Implementation of the Primal-Dual Path-following IPM

Arnau Pérez Reverte

April 27, 2025

Large-Scale Optimization

Contents

1 Introduction **1**

2 Algorithm Formulation **1**
2.1 Generic Primal-Dual Path-Following Algorithm . . . 2
2.2 Implementation Considerations . . . 2
2.2.1 Starting Point . . . 2
2.2.2 Termination Conditions . . . 3
2.2.3 Step Length . . . 3
2.2.4 Solving the Systems of Equations . . . 3
2.2.5 Mehrotra's Predictor-Corrector Direction . . . 4

3 Implementation **5**
3.1 Overview . . . 6
3.2 Residual and Gap Calculation . . . 7
3.3 Termination Checks . . . 7
3.4 Step length calculation . . . 7
3.5 Direction Computation . . . 8
3.6 Main Iteration Loop . . . 9

4 Results and Conclusion **10**
4.1 Testing framework . . . 10
4.2 Netlib results . . . 10

1 Introduction

The following report outlines the Primal-Dual Path-following Interior Point Method (IPM), as developed in Large-Scale Optimization class, and presents an implementation in the Python programming language using several numerical computing libraries. The code is tested against some of the Netlib problems to verify its correct implementation.

2 Algorithm Formulation

Primal-dual path-following methods are a class of interior-point methods (IPMs) used for solving linear programming (LP) problems, and can be extended to quadratic and convex programming. These methods operate by iteratively moving towards an optimal solution while maintaining strict positivity of the primal and dual variables.

2.1 Generic Primal-Dual Path-Following Algorithm

The generic primal-dual path-following IPM aims to solve the LP primal and dual problems:

$$\begin{aligned} &\text{(P)} \quad \min c^T x \quad \text{s.to} \quad Ax = b, \quad x \geq 0 \\ &\text{(D)} \quad \max b^T \lambda \quad \text{s.to} \quad A^T\lambda + s = c, \quad s \geq 0 \end{aligned}$$

by approximately following the central path. The central path is defined as the set of solutions $(x_\tau, \lambda_\tau, s_\tau)$ to the perturbed KKT-τ system:

$$\begin{aligned} A^T\lambda + s &= c \\ Ax &= b \\ XSe &= \tau e \\ (x, s) &> 0 \end{aligned}$$

where $X = \text{diag}(x)$, $S = \text{diag}(s)$, e is a vector of ones, and $\tau \in \mathbb{R}^+$ is a positive barrier parameter. As τ approaches 0, the solution $(x_\tau, \lambda_\tau, s_\tau)$ approaches the optimal solution of the original LP problem.

The generic algorithm iteratively reduces τ towards zero. At each iteration, given a current point (x^k, λ^k, s^k) with $x^k > 0$ and $s^k > 0$, a Newton-like direction $(\Delta x^k, \Delta\lambda^k, \Delta s^k)$ is computed to approach the KKT-$\sigma_k\mu_k$ system:

$$\begin{aligned} A^T\lambda + s &= c \\ Ax &= b \\ XSe &= \sigma_k\mu_k e \\ (x, s) &> 0 \end{aligned}$$

where $\mu_k = (x^k)^T s^k/n$ is the duality measure and $\sigma_k \in (0, 1)$ is a centering parameter. The linear system to be solved for the direction is:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S^k & 0 & X^k \end{bmatrix} \begin{bmatrix} \Delta x^k \\ \Delta\lambda^k \\ \Delta s^k \end{bmatrix} = \begin{bmatrix} -r_c^k \\ -r_b^k \\ -X^k S^k e + \sigma_k\mu_k e \end{bmatrix}$$

where $r_c^k = A^T\lambda^k + s^k - c$ (dual infeasibility) and $r_b^k = Ax^k - b$ (primal infeasibility).

