A Primal-Dual Operator Splitting Method for Conic Optimization

Eric Chu* Brendan O'Donoghue† Neal Parikh‡ Stephen Boyd*

December 6, 2013

Abstract

We develop a simple operator splitting method for solving a primal conic optimization problem; we show that the iterates also solve the dual problem. The resulting algorithm is very simple to describe and implement and yields solutions of modest accuracy in competitive times. Several versions of the algorithm are amenable to parallelization, either via distributed linear algebra or GPU-accelerated matrix-vector multiplication. We provide a simple, single-threaded C implementation for reference.

*Electrical Engineering Department, Stanford University. Email: {echu508, boyd}@stanford.edu
†Quantcast. Email: bodonoghue85@gmail.com
‡Computer Science Department, Stanford University. Email: npparikh@cs.stanford.edu

Contents

1 Introduction

We present a first-order algorithm for solving large-scale conic optimization problems. While variants of the algorithm have previously been described [4, 9, 39, 50], our aim is to provide a short but complete derivation and a reference C implementation. Our goal is to create a black-box solver that can be used in conjunction with modeling tools such as CVX [23]. Though we do not do so here, this implementation can be parallelized via distributed linear algebra or general-purpose GPU (GPGPU) programming. Many extensions and variations on the algorithm exist, but we focus on the simplest and most generic version. With an interface to CVX [23], this solver provides the ability to rapidly prototype large-scale optimization problems. This solver does not exploit problem structure beyond sparsity and thus is not necessarily the most efficient, but instead provides a simple way to solve problems in a parallel and distributed fashion.

We provide detailed discussions of the implementation and demonstrate that the simple implementation performs reasonably well on a variety of problems, including the DIMACS test problems. This implementation lays the groundwork for further work in distributed (first-order) conic solvers.

2 Cone programming

Let $\mathcal{P}$ be the (primal) cone program

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax + s = b \\ & s \in \mathcal{K}, \end{array}$$

with variables $x \in \mathbf{R}^n$ and $s \in \mathbf{R}^m$, where $\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_q \subset \mathbf{R}^m$ is a Cartesian product of q closed convex cones. Here, each $\mathcal{K}_i$ has dimension m_i, so $\sum_{i=1}^q m_i = m$. We assume that $m \geq n$ and that A has full column rank. A pair of variables $(x, s) \in \mathbf{R}^n \times \mathbf{R}^m$ is *primal feasible* if $Ax + s = b$ and $s \in \mathcal{K}$. The problem data are A, b, c, and (a description of) $\mathcal{K}$.

When we fix particular values for the cones $\mathcal{K}_i$, we recover standard problem families as special cases. The most common cones are

$$\begin{aligned} \mathbf{R}_+ &= \{x \in \mathbf{R} \mid x \geq 0\}, \\ \mathcal{Q}^d &= \{(t, v) \in \mathbf{R}^{d+1} \mid \|v\|_2 \leq t\}, \\ \mathbf{S}_+^n &= \{X \in \mathbf{S}^n \mid X \succeq 0\}, \end{aligned}$$

known as the nonnegative reals, the second-order cone (of order d), and the positive semidefinite cone, respectively. (By convention, we define $\mathcal{Q}^0 = \mathbf{R}_+$.) These three cones are known as the *symmetric cones*, and an instance of $\mathcal{P}$ that restricts each $\mathcal{K}_i$ to be one of these is known as a *symmetric cone program*. If *all* the $\mathcal{K}_i$ are a single type of one of the cones above, then $\mathcal{P}$ is known as a *linear program* (LP), *second-order cone program* (SOCP), and *semidefinite program* (SDP), respectively. The method we describe in this paper allows $\mathcal{K}$ to be an arbitrary closed convex cone, although we focus specifically on SOCPs.

The dual problem of $\mathcal{P}$ is

$$\begin{array}{ll} \text{maximize} & -b^T y \\ \text{subject to} & -A^T y = c \\ & y \in \mathcal{K}^*, \end{array}$$

with (dual) variable $y \in \mathbf{R}^m$, where $\mathcal{K}^* = \mathcal{K}_1^* \times \cdots \times \mathcal{K}_q^*$ is the dual cone of $\mathcal{K}$. We say that y is *dual feasible* if $A^T y + c = 0$ and $y \in \mathcal{K}^*$.

When (x, s) is primal feasible and y is dual feasible, the quantity $\eta = c^T x + b^T y = s^T y$ is called the *duality gap*. The duality gap is always nonnegative; this property, *weak duality*, tells us, for example, that $-b^T y$ is a lower bound on the optimal value $p^\star$ of $\mathcal{P}$ whenever y is dual feasible. When the duality gap is zero, then (x, s) is a solution of $\mathcal{P}$ and (x, s, y) is called a *primal-dual solution* of $\mathcal{P}$. We denote a primal-dual solution by $(x^\star, s^\star, y^\star)$. We say that s and y are *complementary* when the duality gap $s^T y$ is zero.

The goal of computing a primal-dual solution $(x^\star, s^\star, y^\star)$ can thus be expressed as attempting to satisfy the following optimality conditions:

$$Ax^\star + s^\star = b, \quad A^T y^\star + c = 0, \quad s^\star \in \mathcal{K}, \quad (s^\star)^T y^\star = 0, \quad y^\star \in \mathcal{K}^*.$$

These include primal feasibility, dual feasibility, and complementarity, the last of which implies zero duality gap. A primal-dual solution can fail to exist for several reasons, such as when $\mathcal{P}$ is unbounded or infeasible. In this paper, we limit our discussion to finding a solution when one exists.

3 Primal-dual operator splitting method

Our algorithm is a first-order method based on the alternating direction method of multipliers (ADMM), also known as Douglas-Rachford splitting [7, 15].

We first rewrite $\mathcal{P}$ to be in a form amenable to ADMM. Let $I_{\mathcal{S}}$ denote the *indicator function* of the set $\mathcal{S}$, so $I_{\mathcal{S}}(z)$ is 0 for $z \in \mathcal{S}$ and $+\infty$ otherwise. Then $\mathcal{P}$ can be expressed as

$$\text{minimize} \quad c^T x + I_{\mathcal{E}}(x, s) + I_{\mathcal{K}}(s)$$

where $\mathcal{E} = \{(x, s) \mid Ax + s = b\}$, *i.e.*, the equality constraints. We then replicate the variables x and s and write the problem in *consensus form* [7, §7.1],

$$\begin{array}{ll} \text{minimize} & f(\tilde{x}, \tilde{s}) + g(x, s) \\ \text{subject to} & \tilde{x} = x, \quad \tilde{s} = s, \end{array}$$

where $f(\tilde{x}, \tilde{s}) = c^T \tilde{x} + I_{\mathcal{E}}(\tilde{x}, \tilde{s})$ and $g(x, s) = I_{\mathbf{R}^n}(x) + I_{\mathcal{K}}(s)$. The dual of this consensus problem is

$$\begin{array}{ll} \text{maximize} & -b^T \nu \\ \text{subject to} & A^T \nu + c = 0 \\ & \nu \in \mathcal{K}^*, \end{array}$$

with variable $\nu \in \mathbf{R}^m$, which is exactly $\mathcal{D}$.

