Convex Optimization with Abstract Linear Operators

Steven Diamond and Stephen Boyd
Dept. of Computer Science and Electrical Engineering, Stanford University
{stevend2, boyd}@stanford.edu

Abstract

We introduce a convex optimization modeling framework that transforms a convex optimization problem expressed in a form natural and convenient for the user into an equivalent cone program in a way that preserves fast linear transforms in the original problem. By representing linear functions in the transformation process not as matrices, but as graphs that encode composition of abstract linear operators, we arrive at a matrix-free cone program, i.e., one whose data matrix is represented by an abstract linear operator and its adjoint. This cone program can then be solved by a matrix-free cone solver. By combining the matrix-free modeling framework and cone solver, we obtain a general method for efficiently solving convex optimization problems involving fast linear transforms.

1. Introduction

Convex optimization modeling systems like YALMIP [38], CVX [28], CVXPY [16], and Convex.jl [47] provide an automated framework for converting a convex optimization problem expressed in a natural human-readable form into the standard form required by a generic solver, calling the solver, and transforming the solution back to the human-readable form. This allows users to form and solve convex optimization problems quickly and efficiently. These systems easily handle problems with a few thousand variables, as well as much larger problems (say, with hundreds of thousands of variables) with enough sparsity structure, which generic solvers can exploit.

The overhead of the problem transformation, and the additional variables and constraints introduced in the transformation process, result in longer solve times than can be obtained with a custom algorithm tailored specifically for the particular problem. Perhaps surprisingly, the additional solve time (compared to a custom solver) for a modeling system coupled to a generic solver is often not as much as one might imagine, at least for modest sized problems. In many cases the convenience of easily expressing the problem makes up for the increased solve time using a convex optimization modeling system.

Many convex optimization problems in applications like signal and image processing, or medical imaging, involve hundreds of thousands or many millions of variables, and so are well out of the range that current modeling systems can handle. There are two reasons for this. First, the standard form problem that would be created is too large to store on a single machine, and second, even if it could be stored, standard interior-point solvers would be too slow to solve it. Yet many of these problems are readily solved on a single machine by custom solvers, which exploit fast linear transforms in the problems. The key to these custom solvers is to directly use the fast transforms, never forming the associated matrix. For this reason these algorithms are sometimes referred to as *matrix-free* solvers.

The literature on matrix-free solvers in signal and image processing is extensive; see, *e.g.*, [3, 4, 10, 9, 25, 49]. There has been particular interest in matrix-free solvers for LASSO and basis pursuit denoising problems [4, 11, 22, 18, 31, 48]. The most general matrix-free solvers target semidefinite programs [32] or quadratic programs and related problems [43, 26]. The software closest to a convex optimization modeling system for matrix-free problems is TFOCS, which allows users to specify many types of convex problems and solve them using a variety of matrix-free first-order methods [5].

To better understand the advantages of matrix-free solvers, consider the nonnegative deconvolution problem

$$\begin{array}{ll} \text{minimize} & \|c * x - b\|^2 \\ \text{subject to} & x \geq 0, \end{array} \tag{1}$$

where $x \in \mathbf{R}^n$ is the optimization variable, $c \in \mathbf{R}^n$ and $b \in \mathbf{R}^{2n-1}$ are problem data, and $*$ denotes convolution. Note that the problem data has size $O(n)$. There are many custom matrix-free methods for efficiently solving this problem, with $O(n)$ memory and a few hundred iterations, each of which costs $O(n \log n)$ floating point operations (flops). It is entirely practical to solve instances of this problem of size $n = 10^7$ on a single computer [36].

Existing convex optimization modeling systems fall far short of the efficiency of matrix-free solvers on problem (1).

These modeling systems target a standard form in which a problem's linear structure is represented as a sparse matrix. As a result, linear functions must be converted into explicit matrix multiplication. In particular, the operation of convolving by c will be represented as multiplication by a $(2n-1) \times n$ Toeplitz matrix C. A modeling system will thus transform problem (1) into the problem

$$\begin{array}{ll} \text{minimize} & \|Cx - b\|^2 \\ \text{subject to} & x \geq 0, \end{array} \tag{2}$$

as part of the conversion into standard form.

Once the transformation from (1) to (2) has taken place, there is no hope of solving the problem efficiently. The explicit matrix representation of C requires $O(n^2)$ memory. A typical interior-point method for solving the transformed problem will take a few tens of iterations, each requiring $O(n^3)$ flops. For this reason existing convex optimization modeling systems will struggle to solve instances of problem (1) with $n = 10^4$, and when they are able to solve the problem, they will be dramatically slower than custom matrix-free methods.

The key to matrix-free methods is to exploit fast algorithms for evaluating a linear function and its adjoint. We call an implementation of a linear function that allows us to evaluate the function and its adjoint a *forward-adjoint oracle* (FAO). In this paper we describe a new algorithm for converting convex optimization problems into standard form while preserving fast linear transforms. The algorithm expresses the standard form's linear structure as an abstract linear operator (specifically, a graph of FAOs) rather than as an explicit sparse matrix.

Our new algorithm yields a convex optimization modeling system that can take advantage of fast linear transforms, and can be used to solve large problems such as those arising in image and signal processing and other areas, with millions of variables. This allows users to rapidly prototype and implement new convex optimization based methods for large-scale problems. As with current modeling systems, the goal is not to attain (or beat) the performance of a custom solver tuned for the specific problem; rather it is to make the specification of the problem straightforward, while increasing solve times only moderately.

Due to space limitations, we cannot give the full details of our approach. A longer paper still in development contains the details, as well as additional references and numerical examples [15].

