Matrix-Free Convex Optimization Modeling

Steven Diamond and Stephen Boyd

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 linear operators, we arrive at a matrix-free cone program, i.e., one whose data matrix is represented by a 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.

Keywords Convex optimization • Matrix-free optimization • Conic programming • Optimization modeling

1 Introduction

Convex optimization modeling systems like YALMIP [83], CVX [57], CVXPY [36], and Convex.jl [106] provide an automated framework for converting a convex optimization problem expressed in a natural human-readable form into the standard form required by a 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.

S. Diamond (✉)
Department of Computer Science, Stanford University, Stanford, CA 94305, USA
e-mail: diamond@cs.stanford.edu

S. Boyd
Department of Electrical Engineering, Stanford University, Stanford, CA 94305, USA
e-mail: boyd@stanford.edu

© Springer International Publishing Switzerland 2016

B. Goldengorin (ed.), *Optimization and Its Applications in Control and Data Sciences*, Springer Optimization and Its Applications 115,
DOI 10.1007/978-3-319-42056-1_7

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., [9, 10, 22, 23, 51, 97, 117]. There has been particular interest in matrix-free solvers for LASSO and basis pursuit denoising problems [10, 24, 42, 46, 74, 108]. Matrix-free solvers have also been developed for specialized control problems [109, 110]. The most general matrix-free solvers target semidefinite programs [75] or quadratic programs and related problems [52, 99]. 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 [11].

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 [77, 81].

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 functions. (A preliminary version of this paper appeared in [35].) This 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.

The outline of our paper is as follows. In Sect. 2 we give many examples of useful FAOs. In Sect. 3 we explain how to compose FAOs so that we can efficiently evaluate the composition and its adjoint. In Sect. 4 we describe cone programs, the standard intermediate-form representation of a convex problem, and solvers for cone programs. In Sect. 5 we describe our algorithm for converting convex optimization problems into equivalent cone programs while preserving fast linear transforms. In Sect. 6 we report numerical results for the nonnegative deconvolution problem (1) and a special type of linear program, for our implementation of the abstract ideas in the paper, using versions of the existing cone solvers SCS [94] and POGS [45] modified to be matrix-free. (The main modification was using the matrix-free equilibration described in [37].) Even with our simple, far from optimized matrix-free cone solvers, we demonstrate scaling to problems far larger than those that can be solved by generic methods (based on sparse matrices), with acceptable performance loss compared to specialized custom algorithms tuned to the problems.

We reserve certain details of our matrix-free canonicalization algorithm for the appendix. In “Equivalence of the Cone Program” we explain the precise sense in which the cone program output by our algorithm is equivalent to the original convex optimization problem. In “Sparse Matrix Representation” we describe how existing modeling systems generate a sparse matrix representation of the cone program. The details of this process have never been published, and it is interesting to compare with our algorithm.

2 Forward-Adjoint Oracles

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^*. We use n to denote the size of f's input and m to denote the size of f's output.

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).

2.1 Vector Mappings

We present a variety of FAOs for functions that take as argument, and return, vectors.

Scalar Multiplication Scalar multiplication by $\alpha \in \mathbf{R}$ is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f : \mathbf{R}^n \to \mathbf{R}^n$ is given by $f(x) = \alpha x$. The adjoint f^* is the same as f. The algorithms Φ_f and Φ_{f^*} simply scale the input, which requires $O(m + n)$ flops and $O(1)$ bytes of data to store α. Here $m = n$.

Multiplication by a Dense Matrix Multiplication by a dense matrix $A \in \mathbf{R}^{m\times n}$ 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 dense matrix. The algorithms Φ_f and Φ_{f^*} are the standard dense matrix multiplication algorithm. Evaluating Φ_f and Φ_{f^*} requires $O(mn)$ flops and $O(mn)$ bytes of data to store A and A^T.

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 [34, 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 $f^*(u) = C^T B^T u$ 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 [79].

