ABHINAV SINHA

PDIPS.JL

COMPUTATIONAL METHODS IN OPTIMIZATION

Contents

PDIPS.jl (Primal-Dual Interior-Point Solver) is a Julia package that implements the homogeneous self-dual interior point algorithm introduced in Tulip.jl.[1] This report contains the algorithm's theory, implementation details, documentation for the code base, and some basic benchmarking results.

[1] Miguel F. Anjos, Andrea Lodi, and Mathieu Tanneau. Tulip.jl: an open-source interior-point linear optimization solver with abstract linear algebra. *Les Cahiers du Gerad*, 2019. URL `https://www.gerad.ca/fr/papers/G-2019-36`

1 Introduction

After Karmakar's paper in 1984, the focus of interior-point methods (IPMs) as viable options for solving linear programs shifted to a class of algorithms known as *primal-dual methods*.[2] In standard form (or equality form), a linear program is

[2] Stephen J. Wright. *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, 1997. DOI: 10.1137/1.9781611971453

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

for $\mathbf{c}, \mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$ and $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ (with $m \leq n$). Primal-dual methods require creating a linear program's *dual*, or

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

for $\mathbf{s} \in \mathbb{R}^n$, and $\boldsymbol{\lambda} \in \mathbb{R}^m$; after which (1) became the *primal* of a linear program. At an optimal solution $(\mathbf{x}^*, \boldsymbol{\lambda}^*, \mathbf{s}^*)$, where $\mathbf{x}^*$ solves (1) and $(\boldsymbol{\lambda}^*, \mathbf{s}^*)$ solves (2), it can be shown that

Note that $\boldsymbol{\lambda}$ is free.

$$\mathbf{b}^T\boldsymbol{\lambda}^* = \mathbf{c}^T\mathbf{x}^*.$$

In fact, for any feasible vectors $(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s})$, the following property holds:

$$\mathbf{b}^T\boldsymbol{\lambda} \leq \mathbf{c}^T\mathbf{x}.$$

Xu et al. [1996] extensively studied the previously proposed homogeneous and self-dual linear feasibility model as another way of approaching primal-dual methods. This model requires the addition of two scalars τ and κ such that the linear program is transformed into:

$$\begin{aligned} \underset{\mathbf{x},\boldsymbol{\lambda},\mathbf{s},\tau,\kappa}{\text{minimize}} \quad & 0 \\ \text{subject to} \quad & \boldsymbol{A}\mathbf{x} - \tau\mathbf{b} = 0 \\ & \boldsymbol{A}^T\boldsymbol{\lambda} + \mathbf{s} - \tau\mathbf{c} = 0 \\ & \mathbf{b}^T\boldsymbol{\lambda} - \mathbf{c}^T\mathbf{x} - \kappa = 0 \\ & \mathbf{x}, \mathbf{s}, \tau, \kappa \geq 0 \end{aligned} \tag{3}$$

PDIPS.JL AIMS to solve problems of the following form:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \boldsymbol{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned} \tag{4}$$

where $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ are lower and upper bounds of $\mathbf{x}$ by taking advantage of the homogeneous self-dual model through an algorithm described in Tulip.jl.

2 *Homogeneous Self-Dual Algorithm*

WE WILL BEGIN this section by first describing the main iteration of this algorithm, assuming that the problem is in standard from. This includes computing a search direction, applying centrality corrections, and termination criteria. Dealing with bounds on $\mathbf{x}$, converting a problem to standard form, and resolving the Newton systems will be discussed in Section 2.6.

This algorithm takes inspiration from the predictor-corrector method published by Mehrotra in 1992.[3]

[3] Sanjay Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2 (4):575–601, 1992. DOI: 10.1137/0802028

2.1 *HSD Step*

Let $(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}, \tau, \kappa)$ be the current strictly feasible iterate.[4] Take the following primal, dual, and gap residuals

[4] This requirement also includes a non-negativity constraint on the iterate.

$$\begin{aligned} r_p &= \tau\mathbf{b} - \boldsymbol{A}\mathbf{x}, \\ r_d &= \tau\mathbf{c} - \boldsymbol{A}^T\boldsymbol{\lambda} - \mathbf{s}, \\ r_g &= \mathbf{c}^T\mathbf{x} - \mathbf{b}^T\boldsymbol{\lambda} + \kappa, \end{aligned}$$

and the duality measure

$$\mu = \frac{\mathbf{x}^T\mathbf{s} + \tau\kappa}{n+1}.$$

Note that this differs from the common duality measure, which is

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

To obtain a search direction $(p_x, p_\lambda, p_s, p_\tau, p_\kappa)$, solve the system