A step length α is then computed to maintain positivity of $x^{k+1} = x^k + \alpha\Delta x^k$ and $s^{k+1} = s^k + \alpha\Delta s^k$, and the iterates are updated:

$$\begin{aligned} x^{k+1} &= x^k + \alpha\Delta x^k \\ \lambda^{k+1} &= \lambda^k + \alpha\Delta\lambda^k \\ s^{k+1} &= s^k + \alpha\Delta s^k \end{aligned}$$

The process continues until a termination criterion is met.

2.2 Implementation Considerations

Practical implementations of primal-dual path-following IPMs involve several key aspects.

2.2.1 Starting Point

For feasible methods, an initial point (x^0, λ^0, s^0) with $x^0 > 0$, $s^0 > 0$, and satisfying the primal and dual constraints is needed. For infeasible methods, only $x^0 > 0$ and $s^0 > 0$ are required. A common initial choice is $(x^0, s^0) = 10e$, and $\lambda^0 = 0$. Better starting points can be obtained by approximately solving perturbed KKT conditions or using least-squares approaches.

2.2.2 Termination Conditions

The algorithm terminates when the infeasibilities and the duality gap are sufficiently small. Typical termination criteria include:

$$\frac{||A^T\lambda^k + s^k - c||}{1 + ||c||} \leq \epsilon_{feas}$$
$$\frac{||Ax^k - b||}{1 + ||b||} \leq \epsilon_{feas}$$
$$\mu_k = \frac{(x^k)^T s^k}{n} \leq \epsilon_{opt}$$

where ϵ_{feas} and ϵ_{opt} are small tolerances (e.g., 10^{-8}).

2.2.3 Step Length

While theory often suggests the same step length for primal and dual variables, practical implementations often use different step lengths α_p and α_d for primal and dual updates, respectively, to maximize progress while maintaining positivity. These are typically computed as:

$$\alpha_p = \min\{1, \rho \cdot \max\{\alpha \geq 0 : x^k + \alpha \Delta x^k \geq 0\}\}$$
$$\alpha_d = \min\{1, \rho \cdot \max\{\alpha \geq 0 : s^k + \alpha \Delta s^k \geq 0\}\}$$

where $\rho \in [0.95, 0.99995]$ is a reduction parameter close to 1.

2.2.4 Solving the Systems of Equations

At each iteration of the primal-dual path-following IPM, a system of linear equations needs to be solved to determine the search direction $(\Delta x, \Delta\lambda, \Delta s)$. The system is given by:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = \begin{bmatrix} -r_c \\ -r_b \\ -r_{xs} \end{bmatrix}$$

where $r_{xs} = XSe - \sigma\mu e$. This system has $2n + m$ variables and equations and is generally neither symmetric nor positive definite. Directly solving this system using LU factorization is possible but not the most efficient approach in practice. Instead, Gaussian elimination is typically employed to reduce the size and structure of the system, leading to two main alternative systems: the augmented system and the normal equations.

System 1: KKT

The KKT conditions can be written as $F(x, \lambda, s) = 0$, where F is a vector-valued function. Newton's method applied to this system leads to solving $\nabla F(x, \lambda, s)\Delta = -F(x, \lambda, s)$. The Jacobian $\nabla F(x, \lambda, s)$ gives rise to the coefficient matrix in the linear system mentioned above.

System 2: Augmented System

The augmented system is derived by eliminating Δs from the original system. From the third block of equations, $S\Delta x + X\Delta s = -r_{xs}$, we can express Δs as:

$$\Delta s = -X^{-1}(r_{xs} + S\Delta x)$$

Substituting this into the first block of equations, $A^T\Delta\lambda + \Delta s = -r_c$, yields:

$$A^T\Delta\lambda - X^{-1}(r_{xs} + S\Delta x) = -r_c$$
$$-X^{-1}S\Delta x + A^T\Delta\lambda = -r_c + X^{-1}r_{xs}$$

Combining this with the second block of equations, $A\Delta x = -r_b$, we get the augmented system:

$$\begin{bmatrix} -XS^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -r_c + X^{-1}r_{xs} \\ -r_b \end{bmatrix}$$

Letting $\Theta = XS^{-1}$, the system becomes:

$$\begin{bmatrix} -\Theta & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -r_c + X^{-1} r_{xs} \\ -r_b \end{bmatrix}$$

This system has $n + m$ variables and equations, is symmetric, but indefinite. It can be solved using Bunch-Parlett factorization.