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

$$\begin{aligned} f(x)_k &= \tfrac{1}{\sqrt{p}} \textstyle\sum_{j=1}^{p} \mathrm{Re}\left(\omega_p^{(j-1)(k-1)}\right) x_j - \mathrm{Im}\left(\omega_p^{(j-1)(k-1)}\right) x_{j+p} \\ f(x)_{k+p} &= \tfrac{1}{\sqrt{p}} \textstyle\sum_{j=1}^{p} \mathrm{Im}\left(\omega_p^{(j-1)(k-1)}\right) x_j + \mathrm{Re}\left(\omega_p^{(j-1)(k-1)}\right) x_{j+p} \end{aligned}$$

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 [31, 82]. Here $m = n = 2p$. There are many fast transforms derived from the DFT, such as the discrete Hartley transform [17] and the discrete sine and cosine transforms [2, 86], with the same computational complexity as the FFT.

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

$$f(x)_k = \sum_{i+j=k+1} c_i x_j, \quad k = 1, \ldots, m. \tag{3}$$

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 (3) as zero when the index is outside 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 (3) 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:

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

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

$$\mathbf{Col}(c)^T = \mathbf{Row}(\mathbf{rev}(c)), \qquad \mathbf{Row}(c)^T = \mathbf{Col}(\mathbf{rev}(c)),$$

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

Yet another variant on convolution is circular convolution, where we take $p = n$ and interpret the entries of vectors outside their range modulo n. In this case the associated matrix $\mathbf{Circ}(c) \in \mathbf{R}^{n\times n}$ is Toeplitz, with each column and row a (circularly) shifted version of c:

$$\mathbf{Circ}(c) = \begin{bmatrix} c_1 & c_n & c_{n-1} & \ldots & \ldots & c_2 \\ c_2 & c_1 & c_n & \ddots & & \vdots \\ c_3 & c_2 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_n & c_{n-1} \\ \vdots & & \ddots & c_2 & c_1 & c_n \\ c_n & \ldots & \ldots & c_3 & c_2 & c_1 \end{bmatrix}.$$

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^*} are given in Algorithms 1 and 2, and require $O((m+n+p)\log(m+n+p))$ flops. Here $m = n + p - 1$. If the kernel is small (i.e., $p \ll n$), Φ_f and Φ_{f^*} instead evaluate (3) 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 $\mathbf{rev}(c)$ [30, 82].

Circular convolution with $c \in \mathbf{R}^n$ is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f : \mathbf{R}^n \to \mathbf{R}^n$ is given by $f(x) = \mathbf{Circ}(c)x$. The adjoint f^* is circular convolution with

$$\tilde{c} = \begin{bmatrix} c_1 \\ c_n \\ c_{n-1} \\ \vdots \\ c_2 \end{bmatrix}.$$

Algorithm 1 Column convolution $c * x$

Precondition: $c \in \mathbf{R}^p$ is a length p array. $x \in \mathbf{R}^n$ is a length n array. $y \in \mathbf{R}^{n+p-1}$ is a length $n + p - 1$ array.

Extend c and x into length $n + p - 1$ arrays by appending zeros.
$\hat{c} \leftarrow$ FFT of c.
$\hat{x} \leftarrow$ FFT of x.
for $i = 1, \ldots, n + p - 1$ **do**
    $y_i \leftarrow \hat{c}_i \hat{x}_i$.
end for
$y \leftarrow$ inverse FFT of y.

Postcondition: $y = c * x$.

Algorithm 2 Row convolution $c * u$

Precondition: $c \in \mathbf{R}^p$ is a length p array. $u \in \mathbf{R}^{n+p-1}$ is a length $n + p - 1$ array. $v \in \mathbf{R}^n$ is a length n array.

Extend $\mathbf{rev}(c)$ and v into length $n + p - 1$ arrays by appending zeros.
$\hat{c} \leftarrow$ inverse FFT of zero-padded $\mathbf{rev}(c)$.
$\hat{u} \leftarrow$ FFT of u.
for $i = 1, \ldots, n + p - 1$ **do**
    $v_i \leftarrow \hat{c}_i \hat{u}_i$.
end for
$v \leftarrow$ inverse FFT of v.
Reduce v to a length n array by removing the last $p - 1$ entries.

