FINAL REPORT

Mathieu Gaillard, Yichen Sheng

April 29, 2019

Abstract

We implemented a Primal-Dual Interior Point LP Solver in Julia. It features:

- Basic presolver (removes zero rows in matrix A)
- Mehrotra algorithm to form the step equation
- Normal equations to solve the step equation
- Cholesky factorization and corresponding solvers

We tested it on a set of problems from the University of Florida Sparse Matrix repository. All of them can be solved with at least 1e-4 tolerance, except lp_ganges. Detailed tolerance results are listed in experiment section. Although we cannot solve all of them with a very high precision, our code might be robust enough for general applications.

Problem

In this project we focus on solving Linear programming problems in standard form:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \boldsymbol{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned} \tag{1}$$

where $\mathbf{c}$ and $\mathbf{x}$ are vectors in $\mathcal{R}^n$, $\mathbf{b}$ is a vector in $\mathcal{R}^m$, and $\boldsymbol{A}$ is an $m \times n$ matrix. The two constraints $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ and $\mathbf{x} \geq 0$ define a convex polytope over which the objective function is minimized.

If $\mathbf{x}$ satisfies the constraints $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ and $\mathbf{x} \geq 0$ then we call it a feasible point. The set of all feasible points is called the feasible set. Usually $m \leq n$, because otherwise, the system would be over-determined and the feasible set could be empty.

It's always possible to convert a problem with the constraint $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$ to a problem in standard form with $\boldsymbol{A}\mathbf{x} = \mathbf{b}$. For that we can add slack and surplus variables to the vector $\mathbf{x}$.

The dual form of the linear program above is:

$$\begin{aligned} \underset{\lambda}{\text{maximize}} \quad & \mathbf{b}^T\lambda \\ \text{subject to} \quad & \boldsymbol{A}^T\lambda + \mathbf{s} = \mathbf{c} \\ & \mathbf{s} \geq 0 \end{aligned} \tag{2}$$

where λ is a vector in $\mathcal{R}^m$ and $\mathbf{s}$ is a vector in $\mathcal{R}^n$. λ and $\mathbf{s}$ are respectively called the dual variables and dual slacks.

We call the first problem (1) the Primal form and the second problem (2) the Dual form. The two problems together are called the primal-dual pair. Primal-Dual algorithms take advantages of the relations between the two forms to solve both of them efficiently. s

Method

The optimality conditions for both problems 1 and 2 are the KKT conditions:

$$\begin{aligned} \boldsymbol{A}^T\lambda + \mathbf{s} &= \mathbf{c} \\ \boldsymbol{A}\mathbf{x} &= \mathbf{b} \\ \mathbf{x}_i * \mathbf{s}_i &= 0 \\ (\mathbf{x}, \mathbf{s}) &\geq 0 \end{aligned}$$

Note that the optimality conditions are the same for both problems. It is the fundamental relationship between the primal and the dual problems.

The main idea of the primal-dual interior-point method is to apply a variant of Newton's method to the equality conditions in the KKT conditions. To make sure the inequality constraint with $\mathbf{x}$ and $\mathbf{s}$ is always strictly true, the search direction is biased and a line search method is used at every iteration. To successfully use Newton's method we build a function F, when the value of this function is zero, it means we have found the solution of the linear programming problem.

$$F(\mathbf{x}, \lambda, \mathbf{s}) = \begin{bmatrix} \boldsymbol{A}^T\lambda + \mathbf{s} - \mathbf{c} \\ \boldsymbol{A}\mathbf{x} - \mathbf{b} \\ \boldsymbol{X}\boldsymbol{S}\mathbf{e} \end{bmatrix} = 0$$

where $\boldsymbol{X} = diag(\mathbf{x})$ and $\boldsymbol{S} = diag(\mathbf{s})$