Applying ADMM to the (primal) consensus problem gives the algorithm

$$\begin{aligned}(\tilde{x}^{k+1}, \tilde{s}^{k+1}) &= \mathbf{prox}_{\lambda f}(x^k + \lambda u^k, s^k + \lambda y^k)\\ (x^{k+1}, s^{k+1}) &= \mathbf{prox}_{\lambda g}(\tilde{x}^{k+1} - \lambda u^k, \tilde{s}^{k+1} - \lambda y^k)\\ u^{k+1} &= u^k + (1/\lambda)(x^{k+1} - \tilde{x}^{k+1})\\ y^{k+1} &= y^k + (1/\lambda)(s^{k+1} - \tilde{s}^{k+1}),\end{aligned}$$

where

$$\mathbf{prox}_{\lambda h}(v) = \operatorname*{argmin}_z \left(h(z) + (1/2\lambda)\|z - v\|_2^2\right)$$

is the *proximal operator* of h with parameter $\lambda > 0$ [37]. Here, the parameter λ is an algorithm parameter.

We can simplify the algorithm above. Since f is an indicator function plus a linear function, we can include the linear term in the argument of the proximal operator. This reduces the first step to a projection onto $\mathcal{E}$. Since g is an indicator function over $\mathbf{R}^n \times \mathcal{K}$, this means $x^k = \tilde{x}^k$ and the second step reduces to a projection onto $\mathcal{K}$. Finally, because $x^k = \tilde{x}^k$, if $u^0 = 0$, then $u^k = 0$ for all iterates k. With these simplifications, the final form of our *primal-dual operator splitting* (PDOS) method is

$$\begin{aligned}(x^{k+1}, \tilde{s}^{k+1}) &= \Pi_{\mathcal{E}}(x^k - \lambda c, s^k + \lambda y^k)\\ s^{k+1} &= \Pi_{\mathcal{K}}(\tilde{s}^{k+1} - \lambda y^k)\\ y^{k+1} &= y^k + (1/\lambda)(s^{k+1} - \tilde{s}^{k+1}),\end{aligned}$$

where $\Pi_{\mathcal{S}}$ denotes Euclidean projection onto the set $\mathcal{S}$. Although the algorithm derives from an operator-splitting applied to the primal problem $\mathcal{P}$, the iterates asymptotically yield primal *and* dual solutions; hence, the name *primal-dual operator splitting*.

3.1 Extensions

There are a few variations on the basic method that can be very helpful in practice.

Over-relaxation. Numerical experiments in [14, 16] suggest that the algorithm can be improved by *over-relaxation*, which consists of replacing occurrences of $\tilde{s}^{k+1}$ with

$$\alpha\tilde{s}^{k+1} + (1 - \alpha)s^k,$$

where $\alpha \in (1, 2)$ is the over-relaxation parameter. Typically, $\alpha \in [1.5, 1.8]$ yields some improvement. The convergence results for ADMM (and therefore, PDOS) hold without modification; see, *e.g.*, [15].

Warm starting. The PDOS algorithm can be started from default values such as 0; it can also be started from a good guess of the optimal values of x, s, and y. (This is called *warm-starting*.) One common situation where this occurs is when we solve a sequence of similar problems, with perturbed values of the data A, b, and c. In this case, we can warm start PDOS with the solution to the previous problem. Warm starting can significantly reduce the number of iterations required to converge. See §7.2.4 for an example.

Approximate projection. Another variation replaces the (subspace projection) update $(x^{k+1}, \tilde{s}^{k+1}) = \Pi_{\mathcal{E}}(x^k - \lambda c, s^k + \lambda y^k)$ with a suitable approximation. We replace $(x^{k+1}, \tilde{s}^{k+1})$ with any $(x^{k+1}, \tilde{s}^{k+1})$ that satisfies

$$\|(x^{k+1}, \tilde{s}^{k+1}) - \Pi_{\mathcal{E}}(x^k - \lambda c, s^k + \lambda y^k)\|_2 \leq \mu^k,$$

where $\mu^k > 0$ satisfy $\sum_k \mu_k < \infty$. This variation is particularly useful when an iterative method is used to compute $(x^{k+1}, \tilde{s}^{k+1})$. The convergence results for ADMM (and therefore, PDOS) hold without modification; see, *e.g.*, [15].

4 Related work

Before continuing a more detailed discussion of the properties and implementation of PDOS, we pause to discuss related work and the similarities and differences of our approach. This section may be skipped with little to no detriment to the reader.

Many methods have been developed to solve $\mathcal{P}$, the most widely-used being interior-point methods [6, 8, 32, 34, 51, 53]. These methods reliably and efficiently solve most small to medium-sized problems. They are known to converge to ϵ accuracy in $O(\log(1/\epsilon))$ iterations. Each iteration is dominated by the solution of the KKT system, which in the worst case requires $O((m+n)^3)$ flops at each iteration. If problem structure or sparsity is exploited, the computational complexity can be substantially less. To solve the KKT system, we must store A and A^T explicitly. In practice, interior-point methods find ϵ accurate solutions in a few tens of iterations independent of problem scale and structure. However, the computational cost of each iteration may be prohibitively expensive, and for larger problems, other approaches must be explored.

This motivates the development of matrix-free interior-point methods such as those proposed by Gondzio which only require the ability to multiply by A and A^T [17, 21]. These need at most $O(k(m+n)^2)$ flops (which can be reduced if sparsity is exploited) per iteration, where k is the number of matrix vector multiplies needed to obtain a desired accuracy. If $k \ll m+n$, then this approach is computationally more attractive than traditional interior-point methods. Furthermore, this approach is amenable to general-purpose GPU (GPGPU) implementations, admitting parallel and large-scale implementations.

Alternatively, first-order methods, such as gradient descent and projected gradient descent can be used to solve $\mathcal{P}$ [5, 25, 28–31]. Gradient-type, first-order methods converge to ϵ accuracy in $O(1/\epsilon^2)$ iterations with at most $O(mn)$ flops required for each iteration.

With additional regularity assumptions (*e.g.*, Lipschitz continuous gradients, strong convexity, *etc.*), the convergence rate can be substantially improved. In practice, these methods achieve low accuracy in a few hundred of iterations and take significantly more iterations to achieve high accuracy. Nonetheless, the computational complexity for each iteration may be substantially less than that required for an interior-point method and thus provide a reasonable way to obtain low accuracy solutions. Additionally, many first-order methods admit trivially parallelizable implementations. These methods work extremely well when tuned to very *specific* application domains, such as in compressed sensing and image reconstruction, *etc.* Although many attempts have been made to generalize first-order methods for large-scale optimization, no de facto standard has yet emerged.

PDOS is an attempt to design and implement a general conic solver for large-scale optimization. It is a first-order penalty method that solves a sequence of quadratic penalty problems via distributed linear algebra. In particular, the projection onto $\mathcal{E}$ can be computed first by finding x^{k+1},

$$x^{k+1} = \operatorname*{argmin}_x \left(c^T x + (1/2\lambda)\|x - x^k\|_2^2 + (1/2\lambda)\|b - Ax - s^k - \lambda y^k\|_2^2 \right),$$

and then $\tilde{s}^{k+1} = b - Ax^{k+1}$. Without further assumptions, it has a worst-case convergence rate of $O(1/\epsilon^2)$. Computing x^{k+1} requires, in the worst case, $O(mn^2)$ flops, but several tricks can be used to reduce the computation cost (factorization caching, warm-starting iterative solvers, *etc.*). We now focus on three other works that are more closely related to ours.

Wen, *et al.*, provide an ADMM-based SDP solver [50]. Minor differences aside, their algorithm and ours are essentially the same; in this sense, our algorithm is not new. The main difference is that we focus primarily on SOCPs [1,26], since our emphasis is to provide an algorithm that is easily parallelized and distributed. While projections onto large second-order cones can be parallelized, it is at present unknown how to parallelize or distribute the projection onto large semidefinite cones.