System 3: Normal Equations

The normal equations approach aims to solve for $\Delta\lambda$ first. From the augmented system, we have $-\Theta\Delta x + A^T\Delta\lambda = -r_c + X^{-1}r_{xs}$, which gives $\Delta x = \Theta^{-1}(A^T\Delta\lambda + r_c - X^{-1}r_{xs})$. Substituting this into $A\Delta x = -r_b$, we obtain:

$$(A\Theta A^T)\Delta\lambda = -r_b + A(-XS^{-1}r_c + e - \sigma\mu S^{-1}e)$$

If A has full row rank, the matrix $A\Theta A^T$ is symmetric and positive definite, allowing the use of sparse Cholesky factorization for efficient solution. Once $\Delta\lambda$ is found, Δs and Δx can be computed by back-substitution:

$$\Delta s = -r_c - A^T\Delta\lambda$$
$$\Delta x = -S^{-1}(r_{xs} + X\Delta s)$$

For most LP problems, the normal equations approach with Cholesky factorization is more effective due to its efficiency in handling sparse matrices. However, for certain problems, especially QPs and general convex programs, the augmented system might be preferred.

2.2.5 Mehrotra's Predictor-Corrector Direction

Mehrotra's predictor-corrector direction is a widely used heuristic to enhance the performance of primal-dual IPMs. It employs an adaptive adjustment of the centrality parameter σ at each iteration and utilizes a second-order Taylor approximation of the KKT conditions. The method computes the search direction in two main phases: the predictor step and the corrector step (which is often combined with a centering step).

Predictor Direction

The predictor step aims to find an affine-scaling direction by neglecting the second-order term and setting $\sigma = 0$ in the perturbed KKT system. The following linear system is solved:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x^{aff} \\ \Delta \lambda^{aff} \\ \Delta s^{aff} \end{bmatrix} = \begin{bmatrix} -r_c \\ -r_b \\ -XSe \end{bmatrix}$$

After computing the affine-scaling direction, the maximum step lengths α_p^{aff} and α_d^{aff} that maintain non-negativity of x and s are determined:

$$\alpha_p^{aff} = \max\{\alpha \in [0,1] : x + \alpha\Delta x^{aff} \geq 0\}$$
$$\alpha_d^{aff} = \max\{\alpha \in [0,1] : s + \alpha\Delta s^{aff} \geq 0\}$$

The predicted duality measure μ_{aff} is then calculated for the point obtained by taking these maximal steps:

$$\mu_{aff} = \frac{(x + \alpha_p^{aff}\Delta x^{aff})^T(s + \alpha_d^{aff}\Delta s^{aff})}{n}$$

The centrality parameter σ for the current iteration is then estimated based on the ratio of μ_{aff} to the current duality measure μ:

$$\sigma = \left(\frac{\mu_{aff}}{\mu}\right)^3$$

This heuristic for σ aims to be aggressive (small σ) when the affine step makes good progress and more centering (larger σ) when the affine step is poor.

Centering Direction

The centering direction $(\Delta x^{cen}, \Delta\lambda^{cen}, \Delta s^{cen})$ aims to move the iterates closer to the central path. It is obtained by solving the system:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x^{cen} \\ \Delta\lambda^{cen} \\ \Delta s^{cen} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sigma\mu e \end{bmatrix}$$

Corrector Direction

The corrector step accounts for the second-order term $\Delta X \Delta S e$ that was ignored in the predictor step. The system for the corrector direction $(\Delta x^{co}, \Delta\lambda^{co}, \Delta s^{co})$ is:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x^{co} \\ \Delta\lambda^{co} \\ \Delta s^{co} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\Delta X^{aff} \Delta S^{aff} e \end{bmatrix}$$

where $\Delta X^{aff} = \text{diag}(\Delta x^{aff})$ and $\Delta S^{aff} = \text{diag}(\Delta s^{aff})$.