We use a modified version of Newton's method to find the 0 of function F. Because we want to stay feasible and avoid hitting some constraint boundary too early, which will affect the convergence speed, we bias the step direction and perform a line search. At each iteration, Newton's method builds a linear model of the function around the current point and solve the following system to get the step direction:

$$\begin{bmatrix} 0 & \boldsymbol{A}^T & \boldsymbol{I} \\ \boldsymbol{A} & 0 & 0 \\ \boldsymbol{S} & 0 & \boldsymbol{X} \end{bmatrix} \begin{pmatrix} \Delta_x \\ \Delta_\lambda \\ \Delta_s \end{pmatrix} = \begin{pmatrix} -r_c \\ -r_b \\ -r_{xs} \end{pmatrix}$$

Usually a full step along this direction is not possible because it would violate the bound $(\mathbf{x}, \mathbf{s}) \geq 0$. By using a line search we can circumvent this problem.

$$(\mathbf{x}, \lambda, \mathbf{s}) + \alpha(\Delta_x, \Delta_\lambda, \Delta_s)$$

Depending on the specific variant of Newton's method implemented r_c, r_b and r_{xs} take specific values. The way to pick α is also dependent on the method. We give more details on our own implementation in the next section.

Implementation details

Presolving

Currently, we remove zero rows in $\boldsymbol{A}$ and corresponding rows in $\mathbf{b}$. We noticed that in some problems without this presolving step, the matrix $\boldsymbol{A}$ is singular.

Since it has no consequence on $\mathbf{x}$, which remains the same, it is straightforward to implement.

Other strategies are detailed in [2] on page 231. We could remove zero columns because it is not constraining the corresponding row in $\mathbf{x}$. We could look for duplicate rows in $\boldsymbol{A}$ and merge them in order to reduce the size of the matrix by 1. If a row contains only one element, the solution of the equation $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ is straightforward for this variable and we can remove the corresponding row and column from $\boldsymbol{A}$.

Conversion to standard form

The conversion process is to convert the problem from:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \boldsymbol{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq low \\ & \mathbf{x} \leq high \end{aligned}$$

to this form:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \mathbf{c}'^T\mathbf{x}' \\ \text{subject to} \quad & \boldsymbol{A}'\mathbf{x}' = \mathbf{b}' \\ & \mathbf{x}' \geq 0 \end{aligned}$$

As for $x \geq low$, we can shift x: $x' = x - low, x' \geq 0$. But we also need to shift it back after finding an optimal standard x'.

As for $x \leq high$, we can add slacks as learned from class. Also, we need to change $\mathbf{c}$ to $\mathbf{c}'$, $\boldsymbol{A}$ to $\boldsymbol{A}'$, $\mathbf{b}$ to $\mathbf{b}'$ to have corresponding terms for slacks.

When implementing this algorithm, we also find there are some practical issues since there are some constraints that are ineffective: low is 0.0 and high is ∞. We treated these constraints as ineffective constraints since they are not useless for finding the optimal standard x. Thus we skip these constraints.

Starting point

We implemented the method given in [2] on page 224. Although any starting point with $\mathbf{x}, \mathbf{s} \geq 0$ could theoretically work with our infeasible interior point algorithm, having a starting point with nice properties is more convenient and considerably reduce the number of iterations needed to solve the problem.

We solve these two constrained least norm problems:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \tfrac{1}{2}\left\|\mathbf{x}\right\|^2 \\ \text{subject to} \quad & \boldsymbol{A}\mathbf{x} = \mathbf{b} \end{aligned}$$

$$\begin{aligned} \underset{\lambda,\mathbf{s}}{\text{minimize}} \quad & \tfrac{1}{2}\left\|\mathbf{s}\right\|^2 \\ \text{subject to} \quad & \boldsymbol{A}^T\lambda + \mathbf{s} = \mathbf{c} \end{aligned}$$

The resulting $(\mathbf{x}, \lambda, \mathbf{s})$ are the vectors of least norm for which the two constraints are satisfied. These constraints are also the first two residuals r_b and r_c of the step equation.