Pock and Chambolle design a family of diagonal preconditioners for solving the saddle-point problem associated with the Lagrangian of $\mathcal{P}$ [9, 39]. Their preconditioners can be interpreted as a scaling of the primal and dual space and correspond directly to our concept of scalings in §6.3. Although they demonstrate on several examples that diagonal preconditioning can improve the performance of their first-order solver, they only consider the case of linear programs. Our observation has been that their proposed diagonal preconditioner works well on linear programs, but can be inadequate for second-order cone programming.

More recently, Aybat and Iyengar present the augmented Lagrangian algorithm (ALCC), a first-order algorithm proven to converge to ϵ accuracy in $O(\log(1/\epsilon))$ (outer) iterations [4]. Its iterates are extremely similar to PDOS. Instead of computing the Euclidean projection onto $\mathcal{E}$, they compute

$$x^{k+1} = \operatorname*{argmin}_x \left(\min_{s \in \mathcal{K}} \left(c^T x + (1/2\lambda)\|b - Ax - s - \lambda y^k\|_2^2 \right) \right),$$

which they solve with projected gradient descent. Thus, the *aggregate* convergence rate is $O(1/\epsilon \log(1/\epsilon))$ with each (outer) iteration requiring at most $O(kmn)$ flops. Each iteration

requires solving a (simple) optimization problem. This relationship between PDOS and ALCC suggests that if $s^k \approx \bar{s}^\star$ (where $\bar{s}^\star$ is the solution to the partial minimization problem over s), one might be able to accelerate the PDOS algorithm.

Finally, we point out that although we do not consider problems that may be infeasible or unbounded, we can also apply similar operator splitting techniques to solve a homogeneous self-dual embedding (HSD) [52, 54] of $\mathcal{P}$ to address these pathologies. We plan to detail this approach in a separate paper.

5 Convergence theory

5.1 Asymptotic optimality

In this section, we will show that, assuming a primal-dual solution exists, the iterates (x^k, s^k, y^k) asymptotically satisfy the optimality conditions. To show that the optimality conditions hold asymptotically, we use general ADMM convergence theory; see, *e.g.*, [7, §3.2], or [15] for the case of approximate projections and dual variable convergence. In our argument, we assume the subspace projection $\Pi_{\mathcal{E}}$ is done exactly. In the case of approximate projections, our argument can be made correct with only minor modifications.

The cone constraints and the complementarity condition are satisfied *exactly* by the iterates (x^k, s^k, y^k), so

$$s^k \in \mathcal{K}, \quad y^k \in \mathcal{K}^*, \quad s^{kT} y^k = 0,$$

for all k. Since s^k is the Euclidean projection of $\tilde{s}^k - \lambda y^{k-1}$ onto $\mathcal{K}$, this implies that $s^k \in \mathcal{K}$, and $\lambda y^k = s^k - (\tilde{s}^k - \lambda y^{k-1})$ is orthogonal to s^k (implying $s^{kT} y^k = 0$). That $y^k \in \mathcal{K}^*$ can be proved via the Moreau decomposition [37]

$$v = \Pi_{\mathcal{K}}(v) + \Pi_{\mathcal{K}^\circ}(v),$$

where $\mathcal{K}^\circ = -\mathcal{K}^*$ is the polar cone (*i.e.*, the negative dual cone). With $v = \tilde{s}^k - \lambda y^{k-1}$ and using the fact that $\lambda > 0$, we obtain

$$-y^k = (1/\lambda)\Pi_{\mathcal{K}^\circ}(\tilde{s}^k - \lambda y^{k-1}),$$

which is in the polar cone, so clearly $y^k \in \mathcal{K}^*$.

It remains to show that as $k \to \infty$, the primal and dual equality constraints are satisfied. ADMM convergence theory guarantees that we have

$$s^k - \tilde{s}^k = Ax^k + s^k - b \to 0,$$

as $k \to \infty$ [7, §3.2]. This implies that (x^k, s^k) asymptotically satisfy the primal equality constraints. Furthermore, the theory also states that $y^k \to y^\star$, an optimal solution to $\mathcal{D}$ [15]. Therefore, we conclude that as $k \to \infty$,

$$A^T y^k + c \to 0.$$

In summary, the iterates (x^k, s^k, y^k) asymptotically satisfy the equality constraints and exactly satisfy the cone constraints and the complementarity condition.

5.2 Stopping conditions

The convergence analysis above tells us that a stopping criterion of the form

$$\|Ax^k + s^k - b\|_2 \leq \epsilon^{\text{primal}}, \qquad \|A^T y^k + c\|_2 \leq \epsilon^{\text{dual}}$$

where $\epsilon^{\text{primal}}, \epsilon^{\text{dual}} > 0$ are tolerances, will eventually be satisfied when a primal-dual solution exists. A typical choice for the tolerances is

$$\epsilon^{\text{primal}} = \epsilon(1 + \|b\|_2), \quad \epsilon^{\text{dual}} = \epsilon(1 + \|c\|_2),$$

where $\epsilon > 0$ corresponds roughly to the required precision. (These are error measures similar to the ones used in the 7$^{\text{th}}$ DIMACS challenge [38].)

We can add an additional condition on the duality gap,

$$c^T x^k + b^T y^k \leq \epsilon^{\text{gap}},$$

where $\epsilon^{\text{gap}} > 0$ is a duality gap tolerance. The quantity on the left is often called the *surrogate gap*, since it is equal to the duality gap only when x^k and y^k exactly satisfy the equality constraints. A typical choice of the tolerance is

$$\epsilon^{\text{gap}} = \epsilon(1 + |c^T x^k| + |b^T y^k|).$$

However, this condition is in some sense redundant, since the surrogate duality gap can be bounded in terms of the primal and dual equality constraint residuals.

6 Implementation

In this section, we discuss a number of different details that are critical to an efficient implementation of the basic method.

6.1 Projecting onto cones

At each iteration, we project $v = \tilde{s}^{k+1} - \lambda y^k$ onto $\mathcal{K}$. We project onto $\mathcal{K}$ by projecting onto $\mathcal{K}_i$ separately and in parallel. Typically, computing the projection onto $\mathcal{K}_i$ can be done with little computational effort, and in many cases analytical solutions exist. We describe some typical ones below.

Projecting v onto the free cone $\mathbf{R}$ gives $s = v$; projecting v onto the zero cone $\{0\}$ gives $s = 0$. Projecting v onto $\mathbf{R}_+$, the cone of nonnegative reals (which is the second-order cone $\mathcal{Q}^0$), only takes the nonnegative part, $s = (v)_+ = \max(v, 0)$. Projecting v onto the second-order cone $\mathcal{Q}^k$ has the closed-form solution

$$\Pi_{\mathcal{Q}^k}(t, v) = \begin{cases} 0 & \|v\|_2 \leq -t, \\ (t, v) & \|v\|_2 \leq t, \\ (\alpha, \alpha v/\|v\|_2) & \text{otherwise}, \end{cases}$$

where $\alpha = (1/2)(\|v\|_2 + t)$. Projecting onto the positive semidefinite cone can be done by eigendecomposition, followed by taking the nonnegative parts of the eigenvalues [50]. For other conic projections (among other proximal operators), see [37].