Denote $\boldsymbol{X} = \operatorname{diag}(\mathbf{x})$ and $\boldsymbol{S} = \operatorname{diag}(\mathbf{s})$.

$$\begin{aligned} \boldsymbol{A}p_x - \mathbf{b}p_\tau &= \eta r_p \\ \boldsymbol{A}^T p_\lambda + p_s - \mathbf{c}p_\tau &= \eta r_d \\ -\mathbf{c}^T p_x + \mathbf{b}^T p_\lambda - p_\kappa &= \eta r_g \\ \boldsymbol{S}p_x + \boldsymbol{X}p_s &= \gamma\mu\mathbf{e} - \boldsymbol{X}\boldsymbol{S}\mathbf{e} \\ \kappa p_\tau + \tau p_\kappa &= \gamma\mu - \tau\kappa \end{aligned}$$

obtained from Newton's method, where $\gamma > 0$ is a centering parameter and $\eta > 0$ is a scalar.

Choose a step size $\alpha > 0$ so that the next iterate is

$$(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}, \tau, \kappa) + \alpha(p_x, p_\lambda, p_s, p_\tau, p_\kappa).$$

We also have that the residuals can be updated as

$$(r_p^+, r_g^+, r_d^+) = (1 - \alpha\eta)(r_p, r_g, r_d). \tag{7}$$

Although this is the case, after applying centrality correctors to the search direction, no performance difference was found by recomputing the residuals at the start of the next iteration.

In general, we have the following

$$\mu^0 = \frac{(\mathbf{x}^0)^T\mathbf{z}^0 + \tau^0\kappa^0}{n+1}, \; \mu^{k+1} = \left(1 - \alpha^k\eta\right)\left[1 - \alpha^k\left(1 - \gamma - \eta\right)\right]\mu^k,$$

and

$$\theta^0 = 1, \; \theta^{k+1} = \left(1 - \alpha^k\eta\right)\theta^k.$$

Applying this to (7), it follows that

$$r_p^k = \theta_k r_p^0,$$

$$r_d^k = \theta_k r_d^0,$$

$$r_g^k = \theta_k r_g^0.$$

This property is crucial in regards to free variables (see Section 3.1).

2.2 *Search Direction Computation*

The predictor-corrector method essentially computes two search directions, first in the affine-scaling direction and then in a corrected direction to improve the iterate's centrality. The predictor step, along the affine-scaling direction, is computed by solving the previous linear system with $\eta = 1$ and the centering parameter, $\gamma = 0$; i.e.

$$\begin{bmatrix} \boldsymbol{A} & & & -\mathbf{b} & \\ & \boldsymbol{A}^T & \boldsymbol{I} & -\mathbf{c} & \\ -\mathbf{c}^T & \mathbf{b}^T & & & -1 \\ \boldsymbol{S} & & \boldsymbol{X} & & \\ & & & \kappa & \tau \end{bmatrix} \begin{bmatrix} p_x^{\text{aff}} \\ p_\lambda^{\text{aff}} \\ p_s^{\text{aff}} \\ p_\tau^{\text{aff}} \\ p_\kappa^{\text{aff}} \end{bmatrix} = \begin{bmatrix} r_p \\ r_d \\ r_g \\ -\boldsymbol{X}\boldsymbol{S}\mathbf{e} \\ -\tau\kappa \end{bmatrix}. \tag{8}$$

Ideally, we would like to choose the max step length of $\alpha = 1$ along the affine-scaling direction, but the non-negativity constraint of the iterate requires us to be more judicious in our choice. Hence, we choose

$$\alpha^{\text{aff}} = \max\left\{0 \le \alpha \le 1 \mid 0 \le (\mathbf{x}, \mathbf{s}, \tau, \kappa) + \alpha\left(p_x^{\text{aff}}, p_s^{\text{aff}}, p_\tau^{\text{aff}}, p_\kappa^{\text{aff}}\right)\right\}.$$

Based on α^{aff}, let $\eta = 1 - \gamma$, and for some $\beta > 0$, compute

$$\gamma = \left(1 - \alpha^{\text{aff}}\right)^2 \min\left(\beta, 1 - \alpha^{\text{aff}}\right).$$

When choosing γ and η, we strive to ensure that $\gamma \leq 0.1$ (and consequently, $\eta \geq 0.9$) for long steps in a search direction so that there is significant reduction in r_p and r_d.