3 Implementation

Attached to this report you will find the code with the implementation of the algorithm. Our implementations is based on Python. In order to run the code, you will need a **Python** base installation with version greater than or equal to 3.12.3. Moreover, the following external libraries need to be appended (you can find a `pyproject.toml` attached aswell):

1. **numpy**: Version 2.2.5 or greater.
2. **ipykernel**: Version 6.29.5 or greater.
3. **scipy**: Version 1.15.2 or greater.
4. **amplpy**: Version 0.14.0 or greater.
5. **pytest**: Version 8.3.5 or greater.
6. **scikit-sparse**: Version 0.4.15 or greater.

These libraries can be installed using **pip** or other library management tools like **uv**; we recommend the building of a Python virtual enviroment to do so using **rye**.

The given libraries are mostly destined for numerical computing purposes. In particular, we put emphasis into **scipy** and **scikit-parse**:

- **Scipy**: SciPy is a comprehensive Python library for scientific and technical computing. In this project, we use this library to use its linear algebra features, specially those dealing with **sparse matrices**.

  The modules used in this case are `scipy.sparse` and `scipy.sparse.linalg`. The Netlib library, and most real problems, present constraint matrices A which are sparse. Storing and manipulating these matrices efficiently is crucial for the performance of the algorithm, especially for large-scale problems. In our case, `scipy.sparse` provides tools to define, operate on and solve linear systems with matrices of such characteristics, while efficiently handling the inner workings. Imported as `sp`, the data structure handling these matrices is the **Compressed Sparse Column (CSC)**, and by calling methods like `sp.csc_matrix()` and `A.tocsc()`, one can convert other standard format arrays like Numpy into CSC. Then, linear systems of equations on these matrices are solved using the `scipy.sparse.linalg` module (imported as `spla`), with `spla.spsolve()`, which is a direct sparse solver that can handle general sparse matrices, which may not be symmetric or positive definite.

- **Scikit-sparse**: This library is actually an interface for the CHOLMOD library, which is an efficient and robust sparse Cholesky factorization package. We use this library and its `ch.cholesky()` method to compute the Cholesky factorization of the symmetric positive-definite matrix $A\Theta A^\top$ which arises in the normal equations.

Nonetheless, note that several shortcomings were found during the development of this project. The main development was performed using an Apple Silicon-based Mac, which lead to not being able to succesfully install the **scikit-sparse** library. For running all instances using the System 3 computation, we overcame the problem by running our code in Google Colab.

3.1 Overview

All the algorithm is encapsulated by a single class `PrimalDualPathFollowing`. This object-oriented approach encapsulates both the data and the functions necessary to solve a linear programming problem using this IPM. The class is initialized with the problem data and algorithm-specific parameters. In particular, these are:

- `c`: `numpy.ndarray`. This parameter represents the cost vector (c) in the linear programming problem: minimize $c^T x$ subject to $Ax = b$, $x \geq 0$.
- `A`: `numpy.ndarray` or `scipy.sparse.csc_matrix`. This is the constraint matrix (A) in the LP problem $Ax = b$. The code converts `A` to a sparse Compressed Sparse Column (CSC) format for efficiency.
- `b`: `numpy.ndarray`. This parameter is the right-hand side vector (b) of the equality constraints $Ax = b$.
- `x0`: `numpy.ndarray`. This provides the initial strictly positive primal variable vector ($x^0 > 0$).
- `lambda0`: `numpy.ndarray`. This is the initial estimate for the dual variable vector (λ^0) associated with the equality constraints $Ax = b$.
- `s0`: `numpy.ndarray`. This is the initial strictly positive dual slack variable vector ($s^0 > 0$) associated with the non-negativity constraints $x \geq 0$.
- `iter_max`: `int`. This sets the maximum number of iterations.
- `epsilon_feas`: `float`. This parameter defines the tolerance for feasibility of the primal and dual solutions [6]. The algorithm checks if the normalized primal residual $||Ax - b||/(1 + ||b||) \leq \epsilon_{feas}$ and the normalized dual residual $||A^T\lambda + s - c||/(1 + ||c||) \leq \epsilon_{feas}$.
- `epsilon_opt`: `float`. This parameter sets the tolerance for optimality, based on average complementarity is $\mu = x^T s/n$, to check $\mu \leq \epsilon_{opt}$ alongside the previous checks.
- `rho`: `float` ($0 < \rho < 1$). This is the step length reduction factor. Typically set to $\rho \in [0.95, 0.99995]$.
- `min_step`: `float`. This parameter sets the minimum allowed value for x or s components. This helps maintain the interior point condition ($x > 0, s > 0$) and prevents numerical issues.
- `regularization`: `float`. This introduces regularization (a small positive value) to the diagonal of matrices to enhance numerical stability.
- `method`: `str` ("predictor-corrector" or "standard"). This selects the algorithm variant:
  - `"predictor-corrector"`: Implements Mehrotra's predictor-corrector method.
  - `"standard"`: Uses a primal-dual path-following method with a standard Newton direction.
- `solver`: `str` ("normal", "augmented", or "kkt"). This chooses the linear system solver:
  - `"normal"`: Solves the normal equations (System 3 in the slides of the course).
  - `"augmented"`: Solves the augmented system (System 2 in the slides of the course
  - `"kkt"`: Solves the full Karush-Kuhn-Tucker (KKT) system (System 1 in the slides of the course).

- `sigma_standard`: `float` ($0 < \sigma < 1$). This is the fixed centering parameter (σ) used only when `method` is set to `"standard"`.
- `verbose`: `bool`. If `True`, the algorithm will print iteration details.

The class is then structured into different methods which orchestrated result in the same logic flow of the algorithm as explained in class. We will individually discuss each block in more detail.

3.2 Residual and Gap Calculation

The `_calculate_residuals_and_gap` function computes the following measures of how close the current iterates (x, λ, s) are to satisfying the optimality conditions which will later be used by `_check_termination` to asses whether to stop the algorithm or not.

- **Dual Residual (r_c)**: Calculated as `self.A.T @ self.lambda_ + self.s - self.c`. This measures dual feasibility.
- **Primal Residual (r_b)**: Calculated as `self.A @ self.x - self.b`, which is $\mathbf{r_b} = \mathbf{Ax} - \mathbf{b}$. This measures primal feasibility.
- **Duality Gap (gap)**: Computed as `self.x @ self.s`. It's also ensured to be non-negative with `max(0.0, gap)`. This represents the difference between primal and dual objective values.
- **Average Complementarity (μ)**: Calculated as `gap / self.n if self.n > 0 else 0.0`.

3.3 Termination Checks

The `_check_termination` method checks if the IPM should stop based on feasibility and optimality criteria by retrieving the metrics computed by the last function. It assesses three main conditions:

- **Primal Feasibility**: Checks if the normalized primal residual is below the feasibility tolerance (`self.epsilon_feas`). The condition is $\frac{||r_b||}{(1+||b||)} \leq \epsilon_{feas}$, implemented in the code as:

  ```
  rb_norm / (1 + norm_b) <= self.epsilon_feas
  ```

- **Dual Feasibility**: Checks if the normalized dual residual is below `self.epsilon_feas`. The condition is $\frac{||r_c||}{(1+||c||)} \leq \epsilon_{feas}$, implemented as:

  ```
  rc_norm / (1 + norm_c) <= self.epsilon_feas
  ```

- **Optimality**: Checks if the average complementarity (`mu = gap  self.n`) is below the optimality tolerance (`self.epsilon_opt`). The condition is $\mu \leq \epsilon_{opt}$, implemented as:

  ```
  gap / self.n <= self.epsilon_opt
  ```

The method returns `True` only if all three conditions are met. In particular, the implementation uses `max(1,np.linalg.norm(self.c))` and `max(1, np.linalg.norm(self.b))` for normalization, which prevents division by zero and provides a lower bound for the normalization factor.