6.2 Projecting onto equality constraints

At each iteration, we project $(u, v) = (x^k - \lambda c, s^k + \lambda y^k)$ onto $\mathcal{E}$, which has the analytical solution $\Pi_{\mathcal{E}}(u, v) = (x, s)$, where

$$x = \left(I + A^T A\right)^{-1} \left(u + A^T(b - v)\right), \quad s = b - Ax.$$

We describe below two basic methods for doing this as well as a hybrid approach. For other general approaches to solving this linear system, we refer the reader to [20, 42, 49].

6.2.1 Direct method

When A is dense, several standard methods can be used to compute the projection. One approach is to form $I + A^T A$ and compute its Cholesky factorization $I + A^T A = LL^T$, which costs $O(mn^2)$ flops. We then compute x and s as

$$x = L^{-T} L^{-1} \left(u + A^T(b - v)\right), \quad s = b - Ax,$$

where the multiplications by L^{-1} and L^{-T} are carried out by forward and backward substitution, respectively. This costs $O(mn)$ flops. Thus the first projection costs $O(mn^2)$ flops; by caching L, subsequent iterations cost $O(mn)$ flops.

A variation on this, with better numerical properties, is to compute the QR decomposition

$$QR = \begin{bmatrix} A \\ I \end{bmatrix}.$$

and compute the projection using

$$x = R^{-1} R^{-T} \left(u + A^T(b - v)\right), \quad s = b - Ax.$$

The complexity of this method is the same as the one above: $O(mn^2)$ flops for the first projection, and $O(mn)$ flops for subsequent projections.

When A is sparse, we compute the sparse LDL factorization

$$PLDL^T P^T = \begin{bmatrix} I & A \\ A^T & -I \end{bmatrix},$$

where P is a permutation matrix chosen to reduce fill-in, L is a lower triangular matrix, and D a diagonal matrix. (This is in contrast to the more generic LDL factorization in which D must include 2×2 blocks.) This (simpler) factorization is guaranteed to exist for any permutation P, since the coefficient matrix is quasi-definite [19, 43].

The projection can be computed using

$$\begin{bmatrix} z \\ x \end{bmatrix} = P^{-T} L^{-T} D^{-1} L^{-1} P^{-1} \begin{bmatrix} b - v \\ u \end{bmatrix}, \quad s = z + v,$$

where $z = b - v - Ax$ is a dual variable.

Since the coefficient matrix is quasi-definite, the permutation matrix can be chosen based on the sparsity pattern of A and not its entries. Moreover, we can determine the signs of the entries in D before the factorization is computed; this provides a numerically stable LDL solver without the need for dynamic pivoting, allowing us to implement more efficient algorithms. This property is exploited in the code generation system CVXGEN [27] and interior-point solvers such as ECOS [13].

This LDL factorization is carried out *once* and cached; the computational effort of this factorization depends on the fill-in of the factorization, which determines the number of nonzeros entries in L, n_L. The number of nonzeros in L is always at least as many in A, which is n_A. After the factorization is computed, subsequent projections can be computed by a forward and backward substitution, which requires $O(n_L)$ flops. In this case, the ratio of the factor effort to the solve effort is less than m (which is the factor in the dense case), but always more than one.

6.2.2 Indirect method

An indirect method such as conjugate gradients (CG) [24] or LSQR [36], solves for x using an iteration that only requires multiplication by A and by A^T. In theory the iteration terminates with the solution in n steps, but the main value in practice is when an adequate approximation is obtained in far fewer iterations. This can be aided by an appropriately chosen preconditioner $M \approx I + A^T A$. (The preconditioner M is chosen so that finding the solution q to $Mq = r$ is computationally easy.)

Starting with an initial value $x = x_0$, we form

$$r_0 = u - x_0 - A^T(Ax_0 - b + v), \quad q_0 = M^{-1}r_0, \quad p_0 = q_0,$$

and then repeat the iteration

$$\begin{array}{lcl} \alpha_i & = & (q_i^T r_i)/(\|p_i\|_2^2 + \|Ap_i\|_2^2) \\ x_{i+1} & = & x_i + \alpha_i p_i \\ r_{i+1} & = & r_i - \alpha_i \left(p_i + A^T(Ap_i)\right) \\ q_{i+1} & = & M^{-1} r_{i+1} \\ \beta_i & = & (q_{i+1}^T r_{i+1})/(q_i^T r_i) \\ p_{i+1} & = & q_{i+1} + \beta_i p_i, \end{array}$$

until $\|r_{i+1}\|_2$ (which is the norm of the residual) is sufficiently small. Once an x is obtained, we compute $s = b - Ax$. Note that while $(x, s) \in \mathcal{E}$, it is only approximately equal to the Euclidean projection of (u, v) (*i.e.*, the point in $\mathcal{E}$ that is closest to (u, v)).

Each CG iteration requires one multiplication of a vector by A and one by A^T. Thus each CG step has a complexity of $O(n_A)$, where n_A is the number of nonzero entries in A. (The matrix-vector multiplies can be distributed.)

Although an indirect method only yields an approximation to the Euclidean projection, ADMM convergence theory tells us that if the projection error is bounded by a summable sequence, then the algorithm will converge [15]. One strategy is to require a CG tolerance

of $1/k^2$ at every iteration. We, however, employ a heuristic strategy: we fix the maximum number of CG iterations and warm start CG with $x_0 = x^k$, the previous PDOS solution of x. This yields inaccurate projections at first, but more accurate ones as ADMM begins to converge. It is typical for each step to involve a number of iterations ranging from tens (at the beginning) to just a few (at the end). We emphasize that this heuristic *does not* guarantee convergence in theory, although the algorithm terminates in practice (and can be made to converge in theory by tightening the tolerances after a fixed number of PDOS iterations).

6.2.3 Hybrid method

In the previous section, if we choose the preconditioner $M = I$ (*i.e.*, no preconditioning whatsoever), CG requires at most n iterations to solve the linear system; at the other extreme, with $M = I + A^TA$, CG requires only one iteration to converge to high accuracy, reducing the indirect method to a direct one. Therefore, we can hybridize the two approaches by choosing preconditioners that are more similar to $I + A^TA$ (a direct method) or more similar to I (an indirect method).

Some possibilities for M are $M = \mathbf{diag}(I + A^TA)$ (Jacobi preconditioner), the diagonal *blocks* of $I + A^TA$, or the incomplete Cholesky factorization of $I + A^TA$ [42]. When the full Cholesky factorization is used, the indirect and direct method agree.

6.3 Problem scaling

Although PDOS converges in theory for any choice of λ, it has been observed that problem scalings and preconditioning can affect the convergence rate in practice [18, 39]. Instead of solving $\mathcal{P}$, we solve an equivalent problem (denoted $\hat{\mathcal{P}}$) with variables $\hat{x}$, $\hat{s}$, and $\hat{y}$,

$$\begin{array}{ll} \text{minimize} & \hat{c}^T\hat{x} \\ \text{subject to} & \hat{A}\hat{x} + \hat{s} = \hat{b} \\ & \hat{s} \in \mathcal{K}. \end{array}$$

This problem is solved with data $\mathcal{K}$, $\hat{A} = DAE$, $\hat{b} = Db$, and $\hat{c} = Ec$, where $D \in \mathbf{R}^{m\times m}$ and $E \in \mathbf{R}^{n\times n}$ are diagonal matrices of the form

$$D = \mathbf{diag}(\pi_1 I_{m_1}, \ldots, \pi_q I_{m_q}), \quad E = \mathbf{diag}(\delta),$$

with $\pi = (\pi_1, \ldots, \pi_q) \in \mathbf{R}^q_{++}$ and $\delta \in \mathbf{R}^n_{++}$, and where I_n is the $n \times n$ identity matrix. The variables (x, s, y) can be related to $(\hat{x}, \hat{s}, \hat{y})$ via the change of variables

$$x = E\hat{x}, \quad s = D^{-1}\hat{s}, \quad y = D\hat{y}.$$

With this change of variables, the optimality conditions of $\mathcal{P}$ and $\hat{\mathcal{P}}$ are equivalent.