These values of η and γ dictate the corrected search direction. Denote $\Delta_x^{\text{aff}} = \text{diag}\left(\delta_x^{\text{aff}}\right)$ and $\Delta_s^{\text{aff}} = \text{diag}\left(\delta_s^{\text{aff}}\right)$. The corrected search direction is computed by solving the system

$$\begin{bmatrix} \boldsymbol{A} & & & -\mathbf{b} & \\ & \boldsymbol{A}^T & \boldsymbol{I} & -\mathbf{c} & \\ -\mathbf{c}^T & \mathbf{b}^T & & & -1 \\ \boldsymbol{S} & & \boldsymbol{X} & & \\ & & & \kappa & \tau \end{bmatrix} \begin{bmatrix} p_x \\ p_\lambda \\ p_s \\ p_\tau \\ p_\kappa \end{bmatrix} = \begin{bmatrix} \eta r_p \\ \eta r_d \\ \eta r_g \\ \gamma\mu\mathbf{e} - \boldsymbol{X}\boldsymbol{S}\mathbf{e} - \Delta_x^{\text{aff}}\Delta_s^{\text{aff}}\mathbf{e} \\ \gamma\mu - \tau\kappa - \delta_\tau^{\text{aff}}\delta_\kappa^{\text{aff}} \end{bmatrix}. \quad (9)$$

Further higher-order corrections can be computed, as suggested in Tulip.jl to improve the centrality of the iterate. However, for the sake of maintaing a level of simplicity in this implementation those corrections were forgone. Including those corrections would, in general, reduce the number of iterations the HSD algorithm requires to reach an optimal solution (or prove infeasibility), making IPMs more competitive with the Simplex method.

2.3 *Step Size*

Let $(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}, \tau, \kappa)$ be the current iterate and let $(p_x, p_\lambda, p_s, p_\tau, p_\kappa)$ be the corrected search direction. Following Xu et al. [1996], the final step size is chosen by computing

$$\alpha_x = \min\left\{-\frac{\mathbf{x}}{p_x} \;\middle|\; p_x < 0\right\},$$

$$\alpha_s = \min\left\{-\frac{\mathbf{s}}{p_s} \;\middle|\; p_s < 0\right\},$$

$$\alpha_\tau = -\frac{\tau}{p_\tau}, \text{ if } p_\tau < 0,$$

$$\alpha_\kappa = -\frac{\kappa}{p_\kappa}, \text{ if } p_\kappa < 0,$$

and then letting

$$\alpha = \omega \times \min\{\alpha_x, \alpha_s, \alpha_\tau, \alpha_\kappa\},$$

where $0 < \omega \leq 1$ is a damping factor.

Some research has shown that IPMs can benefit from taking different step sizes in the primal and dual case, but for the sake of simplicity, this implementation uses the same step size.

2.4 Starting Point

A universal starting point of

$$(\mathbf{x}_0, \boldsymbol{\lambda}_0, \mathbf{s}_0, \tau_0, \kappa_0) = (\mathbf{e}, 0, \mathbf{e}, 1, 1)$$

where $\mathbf{e}$ is the vector of all ones, was suggested in Xu et al. [1996] because of simplicity and desirable properites.

IPMs tend to have issues because their central path requirements prevents them from a warm start. This has led to them taking longer to converge to an optimal solution. Though this algorithm addresses has certain aspects that help it recouperate from many of those issues, it could benefit from some pre-solver that chooses a more specialized starting point for the problem.

2.5 Termination Criteria

Besides numerical instability and iteration limit, the algorithm terminates when the iterate is proven to be optimal, or either the primal or dual is declared infeasible. Denote ε_p, ε_g, ε_g, and ε_i to be positive (specifiable) tolerances. Choosing these positive parameters will be discussed in Section 3.3. Infeasibility is found when either

$$\mu < \varepsilon_i, \text{ or}$$

$$\frac{\tau}{\kappa} < \varepsilon_i.$$

If $\mathbf{b}^T\boldsymbol{\lambda} > \varepsilon_i$, then the primal is infeasible; if $-\mathbf{c}^T\mathbf{x} > \varepsilon_i$, then the dual is infeasible.