3.4 Step length calculation

The `_calculate_step_length` method determines the **maximum step length (α)** that can be taken along a direction (Δv) such that the variables (v) remain non-negative. It is used to compute the primal step length (α_p) for x and the dual step length (α_d) for s.

The method finds the largest $\alpha \geq 0$ such that `current_v + alpha * delta_v >= 0`, where v represents any of the variables. It focuses on components of `delta_v` that are significantly negative (less than `-self.min_neg_delta`) and calculates the ratio `-current_v[index] / delta_v[index]` for these

components. The `alpha_max` is then the minimum of these positive ratios and 1.0.

Observe that we use `self.min_neg_delta` to restrict the values of `delta_v` being significantly below zero, and therefore the method avoids being overly restrictive due to very small negative perturbations in the direction, adding robustness to the step length calculation.

3.5 Direction Computation

The `_solve_direction` method in the IPM implementation is responsible for calculating the **search direction** $(\Delta x, \Delta\lambda, \Delta s)$ at each iteration of the algorithm. This direction is obtained by solving a **system of linear equations** that arises from applying Newton's method to the conditions of the perturbed Karush-Kuhn-Tucker (KKT) system. Recall that applying Newton's method to the previously defined nonlinear system leads to a linear system for the search direction $(\Delta x, \Delta\lambda, \Delta s)$:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = - \begin{bmatrix} rc \\ rb \\ XSe - \sigma\mu e \end{bmatrix}$$

where $rc = A^T\lambda + s - c$ is the dual residual and $rb = Ax - b$ is the primal residual [9].

The `_solve_direction` method in the implementation takes these residuals rc and rb as inputs, along with a `rhs_comp` term, which corresponds to the last block of the right-hand side. Depending on the chosen method and the iteration, `rhs_comp` is $-XSe$ (for the predictor step where $\sigma = 0$) or $-XSe+\sigma\mu e$.

The implementation provides different solvers for this Newton system, selectable via the `self.solver` parameter:

- **Normal Equations (System 3):** This approach aims to solve a reduced system involving $(A\Theta A^T)\Delta\lambda$. Θ is related to X and S ($\Theta = XS^{-1}$). After solving for $\Delta\lambda$, Δs and Δx can be found through back-substitution. The system $A\Theta A^T$ is symmetric and positive definite (if A has full row rank) and can be solved using Cholesky factorization. The implementation uses sparse Cholesky factorization (`sksparse.cholmod`) and might include AMD ordering to reduce fill-in.
- **Augmented System: (System 2)** This method directly solves the $(n+m) \times (n+m)$ indefinite system :
$$\begin{bmatrix} -\Theta^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \end{bmatrix} = - \begin{bmatrix} -\Theta^{-1}(-rc + X^{-1}rxs) \\ rb \end{bmatrix}$$
where $rxs = XSe - \sigma\mu e$. The implementation uses SciPy's `spla.spsolve` which can handle such indefinite systems. This might be preferred when the normal equations system becomes dense.
- **Full KKT System: (System 1)** This method directly solves the $(2n+m) \times (2n+m)$ system:
$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = - \begin{bmatrix} rc \\ rb \\ XSe - \sigma\mu e \end{bmatrix}$$
The implementation again uses `spla.spsolve` for this potentially non-symmetric and indefinite system.