We apply the PDOS algorithm to $\hat{\mathcal{P}}$, but evaluate the stopping conditions in terms of the original variables. Simple algebra will verify that the following stopping criterion is equivalent to the stopping criterion as measured in the original variables,

$$\|D^{-1}(\hat{A}\hat{x}^k + \hat{s}^k - \hat{b})\|_2 \leq \epsilon^{\text{primal}}, \qquad \|E^{-1}(\hat{A}\hat{y}^k + \hat{c})\|_2 \leq \epsilon^{\text{dual}},$$

$$|\hat{c}^T\hat{x}^k + \hat{b}^T\hat{y}^k| \leq \epsilon^{\text{gap}}.$$

We now turn our attention to the special case when $D = I$ and $E = (1/\sqrt{\mu})I$, with $\mu > 0$. In this case, the first step of PDOS can be evaluated in terms of the original coordinates as

$$x^{k+1} = \operatorname*{argmin}_x \left(c^T x + (\mu/2\lambda)\|x - x^k\|_2^2 + (1/2\lambda)\|b - Ax - s^k - \lambda y^k\|_2^2\right),$$

and $\tilde{s}^k = b - Ax^{k+1}$. This means that μ (and, more generally, E) controls the ratio between the two quadratic penalties.

Choice of D and E. Empirically, we have observed that choosing D and E to equilibrate the norms of the rows and columns of $\hat{A} = DAE$ appears to reduce the number of iterations needed to obtain an ϵ-solution. It also has the additional effect of reducing the condition number of $I + \hat{A}^T\hat{A}$. Many algorithms exist to equilibrate or balance matrices, specifically in the context of matrix preconditioners [35, 41]. We present one such algorithm here, although many others can be used.

We first group together the rows of A to form $\tilde{A} \in \mathbf{R}^{q\times n}$ as follows,

$$\tilde{A}_{ij}^2 = (1/|\kappa_i|)\sum_{k\in\kappa_i} A_{kj}^2$$

where κ_i is the index set corresponding to the elements of s which are in $\mathcal{K}_i$, and so $|\kappa_i| = m_i$. We equilibrate the rows and columns of $\tilde{A}$ to obtain the scaling vectors $\pi \in \mathbf{R}^q$ and $\delta \in \mathbf{R}^n$:

$$\pi_i = \left(\sum_{j=1}^n \tilde{A}_{ij}^2\right)^{-1/2},$$
$$\delta_j = \left(\sum_{i=1}^q \tilde{A}_{ij}^2\pi_i^2\right)^{-1/2}.$$

We then construct D and E and scale the data *once* before running the PDOS algorithm.

Note that this is equivalent to applying diagonal preconditioners, such as ones proposed in [39], to the problem. However, we have found that this matrix equilibration appears to yield better results for second-order cone programs than the ones described in [39] (which are designed for linear programs). More sophisticated preconditioners can (and should) be designed to improve the convergence behavior of PDOS.

Choice of λ. With $\hat{A}$ equilibrated and the problem data scaled, a good choice of λ appears to be

$$\lambda = \|\hat{b}\|_2/\|\hat{c}\|_2.$$

Intuitively, this choice of λ attempts to balance $\|\hat{s}^k\|_2$ (which, when $\hat{A}$ is equilibrated, is on the order of $\|\hat{b}\|_2$) and $\lambda\|\hat{y}^k\|_2$ (which is on the order of $\|\hat{c}\|_2$). There is no reason to believe that this is the best choice of λ, although it seems to perform well in our experiments in §7.

6.4 Parallel and distributed implementations

The main advantage of PDOS is that each step of the algorithm is simple and easily parallelized or distributed, allowing PDOS to scale to multiprocessing environments such as compute clusters. Cone projections are trivially parallelized (by projecting onto $\mathcal{K}_i$ separately and in parallel), and second-order cone projections can be further subdivided by exploiting the property

$$\|(x, y)\|_2 = \|(\|x\|_2, \|y\|_2)\|_2,$$

thus reducing large, second-order cones into smaller, more manageable cones.

Any of the approaches—the direct, indirect, or hybrid method—for projecting onto the linear system can be parallelized by leveraging parallel algorithms for LDL factorization and sparse matrix-vector multiplication. For instance, Elemental / Clique can be used to distribute the direct solver on distributed memory machines [40], while NVIDIA's CUDA could be used to accelerate the indirect solver [33].

7 Numerical examples

The C source, along with a Python extension module and CVX shim, are available for download on Github at `http://www.github.com/cvxgrp/pdos`. All numerical examples are run in Matlab on a 4-core, 3.4GHz Intel Xeon processor with hyperthreading and 16GB of RAM.

We begin with a description of our experimental setup in §7.1, which is the same for all examples. We then describe the family of portfolio optimization problems in §7.2 and report several numerical results, including an example of warmstarting in §7.2.4 by tracing out a risk-return tradeoff curve. We also report the results of PDOS on a family of ℓ_1-regularized least-squares problems in §7.3. Finally, we run PDOS on the DIMACS benchmarks and compare its results to SeDuMi in §7.4.

7.1 Experimental setup

For our experiments, we compare both the direct and indirect PDOS solver against SeDuMi (version 1.21) [48]. Unless otherwise specified, we use default SeDuMi tolerances. SeDuMi utilizes a multithreaded BLAS, so in our case, it can use up to eight independent threads. This is in contrast to our implementation of the direct and indirect PDOS solvers, which are both single-threaded. In our implementation of the direct and indirect PDOS solvers, we use the (default) values of $\epsilon = 10^{-3}$ and use an overrelaxation parameter of $\alpha = 1.8$. For the direct solver, we use AMD to select the permutation and use a simple LDL to solve the subsequent linear system [2, 3, 10–12]. For the indirect solver, we use our own conjugate gradient algorithm with a maximum of 10 CG iterations. Our CG code has a (default) tolerance of 10^{-4}.

7.2 Portfolio optimization

We consider a simple long-only portfolio optimization problem [8, p. 185–186], where we choose relative weights of assets to maximize risk-adjusted return. The problem is

$$\begin{array}{ll} \text{maximize} & \mu^T x - \gamma(x^T \Sigma x) \\ \text{subject to} & \mathbf{1}^T x = 1, \quad x \geq 0, \end{array}$$

where the variable $x \in \mathbf{R}^n$ represents the portfolio, $\mu \in \mathbf{R}^n$ is the vector of expected returns, $\gamma > 0$ is the risk-aversion parameter and $\Sigma \in \mathbf{R}^{n \times n}$ is the asset return covariance, given in factor model form,

$$\Sigma = FF^T + D.$$

Here $F \in \mathbf{R}^{n \times m}$ is the factor-loading matrix, and $D \in \mathbf{R}^{n \times n}$ is a diagonal matrix (of *idiosyncratic risk*). The number of factors in the risk model is m, which we assume is substantially smaller than n, the number of assets.

