FINAL REPORT

Jiasong Dong & Yunhan Huang

May 1, 2023

Abstract

Linear programming stands as a remarkable success story in the field of optimization, with applications across various problem-solving and decision-making domains. The Simplex method, introduced by George Dantzig, has been a popular technique for solving linear programming problems. However, the primal-dual method has shown better performance and convergence for many large, complex problems.

Although Simplex codes are faster for small to medium-sized problems, interior point primal-dual methods offer significant speed improvements for large-scale problems. Consequently, we implemented the Primal-Dual Interior Point (PDIP) method to address linear programming problems.

Throughout the PDIP solver's implementation, we faced numerous challenges that demanded critical thinking, research, and collaboration. By overcoming these obstacles, we developed features that enhanced the solver's ability to solve linear programming problems.

Our solver may not be as robust as expected, but it delivers remarkable speed when addressing linear programming problems. This advantage is particularly valuable for applications that require efficient solutions. Experimental results indicate that there is still room for improvement in our solver, and we plan to continue refining it to achieve better performance and reliability.

In summary, the Primal-Dual Interior Point method has demonstrated its value as an efficient approach to solving large-scale linear programming problems. While our implementation has its flaws, the solver's fast performance and potential for further optimization make it a promising tool for future work in this area.

Primal-Dual method

The standard form of the linear programming problem is a common way to explain and examine algorithms. Here is the standard form (primal form)

$$\min c^T x \text{ subject to } Ax = b, x \geq 0$$

where c and x are vectors in $\mathbb{R}^n$, b is a vector in $\mathbb{R}^m$, and A is an $m \times n$ matrix. If x satisfies the constraints $Ax = b, x \geq 0$, we call it a feasible point. If there is the set of all feasible points, we call it feasible set.

By introducing slack variables and artificial variables, we can construct a dual form

$$\max b^T \lambda \text{ subject to } A^T \lambda + s = c, s \geq 0$$

where λ is a vector in $\mathbb{R}^m$ and s is a vector in $\mathbb{R}^n$.

Given any feasible vectors x for primal form and (λ, s) for dual form, we have

$$b^T \lambda \leq c^T x$$

The appendix theorems A.1 and A.2 from the book states that:
The vector $x^* \in \mathbb{R}^n$ is a solution of primal form if and only if there exists vectors $s^* \in \mathbb{R}^n$ and $\lambda^* \in \mathbb{R}^m$ for which the following conditions hold for $(x, \lambda, s) = (x^*, \lambda^*, s^*)$:

$$A^T\lambda + s = c \tag{1}$$
$$Ax = b \tag{2}$$
$$x_i s_i = 0, i = 1, 2, ..., n \tag{3}$$
$$x \geq 0 \tag{4}$$
$$s \geq 0 \tag{5}$$

This is also known as Karush-Kuhn-Tucker(KKT) conditions.

Primal-Dual restates the KKT conditions by mapping F from $\mathbb{R}^{2n+m}$ to $\mathbb{R}^{2n+m}$:

$$F(x, \lambda, s) = \begin{bmatrix} A^T + s - c \\ Ax - b \\ XSe \end{bmatrix} = 0,$$
$$(x, s) \geq 0,$$

where

$$X = \text{diag}(x_1, x_2, \ldots, x_n)$$
$$S = \text{diag}(s_1, s_2, \ldots, s_n)$$
$$e = (1, 1, ..., 1)^T$$

Newton Step Equation

By creating a linear approximation of F centered at the present point, Newton's method calculates the search direction $(\Delta x, \Delta A, \Delta s)$ through solving the following linear equations:

$$J(x, \lambda, s) \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = -F(x, \lambda, s)$$

The search direction procedure has its origins in Newton's method for the nonlinear equations. If the current point is strictly feasible, the Newton step equation is

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

A full step along this direction usually is not permissible because it would violate the bound $(\mathbf{x}, \mathbf{s}) \geq 0$. To avoid, we perform a line search along the Newton direction so that the new iterate is

$$\mathbf{x}, \lambda, \mathbf{s} + \alpha(\Delta\mathbf{x}, \Delta\lambda, \Delta\mathbf{s})$$

for some line search parameter $\alpha \in (0, 1]$

Primal-Dual Framework

Based on above concepts, we have a framework for Primal-Dual.