2. Forward-adjoint oracles

2.1. Definition

A general linear function $f : \mathbf{R}^n \to \mathbf{R}^m$ can be represented on a computer as a dense matrix $A \in \mathbf{R}^{m \times n}$ using $O(mn)$ bytes. We can evaluate $f(x)$ on an input $x \in \mathbf{R}^n$ in $O(mn)$ flops by computing the matrix-vector multiplication Ax. We can likewise evaluate the adjoint $f^*(y) = A^T y$ on an input $y \in \mathbf{R}^m$ in $O(mn)$ flops by computing $A^T y$.

Many linear functions arising in applications have structure that allows the function and its adjoint to be evaluated in fewer than $O(mn)$ flops or using fewer than $O(mn)$ bytes of data. The algorithms and data structures used to evaluate such a function and its adjoint can differ wildly. It is thus useful to abstract away the details and view linear functions as *forward-adjoint oracles* (FAOs), *i.e.*, a tuple $\Gamma = (f, \Phi_f, \Phi_{f^*})$ where f is a linear function, Φ_f is an algorithm for evaluating f, and Φ_{f^*} is an algorithm for evaluating f^*. For simplicity we assume that the algorithms Φ_f and Φ_{f^*} in an FAO read from an input array and write to an output array (which can be the same as the input array). We use n to denote the length of the input array and m to denote the length of the output array.

While we focus on linear functions from $\mathbf{R}^n$ into $\mathbf{R}^m$, the same techniques can be used to handle linear functions involving complex arguments or values, *i.e.*, from $\mathbf{C}^n$ into $\mathbf{C}^m$, from $\mathbf{R}^n$ into $\mathbf{C}^m$, or from $\mathbf{C}^n$ into $\mathbf{R}^m$, using the standard embedding of complex n-vectors into real $2n$-vectors. This is useful for problems in which complex data arise naturally (*e.g.*, in signal processing and communications), and also in some cases that involve only real data, where complex intermediate results appear (typically via an FFT).

2.2. Examples

In this section we describe some useful FAOs. In many of the examples the domain or range are naturally viewed as matrices or Cartesian products rather than as vectors in $\mathbf{R}^n$ and $\mathbf{R}^m$. Matrices are treated as vectors by stacking the columns into a single vector; Cartesian products are treated as vectors by stacking the components. For the purpose of determining the adjoint, we still regard these FAOs as functions from $\mathbf{R}^n$ into $\mathbf{R}^m$.

Multiplication by a sparse matrix. Multiplication by a sparse matrix $A \in \mathbf{R}^{m \times n}$, *i.e.*, a matrix with many zero entries, is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f(x) = Ax$. The adjoint $f^*(u) = A^T u$ is also multiplication by a sparse matrix. The algorithms Φ_f and Φ_{f^*} are the standard algorithm for multiplying by a sparse matrix in (for example) compressed sparse row format. Evaluating Φ_f and Φ_{f^*} requires $O(\mathbf{nnz}(A))$ flops and $O(\mathbf{nnz}(A))$ bytes of data to store A and A^T, where $\mathbf{nnz}$ is the number of nonzero elements in a sparse matrix [14, Chap. 2].

Multiplication by a low-rank matrix. Multiplication by a matrix $A \in \mathbf{R}^{m \times n}$ with rank k, where $k \ll m$ and $k \ll n$, is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f(x) = Ax$. The matrix A can be factored as

$A = BC$, where $B \in \mathbf{R}^{m\times k}$ and $C \in \mathbf{R}^{k\times n}$. The adjoint $f^*(u) = C^TB^Tu$ is also multiplication by a rank k matrix. The algorithm Φ_f evaluates $f(x)$ by first evaluating $z = Cx$ and then evaluating $f(x) = Bz$. Similarly, Φ_{f^*} multiplies by B^T and then C^T. The algorithms Φ_f and Φ_{f^*} require $O(k(m+n))$ flops and use $O(k(m+n))$ bytes of data to store B and C and their transposes. Multiplication by a low-rank matrix occurs in many applications, and it is often possible to approximate multiplication by a full rank matrix with multiplication by a low-rank one, using the singular value decomposition or methods such as sketching [34].

Discrete Fourier transform. The discrete Fourier transform (DFT) is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f : \mathbf{R}^{2p} \to \mathbf{R}^{2p}$ is given by

$$\begin{array}{rcl} f(x)_k & = & \frac{1}{\sqrt{p}} \sum_{j=1}^{p} \Re\left(\omega_p^{(j-1)(k-1)}\right) x_j \\ & & \quad -\Im\left(\omega_p^{(j-1)(k-1)}\right) x_{j+p} \\ f(x)_{k+p} & = & \frac{1}{\sqrt{p}} \sum_{j=1}^{p} \Im\left(\omega_p^{(j-1)(k-1)}\right) x_j \\ & & \quad +\Re\left(\omega_p^{(j-1)(k-1)}\right) x_{j+p} \end{array} \tag{3}$$

for $k = 1, \ldots, p$. Here $\omega_p = e^{-2\pi i/p}$. The adjoint f^* is the inverse DFT. The algorithm Φ_f is the fast Fourier transform (FFT), while Φ_{f^*} is the inverse FFT. The algorithms can be evaluated in $O((m+n)\log(m+n))$ flops, using only $O(1)$ bytes of data to store the dimensions of f's input and output [13, 37]. Here $m = n = 2p$.

The 2-D DFT has the same computational complexity. In its FAO representation $\Gamma = (f, \Phi_f, \Phi_{f^*})$, the algorithms Φ_f and Φ_{f^*} also require $O((m+n)\log(m+n))$ flops, using only $O(1)$ bytes of data to store the dimensions of f's input and output [35, 37].