This problem can be converted in the standard way (say, via a parser-solver such as CVX [22, 23]) into an SOCP,

$$\begin{array}{ll} \text{maximize} & \mu^T x - \gamma(t + s) \\ \text{subject to} & \mathbf{1}^T x = 1, \quad x \geq 0 \\ & \|D^{1/2} x\|_2 \leq u, \quad \|F^T x\|_2 \leq v \\ & \|(1 - t, 2u)\|_2 \leq 1 + t \\ & \|(1 - s, 2v)\|_2 \leq 1 + s \end{array}$$

with variables $x \in \mathbf{R}^n$, $t \in \mathbf{R}$, $s \in \mathbf{R}$, $u \in \mathbf{R}$, and $v \in \mathbf{R}$.

7.2.1 Problem instances

We solve the portfolio problem in four different sizes: small ($m = 10$, $n = 100$), medium ($m = 30$, $n = 1000$), large ($m = 100$, $n = 10000$), and massive ($m = 300$, $n = 100000$). These names only correspond to their sizes relative to each other. For all problems, we choose $\gamma = n$.

7.2.2 Convergence

Figure 1 shows the convergence plot of a large portfolio problem ($m = 100$ and $n = 10000$) using a direct method. Note that the primal and dual residuals and the pseudo-gap are plotted on an *absolute* scale. The plots show convergence behavior as a function of the iterates. The linear convergence rate in Figure 1 is atypical and is a result of the regularity structure in the portfolio problem.[1] Typical PDOS convergence behavior is slow to achieve high accuracy, although significant progress is made early on. Depending on the application, if acceptable, a low-accuracy solution with a certifiable (pseudo-)gap can be found fairly quickly.

[1] It is suspected that PDOS automatically exploits the underlying strong concavity of the objective in the portfolio problem even when transformed to a *linear* cone program, although we did not explore this further.

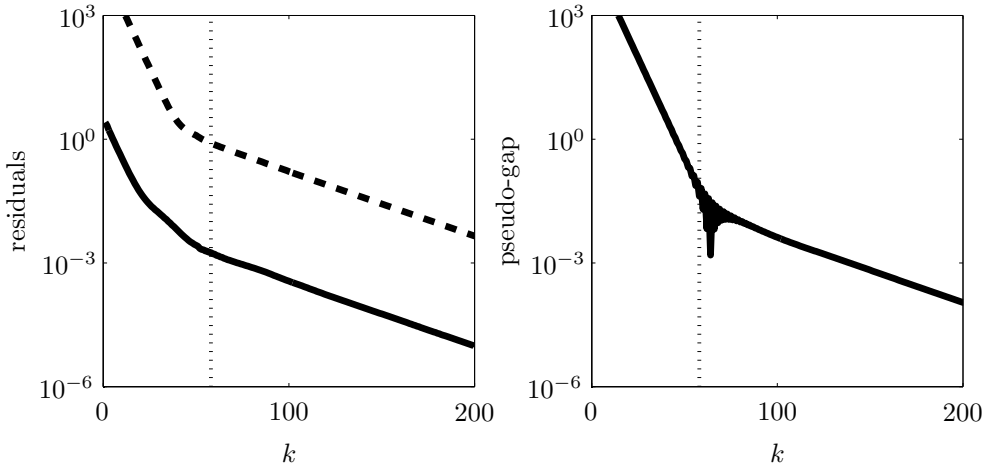

Figure 1: The convergence of the primal residual $\|Ax^k + s^k - b\|_2$ (left, solid) and dual residual $\|A^T y^k + c\|_2$ (left, dashed), along with the pseudo-gap $|c^T x^k + b^T y^k|$ (right) for the large portfolio problem using a direct method. The vertical dotted line in all plots shows where the stopping conditions are satisfied (at iteration 58).

factors (m)	assets (n)	SeDuMi	PDOS (d)	PDOS (i)
10	100	0.163s	0.001s	0.001s
30	1000	0.276s	0.016s	0.026s
100	10000	5.86s	0.812s	1.299s
300	100000	328.5s	58.73s	81.17s

Table 1: Comparison of average runtime (over 10 problem instances) between SeDuMi (multithreaded), PDOS with a direct solver (single-threaded, denoted with '(d)'), and PDOS with an indirect solver (single-threaded, denoted with '(i)') for solving portfolio problem instances.

7.2.3 Timing results

For each problem size, we solve 10 different instances and report the average time needed to solve the problem instance with SeDuMi (using CVX) [23, 48], PDOS with a direct solver, and PDOS with an indirect solver. The timing results are summarized in Table 1. SeDuMi is run with a lower tolerance (10^{-3}) to make a fair comparison with PDOS. In all instances, the average relative error of PDOS's reported objective value is within 1% of the optimal (as found by SeDuMi at a higher precision).

7.2.4 Warm start

In portfolio optimization (§7.2) it may be desirable to plot a tradeoff curve as the parameter γ is varied. In these situations—when only the problem *data* has changed—warm starting PDOS yields significant benefits.

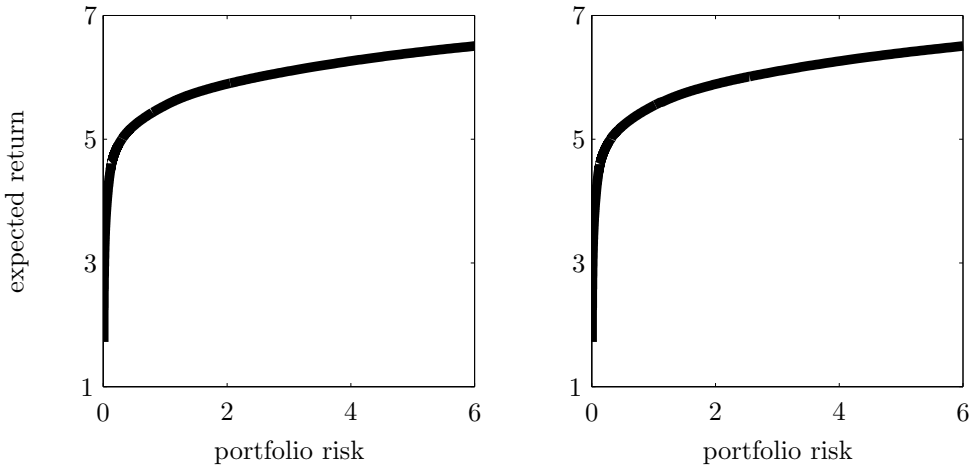

Figure 2: Portfolio tradeoff curve as computed by SeDuMi (left) and PDOS (right).

To illustrate the effect of warm starting, we solve the small portfolio problem ($m = 10$, $n = 100$) with 1000 different values of γ, evenly spaced on a log scale between 10^{-1} and 10^3. (This scenario occurs when determining the tradeoff between risk and return.) We then solve the same sequence of problems with SeDuMi.

Figure 2 shows the resulting tradeoff curve when solved via SeDuMi and PDOS. Qualitatively, these two curves agree. SeDuMi took 3.1 minutes to compute this tradeoff curve while direct PDOS, because of warmstarting, took 5.1 seconds, and indirect PDOS method took 3.5 seconds, almost a factor of 50 in savings.

7.3 ℓ_1-regularized least-squares

The ℓ_1-regularized least-squares, or lasso, problem is

$$\text{minimize} \quad \|Ax - b\|_2^2 + \gamma\|x\|_1,$$

with variable $x \in \mathbf{R}^n$, data $A \in \mathbf{R}^{m\times n}$, and $b \in \mathbf{R}^m$. Typically, $n \gg m$, where m is the number of observations. The parameter $\gamma > 0$ (typically called λ, but renamed to avoid confusion with the PDOS algorithm parameter) trades off the strength of regularization with the goodness of fit. In particular, if $\gamma = 0$, then the solution will be dense; if $\gamma > \gamma_{\max} = \|2A^T b\|_\infty$, then the solution will be trivial. In the interval $[0, \gamma_{\max}]$, the solution is typically (very) sparse.