Optimality is found when all the following conditions hold:

$$\frac{\|r_p\|}{\tau(1 + \|\mathbf{b}\|)} < \varepsilon_p,$$

$$\frac{\|r_d\|}{\tau(1 + \|\mathbf{c}\|)} < \varepsilon_d,$$

$$\frac{\|\mathbf{c}^T\mathbf{x} - \mathbf{b}^T\boldsymbol{\lambda}\|}{\tau + \|\mathbf{b}^T\boldsymbol{\lambda}\|)} < \varepsilon_g,$$

where $\|\cdot\|$ denotes $\|\cdot\|_\infty$.

2.6 Solving Linear Systems

The crux of interior-point methods is resolving the various Newton systems at each iteration of the algorithm. This simple implementation, for instance, requires solving two such systems for first finding the affine-scaling direction and then the corrected search direction. If the aforementioned higer-order corrections were to be added, the

algorithm would require an additional system for each iteration. As a result, the performance of the algorithm is heavily reliant on the efficiency of its linear algebra routines.

As we will soon see, the HSD algorithm gives rise to certain block-matrix structures to which we can tailor a specialized linear system solver. Another aspect to note is that in each system, the factorization of the left-hand side can be reused since only the right-hand side is modified. Using a general purpose solver makes it more difficult to efficiently resuse the Cholesky factorizations used to solve these systems. So, in this implementation, we have adapted the specialized solver written for this algorithm in Tulip.jl.[5]

We are actually using a variant of the Cholesky factorization – LDLT – since some matrices can be decomposed into $\boldsymbol{LDL}^T$ form, even if they are not a candidate for Cholesky.

[5] This solver can be found at `https://github.com/ds4dm/Tulip.jl/tree/master/src/LinearAlgebra/LinearSolvers`.

The Newton systems we have seen so far are of the form

$$\begin{bmatrix} \boldsymbol{A} & & & -\mathbf{b} & \\ & \boldsymbol{A}^T & \boldsymbol{I} & -\mathbf{c} & \\ -\mathbf{c}^T & \mathbf{b}^T & & & -1 \\ \boldsymbol{S} & & \boldsymbol{X} & & \\ & & & \kappa & \tau \end{bmatrix} \begin{bmatrix} p_x \\ p_\lambda \\ p_s \\ p_\tau \\ p_\kappa \end{bmatrix} = \begin{bmatrix} \xi_p \\ \xi_d \\ \xi_g \\ \xi_{xs} \\ \xi_{\tau\kappa} \end{bmatrix}. \tag{10}$$

for given vectors $\xi_p, \xi_d, \xi_g, \xi_{xs}, \xi_{\tau\kappa}$. We can compute δ_s and δ_κ as

$$\delta_s = \boldsymbol{X}^{-1}\left(\xi_{xs} - \boldsymbol{S}\delta_x\right), \text{ and}$$

$$\delta_\kappa = \frac{1}{\tau}\left(\xi_{\tau\kappa} - \kappa\delta_\tau\right)$$

to get a simplified system of

$$\begin{bmatrix} -\boldsymbol{M}^{-1} & \boldsymbol{A}^T & -\mathbf{c} \\ \boldsymbol{A} & & -\mathbf{b} \\ -\mathbf{c}^T & \mathbf{b}^T & \tau^{-1}\kappa \end{bmatrix} \begin{bmatrix} \delta_x \\ \delta_\lambda \\ \delta_\tau \end{bmatrix} = \begin{bmatrix} \xi_d - \boldsymbol{X}^{-1}\xi_{xs} \\ \xi_p \\ \xi + \tau^{-1}\xi_{\tau\kappa} \end{bmatrix} \tag{11}$$

for $\boldsymbol{M} = \boldsymbol{X}\boldsymbol{S}^{-1}$. This system can be broken down into two augmented systems

$$\begin{bmatrix} -\boldsymbol{M}^{-1} & \boldsymbol{A}^T \\ \boldsymbol{A} & \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ \mathbf{q} \end{bmatrix} = \begin{bmatrix} \mathbf{c} \\ \mathbf{b} \end{bmatrix} \tag{12}$$

and

$$\begin{bmatrix} -\boldsymbol{M}^{-1} & \boldsymbol{A}^T \\ \boldsymbol{A} & \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} \xi_d - \boldsymbol{X}^{-1}\xi_{xs} \\ \xi_p \end{bmatrix}. \tag{13}$$

We transform these two augmented systems into normal equations systems. Equation (13) becomes

$$\left(\boldsymbol{A}\boldsymbol{M}\boldsymbol{A}^T\right)\mathbf{z} = \xi_p + \xi_d - \boldsymbol{X}^{-1}\xi_{xs} \tag{14}$$

so that

$$\mathbf{y} = \boldsymbol{M}^{-1}\left(\boldsymbol{A}^T\mathbf{z} - \xi_d + \boldsymbol{X}^{-1}\xi_{xs}\right).$$