Convolution. Convolution with a kernel $c \in \mathbf{R}^p$ is defined as $f : \mathbf{R}^n \to \mathbf{R}^m$, where

$$f(x)_k = \sum_{i+j=k+1} c_i x_j, \quad k = 1, \ldots, m. \tag{4}$$

Different variants of convolution restrict the indices i, j to different ranges, or interpret vector elements outside their natural ranges as zero or using periodic (circular) indexing.

Standard (column) convolution takes $m = n + p - 1$, and defines c_i and x_j in (4) as zero when the index is ouside its range. In this case the associated matrix $\mathbf{Col}(c) \in \mathbf{R}^{n+p-1\times n}$ is Toeplitz, with each column a shifted version of c:

$$\mathbf{Col}(c) = \begin{bmatrix} c_1 & & \\ c_2 & \ddots & \\ \vdots & \ddots & c_1 \\ c_p & & c_2 \\ & \ddots & \vdots \\ & & c_p \end{bmatrix}.$$

Another standard form, row convolution, restricts the indices in (4) to the range $k = p, \ldots, n$. For simplicity we assume that $n \geq p$. In this case the associated matrix $\mathbf{Row}(c) \in \mathbf{R}^{n-p+1\times n}$ is Toeplitz, with each row a shifted version of c, in reverse order:

$$\mathbf{Row}(c) = \begin{bmatrix} c_p & c_{p-1} & \cdots & c_1 & & \\ & \ddots & \ddots & & \ddots & \\ & & c_p & c_{p-1} & \cdots & c_1 \end{bmatrix}.$$

The matrices $\mathbf{Col}(c)$ and $\mathbf{Row}(c)$ are related by the equalities

$$\begin{array}{rcl} \mathbf{Col}(c)^T & = & \mathbf{Row}(\mathbf{rev}(c)) \\ \mathbf{Row}(c)^T & = & \mathbf{Col}(\mathbf{rev}(c)), \end{array} \tag{5}$$

where $\mathbf{rev}(c)_k = c_{p-k+1}$ reverses the order of the entries of c.

Column convolution with $c \in \mathbf{R}^p$ is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f : \mathbf{R}^n \to \mathbf{R}^{n+p-1}$ is given by $f(x) = \mathbf{Col}(c)x$. The adjoint f^* is row convolution with $\mathbf{rev}(c)$, *i.e.*, $f^*(u) = \mathbf{Row}(\mathbf{rev}(c))u$. The algorithms Φ_f and Φ_{f^*} use the DFT to transform convolution into elementwise multiplication and require $O((m+n)\log(m+n))$ flops [37]. Here $m = n + p - 1$. If the kernel is small (*i.e.*, $p \ll n$), Φ_f and Φ_{f^*} instead evaluate (4) directly in $O(np)$ flops. In either case, the algorithms Φ_f and Φ_{f^*} use $O(p)$ bytes of data to store c and $\mathbf{rev}(c)$.

The 2-D analogue of column convolution has the same computational complexity as the 1-D case. The adjoint of 2-D column convolution is a 2-D analogue of row convolution. 2-D column convolution with a kernel $C \in \mathbf{R}^{p\times q}$ has an FAO representation $\Gamma = (f, \Phi_f, \Phi_{f^*})$ where the algorithms Φ_f and Φ_{f^*} require

$$O(\min\{(m+n)\log(m+n), pqn\}),$$

flops and use $O(pq)$ bytes of data to store C [37, Chap. 4]. Often the kernel is parameterized (*e.g.*, a Gaussian kernel), in which case more compact representations of C are possible [20, Chap. 7].

Fast transforms. There are many other linear functions for which the function and its adjoint can be computed efficiently. These typically have $m = n$, and are called transforms. Examples include the discrete wavelet, Hartley,

Haar, and Walsh-Hadamard transforms, which can be evaluated in $O(n)$ or $O(n \log n)$ flops (and the same for their adjoints). Due to space limitations we omit the details here.

Matrix product. Multiplication on the left by a matrix $A \in \mathbf{R}^{s \times p}$ and on the right by a matrix $B \in \mathbf{R}^{q \times t}$ is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f : \mathbf{R}^{p \times q} \to \mathbf{R}^{s \times t}$ is given by $f(X) = AXB$. The adjoint $f^*(U) = A^T U B^T$ is also a matrix product. The linear functions f and f^* can be represented as matrix multiplication (with vector input and output) using Kronecker products.

There are two ways to implement Φ_f efficiently, corresponding to different orders of operations in multiplying out AXB. In one method we multiply by A first and B second, for a total of $O(s(pq + qt))$ flops (assuming that A and B are dense). In the other method we multiply by B first and A second, for a total of $O(p(qt + st))$ flops. The former method is more efficient if

$$\frac{1}{t} + \frac{1}{p} < \frac{1}{s} + \frac{1}{q}.$$

Similarly, there are two ways to implement Φ_{f^*}, one requiring $O(s(pq+qt))$ flops and the other requiring $O(p(qt+st))$ flops. The algorithms Φ_f and Φ_{f^*} use $O(sp + qt)$ bytes of data to store A and B and their transposes. When $p = q = s = t$, the flop count for Φ_f and Φ_{f^*} simplifies to $O\left((m+n)^{1.5}\right)$ flops. Here $m = n = pq$. (When the matrices A or B are sparse, evaluating $f(X)$ and $f^*(U)$ can be done even more efficiently.)