The lasso problem can be converted in the standard way (say, via a parser-solver such as CVX [22, 23]) into an SOCP,

$$\begin{array}{ll} \text{minimize} & t + \gamma \mathbf{1}^T s \\ \text{subject to} & -s \le x \le s \\ & \|Ax - b\|_2 \le u \\ & \|(1 - t, 2u)\|_2 \le 1 + t \end{array}$$

observations (m)	variables (n)	SeDuMi	PDOS (d)	PDOS (i)
250	5000	13.38s	12.74s	16.82s
1250	10000	418.5s	113.7s	132.1s
2500	50000	12631s	2144s	2144s

Table 2: Comparison of average runtime (over 10 problem instances) between SeDuMi (multithreaded), PDOS with a direct solver (single-threaded, denoted with '(d)'), and PDOS with an indirect solver (single-threaded, denoted with '(i)') for solving lasso problem instances.

with variables $x \in \mathbf{R}^n$, $t \in \mathbf{R}$, $s \in \mathbf{R}^n$, and $u \in \mathbf{R}$.

7.3.1 Problem instances

We solve the lasso problem in three different sizes: small ($m = 250$, $n = 5000$), medium ($m = 1250$, $n = 10000$), and large ($m = 2500$, $n = 50000$). For all problems, we choose $\gamma = 0.01\gamma_{\max}$. The large problem is chosen such that the lasso problem has approximately 1GB of nonzeros.

7.3.2 Timing results

For each problem size, we solve 10 different instances and report the average time needed to solve the problem instance with SeDuMi (using CVX) [23, 48], PDOS with a direct solver, and PDOS with an indirect solver. The timing results are summarized in Table 2. SeDuMi is run with a lower tolerance (10^{-3}) to make a fair comparison with PDOS. In all instances, the average relative error of PDOS's reported objective value is within 1% of the optimal (as found by SeDuMi at a higher precision).

7.4 DIMACS benchmarks

In this section, we present the results of running our algorithm on the 7^{th} DIMACS Challenge problems [38]. Table 3 summarizes the results.

With the maximum number of iterations set to 10000, PDOS is able to solve 12 of the 16 challenge problems to within 1% accuracy, requiring fewer than 1500 iterations on average to converge to the desired tolerance.

The problems for which PDOS fails result from poor problem scaling and is hardly surprising, since PDOS is a first-order method and, like other first-order methods, is sensitive to problem scalings. It also illustrates the limitations of our matrix equilibration routine: a well-chosen scaling is superior to the automatic scaling selected by matrix equilibration. More interestingly, SeDuMi also runs into numerical problems with these problems as well. It is unsurprising that in some cases, SeDuMi outperforms PDOS—especially considering that SeDuMi is multithreaded and PDOS is single-threaded—but the benchmark demonstrates our simple algorithm appears to sufficiently solve a range of SOCPs to modest accuracy.

	runtime			objective value		
name	SeDuMi	PDOS (d)	PDOS (i)	SeDuMi	PDOS (d)	PDOS (i)
nb	2.1s	1.1s	4.1s	-0.0507	-0.0503	-0.0503
nb_L1	1.4s	5.9s	21.6s	-13.012	-12.920	-12.922
nb_L2	1.3s	1.6s	7.5s	-1.629	-1.625	-1.625
nb_L2_bessel	0.9s	0.1s	0.5s	-0.103	-0.102	-0.102
nql30	0.3s	0.1s	0.3s	-0.946	-0.945	-0.945
nql60	1.1s	1.5s	3.4s	-0.935	-0.933	-0.956
qssp30	0.5s	0.4s	1.0s	-6.497	-6.489	-6.492
qssp60	2.8s	4.7s	48.1s	-6.563	-6.556	-6.563
sched_50_50_orig	0.9s	4.3s	11.0s	2.67×10^4	$2.46 \times 10^{4*}$	$2.46 \times 10^{4*}$
sched_50_50_scaled	0.5s	3.2s	5.1s	7.852	7.852	7.852^*
sched_100_50_orig	1.6s	9.3s	22.1s	—	—	—
sched_100_50_scaled	1.3s	9.7s	19.0s	67.17	66.75^*	66.75^*
sched_100_100_orig	6.7s	17.4s	43.2s	—	—	—
sched_100_100_scaled	2.9s	18.4s	43.0s	27.33	27.52^*	27.52^*
sched_200_100_orig	17.6s	45.3s	114.1s	—	—	—
sched_200_100_scaled	10.1s	47.4s	121.8s	51.81	51.76^*	51.77^*

Table 3: DIMACS test problems solved with SeDuMi (multithreaded) and PDOS (single-threaded direct, denoted with '(d)', and single-threaded indirect, denoted with '(i)'). Failed problems are marked with a dash; PDOS objective values marked with an asterisk are the reported values after PDOS took the maximum number of iterations.

7.5 Summary

From the portfolio problems, lasso problems, and DIMACS benchmarks, we see that the direct PDOS method not only produces competitive solutions, but can do so at a fraction of the time needed by SeDuMi (and a fraction of the computing resources; recall that PDOS is single-threaded). Even without threading or parallelization, our PDOS implementation is competitive with SeDuMi: the largest lasso problem instance is solved in 3.5 hours with SeDuMi, while direct PDOS solved the same problem in 35 minutes. The performance of the direct PDOS method can be improved by using a large-scale linear system solver such as PARDISO [44–47], which can reduce the factorization times. For instance, the largest lasso problem is solved in about 22 minutes with PARDISO, with the majority of savings coming from the reduction of the initial factor time.

As problem sizes grow, the indirect PDOS method becomes more viable, although its performance is highly dependent on the chosen matrix equilibration routine. If the equilibration results in a small condition number, then each iteration of PDOS requires two or three CG steps to converge to the desired CG accuracy. Otherwise, more CG steps may be required and the total runtime increased.

While these problems fit in the shared-memory of a parallel machine, the real advantage of PDOS is it ability to leverage distributed or parallel linear algebra to solve larger problems.

The DIMACS problems illustrate the importance of problem scaling and equilibration. Further research is required to determine good equilibration routines in these contexts.

8 Conclusion

In this paper, we have presented an operator-splitting applied to the primal cone problem $\mathcal{P}$, PDOS—a first-order penalty-method for conic optimization. Incidentally, the PDOS iterates not only solve the primal, but also the dual. The PDOS algorithm is guaranteed to converge (provided a solution exists). With the proper scaling and choice of λ, the algorithm converges to modest accuracy in several hundreds of iterations. If more iterations can be afforded, this simple algorithm can solve a wide range of SOCPs.

More importantly, the (generic) algorithm can scale to large problems by leveraging parallel sparse direct or sparse iterative solvers on distributed- or shared-memory machines (*e.g.*, Clique, Elemental, PARDISO, *etc.* [40, 44–47]).

Since many convex optimization problems can be reformulated as cone programs, this approach lets us solve a number of (large-scale) convex optimization problems and provides the groundwork for developing massively parallel and distributed, conic solvers. We hope that PDOS' simplicity along with our implementation may spur interested readers in developing both theory and code for *general* first-order conic solvers.