Algorithm 1: PD Framework

Given the starting points $(x^0, \lambda^0, s^0) \in \mathcal{F}^0$
for *k=0,1,2,...* **do**
  solve
  $$\begin{bmatrix} 0 & \boldsymbol{A}^T & \boldsymbol{I} \\ \boldsymbol{A} & 0 & 0 \\ \boldsymbol{S}^k & 0 & \boldsymbol{X}^k \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \lambda \\ \Delta \mathbf{s} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\boldsymbol{X}^k \boldsymbol{S}^k \mathbf{e} + \sigma_k \mu_k \mathbf{e} \end{bmatrix}$$
  where $\sigma_k \in [0, 1]$ and $\mu_k = (\mathbf{x}^k)^T \mathbf{s}^k / n$
  set
  $(\mathbf{x}^{k+1}, \lambda^{k+1}, \mathbf{s}^{k+1}) \leftarrow (\mathbf{s}^k, \lambda^k, \mathbf{s}^k) + \alpha_k (\Delta \mathbf{x}^k, \Delta \lambda^k, \Delta \mathbf{s}^k)$
  choosing α_k so that $(\mathbf{x}^{k+1}, \mathbf{s}k + 1) > 0$
end for

Implementation

Presolving

Presolving is a crucial step in solving linear programming (LP) problems because it simplifies the problem, reduces its size, and improves the efficiency of the solving process.
In presolving step, there are several things that need to be checked because our sparse matrix A has a large scale dimensions. Our LP solver checks zero rows and columns.

- *Zero columns in A*: When the column A_{*i} is zero, we need to eliminate the zero column. The reason is that no matter what value x_i is, it has no effect on the product Ax. After removing the zero columns, we need to adjust the objective function coefficients and bounds. The constraint vector b is going to be adjusted as well.
- *Zero rows in A*: When the row $A)i*$ is zero, we just need to eliminate the zero row. If all of the entries of A are zero, and if the entry of b is zero, then x must be zero. Otherwise, $Ax = b$ is infeasible.

Although other presolve techniques, for instances, merging duplicated rows and columns, row singleton, and forced row will help to downsize the volume of the problem and theoretically increase the solving speed, we found those steps will increase the time cost significantly in practice since multiple nested loops will be used in those operations. Therefore, we choose not to implement those parts in favor of the speed.

There are still several features that we didn't implement them.

- *Duplicate rows and columns in A*: If two columns of matrix A are scalar multiples of each other, we can merge them into a single primal variable and reduce the dimension of A by 1. Likewise, if A has repeated rows, we can combine two rows of A and reduce the number of rows by 1.
- *Row singleton*: If the ith row of matrix A has only one nonzero element, A_{ij}, then we can directly solve for x_j by setting it equal to b_i/A_{ij}. After that, we can remove row i and column j from the problem, which reduces both the number of rows and columns by 1.
- *Forced row*: If a constraint and some bounds on a set of variables imply that these variables must take on fixed values, then we can remove these variables and the corresponding row of matrix A from the linear program.

Convert To Standard Form

The LP problem loaded in from LPNetLib is not guaranteed to be in standard form. Standard form serves as a crucial role in solving LP problems, it ensures that all problems have the same structure, generalize the problem for the solver, help with the theoretical analysis of the problem and its properties. In short, standard form provides consistency, simplification, stability, and easy analysis for the solver. Before we standardized the input LP problem, we reconstruct the original problem base to handle the stability issue:

1. If a variable is unbounded in both directions (lower and upper bounds are infinite), the variable is split into two non-negative variables (x_i^+ and x_i^-)
2. If a variable has only an upper bound, the function negates the variable
3. If a variable has a lower bound (or both lower and upper bounds), the function shifts the variable by subtracting the lower bound multiplied by its corresponding column from the right-hand side vector P.b

At the end of each condition, we modified the newly constructed matrix A, c, and b if applicable. There are more operation for variables with both lower and upper bounds since they are fully bounded, the function adds slack variables to convert the inequality constraints into equality constraints. It first finds the indices of such variables and initializes a zero matrix of the appropriate size. With the new A, b, and c, the function concatenate an Identity matrix to A and return the standard form of the input LP problem.

Starting Point

The starting point is defined as

$$(x^0, \lambda^0, s^0) = (\tilde{x} + \tilde{\delta}_x e, \tilde{\lambda}, \tilde{s} + \tilde{\delta}_s e)$$

where the scalars $\tilde{\delta}_x$ and $\tilde{\delta}_s$ are calculated to satisfy the two following conditions.