Sum and copy. The function $\mathbf{sum} : \mathbf{R}^m \times \cdots \times \mathbf{R}^m \to \mathbf{R}^m$ with k inputs is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f(x_1, \ldots, x_k) = x_1 + \cdots + x_k$. The adjoint f^* is the function $\mathbf{copy} : \mathbf{R}^m \to \mathbf{R}^m \times \cdots \times \mathbf{R}^m$, which outputs k copies of its input. The algorithms Φ_f and Φ_{f^*} require $O(m + n)$ flops to sum and copy their input, respectively, using only $O(1)$ bytes of data to store the dimensions of f's input and output. Here $n = km$.

2.3. Compositions

In this section we consider compositions of FAOs. In fact we have already discussed several linear functions that are naturally and efficiently represented as compositions, such as multiplication by a low-rank matrix and matrix product. Here though we present a data structure and algorithm for efficiently evaluating any composition and its adjoint, which gives us an FAO representing the composition.

A composition of FAOs can be represented using a directed acyclic graph (DAG) with exactly one node with no incoming edges (the start node) and exactly one node with no outgoing edges (the end node). We call such a representation an *FAO DAG*.

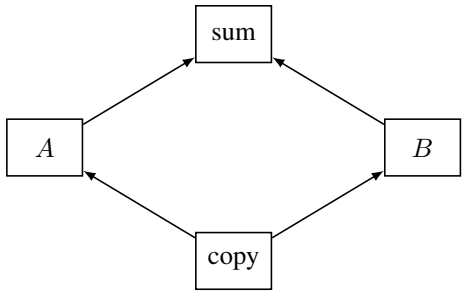

Figure 1: The FAO DAG for $f(x) = Ax + Bx$.

Each node in the FAO DAG stores the following attributes:

- An FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$. Concretely, f is a symbol identifying the function, and Φ_f and Φ_{f^*} are executable code.
- The data needed to evaluate Φ_f and Φ_{f^*}.
- An input array.
- An output array.
- A list E_{in} of incoming edges.
- A list E_{out} of outgoing edges.

The input and output arrays can be the same when the FAO algorithms operate in-place, *i.e.*, write the output to the array storing the input. The total bytes needed to store an FAO DAG is dominated by the sum of the bytes of data on each node. When the same FAO occurs more than once in the FAO DAG, we can reduce the total bytes of data needed by sharing data across duplicate nodes.

As an example, figure 1 shows the FAO DAG for the composition $f(x) = Ax + Bx$, where $A \in \mathbf{R}^{m \times n}$ and $B \in \mathbf{R}^{m \times n}$ are dense matrices. The $\mathbf{copy}$ node duplicates the input $x \in \mathbf{R}^n$ into the multi-argument output $(x, x) \in \mathbf{R}^n \times \mathbf{R}^n$. The A and B nodes multiply by A and B, respectively. The $\mathbf{sum}$ node sums two vectors together. The $\mathbf{copy}$ node is the start node, and the $\mathbf{sum}$ node is the end node. The FAO DAG requires $O(mn)$ bytes to store, since the A and B nodes store the matrices A and B and their tranposes.

Forward evaluation. To evaluate the composition $f(x) = Ax + Bx$ using the FAO DAG in figure 1, we first evaluate the start node on the input $x \in \mathbf{R}^n$, which copies x and sends it out on both outgoing edges. We evaluate the A and B nodes (serially or in parallel) on their incoming argument, and send the results (Ax and Bx) to the end node. Finally, we evaluate the end node on its incoming arguments to obtain the result $Ax + Bx$.

The general procedure for evaluating an FAO DAG is given in algorithm 1. The algorithm evaluates the nodes in a topological order. The total flop count is the sum of the flops from evaluating the algorithm Φ_f on each node. If we

allocate all scratch space needed by the FAO algorithms in advance, then no memory is allocated during the algorithm.

Algorithm 1 Evaluate an FAO DAG.

Input: $G = (V, E)$ is an FAO DAG representing a function f. V is a list of nodes. E is a list of edges. I is a list of inputs to f. O is a list of outputs from f. Each element of I and O is represented as an array.

```
Copy the elements of I onto the start node's input array.
Create an empty queue Q for nodes that are ready to evaluate.
Create an empty set S for nodes that have been evaluated.
Add G's start node to Q.
while Q is not empty do
    u ← pop the front node of Q.
    Evaluate u's algorithm Φ_f on u's input array,
    writing the result to u's output array.
    Add u to S.
    for each edge e = (u, v) in u's E_out do
        i ← the index of e in u's E_out.
        j ← the index of e in v's E_in.
        Copy the segment of u's output array holding
        output i onto the segment of v's input array
        holding input j.
        if for all edges (p, v) in v's E_in, p is in S then
            Add v to the end of Q.
Copy the output array of G's end node onto the elements of O.
```

Postcondition: O contains the outputs of f applied to inputs I.

Adjoint evaluation. Given an FAO DAG G representing a function f, we can easily generate an FAO DAG G^* representing the adjoint f^*. We modify each node in G, replacing the node's FAO (f, Φ_f, Φ_{f^*}) with the FAO $(f^*, \Phi_{f^*}, \Phi_f)$ and swapping E_{in} and E_{out}. We also reverse the orientation of each edge in G. We can apply algorithm 1 to the resulting graph G^* to evaluate f^*. Figure 2 shows the FAO DAG in figure 1 transformed into an FAO DAG for the adjoint.