From this, we can recover

$$\delta_\tau = \frac{\xi_g + \tau^{-1}\xi_{\tau\kappa} + \mathbf{c}^T\mathbf{y} + \mathbf{b}^T\mathbf{y}}{\tau^{-1}\kappa - \mathbf{c}^T\mathbf{p} + \mathbf{b}^T\mathbf{q}},$$

$$\delta_x = \mathbf{y} + \delta_\tau \mathbf{p},$$

$$\delta_\lambda = \mathbf{z} + \delta_\tau \mathbf{q}.$$

In solving the normal equations of the form

$$\boldsymbol{AMA}^T\mathbf{s} = \mathbf{r},$$

we perform the an LDLT facorization of $\boldsymbol{AMA}^T \succ 0$, where $\boldsymbol{L}$ is as lower triangular matrix with unitary elements on the diagonal and $\boldsymbol{D}$ is a positive diagonal matrix. When pivoting, we check if the pivot element is less than a certain numerical tolerance so that the current row can be dismissed as almost linearly dependent. This introduces more numerical stability into the algorithm.[6]

[6] Xiaojie Xu, Pi-Fang Hung, and Yinyu Ye. A simplified homogeneous and self-dual linear programming algorithm and its implementation. *Annals of Operations Research*, 62(1):151–171, 1996. DOI: 10.1007/BF02206815

3 Implementation

This section contains the implementation details for PDIPS.jl. The focus of this implementation was on speed, even though we might have to sacrifice robustness as a result. Benchmark results are shown in Section 5

3.1 Bounded Variables

In dealing with bounded variables, we consider lower bounds, upper bounds, and free variables. Suppose $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$.

Free variables may be split into two nonnegative components, i.e.

$$x_i = x_i^+ - x_i^-.$$

Splitting free variables into two components has been shown to produce numerical instability because large pairs can make $\boldsymbol{AMA}^T$ ill-conditioned. However the addition of $\eta = 1 - \gamma$ and the property from Section 2.1 prevents the free variables from rapidly increasing by essentially normalizing the iterates.

Lower bounds like $l_i \leq x_i$ can be dealt with with a simple shift so that

$$0 \leq x_i - l_i.$$

Upper bounds, however, require more consideration. For this, we follow the procedure recommended by Tulip.jl.[7] Let $\mathcal{I} = [1, n]$ and let

[7] Miguel F. Anjos, Andrea Lodi, and Mathieu Tanneau. Tulip.jl: an open-source interior-point linear optimization solver with abstract linear algebra. *Les Cahiers du Gerad*, 2019. URL `https://www.gerad.ca/fr/papers/G-2019-36`

$\mathcal{J} \subseteq \mathcal{I}$ be the set of indices whose corresponding component in $\mathbf{x}$ is upper bounded. The upper bounds are then

$$x_j \leq u_j, \quad \forall j \in \mathcal{J}.$$

If we introduce $\boldsymbol{U} \in \mathbb{R}^{|\mathcal{J}| \times n}$, then the upper bounds can be expressed as $\boldsymbol{U}\mathbf{x} \leq \mathbf{u}$, where

$$\boldsymbol{U}_{i,j} = \begin{cases} 1, & \text{if } i = j \in \mathcal{J} \\ 0, & \text{otherwise} \end{cases}.$$

We apply this modification to the Newton system as

$$\begin{bmatrix} \boldsymbol{A} & & & & & -\mathbf{b} & \\ \boldsymbol{U} & \boldsymbol{I} & & & -\mathbf{u} & & \\ & & \boldsymbol{A}^T & \boldsymbol{I} & -\boldsymbol{U}^T & -\mathbf{c} & \\ -\mathbf{c}^T & \mathbf{b}^T & & -\mathbf{u}^T & & & -1 \\ \boldsymbol{S} & & & \boldsymbol{X} & & & \\ & \boldsymbol{W} & & & \boldsymbol{V} & & \\ & & & & & \kappa & \tau \end{bmatrix} \begin{bmatrix} p_x \\ p_v \\ p_\lambda \\ p_s \\ p_w \\ p_\tau \\ p_\kappa \end{bmatrix} = \begin{bmatrix} \xi_p \\ \xi_u \\ \xi_d \\ \xi_g \\ \xi_{xs} \\ \xi_{vw} \\ \xi_{\tau\kappa} \end{bmatrix}. \quad (15)$$

The method for resolving the Newton systems from Section 2.6 still holds with slight modification.