Postcondition: $v = c * u$.

Algorithm 3 Circular convolution $c * x$

Precondition: $c \in \mathbf{R}^n$ is a length n array. $x \in \mathbf{R}^n$ is a length n array. $y \in \mathbf{R}^n$ is a length n array.

$\hat{c} \leftarrow$ FFT of c.
$\hat{x} \leftarrow$ FFT of x.
for $i = 1, \ldots, n$ **do**
    $y_i \leftarrow \hat{c}_i \hat{x}_i$.
end for
$y \leftarrow$ inverse FFT of y.

Postcondition: $y = c * x$.

The algorithms Φ_f and Φ_{f^*} are given in Algorithm 3, and require $O((m+n)\log(m+n))$ flops. The algorithms Φ_f and Φ_{f^*} use $O(m+n)$ bytes of data to store c and $\tilde{c}$ [30, 82]. Here $m = n$.

Discrete Wavelet Transform The discrete wavelet transform (DWT) for orthogonal wavelets is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where the function $f : \mathbf{R}^{2^p} \to \mathbf{R}^{2^p}$ is given by

$$f(x) = \begin{bmatrix} D_1 G_1 & \\ D_1 H_1 & \\ & I_{2^p-2} \end{bmatrix} \cdots \begin{bmatrix} D_{p-1} G_{p-1} & \\ D_{p-1} H_{p-1} & \\ & I_{2^{p-1}} \end{bmatrix} \begin{bmatrix} D_p G_p \\ D_p H_p \end{bmatrix} x, \tag{4}$$

where $D_k \in \mathbf{R}^{2^{k-1} \times 2^k}$ is defined such that $(D_k x)_i = x_{2i}$ and the matrices $G_k \in \mathbf{R}^{2^k \times 2^k}$ and $H_k \in \mathbf{R}^{2^k \times 2^k}$ are given by

$$G_k = \mathbf{Circ}\left(\begin{bmatrix} g \\ 0 \end{bmatrix}\right), \qquad H_k = \mathbf{Circ}\left(\begin{bmatrix} h \\ 0 \end{bmatrix}\right).$$

Here $g \in \mathbf{R}^q$ and $h \in \mathbf{R}^q$ are low and high pass filters, respectively, that parameterize the DWT. The adjoint f^* is the inverse DWT. The algorithms Φ_f and Φ_f^* repeatedly convolve by g and h, which requires $O(q(m+n))$ flops and uses $O(q)$ bytes to store h and g [85]. Here $m = n = 2^p$. Common orthogonal wavelets include the Haar wavelet and the Daubechies wavelets [32, 33]. There are many variants on the particular DWT described here. For instance, the product in (4) can be terminated after fewer than p multiplications by G_k and H_k [70], G_k and H_k can be defined as a different type of convolution matrix, or the filters g and h can be different lengths, as in biorthogonal wavelets [28].

Discrete Gauss Transform The discrete Gauss transform (DGT) is represented by the FAO $\Gamma = (f_{Y,Z,h}, \Phi_f, \Phi_{f^*})$, where the function $f_{Y,Z,h} : \mathbf{R}^n \to \mathbf{R}^m$ is parameterized by $Y \in \mathbf{R}^{m \times d}$, $Z \in \mathbf{R}^{n \times d}$, and $h > 0$. The function $f_{Y,Z,h}$ is given by

$$f_{Y,Z,h}(x)_i = \sum_{j=1}^{n} \exp(-\|y_i - z_j\|^2/h^2) x_j, \quad i = 1, \ldots, m,$$

where $y_i \in \mathbf{R}^d$ is the ith column of Y and $z_j \in \mathbf{R}^d$ is the jth column of Z. The adjoint of $f_{Y,Z,h}$ is the DGT $f_{Z,Y,h}$. The algorithms Φ_f and Φ_{f^*} are the improved fast Gauss transform, which evaluates $f(x)$ and $f^*(u)$ to a given accuracy in $O(d^p(m+n))$ flops. Here p is a parameter that depends on the accuracy desired. The algorithms Φ_f and Φ_{f^*} use $O(d(m+n))$ bytes of data to store Y, Z, and h [114]. An interesting application of the DGT is efficient multiplication by a Gaussian kernel [113].