Parallelism. Algorithm 1 can be easily parallelized, since the nodes in the ready queue Q can be evaluated in any order. A simple parallel implementation could use a thread pool with t threads to evaluate up to t nodes in the ready queue at a time. The extent to which parallelism speeds up evaluation of the composition graph depends on how many parallel paths there are in the graph, *i.e.*, paths with no shared nodes. The evaluation of individual nodes can also be parallelized by replacing a node's algorithm Φ_f with a parallel variant. For example, the standard algorithms for dense and sparse matrix multiplication can be trivially parallelized.

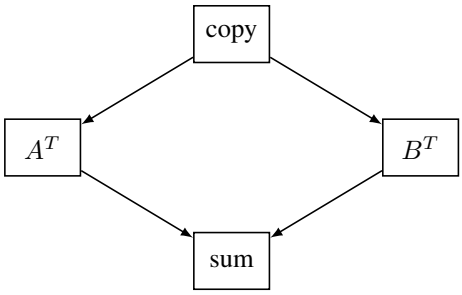

Figure 2: The FAO DAG for the adjoint $f^*(u) = A^T u + B^T u$.

3. Cone programs and solvers

3.1. Cone programs

A cone program is a convex optimization problem of the form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax + b \in \mathcal{K}, \end{array} \tag{6}$$

where $x \in \mathbf{R}^n$ is the optimization variable, $\mathcal{K}$ is a convex cone, and $A \in \mathbf{R}^{m \times n}$, $c \in \mathbf{R}^n$, and $b \in \mathbf{R}^m$ are problem data. Cone programs are a broad class that include linear programs, second-order cone programs, and semidefinite programs as special cases [40, 8]. We call the cone program *matrix-free* if A is represented implicitly as an FAO, rather than explicitly as a dense or sparse matrix.

The convex cone $\mathcal{K}$ is typically a Cartesian product of simple convex cones from the following list:

- Zero cone: $\mathcal{K}_0 = \{0\}$.
- Free cone: $\mathcal{K}_{\text{free}} = \mathbf{R}$.
- Nonnegative cone: $\mathcal{K}_+ = \{x \in \mathbf{R} \mid x \geq 0\}$.
- Second-order cone:

  $$\mathcal{K}_{\text{soc}} = \{(x, t) \in \mathbf{R}^{n+1} \mid x \in \mathbf{R}^n,\ t \in \mathbf{R},\ \|x\|_2 \leq t\}.$$

- Positive semidefinite cone:

  $$\mathcal{K}_{\text{psd}} = \{X \mid X \in \mathbf{S}^n,\ z^T X z \geq 0 \text{ for all } z \in \mathbf{R}^n\}.$$

- Exponential cone:

  $$\begin{aligned} \mathcal{K}_{\text{exp}} = \quad & \{(x, y, z) \in \mathbf{R}^3 \mid y > 0,\ ye^{x/y} \leq z\} \\ & \cup \{(x, 0, z) \in \mathbf{R}^3 \mid x \leq 0,\ z \geq 0\}. \end{aligned}$$

- Power cone:

  $$\mathcal{K}^a_{\text{pwr}} = \{(x, y, z) \in \mathbf{R}^3 \mid x^a y^{(1-a)} \geq |z|,\ x, y \geq 0\},$$

  where $a \in [0, 1]$.

These cones are useful in expressing common problems (via canonicalization), and can be handled by various solvers (as discussed below).

Cone programs that include only cones from certain subsets of the list above have special names. For example, if the only cones are zero, free, and nonnegative cones, the cone program is a linear program; if in addition it includes the second-order cone, it is called a second-order cone program. A well studied special case is so-called symmetric cone programs, which include the zero, free, nonnegative, second-order, and positive semidefinite cones. Semidefinite programs, where the cone constraint consists of a single positive semidefinite cone, are another common case.

3.2. Cone solvers

Many methods have been developed to solve cone programs, the most widely used being interior-point methods.

Interior-point. A large number of interior-point cone solvers have been implemented. Most support symmetric cone programs. For example, SDPT3 [46] and SeDuMi [44] are open-source solvers implemented in MATLAB; CVXOPT [2] is an open-source solver implemented in Python; MOSEK [39] is a commercial solver with interfaces to many languages. ECOS is an open-source cone solver written in library-free C that supports second-order cone programs [17]; Akle extended ECOS to support the exponential cone [1]. DSDP5 [6] and SDPA [23] are open-source solvers for semidefinite programs implemented in C and C++, respectively.

First-order. First-order methods are an alternative to interior-point methods that scale more easily to large cone programs, at the cost of lower accuracy. PDOS [12] is a first-order cone solver based on the alternating direction method of multipliers (ADMM) [7]. PDOS supports second-order cone programs. POGS [21] is an ADMM based solver that runs on a GPU, with a version that is similar to PDOS and targets second-order cone programs. SCS is another ADMM-based cone solver, which supports symmetric cone programs as well as the exponential and power cones [41]. Many other first-order algorithms can be applied to cone programs (*e.g.*, [33, 9, 42]), but none have been implemented as a robust, general purpose cone solver.

Matrix-free. Matrix-free cone solvers are an area of active research, and a small number have been developed. PENNON is a matrix-free semidefinite program (SDP) solver [32]. PENNON solves a series of unconstrained optimization problems using Newton's method. The Newton step is computed using a preconditioned conjugate gradient method, rather than by factoring the Hessian directly [30]. Many other matrix-free algorithms for solving SDPs have been proposed; see, *e.g.*, [24, 45, 50].

Several matrix-free solvers have been developed for quadratic programs (QPs), which are a superset of linear programs and a subset of second-order cone programs. Gondzio developed a matrix-free interior-point method for QPs that solves linear systems using a preconditioned conjugate gradient method [26]. PDCO is a matrix-free interior-point solver that can solve QPs [43], using LSMR to solve linear systems [19].