- To ensure similarity in pairwise products $x_i^0 s_i^0$ for all $i = 1, 2, ..., n$, it is important to center the points effectively.
- It is advisable to choose points that are not excessively infeasible, (i.e. the ratio $||(r_b^0, r_c^0)||/\mu_0$ of infeasibility to duality measure should not be too high.)

To determine the starting point, we can calculate the solution of two least-squares equations:
Primal constraint are given by

$$\begin{aligned} \min_x \quad & ||x||^2 \\ \text{subject to} \quad & Ax = b \end{aligned}$$

and dual constraints are given by

$$\begin{aligned} \min_{(\lambda,s)} \quad & ||s||^2 \\ \text{subject to} \quad & A^T\lambda + s = c \end{aligned}$$

To solve above problems, we can use the Lagrangian method to derive the KKT conditions.
The optimal solution are

$$\begin{aligned} \tilde{x} &= A^T(AA^T)^{-1}b \\ \tilde{s} &= c - A^T(AA^T)^{-1}b \end{aligned}$$

Because $\tilde{x}$ and $\tilde{s}$ are not all positive, hence we take $\delta_p = \max(-1.5 * \min_i(\tilde{x}_i), 0)$ and $\delta_d = \max(-1.5 * \min_i \tilde{s}_i, 0)$

Now,

$$\tilde{\delta}_p = \delta_p + 0.5 \times \frac{(\tilde{x} + \delta_p e)^T (\tilde{s} + \delta_d e)}{\Sigma_{i=1}^n (\tilde{s}_i + \delta_d)}$$
$$\tilde{\delta}_d = \delta_d + 0.5 \times \frac{(\tilde{x} + \delta_p e)^T (\tilde{s} + \delta_d e)}{\Sigma_{i=1}^n (\tilde{x}_i + \delta_p)}$$

Finally, we have our starting point function strictly follows the steps described by Mehrotra

$$x^0 = \tilde{x} + \tilde{\delta}_p e$$
$$\lambda^0 = \tilde{\lambda}$$
$$s^0 = \tilde{s} + \tilde{\delta}_d e$$

```
function starting_point(P::IplpProblem, default_cholesky=true)
    # 4 Steps
    # (1) Solving Primal and Dual starting points least squares
    # (2) Calculate delta primal and delta dual
    # (3) Calculate delta primal hat and delta dual hat
    # (4) Get starting point

    symm = P.A*P.A'
    if default_cholesky
        f = get_cholesky_lowtriangle(symm)
    else
        f = get_cholesky_lowtriangle(symm, false)
    end

    # (1)
    # Solve Primal
    d = f\P.b
    x_hat = P.A'*d

    # Solve Dual
    λ_hat = f\(P.A*P.c)
    s_hat = P.c - P.A'*λ_hat

    # (2)
    delta_p = max(-1.5*minimum(x_hat), 0)
    delta_d = max(-1.5*minimum(s_hat), 0)

    n = size(x_hat,1)
    e = ones(n,1)

    # (3)
    x_hat = x_hat + delta_p*e
    s_hat = s_hat + delta_d*e

    numerator = 0.5 * (x_hat'*s_hat)
    delta_p_hat = (numerator / (e'*s_hat))[1]
    delta_d_hat = (numerator / (e'*x_hat))[1]

    # (4)
    x_0 = x_hat + delta_p_hat*e
```

```
    λ_0 = λ_hat
    s_0 = s_hat + delta_d_hat*e

    return (x=x_0, λ=λ_0, s=s_0, f=f)
end
```

Cholesky Factorization

Let M denote $AD^2A^T\Delta\lambda = -r_b + A(-S^{-1}Xr_c + S^{-1}r_{xs})$ after a possible symmetric reordering of its rows and columns, $M = P(AD^2A^T)P^T$, where P is an $m \times m$ permutation matrix. $M = LL^T$ where L is a lower triangular matrix with positive diagonal elements. By the pseudocode of Cholesky algorithm, we can write the update step as

$$M_{i+1:m,i+1:m} \leftarrow M_{i+1:m,i+1:m} - M_{i+1:m,i}M^T_{i+1:m,i}/M_{ii}$$

where M_{ii} is the pivot element. Given the factorization $M = LL^T$, we can solve the linear system $Mz = r$ by using two triangular substitutions:

$$\text{solve } Ly = r \text{ to find } y$$

$$\text{solve } L^Tz = y \text{ to find } z$$

We also take the modified version of Cholesky factorization with pivoting and reordering, diagonal perturbation, diagonal scaling, and separate L and D factors, which theoretically offers the advantage:

- Improving stability by enforcing a minimum threshold on the diagonal elements and scales the off-diagonal elements to ensure better stability
- Maintain positive semi-definite matrices in order to handle wider range of problems
- Uses Approximate Minimum Degree (AMD) ordering method to reorder the input matrix, which improves the sparsity structure and helps preserve the sparsity during the factorization
- Provides flexibility to outputs by separate L and D factors

the code is shown below:

```
function modified_cholesky(A::SparseMatrixCSC{Float64}, delta::Real, beta::Real)
    @assert (n=size(A, 1)) == size(A, 2) "Input matrix must be a square matrix!"
    reorder = amd(A)
    L = A[reorder,reorder]
    d = ones(n)
    D = Diagonal(d)
    for i = 1:n-1
        theta = maximum(abs.(L[i+1:n, i]))
        d[i] = max(abs(L[i, i]), (theta / beta)^2, delta)
        L[i:n, i] ./= d[i]
        L[i, i] = 1.0
        L[i+1:n, i+1:n] .-= d[i] * (L[i+1:n, i] * L[i+1:n, i]')
    end

    d[n] = max(abs(L[n, n]), delta)
    L[n, n] = 1.0

    return (L=LowerTriangular(L), D=D, M=L, O=reorder)
end
```

Delta (δ) and beta (β) values in the modified version is use to control the numerical stability of the factorization process which δ is a small positive value that ensures the diagonal elements of the resulting factor are greater than or equal to a given threshold, and β is a parameter used to bound the off-diagonal elements of the resulting factor. The choice of these 2 values should be decide case-by-case. We choose them base off the matrix A in the LP problem:

- β = the square root of the ratio of the largest singular value to the smallest singular value of A, which is a value that reflects the degree of ill-conditioning of the input matrix
- δ = product of the machine epsilon and the Frobenius norm, which ensures that the diagonal perturbation is small enough to not significantly change the matrix properties but large enough to improve numerical stability for the specific problem

```
function select_modified_cholesky_parameters(A::SparseMatrixCSC{Float64})
    beta = sqrt(cond(Matrix(A)))
    delta = eps(Float64) * norm(Matrix(A))
    return delta, beta
end
```

Similar the presolving part, with all these benefits brought the modified Cholesky factorization algorithm, it actually slows our entire algorithm down in practically. We tested the code using “lp_ganges", the algorithm takes 21.186804 seconds to finsh with modified version, and 1.016800 seconds with original version. Hence, we provided a function that users can choose which version they prefer.

```
function get_cholesky_lowtriangle(matrix, default=true)
    if default
        return cholesky(matrix)
    else
        delta, beta = select_modified_cholesky_parameters(matrix)
        F = modified_cholesky(matrix, delta, beta)
        return F.L
    end
end
```

By default, the solver runs with original Cholesky factorization for the sake of speed.

Step Equation

Here we just put three forms of the step equations.
The unreduced form is:

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

, where $r_{xs} = XSe - \sigma\mu e$ for the generic primal-dual step. The calculation of the residual values corresponds to the following lines in our code:

```
rb  = P.A*x_0-P.b
rc  = P.A'*λ_0+s_0-P.c
rxs = x_0.*s_0
```

where x_0, λ_0, and s_0 are initialized to the starting points, and update after step out in the current iteration. By eliminating Δs and using the notation $D = S^{-1/2}X^{1/2}$, we get the augmented system form:

$$\begin{bmatrix} 0 & A \\ A^T & -D^{-2} \end{bmatrix} \begin{bmatrix} \Delta\lambda \\ \Delta x \end{bmatrix} = \begin{bmatrix} -r_b \\ -r_c + X^{-1}r_{xs} \end{bmatrix}$$

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

Finally, by eliminating Δx, we get the most compact of the three forms:

$$AD^2A^T\Delta\lambda = -r_b + A(-S^{-1}Xr_c + S^{-1}r_{xs})$$

$$\Delta s = -r_c - A^T\Delta\lambda$$

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

Our code perform the following to solve the augmented system form

```
function solve_linear_system(A,x,s,rb,rc,rxs)
    KKT = build_kkt_matrix(A, x, s)
    f = lu(KKT)
    m = length(rb)
    n = length(rc)
    b = Array{Float64}([-rb; -rc; -rxs])
    b = f\b
    dlam = b[1:m]
    dx = b[1+m:m+n]
    ds = b[1+m+n:m+2*n]
    return dlam,dx,ds
end
```

where we used LU Factorization since it is can apply to any square matrix whether the matrix is symmetric or not, and whether it is positive definite or not. In other words, it is more generic and able to handle wider range of cases.