Upon testing the code, several regularization techniques have been applied to try to improve the numerical stability of the program. For instance, during the construction of the augmented KKT matrix (`KKT_aug`), regularization terms are added to the diagonal blocks:

- **Regularized (1,1) block ($-\Theta^{-1}$):** A term `-sp.identity(n) * delta_p` is subtracted from the $-\Theta^{-1}$ block. Here, `delta_p` is a small positive value ($1e-8$ by default), and I is the identity matrix. This adds a small negative value to the diagonal elements: $-\Theta^{-1} \coloneqq -\Theta^{-1} - \delta_p I$.
- **Regularized (2,2) block (0):** A term `-sp.identity(m) * delta_d` is used in place of the zero block. Here, `delta_d` is another small positive value ($1e-12$ by default). This adds a small negative value to the diagonal elements of what would otherwise be a zero block: $\mathbf{0} \coloneqq -\delta_d I$

These regularization terms, `delta_p` and `delta_d`, are intended to prevent the matrices from becoming singular or ill-conditioned, which can lead to numerical instability during the solution process. It's worth noting that 3 iterations of iterative refinement are performed as well after solving the linear system to maintain stability.

For the normal equations solver, the method relies on `sksparse.cholmod` for a sparse Cholesky factorization. Cholesky factorization is numerically stable even without explicit row/column pivoting, provided that all diagonal pivots are greater than zero and the matrix symmetry is preserved. This stability avoids the necessity for explicit regularization in the same manner as applied to the previous system.

3.6 Main Iteration Loop

The `solve` method forms the central iterative process of the primal-dual interior point method implemented. This method continues to execute until predefined termination criteria are met or a maximum number of iterations is reached.

At the beginning of each iteration k, the implementation ensures that the primal variables x and the dual slack variables s remain strictly positive by taking their maximum with a small positive value `self.min_step`. Next, the method calculates the primal residual $r_b = Ax - b$, the dual residual $r_c = A^T\lambda + s - c$, the duality gap $gap = c^Tx - b^T\lambda$, and the average complementarity measure $\mu = x^Ts/n$ using the `_calculate_residuals_and_gap()` method. If the `verbose` flag is set, these values, along with the primal objective c^Tx and dual objective $b^T\lambda$, are printed to monitor the algorithm's progress.

The implementation then checks for termination using the `_check_termination()` method. If the criteria are satisfied, the algorithm's status is updated to "Optimal solution found." and the loop terminates. If termination conditions are not met, the `_calculate_search_direction()` method is invoked to compute the Newton direction $(\Delta x, \Delta\lambda, \Delta s)$. The specific calculation of this direction depends on the chosen method (`predictor-corrector` or `standard`) and solver (`normal`, `augmented`, or `kkt`). The implementation includes a check for non-finite values in the calculated search directions, raising a `ValueError` if any are detected.

When the implementation uses the predictor-corrector method, each iteration involves a predictor step followed by a corrector step. In the predictor step (affine scaling), an affine-scaling direction is computed by solving the system with the right-hand side `rhs_comp_aff = -self.x * self.s`. This step aims to reduce the complementarity gap aggressively. Subsequently, the maximum primal (α_{p_aff}) and dual (α_{d_aff}) step lengths that maintain positivity along this direction are calculated. An estimate of the duality gap after the affine step, μ_{aff}, is then computed. The centering parameter σ is then adaptively updated based on the ratio $(\mu_{aff}/\mu)^3$ and forced to remain within $[0.01, 0.99]$. In the corrector step, the combined centering and corrector direction $(\Delta x, \Delta\lambda, \Delta s)$ is computed using the right-hand side `rhs_comp_cc = -self.x * self.s + sigma * mu * e - delta_x_aff * delta_s_aff`. This direction incorporates a term to move towards the central path ($\sigma\mu e$) and a second-order correction term ($-\Delta x_{aff}\Delta s_{aff}$).

If the `self.method` is set to `"standard"`, the implementation directly computes the search direction using a fixed centering parameter $\sigma =$ `self.sigma_standard` . The right-hand side for the Newton system in this case is `rhs_comp = -self.x * self.s + sigma * mu * e`.