4. Matrix-free canonicalization

4.1. Canonicalization

Canonicalization is an algorithm that takes as input a data structure representing a general convex optimization problem and outputs a data structure representing an equivalent cone program. By solving the cone program, we recover the solution to the original optimization problem. This approach is used by convex optimization modeling systems such as YALMIP [38], CVX [28], CVXPY [16], and Convex.jl [47]. Current methods of canonicalization convert fast linear transforms in the original problem into multiplication by a dense or sparse matrix, which makes the final cone program far more costly to solve than the original problem.

The canonicalization algorithm can be modified, however, so that fast linear transforms are preserved. The key is to represent all linear functions arising during the canonicalization process as FAO DAGs instead of as sparse matrices. The FAO DAG representation of the final cone program can be used by a matrix-free cone solver to solve the cone program. The modified canonicalization algorithm never forms explicit matrix representations of linear functions. Hence we call the algorithm *matrix-free canonicalization*.

4.2. Informal overview

In this section we give an informal overview of the matrix-free canonicalization algorithm. A longer paper still in development contains the full details [15].

We are given an optimization problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\ & h_i(x) + d_i = 0, \quad i = 1, \ldots, q, \end{array} \tag{7}$$

where $x \in \mathbf{R}^n$ is the optimization variable, $f_0 : \mathbf{R}^n \to \mathbf{R}, \ldots, f_p : \mathbf{R}^n \to \mathbf{R}$ are convex functions, $h_1 : \mathbf{R}^n \to \mathbf{R}^{m_1}, \ldots, h_q : \mathbf{R}^n \to \mathbf{R}^{m_q}$ are linear functions, and $d_1 \in \mathbf{R}^{m_1}, \ldots, d_q \in \mathbf{R}^{m_q}$ are vector constants. Our goal is to convert the problem into an equivalent matrix-free cone program, so that we can solve it using a matrix-free cone solver.

We assume that the problem satisfies a set of requirements known as *disciplined convex programming* [27, 29].

The requirements ensure that each of the $f_0, \ldots, f_p$ can be represented as partial minimization over a cone program. Let each function f_i have the cone program representation

$$f_i(x) \quad = \quad \begin{array}{ll} \text{minimize} & g_0^{(i)}(x, t^{(i)}) + e_0^{(i)} \\ \text{subject to} & g_j^{(i)}(x, t^{(i)}) + e_j^{(i)} \in \mathcal{K}_j^{(i)}, \end{array} \tag{8}$$

where the minimization is over the variable $t^{(i)} \in \mathbf{R}^{s^{(i)}}$, the constraints are indexed over $j = 1, \ldots, r^{(i)}$, $g_0^{(i)}, \ldots, g_{r^{(i)}}^{(i)}$ are linear functions, $e_0^{(i)}, \ldots, e_{r^{(i)}}^{(i)}$ are vector constants, and $\mathcal{K}_1^{(i)}, \ldots, \mathcal{K}_{r^{(i)}}^{(i)}$ are convex cones. For simplicity we assume here that all cone elements are real vectors.

We rewrite problem (7) as the equivalent cone program

$$\begin{array}{ll} \text{minimize} & g_0^{(0)}(x, t^{(0)}) + e_0^{(0)} \\ \text{subject to} & -g_0^{(i)}(x, t^{(i)}) - e_0^{(i)} \in \mathcal{K}_+ \\ & g_j^{(i)}(x, t^{(i)}) + e_j^{(i)} \in \mathcal{K}_j^{(i)} \\ & h_i(x) + d_i \in \mathcal{K}_0^{m_i}, \end{array} \tag{9}$$

where constraints indexed by i and j are over the obvious ranges. We convert problem (9) into the standard form for a matrix-free cone program given in (6) by representing $g_0^{(0)}$ as the inner product with a vector $c \in \mathbf{R}^{n+s^{(0)}}$, concatenating the d_i and $e_j^{(i)}$ vectors into a single vector b, and representing the matrix A implicitly as the linear function that stacks the outputs of all the h_i and $g_j^{(i)}$ (excluding the objective $g_0^{(0)}$) into a single vector.

5. Numerical results

5.1. Implementation

We have implemented the matrix-free canonicalization algorithm as an extension of CVXPY [16], available at

`https://github.com/SteveDiamond/cvxpy`.

To solve the resulting matrix-free cone programs, we implemented modified versions of SCS [41] and POGS [21] that are truly matrix-free, available at

`https://github.com/SteveDiamond/scs`,
`https://github.com/SteveDiamond/pogs`.

(The details of these modification will be described in future work.) Our implementations are still preliminary and can be improved in many ways. We also emphasize that the canonicalization is independent of the particular matrix-free cone solver used.

5.2. Nonnegative deconvolution

We applied our matrix-free convex optimization modeling system to the nonnegative deconvolution problem (1). We compare the performance with that of the current

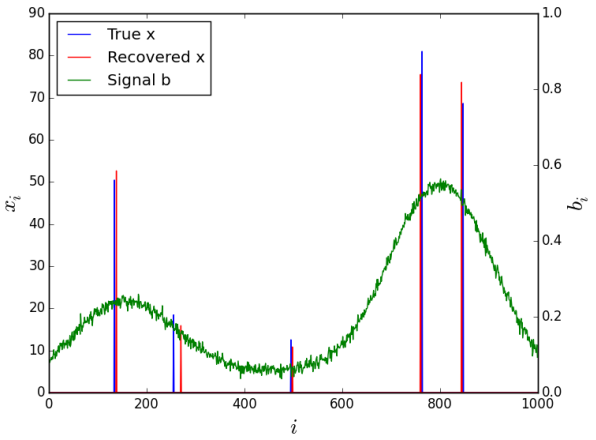

Figure 3: Results for a problem instance with $n = 1000$.