Mehrotra's Algorithm

The algorithm developed by Mehrotra produces a series of unfeasible iterations (x^k, λ^k, s^k) in which $(x^k, s^k) > 0$. At each iteration, the search direction is composed of three components:

- A "predictor" direction using affine-scaling is computed at each iteration, which is the pure Newton direction for the function F(x, A, s) defined by

$$F(x, \lambda, s) = \begin{bmatrix} A^T + s - c \\ Ax - b \\ XSe \end{bmatrix} = 0,$$
$$(x, s) \geq 0,$$

- The algorithm includes a centering term that is controlled by the adaptively selected centering parameter σ, which determines its magnitude.
- A "corrector" direction aims to counterbalance some of the nonlinearity present in the predictor direction.

Consider the component 1, we can solve the following system to find the affine-scaling direction by building KKT matrix.

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

where the residuals are defined as before by

$$r_b = Ax - b$$
$$r_c = A^T\lambda + s - c$$

Now that this system can be solved by setting $\sigma = 0$.

Since we are given the starting point, we can first solve for $(\Delta x^{aff}, \Delta\lambda^{aff}, \Delta s^{aff})$
Now, we need to find the step lengths to the boundary along this direction.
We calculate the α_{aff}^{pri} and α_{aff}^{dual} where these values need to be arg max by the following equations

$$\alpha_{aff}^{pri} = \text{ arg max } \{\alpha \in [0,1] | x + \alpha\Delta x^{aff} \geq 0\}$$
$$\alpha_{aff}^{dual} = \text{ arg max } \{\alpha \in [0,1] | s + \alpha\Delta s^{aff} \geq 0\}$$

The following is the code for finding α for both primal and dual.

```
function alpha_max(x,dx,hi=1.0)
    alpha = hi
    for i = 1:length(x)
        alpha = dx[i] < 0 ? min(alpha, -x[i] / dx[i]) : alpha
    end
    if alpha < 0
        alpha = Inf
    end
    alpha = min(alpha,hi)
    return alpha
end
```

To evaluate the effectiveness of the affine-scaling direction, the algorithm uses a metric called μ_{aff}, which represents the theoretical value of μ that would be obtained if a full step were taken to the boundary. Mathematically, this is expressed as follows:

$$\mu_{aff} = (x + \alpha_{aff}^{pri}\Delta x^{aff})^T(s + \alpha_{aff}^{dual}\Delta s^{aff})/n$$

Given that

$$\mu = x^s/n$$

and we have μ_{aff}, if $\mu_{aff} \ll \mu$, the affine-scaling direction is a good search direction so that setting the centering parameter σ close to zero. If μ_{aff} is just a little smaller than μ, then setting σ close to 1 is a better choice. Mehrotra suggests that

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

Now, we move on to the "centering-corrector" step. The combinned centering-corrector step $(\Delta x^{cc}, \Delta\lambda^{cc}, \Delta s^{cc})$ can be obtained by solving the linear system below:

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

The coefficients of the matrix are already obtained when we solve the started linear system, hence there is no need to compute them again. It is also possible to solve the above equation by performing a single back-substitution operation, which is computationally less expensive than other methods. To improve the efficiency and robustness for our solver, we need to modify the step length, the centering parameter, and the search direction.

In our solver, we implemented the γ so that we can choose how large the γ is. Typically, the value of the parameter γ is 0.01. We checks that the problem size is less than 100 or less than 1000. Otherwise, the value of γ will be

0.1. By doing so, our solver will be more efficient and faster.
By using γ, we have our new α to calculate. Firstly to calculate μ_+:

$$\mu_+ = (x^k + \alpha_{max}^{prim}\Delta x^k)^T(s^k + \alpha_{max}^{dual}\Delta s^k)/n$$

for the particular index i for which $x_i^k + \alpha_{max}^{prim}\Delta x_i^k = 0$, define f^{prim} by

$$(x_i^k + f^{prim}\alpha_{max}^{prim}\Delta x_i^k)^T(s_i^k + \alpha_{max}^{dual}\Delta s_i^k) = \gamma_f\mu_+$$

for the particular index i for which $s_i^k + \alpha_{max}^{dual}\Delta s_i^k = 0$, define f^{dual} by

$$(x_i^k + \alpha_{max}^{prim}\Delta x_i^k)^T(s_i^k + f^{prim}\alpha_{max}^{dual}\Delta s_i^k) = \gamma_f\mu_+$$