To solve these two optimization problems we use an augmented Lagrangian system.

For the first problem:

$$\mathcal{L}(\mathbf{x}, \lambda) = \frac{1}{2}\mathbf{x}^T\mathbf{x} + \lambda^T(\boldsymbol{A}\mathbf{x} - \mathbf{b})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \mathbf{x} + \boldsymbol{A}^T \lambda = 0$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \boldsymbol{A}\mathbf{x} - \mathbf{b} = 0$$

The augmented system is:

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{A}^T \\ \boldsymbol{A} & 0 \end{bmatrix} \begin{pmatrix} \mathbf{x} \\ \lambda \end{pmatrix} = \begin{bmatrix} 0 \\ \mathbf{b} \end{bmatrix}$$

For the second problem, we pack the two variables in one vector and we use the same augmented Lagrangian system:

$$\boldsymbol{A}^T \lambda + \mathbf{s} - \mathbf{c} \quad \equiv \quad \begin{bmatrix} A^T & I \end{bmatrix} \begin{pmatrix} \lambda \\ \mathbf{s} \end{pmatrix} = \mathbf{c} \quad \equiv \quad \boldsymbol{A}'\mathbf{x}' = \mathbf{c}$$

$$\begin{bmatrix} \boldsymbol{I} & 0 & \boldsymbol{A} \\ 0 & \boldsymbol{I} & \boldsymbol{I} \\ \boldsymbol{A}^T & \boldsymbol{I} & 0 \end{bmatrix} \begin{pmatrix} \lambda \\ \mathbf{s} \\ \mathbf{z} \end{pmatrix} = \begin{bmatrix} 0 \\ 0 \\ \mathbf{c} \end{bmatrix}$$

Then, the starting point is defined as:

$$(\mathbf{x}^0, \lambda^0, \mathbf{s}^0) = (\mathbf{x} + \hat{\delta_x}\mathbf{e}, \lambda, \mathbf{s} + \hat{\delta_s}\mathbf{e})$$

The computation of $delta_x$ and $delta_s$ is detailed in [1] (on page 589, section 7).

$$\delta_x = \max(-1.5 \times \min(\mathbf{x}_i), 0)$$

$$\delta_s = \max(-1.5 \times \min(\mathbf{s}_i), 0)$$

$$\hat{\delta_x} = \delta_x + 0.5 \times \frac{(\mathbf{x} + \delta_x\mathbf{e})^T(\mathbf{s} + \delta_s\mathbf{e})}{\sum_1^n(\mathbf{s} + \delta_s)}$$

$$\hat{\delta_s} = \delta_s + 0.5 \times \frac{(\mathbf{x} + \delta_x\mathbf{e})^T(\mathbf{s} + \delta_s\mathbf{e})}{\sum_1^n(\mathbf{x} + \delta_x)}$$

We compared this method against a starting point of the form $(\zeta\mathbf{e}, 0, \zeta\mathbf{e})$ with $\zeta = 1$ or $\zeta = 128$. We noticed a significant decrease in the number of iterations needed to solve the problem.

Step equation

The book provided a primal-dual framework and three forms of the step equation. We have implemented all of them.

First form

Form 1 is directly coming from KKT conditions.

$$\begin{bmatrix} 0 & \boldsymbol{A} & 0 \\ \boldsymbol{A}^T & 0 & \boldsymbol{I} \\ \boldsymbol{S} & 0 & \boldsymbol{X} \end{bmatrix} \begin{pmatrix} \Delta \mathbf{x} \\ \Delta \lambda \\ \Delta \mathbf{s} \end{pmatrix} = \begin{bmatrix} -\mathbf{r}_b \\ -\mathbf{r}_c \\ -\mathbf{r}_{xs} \end{bmatrix}$$

where $\mathbf{r}_{xs} = \boldsymbol{X}\boldsymbol{S}\mathbf{e} - \sigma\mu\mathbf{e}$, $\sigma \in [0, 1]$, $\mu = x^T s/n$ in each iteration.