CVXPY modeling system, which represents the matrix A in a cone program as a sparse matrix and uses standard cone solvers.

The Python code below constructs and solves problem (1). The constants c and b and problem size n are defined elsewhere. The code is only a few lines, and it could be easily modified to add regularization on x or apply a different cost function to $c * x - b$. The modeling system would automatically adapt to solve the modified problem.

```
# Construct the optimization problem.
x = Variable(n)
cost = sum_squares(conv(c, x) - b)
prob = Problem(Minimize(cost),
               [x >= 0])
# Solve using matrix-free SCS.
prob.solve(solver=MAT_FREE_SCS)
```

Problem instances. We used the following procedure to generate interesting (nontrivial) instances of problem (1). For all instances the vector $c \in \mathbf{R}^n$ was a Gaussian kernel with standard deviation $n/10$. All entries of c less than 10^{-6} were set to 10^{-6}, so that no entries were too close to zero. The vector $b \in \mathbf{R}^{2n-1}$ was generated by picking a solution $\tilde{x}$ with 5 entries randomly chosen to be nonzero. The values of the nonzero entries were chosen uniformly at random from the interval $[0, n/10]$. We set $b = c * \tilde{x} + v$, where the entries of the noise vector $v \in \mathbf{R}^{2n-1}$ were drawn from a normal distribution with mean zero and variance $\|c * \tilde{x}\|^2/(400(2n-1))$. Our choice of v yielded a signal-to-noise ratio near 20.

While not relevant to solving the optimization problem, the solution of the nonnegative deconvolution problem often, but not always, (approximately) recovers the original

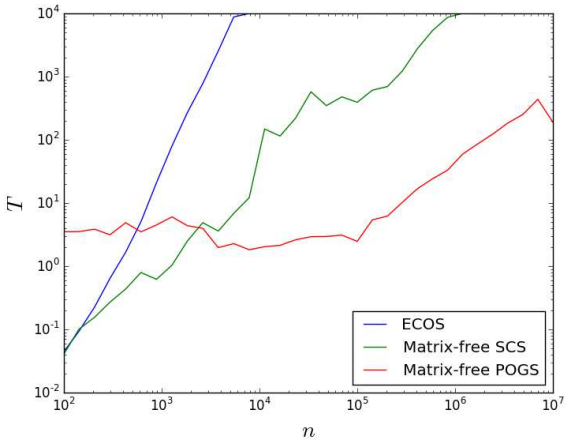

Figure 4: Solve time in seconds T versus variable size n.

vector $\tilde{x}$. Figure 3 shows the solution recovered by ECOS [17] for a problem instance with $n = 1000$. The ECOS solution $x^\star$ had a cluster of 3-5 adjacent nonzero entries around each spike in $\tilde{x}$. The sum of the entries was close to the value of the spike. The recovered x in figure 3 shows only the largest entry in each cluster, with value set to the sum of the cluster's entries.

Results. Figure 4 compares the performance on problem (1) of the interior-point solver ECOS [17] and matrix-free versions of SCS and POGS as the size n of the optimization variable increases. We limited the solvers to 10^4 seconds. ECOS and matrix-free SCS were run serially on a Intel Xeon processor, while matrix-free POGS was run on a Titan X GPU.

For each variable size n we generated ten different problem instances and recorded the average solve time for each solver. ECOS and matrix-free SCS were run with an absolute and relative tolerance of 10^{-3} for the duality gap, ℓ_2 norm of the primal residual, and ℓ_2 norm of the dual residual. Matrix-free POGS was run with an absolute tolerance of 10^{-4} and a relative tolerance of 10^{-3}.

The slopes of the lines show how the solvers scale. The least-squares linear fit for the ECOS solve times has slope 3.1, which indicates that the solve time scales like n^3, as expected. The least-squares linear fit for the matrix-free SCS solve times has slope 1.3, which indicates that the solve time scales like the expected $n \log n$. The least-squares linear fit for the matrix-free POGS solve times in the range $n \in [10^5, 10^7]$ has slope 1.1, which indicates that the solve time scales like the expected $n \log n$. For $n < 10^5$, the GPU overhead (launching kernels, *etc.*) dominates, and the solve time is nearly constant.

References

[1] S. Akle. *Algorithms for unsymmetric cone optimization and an implementation for problems with the exponential cone*. PhD thesis, Stanford University, 2015.

[2] M. Andersen, J. Dahl, and L. Vandenberghe. CVXOPT: Python software for convex optimization, version 1.1. `http://cvxopt.org/`, May 2015.

[3] A. Beck and M. Teboulle. Fast gradient-based algorithms for constrained total variation image denoising and deblurring problems. *IEEE Transactions on Image Processing*, 18(11):2419–2434, Nov. 2009.

[4] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183–202, 2009.

[5] S. Becker, E. Candès, and M. Grant. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical Programming Computation*, 3(3):165–218, 2011.

[6] S. Benson and Y. Ye. DSDP5: Software for semidefinite programming. Technical Report ANL/MCS-P1289-0905, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, Sept. 2005. Submitted to ACM Transactions on Mathematical Software.

[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. Chan, S. Esedoglu, and M. Nikolova. Algorithms for finding global minimizers of image segmentation and denoising models. *SIAM Journal on Applied Mathematics*, 66(5):1632–1648, 2006.