Finally, we reach our α_k^{prim} and α_k^{dual} as below:

$$\alpha_k^{prim} = \max(1-\gamma_f, f^{prim})\alpha_{max}^{prim}$$
$$\alpha_k^{dual} = \max(1-\gamma_f, f^{dual})\alpha_{max}^{dual}$$

Here is our code for choosing the γ value:

```
function select_gamma(P::IplpProblem)
    m, n = size(P.A)
    problem_size = max(m, n)

    if problem_size <= 100
        gamma = 0.01
    elseif problem_size <= 1000
        gamma = 0.05
    else
        gamma = 0.1
    end

    return gamma
end
```

The algorithm 2 below is our predictor-corrector algorithm, which is the main functional algorithm for our solver. We strictly follow the pseudocode.

Termination

Primal-dual Interior-point method never find and exact solution of the Linear program, so we need a condition for termination.

We will have a small positive number, $\epsilon = 10^{-8}$, which we will consider it as tolerance. When we satisfy these three conditions, the solver will be terminated.

$$\frac{||r_b||}{1+||b||} = \frac{||Ax-b||}{1+||b||} \leq \epsilon$$
$$\frac{||r_c||}{1+||c||} = \frac{||A^T\lambda + s - c||}{1+||c||} \leq \epsilon$$
$$\frac{|c^Tx - b^Ty|}{1+|c^Tx|} \leq \epsilon$$

Here are the code implementations:

```
r1 = norm(P.A*x1-P.b)/(1+norm(P.b))
r2 = norm(P.A'*lambda1+s1-P.c)/(1+norm(P.c))
r3 = abs(dot(P.c,x1)-dot(P.b,λ1))/(1+abs(dot(P.c,x1)))
if r1 < tol && r2 < tol && r3 < tol
    flag = true
    break
end
```

When we handle the tolerance, there are some problems that we cannot use 1e-8 to handle. Otherwise, it will be infeasible. Therefore, most of the problems can be solved by 1e-8 tolerance. There are a few specific case that we need to change the tolerance to 1e-4 so that can be solved quickly.

Algorithm 2: MPC

Given the starting points (x^0, λ^0, s^0) with $(x^0, s^0) > 0$

for *k=0,1,2,...* **do**

set $(x, \lambda, s) = (x^k, \lambda^k, s^k)$ and solve the linear system and find the affine-scaling direction

Then calculate the α_{aff}^{prim}, α_{aff}^{dual}, and μ_{aff}

$$\alpha_{aff}^{pri} = \text{ arg max } \{\alpha \in [0,1] | x^k + \alpha \Delta x^{aff} \geq 0\}$$
$$\alpha_{aff}^{dual} = \text{ arg max } \{\alpha \in [0,1] | s^k + \alpha \Delta s^{aff} \geq 0\}$$
$$\mu_{aff} = (x + \alpha_{aff}^{pri} \Delta x^{aff})^T (s + \alpha_{aff}^{dual} \Delta s^{aff})/n$$

Set the centering parameter to

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

solve the new linear system for $(\Delta x^{cc}, \Delta \lambda^{cc}, \Delta s^{cc})$ Compute the search direction and step to boundary from
$(\Delta x^k, \Delta \lambda^k, \Delta s^k) = (\Delta x^{aff}, \Delta \lambda^{aff}, \Delta s^{aff}) + (\Delta x^{cc}, \Delta \lambda^{cc}, \Delta s^{cc})$

$$\alpha_{max}^{pri} = \text{ arg max } \{\alpha \geq 0 | x^k + \alpha \Delta x^k \geq 0\}$$
$$\alpha_{max}^{dual} = \text{ arg max } \{\alpha \geq 0 | s^k + \alpha \Delta s^k \geq 0\}$$

Set

$$\alpha_k^{prim} = \min(0.99 * \alpha_{max}^{prim}, 1), \alpha_k^{dual} = \min(0.99 * \alpha_{max}^{dual}, 1)$$