Multiplication by the Inverse of a Sparse Triangular Matrix Multiplication by the inverse of a sparse lower triangular matrix $L \in \mathbf{R}^{n \times n}$ with nonzero elements on its diagonal is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f(x) = L^{-1}x$.

The adjoint $f^*(u) = (L^T)^{-1}u$ is multiplication by the inverse of a sparse upper triangular matrix. The algorithms Φ_f and Φ_{f^*} are forward and backward substitution, respectively, which require $O(\mathbf{nnz}(L))$ flops and use $O(\mathbf{nnz}(L))$ bytes of data to store L and L^T [34, Chap. 3].

Multiplication by a Pseudo-Random Matrix Multiplication by a matrix $A \in \mathbf{R}^{m\times n}$ whose columns are given by a pseudo-random sequence (i.e., the first m values of the sequence are the first column of A, the next m values are the second column of A, etc.) is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f(x) = Ax$. The adjoint $f^*(u) = A^T u$ is multiplication by a matrix whose rows are given by a pseudo-random sequence (i.e., the first m values of the sequence are the first row of A^T, the next m values are the second row of A^T, etc.). The algorithms Φ_f and Φ_{f^*} are the standard dense matrix multiplication algorithm, iterating once over the pseudo-random sequence without storing any of its values. The algorithms require $O(mn)$ flops and use $O(1)$ bytes of data to store the seed for the pseudo-random sequence. Multiplication by a pseudo-random matrix might appear, for example, as a measurement ensemble in compressed sensing [50].

Multiplication by the Pseudo-Inverse of a Graph Laplacian Multiplication by the pseudo-inverse of a graph Laplacian matrix $L \in \mathbf{R}^{n\times n}$ is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f(x) = L^\dagger x$. A graph Laplacian is a symmetric matrix with nonpositive off diagonal entries and the property $L\mathbf{1} = 0$, i.e., the diagonal entry in a row is the negative sum of the off-diagonal entries in that row. (This implies that it is positive semidefinite.) The adjoint f^* is the same as f, since $L = L^T$. The algorithms Φ_f and Φ_{f^*} are one of the fast solvers for graph Laplacian systems that evaluate $f(x) = f^*(x)$ to a given accuracy in around $O(\mathbf{nnz}(L))$ flops [73, 101, 111]. (The details of the computational complexity are much more involved.) The algorithms use $O(\mathbf{nnz}(L))$ bytes of data to store L.

2.2 *Matrix Mappings*

We now consider linear functions that take as argument, or return, matrices. We take the standard inner product on matrices $X, Y \in \mathbf{R}^{p\times q}$,

$$\langle X, Y\rangle = \sum_{i=1,\dots,p,\ j=1,\dots,q} X_{ij}Y_{ij} = \mathbf{Tr}(X^T Y).$$

The adjoint of a linear function $f : \mathbf{R}^{p\times q} \to \mathbf{R}^{s\times t}$ is then the function $f^* : \mathbf{R}^{s\times t} \to \mathbf{R}^{p\times q}$ for which

$$\mathbf{Tr}(f(X)^T Y) = \mathbf{Tr}(X^T f^*(Y)),$$

holds for all $X \in \mathbf{R}^{p\times q}$ and $Y \in \mathbf{R}^{s\times t}$.