3.2 *Standard Form*

Converting a linear program from the form in (4) to standard from in (1) requires careful implementation to ensure that constraints are not lost in the process.

The following code block shows how PDIPS.jl counts the number of free and upper bounded variables in $\mathbf{x}$.

```
free = 0
ub = 0
nzv = 0 # non-zero values
for (i, (l, h)) in enumerate(zip(lp.lo, lp.hi))
    if l == -Inf && h == Inf
        # free
        free += 1
        nzv += length(lp.cols[i].ind)
    elseif isfinite(l) && isfinite(h)
        # l ≤ x ≤ h
        ub += 1
    elseif l == -Inf && isfinite(h)
        # x ≤ h, will be dealt with later
    elseif isfinite(l) && h == Inf
        # l ≤ x, will be dealt with later
    else
        error("unexpected bounds ($l, $h)")
    end
    nzv += length(lp.cols[i].ind)
end
```

Note that `free` is a counter for the number of free variables, `ub` is the same for upper bounded variables, and `nzv` counts the number of non-zero values in the columns of A.

The following block is the method by which we modify the original problem to account for bounded variables. Although this way of reforumlating requires iterating over the bounds twice, it avoids the difficulty in memory management when using a single iteration. Pre-allocating the needed memory actually gives better performance. See Section 4.3 for more details.

```
I = Vector{Int}(undef, nzv) # row indices
J = Vector{Int}(undef, nzv) # column indices
V = Vector{T}(undef, nzv)
ind_ub = Vector{Int}(undef, ub)
val_ub = Vector{T}(undef, ub)
b = copy(lp.b)
c = Vector{T}(undef, nv + free)

free = 0
ub = 0
nzv = 0
for (j, (l, h)) in enumerate(zip(lp.lo, lp.hi))
    column = lp.cols[j] # current column
    if l == -Inf && h == Inf
        # free variable
        c[j + free]  = lp.c[j]
        for (i,v) in zip(column.ind,column.nzval)
            nzv += 1
            I[nzv] = i
            J[nzv] = j + free
            V[nzv] = v
        end
        c[j + free + 1] = -lp.c[j]
        for (i,v) in zip(column.ind,column.nzval)
            nzv += 1
            I[nzv] = i
            J[nzv] = j + free + 1
            V[nzv] = -v
        end
        free += 1
    elseif isfinite(l) && isfinite(h)
        # l ≤ x ≤ h
        c[j + free] = lp.c[j]
        for (i,v) in zip(column.ind,column.nzval)
            b[i] -= (v * l)
            nzv += 1
            I[nzv] = i
            J[nzv] = j + free
            V[nzv] = v
        end
        ub += 1
        ind_ub[ub] = j + free
        val_ub[ub] = h - l
    elseif l == -Inf && isfinite(h)
        # x ≤ h
        c[j + free] = -lp.c[j]
        for (i,v) in zip(column.ind,column.nzval)
            b[i] -= (-v * u)
```

```
                nzv += 1
                I[nzv] = i
                J[nzv] = j + free
                V[nzv] = -v
            end
        elseif isfinite(l) && h == Inf
            # l ≤ x
            c[j + free] = lp.c[j]
            for (i,v) in zip(column.ind,column.nzval)
                b[i] -= (v * l)
                nzv += 1
                I[nzv] = i
                J[nzv] = j + free
                V[nzv] = v
            end
        else
            # this error will have been caught
        end
    end
```

3.3 *Default Values*

The default iteration limit is 100, but can be configured.

The default value of the damping factor in Section 2.3 is

$$\omega = 0.9995.$$

This value is not currently configurable because this value was found to be the most practical after numerical experiments.[8]

[8] Xiaojie Xu, Pi-Fang Hung, and Yinyu Ye. A simplified homogeneous and self-dual linear programming algorithm and its implementation. *Annals of Operations Research*, 62(1):151–171, 1996. DOI: 10.1007/BF02206815

The default value when computing centrality corrections with γ from Section 2.2 is

$$\beta = 10^{-1}.$$

This value is also not currently configurable.