[11] S. Chen, D. Donoho, and M. Saunders. Atomic decomposition by basis pursuit. *SIAM Journal on Scientific Computing*, 20(1):33–61, 1998.

[12] E. Chu, B. O'Donoghue, N. Parikh, and S. Boyd. A primal-dual operator splitting method for conic optimization. *Preprint*, 2013. `http://stanford.edu/~boyd/papers/pdf/pdos.pdf`.

[13] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[14] T. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. SIAM, Philadelphia, PA, USA, 2006.

[15] S. Diamond and S. Boyd. Matrix-free convex optimization modeling. *Preprint*, 2015. `http://arxiv.org/pdf/1506.00760v1.pdf`.

[16] S. Diamond, E. Chu, and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization, version 0.2. `http://cvxpy.org/`, May 2014.

[17] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP solver for embedded systems. In *Proceedings of the European Control Conference*, pages 3071–3076, 2013.

[18] M. Figueiredo, R. Nowak, and S. Wright. Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems. *IEEE Journal of Selected Topics in Signal Processing*, 1(4):586–597, Dec. 2007.

[19] D. Fong and M. Saunders. LSMR: An iterative algorithm for sparse least-squares problems. *SIAM Journal on Scientific Computing*, 33(5):2950–2971, 2011.

[20] D. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002.

[21] C. Fougner and S. Boyd. Parameter selection and preconditioning for a graph form solver. *Preprint*, 2015. `http://arxiv.org/pdf/1503.08366v1.pdf`.

[22] 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`.

[23] K. Fujisawa, M. Fukuda, K. Kobayashi, M. Kojima, K. Nakata, M. Nakata, and M. Yamashita. SDPA (semidefinite programming algorithm) user's manual – version 7.0.5. Technical report, 2008.

[24] M. Fukuda, M. Kojima, and M. Shida. Lagrangian dual interior-point methods for semidefinite programs. *SIAM Journal on Optimization*, 12(4):1007–1031, 2002.

[25] T. Goldstein and S. Osher. The split Bregman method for ℓ_1-regularized problems. *SIAM Journal on Imaging Sciences*, 2(2):323–343, 2009.

[26] J. Gondzio. Matrix-free interior point method. *Computational Optimization and Applications*, 51(2):457–480, 2012.

[27] M. Grant. *Disciplined Convex Programming*. PhD thesis, Stanford University, 2004.

[28] M. Grant and S. Boyd. CVX: MATLAB software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`, Mar. 2014.

[29] 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.

[30] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. N.B.S.*, 49(6):409–436, 1952.

[31] S.-J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky. An interior-point method for large-scale ℓ_1-regularized least squares. *IEEE Journal on Selected Topics in Signal Processing*, 1(4):606–617, Dec. 2007.

[32] M. Kočvara and M. Stingl. On the solution of large-scale SDP problems by the modified barrier method using iterative solvers. *Mathematical Programming*, 120(1):285–287, 2009.

[33] G. Lan, Z. Lu, and R. Monteiro. Primal-dual first-order methods with $O(1/\epsilon)$ iteration-complexity for cone programming. *Mathematical Programming*, 126(1):1–29, 2011.

[34] E. Liberty. Simple and deterministic matrix sketching. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 581–588, 2013.

[35] J. Lim. *Two-dimensional Signal and Image Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[36] Y. Lin, D. Lee, and L. Saul. Nonnegative deconvolution for time of arrival estimation. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 377–380, May 2004.

[37] C. V. Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, 1992.

[38] J. Lofberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proceedings of the IEEE International Symposium on Computed Aided Control Systems Design*, pages 294–289, Sept. 2004.

[39] MOSEK optimization software, version 7. `https://mosek.com/`, Jan. 2015.

[40] Y. Nesterov and A. Nemirovsky. Conic formulation of a convex programming problem and duality. *Optimization Methods and Software*, 1(2):95–115, 1992.

[41] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Preprint*, 2015. `http://stanford.edu/~boyd/papers/pdf/scs.pdf`.

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

[43] M. Saunders, B. Kim, C. Maes, S. Akle, and M. Zahr. PDCO: Primal-dual interior method for convex objectives. `http://web.stanford.edu/group/SOL/software/pdco/`, Nov. 2013.

[44] J. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11(1-4):625–653, 1999.

[45] K.-C. Toh. Solving large scale semidefinite programs via an iterative solver on the augmented systems. *SIAM Journal on Optimization*, 14(3):670–698, 2004.

[46] K.-C. Toh, M. Todd, and R. Tütüncü. SDPT3 — a MATLAB software package for semidefinite programming, version 4.0. *Optimization Methods and Software*, 11:545–581, 1999.

[47] M. Udell, K. Mohan, D. Zeng, J. Hong, S. Diamond, and S. Boyd. Convex optimization in Julia. *SC14 Workshop on High Performance Technical Computing in Dynamic Languages*, 2014.

[48] E. van den Berg and M. Friedlander. Probing the Pareto frontier for basis pursuit solutions. *SIAM Journal on Scientific Computing*, 31(2):890–912, 2009.

[49] C. Zach, T. Pock, and H. Bischof. A duality based approach for realtime TV-ℓ_1 optical flow. In *Pattern Recognition*, volume 4713 of *Lecture Notes in Computer Science*, pages 214–223. Springer Berlin Heidelberg, 2007.

[50] X.-Y. Zhao, D. Sun, and K.-C. Toh. A Newton-CG augmented Lagrangian method for semidefinite programming. *SIAM Journal on Optimization*, 20(4):1737–1765, 2010.