Vec and Mat The function $\mathbf{vec} : \mathbf{R}^{p\times q} \to \mathbf{R}^{pq}$ is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f(X)$ converts the matrix $X \in \mathbf{R}^{p\times q}$ into a vector $y \in \mathbf{R}^{pq}$ by stacking the columns. The adjoint f^* is the function $\mathbf{mat} : \mathbf{R}^{pq} \to \mathbf{R}^{p\times q}$, which outputs a matrix whose columns are successive slices of its vector argument. The algorithms Φ_f and Φ_{f^*} simply reinterpret their input as a differently shaped output in $O(1)$ flops, using only $O(1)$ bytes of data to store the dimensions of f's input and output.

Sparse Matrix Mappings Many common linear functions on and to matrices are given by a sparse matrix multiplication of the vectorized argument, reshaped as the output matrix. For $X \in \mathbf{R}^{p\times q}$ and $f(X) = Y \in \mathbf{R}^{s\times t}$,

$$Y = \mathbf{mat}(A\,\mathbf{vec}(X)).$$

The form above describes the general linear mapping from $\mathbf{R}^{p\times q}$ to $\mathbf{R}^{s\times t}$; we are interested in cases when A is sparse, i.e., has far fewer than $pqst$ nonzero entries. Examples include extracting a submatrix, extracting the diagonal, forming a diagonal matrix, summing the rows or columns of a matrix, transposing a matrix, scaling its rows or columns, and so on. The FAO representation of each such function is $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where f is given above and the adjoint is given by

$$f^*(U) = \mathbf{mat}(A^T\,\mathbf{vec}(U)).$$

The algorithms Φ_f and Φ_{f^*} are the standard algorithms for multiplying a vector by a sparse matrix in (for example) compressed sparse row format. The algorithms require $O(\mathbf{nnz}(A))$ flops and use $O(\mathbf{nnz}(A))$ bytes of data to store A and A^T [34, Chap. 2].

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 $f^*(U) = A^TUB^T$ is also a matrix product. 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.) The matrix product function is used in Lyapunov and algebraic Riccati inequalities and Sylvester equations, which appear in many problems from control theory [49, 110].

2-D Discrete Fourier Transform The 2-D DFT is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f : \mathbf{R}^{2p\times q} \to \mathbf{R}^{2p\times q}$ is given by

$$\begin{aligned} f(X)_{k\ell} &= \tfrac{1}{\sqrt{pq}} \textstyle\sum_{s=1}^{p} \sum_{t=1}^{q} \mathrm{Re}\left(\omega_p^{(s-1)(k-1)} \omega_q^{(t-1)(\ell-1)}\right) X_{st} \\ &\qquad - \mathrm{Im}\left(\omega_p^{(s-1)(k-1)} \omega_q^{(t-1)(\ell-1)}\right) X_{s+p,t} \\ f(X)_{k+p,\ell} &= \tfrac{1}{\sqrt{pq}} \textstyle\sum_{s=1}^{p} \sum_{t=1}^{q} \mathrm{Im}\left(\omega_p^{(s-1)(k-1)} \omega_q^{(t-1)(\ell-1)}\right) X_{st} \\ &\qquad + \mathrm{Re}\left(\omega_p^{(s-1)(k-1)} \omega_q^{(t-1)(\ell-1)}\right) X_{s+p,t}, \end{aligned}$$

for $k = 1, \ldots, p$ and $\ell = 1, \ldots, q$. Here $\omega_p = e^{-2\pi i/p}$ and $\omega_q = e^{-2\pi i/q}$. The adjoint f^* is the inverse 2-D DFT. The algorithm Φ_f evaluates $f(X)$ by first applying the FFT to each row of X, replacing the row with its DFT, and then applying the FFT to each column, replacing the column with its DFT. The algorithm Φ_{f^*} is analogous, but with the inverse FFT and inverse DFT taking the role of the FFT and DFT. The algorithms Φ_f and Φ_{f^*} 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 [80, 82]. Here $m = n = 2pq$.