Following the computation of the search direction, the `_calculate_step_length()` method determines the primal step length α_p and the dual step length α_d. The step length reduction factor ρ is applied to both α_p and α_d, and then the primal and dual variables are then updated taking a step in the computed direction. Our implementation also includes a check for stagnation, where if the step lengths become very small for several consecutive iterations, the algorithm may terminate with a "Stagnation detected" status. If the loop completes `iter_max` iterations without meeting the termination criteria, the status is set to "Maximum number of iterations reached.". Finally, if `self.verbose` is true and the primal variables x are finite, the final primal objective value is printed.

4 Results and Conclusion

4.1 Testing framework

To present our solutions, we propose a testing framework which employs **pytest** to evaluate our implementation. Our primary goal is to assess the custom IPM solver's correctness and robustness by comparing its results against **AMPL/CPLEX**. The testing process involves loading linear programming (LP) problems from the Netlib collection. The framework parameterizes tests across different Netlib problems, IPM variants ("standard" and "predictor-corrector"), and linear system solvers ("normal", "augmented", or "kkt"). For each test case, the custom **PrimalDualPathFollowing** class is instantiated and its `solve()` method is executed. Simultaneously, the same LP problem is solved using **AMPL/CPLEX**. The framework then uses assertion statements to compare the objective values and solution statuses reported by the custom IPM and AMPL/CPLEX. This comparative approach validates the IPM implementation against a trusted solver across various LP problems and configurations.

4.2 Netlib results

The set of Netlib instances contain problems of different sizes and number of non-zero elements. We go from small instances like lp_afiro to large problems like lp_pilot87. Our selection is the following:

Problem Name	Rows	Columns	Non-Zero Elements
lp_israel	174	316	2,443
lp_afiro	27	51	102
lp_stocfor1	117	165	501
lpi_galenet	8	14	22
lp_sierra	1,227	2,735	8,001
lp_stair	356	614	4,003
lp_pilot87	2,030	6,680	74,949

Table 1: Netlib problem dimensionality

Each test case will run with the same parameters, which are the following:

x^0	s^0	λ^0	`iter_max`	ϵ_{feas}	ϵ_{opt}	ρ	`sigma_standard`	`min_step`
$100 \cdot \mathbf{1}_n$	$100 \cdot \mathbf{1}_n$	$\mathbf{0}_m$	500	10^{-6}	10^{-6}	0.9999	0.1	10^{-10}

Table 2: Netlib problem parameters

The following results correspond to only the full KKT system, given that System 2 and System 3 were not able to be succesfully implemented due to numerical instability (contradictory). Observe how in most cases, the Predictor-Corrector method achieves the optimal solution in less iterations than the standard. Nonetheless, the largest instance lp_pilot87 led to divergence in Mehrotra but not in the Newton method.

Table 3: Netlib instances results: KKT system (System 1)

	Standard				Predictor-Corrector				CPLEX
Problem	**Iter**	**Primal Obj**	μ	**Passed?**	**Iter**	**Primal Obj**	μ	**Passed?**	**Objective**
lp_afiro	16	−4.65e+02	2.19e−07	Pass	10	−4.65e+02	1.39e−07	Pass	−4.65e+02
lp_israel	147	−8.97e+05	9.75e−07	Pass	50	−8.97e+05	6.56e−07	Pass	−8.97e+05
lpi_galenet	12	0.00e+00	9.73e−07	Pass	8	0.00e+00	2.39e−08	Pass	0.00e+00
lp_stocfor1	23	−4.11e+04	3.98e−07	Pass	19	−4.11e+04	3.43e−07	Pass	−4.11e+04
lp_sierra	66	1.46e+07	3.44e−07	Pass	39	Fail	Fail	Fail	1.46e+07
lp_stair	48	−4.29e+02	3.60e−07	Pass	500	Fail(Iter)	Fail(Iter)	Fail	−4.29e+02
lp_sctap2	29	1.72e+03	1.02e−07	Pass	17	1.72e+03	6.36e−08	Pass	1.72e+03
lp_pilot87	108	1.03e+02	5.11e−07	Pass	500	Fail(Iter)	Fail(Iter)	Fail	1.03e+02