Denote $\boldsymbol{M} = \begin{bmatrix} 0 & \boldsymbol{A} & 0 \\ \boldsymbol{A}^T & 0 & \boldsymbol{I} \\ \boldsymbol{A}^T & \boldsymbol{I} & 0 \end{bmatrix}$. In practice, we find when x is very close to minimizer, M will become ill, which makes $\begin{pmatrix} \Delta \lambda \\ \Delta \mathbf{x} \\ \Delta \mathbf{s} \end{pmatrix}$ not solved precisely if we use the built in inversion in Julia to solve this system. M is not exactly symmetric, but it is close to symmetric. If it were symmetric, it would give us a lot of benefits. Therefore, the author gives us the second form.

Second form

$$\begin{bmatrix} 0 & \boldsymbol{A} \\ \boldsymbol{A}^T & -\boldsymbol{D}^{-2} \end{bmatrix} \begin{pmatrix} \Delta \lambda \\ \Delta \mathbf{x} \end{pmatrix} = \begin{bmatrix} -\mathbf{r}_b \\ -\mathbf{r}_c + \boldsymbol{X}^{-1}\mathbf{r}_{xs} \end{bmatrix}$$

$$\Delta s = -\boldsymbol{X}^{-1}(\mathbf{r}_{xs} + \boldsymbol{S}\Delta x)$$

where $\boldsymbol{D} = \boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{X}^{\frac{1}{2}}$. Denote $\boldsymbol{M}' = \begin{bmatrix} 0 & \boldsymbol{A} \\ \boldsymbol{A}^T & -\boldsymbol{D}^{-2} \end{bmatrix}$. We have tried to use Julia built-in Cholesky to factorize $\boldsymbol{M}'$, and then solve LL' x = b. This method is the fastest in this way among the three, but we find it is not stable. It can only solve 3 small problems in the testing problems. Especially for Cholesky factorization, the built-in Cholesky is able to solve symmetric positive-definite matrix. It suffers for the illness of $\boldsymbol{M}'$ when x is close to minimizer. Then we further explored the third form: normal equations form.

Normal equations form

$$\begin{aligned} AD^2A^T\Delta\lambda &= -\mathbf{r}_b + \boldsymbol{A}(-\boldsymbol{S}^{-1}\boldsymbol{X}\mathbf{r}_c + \boldsymbol{S}^{-1}\mathbf{r}_{xs}) \\ \Delta s &= -\mathbf{r}_c - \boldsymbol{A}^T\Delta\lambda \\ \Delta x &= -\boldsymbol{S}^{-1}(\mathbf{r}_{xs} + \boldsymbol{X}\Delta s) \end{aligned}$$

This form has some good benefits for solving the systems stable. AD^2A^T is easy to prove that it is symmetric semi-definite positive. Then we can use Cholesky factorization to solve $\Delta\lambda$.

But still we cannot directly use built-in Cholesky function due to illness on the pivot values. In practice, we found that sometimes in the L matrix, some pivots are too large compared to other some small pivots, which makes the result unstable. Thus, we implemented two ways discussed in the book: skip the small pivots or switch the small pivot with a very large pivot to deal with this problem.