2-D Convolution 2-D convolution with a kernel $C \in \mathbf{R}^{p\times q}$ is defined as $f : \mathbf{R}^{s\times t} \to \mathbf{R}^{m_1\times m_2}$, where

$$f(X)_{k\ell} = \sum_{i_1+i_2=k+1, j_1+j_2=\ell+1} C_{i_1 j_1} X_{i_2 j_2}, \quad k = 1, \ldots, m_1, \quad \ell = 1, \ldots, m_2. \tag{5}$$

Different variants of 2-D convolution restrict the indices i_1, j_1 and i_2, j_2 to different ranges, or interpret matrix elements outside their natural ranges as zero or using periodic (circular) indexing. There are 2-D analogues of 1-D column, row, and circular convolution.

Standard 2-D (column) convolution, the analogue of 1-D column convolution, takes $m_1 = s + p - 1$ and $m_2 = t + q - 1$, and defines $C_{i_1 j_1}$ and $X_{i_2 j_2}$ in (5) as zero when the indices are outside their range. We can represent the 2-D column convolution $Y = C * X$ as the matrix multiplication

$$Y = \mathbf{mat}(\mathbf{Col}(C)\,\mathbf{vec}(X)),$$

where $\mathbf{Col}(C) \in \mathbf{R}^{(s+p-1)(t+q-1)\times st}$ is given by:

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

Here $c_1, \ldots, c_q \in \mathbf{R}^p$ are the columns of C and $\mathbf{Col}(c_1), \ldots, \mathbf{Col}(c_q) \in \mathbf{R}^{s+p-1 \times s}$ are 1-D column convolution matrices.

The 2-D analogue of 1-D row convolution restricts the indices in (5) to the range $k = p, \ldots, s$ and $\ell = q, \ldots, t$. For simplicity we assume $s \geq p$ and $t \geq q$. The output dimensions are $m_1 = s - p + 1$ and $m_2 = t - q + 1$. We can represent the 2-D row convolution $Y = C * X$ as the matrix multiplication

$$Y = \mathbf{mat}(\mathbf{Row}(C)\,\mathbf{vec}(X)),$$

where $\mathbf{Row}(C) \in \mathbf{R}^{(s-p+1)(t-q+1) \times st}$ is given by:

$$\mathbf{Row}(C) = \begin{bmatrix} \mathbf{Row}(c_q) & \mathbf{Row}(c_{q-1}) & \ldots & \mathbf{Row}(c_1) & & \\ & \ddots & \ddots & & \ddots & \\ & & \mathbf{Row}(c_q) & \mathbf{Row}(c_{q-1}) & \ldots & \mathbf{Row}(c_1) \end{bmatrix}.$$

Here $\mathbf{Row}(c_1), \ldots, \mathbf{Row}(c_q) \in \mathbf{R}^{s-p+1 \times s}$ are 1-D row convolution matrices. The matrices $\mathbf{Col}(C)$ and $\mathbf{Row}(C)$ are related by the equalities

$$\mathbf{Col}(C)^T = \mathbf{Row}(\mathbf{rev}(C)), \qquad \mathbf{Row}(C)^T = \mathbf{Col}(\mathbf{rev}(C)),$$

where $\mathbf{rev}(C)_{k\ell} = C_{p-k+1, q-\ell+1}$ reverses the order of the columns of C and of the entries in each row.

In the 2-D analogue of 1-D circular convolution, we take $p = s$ and $q = t$ and interpret the entries of matrices outside their range modulo s for the row index and modulo t for the column index. We can represent the 2-D circular convolution $Y = C * X$ as the matrix multiplication

$$Y = \mathbf{mat}(\mathbf{Circ}(C)\,\mathbf{vec}(X)),$$

where $\mathbf{Circ}(C) \in \mathbf{R}^{st \times st}$ is given by:

$$\mathbf{Circ}(C) = \begin{bmatrix} \mathbf{Circ}(c_1) & \mathbf{Circ}(c_t) & \mathbf{Circ}(c_{t-1}) & \ldots & \ldots & \mathbf{Circ}(c_2) \\ \mathbf{Circ}(c_2) & \mathbf{Circ}(c_1) & \mathbf{Circ}(c_t) & \ddots & & \vdots \\ \mathbf{Circ}(c_3) & \mathbf{Circ}(c_2) & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \mathbf{Circ}(c_t) & \mathbf{Circ}(c_{t-1}) \\ \vdots & & \ddots & \mathbf{Circ}(c_2) & \mathbf{Circ}(c_1) & \mathbf{Circ}(c_t) \\ \mathbf{Circ}(c_t) & \ldots & \ldots & \mathbf{Circ}(c_3) & \mathbf{Circ}(c_2) & \mathbf{Circ}(c_1) \end{bmatrix}.$$