Set

$$x^{k+1} = x^k + \alpha_k^{prim} \Delta x^k$$
$$(\lambda^{k+1}, s^{k+1}) = (\lambda^k, s^k) + \alpha_k^{dual} (\Delta \lambda^k, \Delta s^k)$$

end for

Algorithm 3: Primal-Dual LP Solver

Input: IplpProblem(c::Vector, A::SparseMatrixCSC, b::Vector, lo::Vector, hi::Vector)

Output: IplpSolution(x::Vector, flag::Bool, cs::Vector, As::SparseMatrixCSC, bs::Vector, xs::Vector, lam::Vector, s::Vector)

Presolving step: Remove the zero rows and zero columns

Convert the problem to standard form

▷ returns the standardized IplpProblems

Create a new vector c, and checks whether the target is infeasible or not

▷ print the feasibility of the problem

Run MPC algorithm (predictor corrector)

▷ In this function, we find the starting point and select the gamma value

▷ returns x_1, λ_1, s_1, flag

Finding the optimal value with the vector c and solution of x.

Return IplpSolution

Evaluation

The correct result for the feasible LP problems are obtain from feasible problem summary and the infeasible problems are found at infeasible problem summary (these dataset has the prefix of 'lpi' instead of 'lp'), both are provided by LPNetLib.

Required Tests

There are total of 9 required tests from course webpage, the result is shown below (*ov* denotes optimal values):

test name	Feasible	$ov_{correct}$	ov_{pc}	$Iterations_{correct}$	$Iterations_{pc}$	Time(s)
afiro	✓	-4.648e2	-4.648e2	6	7	5.2235
brandy	✓	1.519e3	1.519e3	292	19	5.5057
fit1d*	✓	-9.146e3	✗	✗	16	✗
adlittle	✓	2.255e5	2.255e5	123	10	5.2198
agg	✓	-3.599e7	-3.599e7	✗	31	6.5057
ganges*	✓	-1.096e5	-2.473e5	✗	20	8.6061
stocfor1	✓	-4.113e4	-4.113e4	✗	17	5.4870
25fv47	✓	5.502e3	5.502e3	✗	26	8.9194
chemcom	✗	✗	✗	✗	>100	✗

Table 1: Required Tests Results

Table 1 shows a directly comparison between our solver's result and the correct result provided by LPNetLib:

1. Name: The name of the data sets ("LPnetlib/[name]")
2. Star sign (∗)next to name indicates either our result is inconsistent with the answer or the data set causes our solver ran into errors
3. Feasible: This information is provided by LPNetLib and indicated by the prefix ("lpi" is infeasible, "lp" is feasible)
4. $ov_{correct}$ and ov_{pc} are the optimal value acquired from LPNetLib webpage and from our solver results
5. $Iterations_{correct}$ and $Iterations_{pc}$ are the interation numbers acquired from LPNetLib webpage and from our solver results
6. Times in seconds is acquired by calling Julia macro '@timed' on our 'iplp()' function, which covers the all solver steps and excluding the library loading or testing time.

***Testing environment**: The above results are computed with Julia 1.8.5 on Intel® Core™ i5-10600KF CPU @ 4.10GHz × 12 and 64GB RAM space in Ubuntus 22.04.2 LTS.

We found that Jupyter notebook sometimes make the execution faster after several trails of tests. To ensure the time is correct and will not be affected(accelerated) by cache left from previous executions, we perform all the testing from terminal by calling 'julia run-test.jl'.

Extra Tests

test name	Feasible	$ov_{correct}$	ov_{pc}	$Iterations_{correct}$	$Iterations_{pc}$	Time(s)
agg2	✓	-2.024e7	-2.024e7	✗	20	5.9173
woodw	✓	1.304	1.304	✗	38	94.7758
beaconfd	✓	3.359e4	3.359e4	38	7	5.4392
standata	✓	1.258e3	1.258e3	✗	20	6.6024
e226	✓	-1.875e1	-1.875e1	570	21	0.7447
grow15*	✓	-1.069e8	-9.878e28	✗	✗	✗
degen2*	✓	-1.435e3	✗	✗	✗	✗
klein2	✗	✗	✗	✗	>100	✗
vol1	✗	✗	✗	✗	>100	✗
bgprtr	✗	✗	✗	✗	>100	✗

Table 2: Extra Tests Results

We performed 10 extra tests, the testing environment is the same as where we perform our required tests. From both sets of testing results, our code handles the infeasibility check pretty well but it is still not able to solve all the feasible LP problems, the reasons we suspected and the potential fix will be discussed in next section.