Cholesky factorization

Instead of the built-in Cholesky in Julia, we implemented two different Cholesky factorization process to deal with small pivots. In the book, they claimed theoretically they are identical, which is the same as our experiments. The two are skipping the small pivots, setting the column to be zero and replacing small pivots with very large number like 10^{64}. To define what is small, we did the same as the book. Set up a tolerance, track the biggest pivot max_pivot until that iteration. If current pivot is smaller than max_pivot, then we skip or replace it. Otherwise, it is the same as traditional Cholesky. Here are the two implementations:

```
function cholesky_skip(M, tol=default_tol)
    m,n = size(M)
    @assert m==n "To factorize, M should be a squared matrix"

    L = zeros(m,m)
    max_pivot = Float64(0.0)

    for i in 1:m
        max_pivot = max(max_pivot, M[i,i])
        if M[i,i] >= max_pivot * tol
            L[i,i] = sqrt(M[i,i])
            for j in i + 1 : m
                L[j,i] = M[j,i] / L[i,i]
                for k in i+1 : j
                     M[j,k] = M[j,k] - L[j,i] * L[k,i]
                end
            end
        else
            # skip
            continue
        end
    end
    return L
end

function cholesky_big(M, tol=default_tol)
    m,n = size(M)
    @assert m==n "To factorize, M should be a squared matrix"

    L = zeros(m,m)
    println("Type of L: ",typeof(L))
    max_pivot = 0.0
    for i in 1:m
        max_pivot = max(max_pivot, M[i,i])
        if M[i,i] >= max_pivot * tol
            L[i,i] = sqrt(M[i,i])
            for j in i + 1 : m
                L[j,i] = M[j,i] / L[i,i]
                for k in i+1 : j
                     M[j,k] = M[j,k] - L[j,i] * L[k,i]
                end
            end
        else
            L[i,i] = Float64(1e64)
            L[i+1:m, i] .= Float64(1e-64)
            for j in i + 1 : m
                for k in i+1 : j
                     M[j,k] = M[j,k] - Float64(1e-64)
                end
            end

        end
    end
    return L
end
```

Heuristically, "skip" is a little faster than "big", so our final version uses skip. Our process is slower than the built-in cholesky. Also, the result lower triangular matrix is not exact triangular. It is trapozoid shape, thus we also need a solver to efficiently solve it. Thus we also implemented the co-responding solver for trapozoid shape, which is the same process as L x = b, but we set $x_k = 0$ if it meets zero pivots.

Pick alpha

We compute alpha according to the Mehrotra method given in [2] in chapter 10. Alpha is computed twice: the first time during the predictor step to get an estimate of sigma, and the second time after the corrector step as part of the line search.

During after the predictor step, we compute α_{aff} such that:

$$\alpha_{aff}^{pri} = \underset{\alpha \in [0,1]}{\arg\max} \quad (\mathbf{x}^k + \alpha \Delta \mathbf{x}^{aff} \geq 0)$$

Where $\mathbf{x}^k$ is the current value of $\mathbf{x}$ and $\Delta \mathbf{x}^{aff}$ is the direction computed in the predictor step.

$$\alpha_{aff}^{dual} = \underset{\alpha \in [0,1]}{\arg\max} \quad (\mathbf{s}^k + \alpha \Delta \mathbf{s}^{aff} \geq 0)$$

Where $\mathbf{s}^k$ is the current value of $\mathbf{s}$ and $\Delta \mathbf{s}^{aff}$ is the direction computed in the predictor step.

To compute this is Julia, we are iterating on every negative component of $\mathbf{x}$, looking for the minimum value such that $\alpha = \frac{-x_i^k}{\Delta \mathbf{s}^{aff}}$. Following is the code we use:

```
function alpha_max(x, dx, hi = 1.0)
     n = length(x)
     alpha = hi
     ind = -1

     for i=1:n
          if dx[i] < 0.0
               curr_alpha = -x[i]/dx[i]
               if curr_alpha < alpha
                    alpha = curr_alpha
                    ind = i
               end
          end
     end

     return alpha, ind
 end
```

To compute alpha during the line search step, we implemented one of the heuristic given in the book [2] on page 205.

First we compute α_{max} such that:

$$\alpha_{max}^{pri} = \underset{\alpha \geq 0]}{\arg\max} \quad (\mathbf{x}^k + \alpha \Delta \mathbf{x}^k \geq 0)$$

Where $\mathbf{x}^k$ is the current value of $\mathbf{x}$ and $\Delta \mathbf{x}^{aff}$ is the averaged direction computed in the predictor-corrector step.

$$\alpha_{max}^{dual} = \underset{\alpha \geq 0}{\arg\max} \quad (\mathbf{s}^k + \alpha \Delta \mathbf{s}^k \geq 0)$$

Where $\mathbf{s}^k$ is the current value of $\mathbf{s}$ and $\Delta \mathbf{s}^{aff}$ is the averaged direction computed in the predictor-corrector step.