Here $\mathbf{Circ}(c_1), \ldots, \mathbf{Circ}(c_t) \in \mathbf{R}^{s \times s}$ are 1-D circular convolution matrices.

2-D column convolution with $C \in \mathbf{R}^{p \times q}$ is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f : \mathbf{R}^{s \times t} \to \mathbf{R}^{s+p-1 \times t+q-1}$ is given by

$$f(X) = \mathbf{mat}(\mathbf{Col}(C)\,\mathbf{vec}(X)).$$

The adjoint f^* is 2-D row convolution with $\mathbf{rev}(C)$, i.e.,

$$f^*(U) = \mathbf{mat}(\mathbf{Row}(\mathbf{rev}(C))\,\mathbf{vec}(U)).$$

The algorithms Φ_f and Φ_{f^*} are given in Algorithms 4 and 5, and require $O((m+n)\log(m+n))$ flops. Here $m=(s+p-1)(t+q-1)$ and $n=st$. If the kernel is small (i.e., $p \ll s$ and $q \ll t$), Φ_f and Φ_{f^*} instead evaluate (5) directly in $O(pqst)$ flops. In either case, the algorithms Φ_f and Φ_{f^*} use $O(pq)$ bytes of data to store C and $\mathbf{rev}(C)$ [82, Chap. 4]. Often the kernel is parameterized (e.g., a Gaussian kernel), in which case more compact representations of C and $\mathbf{rev}(C)$ are possible [44, Chap. 7].

Algorithm 4 2-D column convolution $C * X$

Precondition: $C \in \mathbf{R}^{p\times q}$ is a length pq array. $X \in \mathbf{R}^{s\times t}$ is a length st array. $Y \in \mathbf{R}^{s+p-1\times t+q-1}$ is a length $(s+p-1)(t+q-1)$ array.

Extend the columns and rows of C and X with zeros so $C, X \in \mathbf{R}^{s+p-1\times t+q-1}$.
$\hat{C} \leftarrow$ 2-D DFT of C.
$\hat{X} \leftarrow$ 2-D DFT of X.
for $i=1,\ldots,s+p-1$ **do**
    for $j=1,\ldots,t+q-1$ **do**
        $Y_{ij} \leftarrow \hat{C}_{ij}\hat{X}_{ij}$.
    end for
end for
$Y \leftarrow$ inverse 2-D DFT of Y.

Postcondition: $Y = C * X$.

Algorithm 5 2-D row convolution $C * U$

Precondition: $C \in \mathbf{R}^{p\times q}$ is a length pq array. $U \in \mathbf{R}^{s+p-1\times t+q-1}$ is a length $(s+p-1)(t+q-1)$ array. $V \in \mathbf{R}^{s\times t}$ is a length st array.

Extend the columns and rows of $\mathbf{rev}(C)$ and V with zeros so $\mathbf{rev}(C), V \in \mathbf{R}^{s+p-1\times t+q-1}$.
$\hat{C} \leftarrow$ inverse 2-D DFT of zero-padded $\mathbf{rev}(C)$.
$\hat{U} \leftarrow$ 2-D DFT of U.
for $i=1,\ldots,s+p-1$ **do**
    for $j=1,\ldots,t+q-1$ **do**
        $V_{ij} \leftarrow \hat{C}_{ij}\hat{U}_{ij}$.
    end for
end for
$V \leftarrow$ inverse 2-D DFT of V.
Truncate the rows and columns of V so that $V \in \mathbf{R}^{s\times t}$.

Postcondition: $V = C * U$.

