DoolittleLU(A);
x = eye(n);

for i = 1:n
    ei = x(:,i);
    [ci,~] = forwardSubs(L,ei);
    [x(:,i),~] = backSubs(U,ci);
end

end
```

```

% ========================================================================
% FUNCTION NAME: mtx_inv_newton.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  Computes the inverse of the provided matrix using
% quasi-Newton iterations.
%
% INPUTS:
%   - A: square nxn matrix
%   - TOL: tolerance
%   - MAXIT: maximum number of iterations
%   - x0: initial guess
%
% OUTPUTS:
%   - x: computed inverse
%   - iterates: array of all the quasi-Newton iterates
%
% EXIT FLAGS: (none)
%
% =========================================================================

function [x, iterates] = mtx_inv_newton(A, TOL, MAXIT, x0)

n = size(A,1);
R = @(x, A) eye(n) - A*x;

x = x0;
for i = 1:MAXIT
    if norm(R(x, A), inf) < TOL
        return
    else
        x = x + x*R(x, A);
    end
end
```

```
% =========================================================================
% FUNCTION NAME: conjugateGradient.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  Performs conjugate gradient on the SPD system.
%
% INPUTS:
%   - A: nxn matrix
%   - b: RHS vector
%   - M: preconditioner (as a vector)
%   - x0: initial guess
%   - TOL: tolerance for convergence
%   - MAXIT: maximum number of iterations
%
```

```
% OUTPUTS:
%   - x: solution to the linear system
%   - k: iterations to converge
%
% EXIT FLAGS: = -2, incompatible U and C
%             = -1, U not square
%             = 0, successful
%
% ========================================================================

function [x, k] = conjugateGradient(A, b, M, x0, TOL, MAXIT)

if (isempty(M))
    M = ones(length(b), 1);
end
x = x0(:);
r = A*x0 - b;
y = M.*r;
p = -y;

for k = 1:MAXIT
    if norm(r, inf) < TOL
        return
    else
        alpha = r'*y/(p'*A*p);
        x = x + alpha*p;
        rp1 = r + alpha*A*p;
        yp1 = M.*rp1;
        beta = rp1'*yp1/(r'*y);
        p = -yp1 + beta*p;
        r = rp1; y = yp1;
    end
end

end
```

```

% ========================================================================
% FUNCTION NAME: CG_COO.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  Performs conjugate gradient where the sparse matrix is
% represented in COO format.
%
% INPUTS:
%   - arow: row index vector
%   - acol: column index vector
%    - aval: value vector
%    - b: RHS vector
%    - M: preconditioner (as a vector)
%    - x0: initial guess
%    - TOL: tolerance for convergence
%    - MAXIT: maximum number of iterations
%
% OUTPUTS:
%    - x: solution to the linear system
%    - k: iterations to converge
%
% EXIT FLAGS: = -2, incompatible U and C
%             = -1, U not square
%             = 0, successful
%
% ========================================================================

function [x,k] = CG_COO(arow, acol, aval, b, M, x0, TOL, MAXIT)

if (isempty(M))
    M = ones(length(b), 1);
end
x = x0(:);
r = mvmCOO(arow, acol, aval, x) - b;
y = M.*r;
p = -y;

for k = 1:MAXIT
    if norm(r, inf) < TOL
        return
    else
        t = mvmCOO(arow, acol, aval, p);
        alpha = r'*y/(p'*t);
        x = x + alpha*p;
        rp1 = r + alpha*t;
        yp1 = M.*rp1;
        beta = rp1'*yp1/(r'*y);
        p = -yp1 + beta*p;
        r = rp1; y = yp1;
    end
end

end
```

```

% ========================================================================
% FUNCTION NAME: mvmCOO.m
% AUTHOR:        Luke Nuculaj
% CLASS:         APM 5334 - Applied Numerical Methods
% DESCRIPTION:  This function performs matrix vector multiplication where
% the matrix is represented in COO format.
%
% INPUTS:
%   - arow: row index vector
%   - acol: column index vector
%   - aval: value vector
%   - v: vector in the matrix vector multiplication
%
% OUTPUTS:
%   - y: result
%
% EXIT FLAGS: (none)
% ========================================================================

function [y] = mvmCOO(arow, acol, aval, v)

n = length(v);
l = length(aval);
y = zeros(n, 1);

c = 1;
for i = 1:n
    sum = 0;
    while (c <= l & arow(c)==i)
        col = acol(c); a = aval(c);
        sum = sum + a*v(col);
        c = c+1;
    end
    y(i) = sum;
end
end
```

References

[1] D. Q. Mayne, “Model predictive control: Recent developments and future promise,” *Automatica*, vol. 50, no. 12, pp. 2967–2986, 2014.

[2] M. Schwenzer, M. Ay, T. Bergs, and D. Abel, “Review on model predictive control: An engineering perspective,” *The International Journal of Advanced Manufacturing Technology*, vol. 117, no. 5, pp. 1327–1349, 2021.

[3] J. L. Jerez, P. J. Goulart, S. Richter, G. A. Constantinides, E. C. Kerrigan, and M. Morari, “Embedded online optimization for model predictive control at megahertz rates,” *IEEE*

Transactions on Automatic Control, vol. 59, no. 12, pp. 3238–3251, 2014.

[4] D. Axehill, "Controlling the level of sparsity in mpc," *Systems & Control Letters*, vol. 76, pp. 1–7, 2015.

[5] J. Nocedal and S. J. Wright, *Numerical optimization.* Springer, 1999.

[6] R. Schreiber, "A new implementation of sparse gaussian elimination," *ACM Transactions on Mathematical Software (TOMS)*, vol. 8, no. 3, pp. 256–276, 1982.

[7] A. Forsgren, P. E. Gill, and J. D. Griffin, "Iterative solution of augmented systems arising in interior methods," *SIAM Journal on Optimization*, vol. 18, no. 2, pp. 666–690, 2007.

[8] F. Liu, A. Fredriksson, and S. Markidis, "Krylov solvers for interior point methods with applications in radiation therapy and support vector machines," 2024. [Online]. Available: https://arxiv.org/abs/2308.00637

[9] V. Pan and R. Schreiber, "An improved newton iteration for the generalized inverse of a matrix, with applications," *SIAM Journal on Scientific and Statistical Computing*, vol. 12, no. 5, pp. 1109–1130, 1991.