Discussion

This section will mainly focus on the 2 criteria required by the project handout: 1) Robustness, and 2) Speed. In addition, we discuss the success of our solver, the downside of our solver and some potential solution for these errors.

In the aspect of speed, our LP problem solver is fast even for some large data sets, such as “lp_ganges" (1310×1681 with 7021 non-zero data) and “lp_woodw" (1099×8405 with 37478 non-zero data), both takes less than 2 minutes to find the convergence. Although our solver gives out the wrong optimal value for “lp_ganges", the time is still an advantage since in general the error rate is low. We believe our problem-based gamma selection has contribute to this time, and as mentioned, we choose the original version of Cholesky Factorization instead of the modified version. To confirm this guess, we did a small-scale test to compare the time taken when using modified Cholesky and using original Cholesky, we found that the solver uses modified version of Cholesky usually takes over 1 to 2 minute more for large size LP problems (Table 3), while small difference in time is already shown in small size data sets (sc50a and scfxm1). We only used datasets that can terminate successfully for this test since we want to have a correct elapsed time for the iplp function. However, we keep the code of modified Cholesky in our solver in order users would like to compare the results or use modified version for specific problems.

Names	Modified Time (s)	Original Time (s)	Rows	Columns
lp_woodw	150.4721	96.4271	1,098	8,418
$lp\_stocfor2$	99.5612	10.6302	2,157	3,045
lpi_bgindy	189.4207	59.9231	2,671	10,880
lp_sc50a	5.6698	5.5450	50	78
lp_scfxm1	7.4128	6.0977	330	600

Table 3: Time Comparison Test

On the other hand, the robustness of our solver is not as good as our solver's speed. As shown in Table 1 and Table 2 above, there are four feasible LP problems out of total of 15 feasible problems, the solver either returns the incorrect result

or optimal value or exit unexpectedly with exceptions, we categorized them into X types of issues:

- Exit unexpectedly with Exception threw
  - Singular Exception: “lp_fit1d"
  - PosDef Exception: “lp_degen2"
- Incorrect optimal value returned with large difference: “lp_ganges", “lp_grow15"

Since none of them is reporting the "cannot be solved in max iterations", we conclude that the issues may not be fixed just by simply modify the parameters we use.

Some flaw we suspected in our solver code is that we did not fully implemented all 5 of the presolving part. The reason we suspect the presolving is because before we add the removing zero rows and columns into the solver, the “lp_brandy" cannot be solved, and it is solved after adding the removal of zero rows and columns. This is actually a trade-off between speed since presolving operations such as, merging duplicated rows and columns, row singleton, and forced row requires multiple nested loops to sweep through the matrix A in the LP problem multiple times to complete.

Similar to speed test results, there might not be difference for small size LP problems such as “lp_afiro", but it sure will cause increase the execution time of the solver for large scale LP problems. Another reason we suspect it to be the cause of the Exception threw is our standard form conversion process, we tried to save the memory allocation for the solver by making the modification in-place. Maybe separate out the matrix constructions into small size vectors or matrices in order to make the standard form more robust or at least less likely to prone errors. In addition, for the sake of overall robustness (not just focusing on problem solving aspect), we want the solver keep working on other problems even if one of them exit out due to exceptions. To handle these issues, we added the try-catch blocks around the operations/functions that might cause the unexpected exits.

Generally speaking, our solver achieve the speed criteria by using problem-based gamma selection and less presolving operations, and the robustness of the solver is decent and improvable by the potential fixes listed above.

Conclusion

In conclusion, our solver has demonstrated its effectiveness in solving large-scale linear programming problems. Despite the challenges encountered during the implementation process, our solver delivers remarkable speed and has the potential for further optimization. The features developed during the project enhance the solver’s ability to solve linear programming problems. However, there is still room for improvement, and further research can be done to improve its robustness and reliability. Overall, we have had a great experience on this project and learned a tremendous knowledge about the Primal-dual interior point method.

References

[1] Mehrotra, S. (1990, January 1). Higher order methods and their performance. Northwestern Scholars. Retrieved May 1, 2023, from https://www.scholars.northwestern.edu/en/publications/higher-order-methods-and-their-performance

[2] Nocedal, J., & Wright, S. (2006). Numerical optimization. Springer.

[3] The netlib. (n.d.). Retrieved May 1, 2023, from https://netlib.org/lp/data/readme

[4] Wright, S. (1997). Primal-dual interior-point method, SIAM.