The default tolerances in Section 2.5 are

$$\varepsilon_p = 10^{-8},$$

$$\varepsilon_d = 10^{-8},$$

$$\varepsilon_g = 10^{-8},$$

$$\varepsilon_i = 10^{-8}.$$

In the implementation, this is value is coded as

```
tol::T where T <: Real = sqrt(eps(T)))
```

since for a 64-bit floating point number, $\sqrt{\epsilon} \approx 1.5 \times 10^{-8}$. The framework for allowing each parameter to be individually tuned is already in place, but currently, the one specified tolerance is used for all four constants.

4 Package Documentation

This section documents the `PDIPS.jl` package. The package is available at `https://github.com/sinha-abhi/PDIPS.jl`; it can be added to Julia via `REPL` as

```
] add https://github.com/sinha-abhi/PDIPS.jl
```

4.1 API

The API for PDIPS.jl consists of three function calls.

The following empty constructor allocates memory for an empty `Problem` of the from described in Section 1. The type `T` denotes the type of the elements of the solution vector, constraints, etc.

For further details on types in this package, see Section 4.2.

```
Problem{T}() where T <: Real
```

`load_problem!` loads the problem data into `lp` with no return value. Currently, we assume that A is sparse when solving the resulting linear systems.

```
function load_problem!(
    lp::AbstractProblem{T},
    A::AbstractMatrix{T},
    b::Vector{T},
    c::Vector{T},
    lo::Vector{T},
    hi::Vector{T}
) where T <: Real
```

`solve` is the workhorse of the package; it attempts to solve `lp` and returns a `Solution`.

```
function solve(
    lp::AbstractProblem{T},
    maxiter::Int = 100,
    tol::T = sqrt(eps(T))
) where T <: Real
```

4.2 Types

Constructors have been omitted for brevity.

PDIPS.jl has eight types that are used throughout the solver, and two enums to track the solver's status.

The `Problem` type stores the number of constraints and variables the problem has, A as an array of `Columns`, the right-hand side, the cost function, and potential bounds.

```
mutable struct Column{T}
    ind::Vector{Int}
    nzval::Vector{T}
end
```

```
mutable struct Problem{T <: Real}
    nc::Int
    nv::Int

    cols::Vector{Column{T}} # columns of A
    b::Vector{T}
    c::Vector{T}

    lo::Vector{T}
    hi::Vector{T}
end
```

Storing the constraint matrix $\boldsymbol{A}$ as an array of columns makes accessing and tracking elements easier when converting to standard from. After the linear program is in standard from, $\boldsymbol{A}$ is of type `AbstractMatrix` from SparseArrays.jl.

The `StandardProblem` type is the result of reformulating `Problem`. Instead of explicitly storing $\boldsymbol{U}$, we store only the indices with valid upper bounds and their respective values. This saves computation time as well as memory in the algorithm.

```
mutable struct StandardProblem{T}
    nc::Int # number of constraints
    nv::Int # number of variables
    nu::Int # number of upper-bounded variables

    upb_ind::Vector{Int} # indices of upper bounds
    upb_val::Vector{T}   # values of upper bounds

    A::SparseMatrixCSC{T}
    b::Vector{T}
    c::Vector{T}
end
```

The `Solution` type contains the solution vector $\mathbf{x}$ and flag indicating successful optimization. It also includes information about the problem when converted to standard form.

```
mutable struct Solution{T}
    x::Vector{T}
    status::Bool

    # standard form
    A_std::Union{Nothing, AbstractMatrix{T}}
    b_std::Vector{T}
    c_std::Vector{T}
    x_std::Vector{T}
    λ_std::Vector{T}
    s_std::Vector{T}
end
```

However, the matrix `A_std` will not contain all information about the upper bounds because of the method we used to reforumlate the problem.[9]

[9] See Section 3.2.

The most important type is `Solver`, which houses information about the status of the optimization problem, tolerances, current

iterate and its residuals, as well as a reference to the linear system solver.

```
    mutable struct Solver{T}
    lp::StandardProblem{T}

    iter::Iterate{T}
    res::Residuals{T}
    tols::Tolerances{T}

    niter::Int # number of iterations

    # status
    status::SolverStatus
    status_primal::IterateStatus
    status_dual::IterateStatus

    # linear solver -- adapted from Tulip.jl
    ls::AbstractLinearSolver{T}
    regP::Vector{T} # primal regularization
    regD::Vector{T} # dual regularization
    regG::T         # gap regularization
end
```