Acknowledgments

The authors would like to thank AJ Friend, O. Konings, A. Domahidi, and C. De Sa for helpful discussions about the algorithm. In particular, O. Konings and C. De Sa found bugs in early implementations of our code; AJ Friend provided helpful comments in an early draft of this paper. This work was supported by DARPA XDATA (FA8750-12-2-0306) and NASA (NNX07AEIIA). N. Parikh was also supported by a NSF Graduate Research Fellowship (DGE-0645962).

References

[1] F. Alizadeh and D. Goldfarb. Second-order cone programming. *Mathematical Programming Series B*, 95:3–51, 2003.

[2] P. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, Dec 1996.

[3] P. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, An approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 30(3):381–388, Sep 2004.

[4] N. S. Aybat and G. Iyengar. An augmented lagrangian method for conic convex programming. *Preprint*, 2013. `http://arxiv.org/pdf/1302.6322v1.pdf`.

[5] S. R. Becker, E. J. Candès, and M. C. Grant. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical Programming Computation*, pages 1–54, 2010.

[6] A. Ben-Tal and A. Nemirovski. *Lectures on Modern Convex Optimization. Analysis, Algorithms, and Engineering Applications*. Society for Industrial and Applied Mathematics, 2001.

[7] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3:1–122, 2011.

[8] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[9] A. Chambolle and T. Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of Mathematical Imaging and Vision*, 40(1):120–145, may 2011.

[10] T. A. Davis. Algorithm 849: A concise sparse cholesky factorization package. *ACM Transactions on Mathematical Software*, 31(4):587–591, Dec 2005.

[11] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Sep 2006.

[12] T. A. Davis. Summary of available software for sparse direct methods, Apr 2012. `http://www.cise.ufl.edu/research/sparse/codes/codes.pdf`.

[13] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP solver for embedded systems. In *European Control Conference*, 2013. To appear.

[14] J. Eckstein. Parallel alternating direction multiplier decomposition of convex programs. *Journal of Optimization Theory and Applications*, 80(1):39–62, 1994.

[15] J. Eckstein and D. P. Bertsekas. On the Douglas-Rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming*, 55:293–318, 1992.

[16] J. Eckstein and M. C. Ferris. Operator-splitting methods for monotone affine variational inequalities, with a parallel application to optimal control. *INFORMS Journal on Computing*, 10(2):218–235, 1998.

[17] K. Fountoulakis, J. Gondzio, and P. Zhlobich. Matrix-free interior point method for compressed sensing problems. *Preprint*, 2012. `http://arxiv.org/pdf/1208.5435.pdf`.

[18] E. Ghadimi, A. Teixeira, I. Shames, and M. Johansson. Optimal parameter selection for the alternating direction method of multipliers (admm): quadratic problems. *Preprint*, 2013. `http://arxiv.org/pdf/1306.2454.pdf`.

[19] P. E. Gill, M. A. Saunders, and J. R. Shinnerl. On the stability of Cholesky factorization for symmetric quasidefinite systems. *SIAM Journal of Matrix Analysis and Applications*, 17(1):35–46, 1996.

[20] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

[21] J. Gondzio. Convergence analysis of an inexact feasible interior point method for convex quadratic programming. *Preprint*, 2012. `http://arxiv.org/pdf/1208.5960.pdf`.

[22] M. Grant, S. Boyd, and Y. Ye. Disciplined convex programming. In L. Liberti and N. Maculan, editors, *Global Optimization: From Theory to Implementation*, Nonconvex Optimization and its Applications, pages 155–210. Springer, 2006.

[23] M. Grant, S. Boyd, and Y. Ye. CVX: Matlab software for disciplined convex programming, ver. 2.0, build 870. Available at `www.stanford.edu/~boyd/cvx/`, September 2012.

[24] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Research of the National Bureau of Standards*, 49(6), 1952.

[25] G. Lan, Z. Lu, and R. Monteiro. Primal-dual first-order methods with $\mathcal{O}(1/\epsilon)$ iteration-complexity for cone programming. *Math. Program.*, 126(1):1–29, January 2011.

[26] M. S. Lobo, L. Vandenberghe, S. Boyd, and H. Lebret. Applications of second-order cone programming. *Linear Algebra and Its Applications*, 284:193–228, 1998.

[27] J. Mattingley and S. Boyd. CVXGEN: A code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, 2012.

[28] R. Monteiro, C. Ortiz, and B. Svaiter. An adaptive accelerated first-order method for convex optimization. `http://www2.isye.gatech.edu/~monteiro/publications/tech_reports/AA_method.pdf`, 2012. Manuscript.

[29] Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. *Soviet Mathematics Doklady*, 27(2):372–376, 1983.

[30] Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer, 2003.

[31] Y. Nesterov. Gradient methods for minimizing composite functions. *Mathematical Programming*, pages 1–37, 2010.

[32] Y. Nesterov and A. S. Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*, volume 13. Society for Industrial and Applied Mathematics, 1994.

[33] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[34] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Science, 2006.

[35] E. E. Osborne. On pre-conditioning of matrices. *Journal of the ACM*, 7(4):338–345, Oct 1960.

[36] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *TOMS*, 8(1):43–71, 1982.

[37] N. Parikh and S. Boyd. Proximal algorithms. *Foundations and Trends in Optimization*, 2013.

[38] G. Pataki and S. Schmieta. The DIMACS library of mixed semidefinite-quadratic-linear programs. Available at `dimacs.rutgers.edu/Challenges/Seventh/Instances`.

[39] T. Pock and A. Chambolle. Diagonal preconditioning for first order primal-dual algorithms in convex optimization. In *IEEE International Conference on Computer Vision (ICCV)*, pages 1762–1769, 2011.

[40] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 39(2):13:1–13:24, February 2013.

[41] D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical report, Rutherford Appleton Laboratories, 2001.

[42] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.

[43] M. A. Saunders. Cholesky-based methods for sparse least squares: the benefits of regularization. In L. Adams and J. L. Nazareth, editors, *Linear and Nonlinear Conjugate Gradient-Related Methods*, pages 92–100. SIAM, Philadelphia, 1996.

[44] O. Schenk, M. Bollhöfer, and R. A. Römer. On large-scale diagonalization techniques for the Anderson model of localization. *SIAM Review*, 50(1):91–112, 2008.

[45] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[46] O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. *ETNA. Electronic Transactions on Numerical Analysis*, 23:158–179, 2006.

[47] O. Schenk, A. Wächter, and M. Hagemann. Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. *Computational Optimization and Applications*, 36(2-3):321–341, 2007.

[48] J. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11:625–653, 1999. Software available at `http://sedumi.mcmaster.ca`.

[49] L. N. Trefethen and D. Bau. *Numerical linear algebra*. Society for Industrial and Applied Mathematics, 1997.

[50] Z. Wen, D. Goldfarb, and W. Yin. Alternating direction augmented lagrangian methods for semidefinite programming. *Mathematical Programming Computation*, 2(3-4):203–230, 2010.

[51] S. J. Wright. *Primal-Dual Interior-Point Methods*, volume 54. Society for Industrial and Applied Mathematics, 1987.

[52] X. Xu, P.-F. Hung, and Y. Ye. A simplified homogeneous and self-dual linear programming algorithm and its implementation. *Annals of Operations Research*, 62(1):151–171, 1996.

[53] Y. Ye. *Interior Point Algorithms: Theory and Analysis*. Wiley-Interscience, 2011.

[54] Y. Ye, M. Todd, and S. Mizuno. An $O(\sqrt{n}L)$-iteration homogeneous and self-dual linear programming algorithm. *Mathematics of Operations Research*, 19(1):53–67, Feb 1994.