Given a parameter γ_f (equal to 0.05 in our implementation), we compute:

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

For the particular index i for which $\mathbf{x}_i^k + \alpha_{max}^{pri} \Delta \mathbf{x}_i^k = 0$, compute f^{pri}:

$$f^{pri} = \frac{1}{\alpha_{max}^{pri} \Delta \mathbf{x}_i^k} \left(\frac{\gamma_f \mu_+}{\mathbf{s}_i^k + \alpha_{max}^{dual} \Delta \mathbf{s}_i^k} - \mathbf{x}_i^k \right)$$

For the particular index i for which $\mathbf{s}_i^k + \alpha_{max}^{dual} \Delta \mathbf{s}_i^k = 0$, compute f^{dual}:

$$f^{dual} = \frac{1}{\alpha_{max}^{dual} \Delta \mathbf{s}_i^k} \left(\frac{\gamma_f \mu_+}{\mathbf{x}_i^k + \alpha_{max}^{pri} \Delta \mathbf{x}_i^k} - \mathbf{s}_i^k \right)$$

Then, set:

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

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

We noticed a significant increase in speed when using this heuristic. However, on the contrary of what is written in the book, we did not notice any improvement in robustness.

Predictor-corrector method

Predictor-corrector is a smarter way to pick step direction using second order directions. The procedure is to first solve the affine-scaling direction. In form 1, we have:

$$\begin{bmatrix} 0 & \boldsymbol{A} & 0 \\ \boldsymbol{A}^T & 0 & \boldsymbol{I} \\ \boldsymbol{S} & 0 & \boldsymbol{X} \end{bmatrix} \begin{pmatrix} \Delta \mathbf{x} \\ \Delta \lambda \\ \Delta \mathbf{s} \end{pmatrix} = \begin{bmatrix} -\mathbf{r}_b \\ -\mathbf{r}_c \\ -\mathbf{r}_{xs} \end{bmatrix}$$

where $\mathbf{r}_{xs} = \boldsymbol{X}\boldsymbol{S}\mathbf{e} - \sigma\mu\mathbf{e}$. Set $\sigma = 0$, we get affine-scaling direction. Then, we pick alpha_x and alpha_dual for standard x and standard s. After that, we can measure the "quality" of this direction by computing

$$\mu_{aff} = ((xs + \alpha_x * dx_{affine})^T (s + \alpha_{affine} * ds_{affine})/n$$

Use this μ_{aff} we can adaptively choose a better sigma:

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

Use this σ to solve the linear system:

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

We can compute the corrector. Add this corrector to affine-scaling direction. We get the better direction.

End condition

When running the algorithm, the user gives a tolerance *tol*. Its value is very small: usually from 10^{-4} up to 10^{-8}. The iterative algorithm ends when:

$$\mu = \frac{\mathbf{x}^T \mathbf{s}}{n} \leq tol$$

$$\frac{\left\| \begin{matrix} \boldsymbol{A}^T\lambda + \mathbf{s} - \mathbf{c} \\ \boldsymbol{A}\mathbf{x} - \mathbf{b} \\ \mathbf{x}^T\mathbf{s} \end{matrix} \right\|}{\left\| \begin{matrix} \mathbf{b} \\ \mathbf{c} \end{matrix} \right\|} \leq tol$$

In addition to the tolerance, the user gives a maximum number of iterations (100 by default).