The `Iterate` type has the values needed for each step from Section 2.1, as well as dimensions of the problem.

```
mutable struct Iterate{T}
  nc::Int # number of constraints
  nv::Int # number of variables
  nu::Int # number of upper-bounded variables

  # primal
  x::Vector{T} # given variables
  v::Vector{T} # slacks

  # dual
  λ::Vector{T} # dual variables
  s::Vector{T} # dual slacks
  w::Vector{T} # dual upperbound slacks

  τ::T # primal homogenous constant
  κ::T # dual homogenous constant

  μ::T # duality measure
end
```

Along with the solver and iterates, we have the following enums that track the their statuses. If a `verbose` option is added, these will be more useful in determining the reason why a linear program could not be solved.

```
@enum SolverStatus begin
    SolverUndefined

    # problem status
    SolverOptimal
    SolverPrimalInfeasible
    SolverDualInfeasible
```

```
    # computation status
    SolverExceededIterations
    SolverNumericalInstability
    SolverExceededMemory
end

@enum IterateStatus begin
    IterateUndefined

    IterateOptimal
    IterateFeasible
    IterateInfeasible
end
```

`Residuals` includes all the residual values required to check for optimality or infeasibility. These are recomputed and update appropriately at the start of each iteration.

```
mutable struct Residual{T <: Real}
  rp::T # τ * b - A * x
  ru::T # τ * u - v - U * x
  rd::T # τ * c - A' * y - s + U * x
  rg::T # c' * x - b' * λ - u * w + κ

  # norms of above residuals
  rpn::T
  run::T
  rdn::T
  rgn::T
end
```

The final struct `Tolerances` logs the tolerances for each of the positive parameters.

```
mutable struct Tolerances{T}
    εp::T
    εd::T
    εg::T
    εi::T
end
```

4.3 *Performance Notes*

This section is for general notes on writing performant code in Julia.

First, it is important to write "type-stable" functions and `structs`. As such, when creating generic (or parameterized) types, it is best to avoid fields with abstract types; i.e. annotate types in `structs`, for example, so that the compiler can generate high performance code.

Second, it is best to pre-allocate memory as much as possible because dynamically allocating memory can lead to expensive operations, such a resizing an array. Thus, most of the functions in PDIPS.jl modify their arguments instead of returning a new object. Loosely related is the idea of using `@views` instead of slicing arrays.

Third, use the `dot` operator where appropriate when applying basic operations to vectors. For example, for `Vectors` `x` and `y`, use

```
x .+ y
```

as opposed to

```
x + y
```

even though the latter is valid.

Finally, make use of specialized linear algebra routines (BLAS) when applicable since they are optimized for common linear algebra operations. For instance, take the operation $\boldsymbol{Y} = a\boldsymbol{X} + \boldsymbol{Y}$. It is valid to do either of the following.

```
Y = a * X + Y
```

```
Y .+= a .* X
```

However, a higher-performance option is

```
axpy!(a, X, Y)
```

because it avoids miscellaneous allocations (among other things).

5 *Benchmarks*

In this section, we see the computational results of the package in terms of memory used, allocations made, and time taken. The code was run using Julia v1.4 on a Intel i7 5820K, overclocked at 4.5GHz, 32GB RAM machine running Ubuntu. The information reported in Table 1 does not include preprocessing (only the main iterations).

Name	Optimal	Memory	Allocations	Mean Time
25fv47	Y	48.97 MiB	4510	88.956 ms
adlittle	Y	1.47 MiB	2014	935.618 μs
afiro	Y	433.97 KiB	1456	272.505 μs
agg	Y	11.75 MiB	3054	8.408 ms
brandy	Y	6.50 MiB	2884	5.904 ms
chemcom	N	4.58 MiB	1239	3.563 ms
fit1d	Y	27.16 MiB	3050	18.282 ms
ganges	Y	33.17 MiB	3336	26.077 ms
stocfor1	Y	2.35 MiB	2436	1.532 ms

Table 1: Computational results of PDIPS.jl

References

Miguel F. Anjos, Andrea Lodi, and Mathieu Tanneau. Tulip.jl: an open-source interior-point linear optimization solver with abstract linear algebra. *Les Cahiers du Gerad*, 2019. URL `https://www.gerad.ca/fr/papers/G-2019-36`.

Sanjay Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2(4):575–601, 1992. DOI: 10.1137/0802028.

Stephen J. Wright. *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, 1997. DOI: 10.1137/1.9781611971453.

Xiaojie Xu, Pi-Fang Hung, and Yinyu Ye. A simplified homogeneous and self-dual linear programming algorithm and its implementation. *Annals of Operations Research*, 62(1):151–171, 1996. DOI: 10.1007/BF02206815.