These termination conditions are given as a requirement of the final project for this class. Alternatively, [2] gives other termination conditions on page 226.

$$\begin{aligned} \frac{\|r_b\|}{1+\|\mathbf{b}\|} = \frac{\|\boldsymbol{A}\mathbf{x}-\mathbf{b}\|}{1+\|\mathbf{b}\|} &\leq tol \\ \frac{\|r_c\|}{1+\|\mathbf{c}\|} = \frac{\left\|\boldsymbol{A}^T\lambda+\mathbf{s}-\mathbf{c}\right\|}{1+\|\mathbf{c}\|} &\leq tol \\ \frac{\left\|\mathbf{c}^T\mathbf{x}-\mathbf{b}^T\lambda\right\|}{1+\|\mathbf{c}^T\mathbf{x}\|} &\leq tol \end{aligned}$$

Note that in the book [2], the numerator of the last condition is $\left\|\mathbf{c}^T\mathbf{x}-\mathbf{b}^T\mathbf{y}\right\|$. Since $\mathbf{y}$ is not defined anywhere in the book, we replaced it by λ in our formulation.

Experiment

In the following table we gathered the results we can get with our solver. For each problem, we give the reference optimal value, the difference between our result and the reference, the best tolerance, with which our solver can converge, the number of iterations needed for this result and the time needed on a computer equipped with an Intel Xeon W-2145.

		Our MPC solver			
Problem	Optimal	Difference	Best tol	Iterations	Time (s)
lp_afiro	-4.6475314286E+02	2.856041e-09	1e-14	8	0.09
lp_brandy	1.5185098965E+03	-9.095947e-08	1e-8	18	1.09
lp_adlittle	2.2549496316E+05	2.371002e-06	1e-13	25	0.24
lp_agg	-3.5991767287E+07	-4.703382e+02	1e-7	33	31.2
lp_stocfor1	-4.1131976219E+04	-4.364992e-07	1e-12	15	0.34
lp_fit1d	-9.1463780924E+03	-144.864139	1e-5	16	519.289
lp_25fv47	5.5018458883E+03	2.865670e+00	1e-4	23	139
lp_ganges	-1.0958636356E+05	N/A	1e-2	11	N/A
lpi_chemcom	Infeasible	N/A	N/A	N/A	N/A

Discussion / Future work

In summary, we used the PD-framework and tried the three forms of step functions. After explorating the two first options, we focused on the third form and implemented a Cholesky factorization to take advantage of the symmetric semi-definite matrix in it. To deal with illness or small pivot values of the symmetric

semi-definite matrix, we tried the "skip" or "replace it" heuristics. We implemented the Mehrotra predictor-corrector method to pick a better step direction and used a heuristic to pick the value of alpha during the line search step.

The main problem we faced happens when the residual and μ are both very small. Sometimes, with big matrices, some instability arise when near to the solution. For instance: lp_ganges, lp_fit1d and lp_25fv47, when tolerance is very small, e.g. smaller than 10^{-5}. Optimization process will become unstable, and then simply diverge. At first, when testing the first form of the step equation, we had a stricter alpha picking process, although it was usually yielding a too small values and was often stuck in infinite loops, we did not encounter divergence problems. To overcome that, we tried to tweak the parameters, pick different tolerances, switch from one method to another method during the optimization process randomly, etc. But it does not help us to get more robustness.

There are some improvements we did not have time to implement although they are very promising. One such work is to deal with sparse matrices in our Cholesky factorization code. The book [2] give some ways to deal with it, like columns permutations using some heuristics. There also exists some heuristics to find more robust step directions. But due to lack of time, we did not explore those.

About the first form and especially the second form of the step equation, the book explain that some implementations use specific (more robust and faster) factorization methods to solve the step equation. We did not have enough time to explore any of them either.

Conclusion

It was a fun project. We learned a lot both on practical and theoretical part of Linear Programming and gained much experience in developing an Interior-point solver.

References

[1] Sanjay Mehrotra. “On the implementation of a primal-dual interior point method”. In: *SIAM Journal on optimization* 2.4 (1992), pp. 575–601.

[2] Stephen J Wright. *Primal-dual interior-point methods*. Vol. 54. Siam, 1997.