Algorithm 6 2-D circular convolution $C * X$

Precondition: $C \in \mathbf{R}^{s\times t}$ is a length st array. $X \in \mathbf{R}^{s\times t}$ is a length st array. $Y \in \mathbf{R}^{s\times t}$ is a length st array.

$\hat{C} \leftarrow$ 2-D DFT of C.
$\hat{X} \leftarrow$ 2-D DFT of X.
for $i = 1, \ldots, s$ **do**
  for $j = 1, \ldots, t$ **do**
    $Y_{ij} \leftarrow \hat{C}_{ij}\hat{X}_{ij}$.
  end for
end for
$Y \leftarrow$ inverse 2-D DFT of Y.

Postcondition: $Y = C * X$.

2-D circular convolution with $C \in \mathbf{R}^{s\times t}$ is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f : \mathbf{R}^{s\times t} \to \mathbf{R}^{s\times t}$ is given by

$$f(X) = \mathbf{mat}(\mathbf{Circ}(C)\,\mathbf{vec}(X)).$$

The adjoint f^* is 2-D circular convolution with

$$\tilde{C} = \begin{bmatrix} C_{1,1} & C_{1,t} & C_{1,t-1} & \ldots & C_{1,2} \\ C_{s,1} & C_{s,t} & C_{s,t-1} & \ldots & C_{s,2} \\ C_{s-1,1} & C_{s-1,t} & C_{s-1,t-1} & \ldots & C_{s-1,2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_{2,1} & C_{2,t} & C_{2,t-1} & \ldots & C_{2,2} \end{bmatrix}.$$

The algorithms Φ_f and Φ_{f^*} are given in Algorithm 6, and require $O((m+n)\log(m+n))$ flops. The algorithms Φ_f and Φ_{f^*} use $O(m+n)$ bytes of data to store C and $\tilde{C}$ [82, Chap. 4]. Here $m = n = st$.

2-D Discrete Wavelet Transform The 2-D DWT for separable, orthogonal wavelets is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f : \mathbf{R}^{2^p\times 2^p} \to \mathbf{R}^{2^p\times 2^p}$ is given by

$$f(X)_{ij} = W_k \cdots W_{p-1} W_p X W_p^T W_{p-1}^T \cdots W_k^T,$$

where $k = \max\{\lceil \log_2(i)\rceil, \lceil \log_2(j)\rceil, 1\}$ and $W_k \in \mathbf{R}^{2^p\times 2^p}$ is given by

$$W_k = \begin{bmatrix} D_k G_k & \\ D_k H_k & \\ & I \end{bmatrix}.$$

Here D_k, G_k, and H_k are defined as for the 1-D DWT. The adjoint f^* is the inverse 2-D DWT. As in the 1-D DWT, the algorithms Φ_f and Φ_{f^*} repeatedly convolve by the filters $g \in \mathbf{R}^q$ and $h \in \mathbf{R}^q$, which requires $O(q(m+n))$ flops and uses $O(q)$ bytes of data to store g and h [70]. Here $m = n = 2^p$. There are many alternative wavelet transforms for 2-D data; see, e.g., [20, 38, 69, 102].

2.3 Multiple Vector Mappings

In this section we consider linear functions that take as argument, or return, multiple vectors. (The idea is readily extended to the case when the arguments or return values are matrices.) The adjoint is defined by the inner product

$$\langle (x_1, \dots, x_k), (y_1, \dots, y_k) \rangle = \sum_{i=1}^{k} \langle x_i, y_i \rangle = \sum_{i=1}^{k} x_i^T y_i.$$

The adjoint of a linear function $f : \mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_k} \to \mathbf{R}^{m_1} \times \cdots \times \mathbf{R}^{m_\ell}$ is then the function $f^* : \mathbf{R}^{m_1} \times \cdots \times \mathbf{R}^{m_\ell} \to \mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_k}$ for which

$$\sum_{i=1}^{\ell} f(x_1, \dots, x_k)_i^T y_i = \sum_{i=1}^{k} x_i^T f^*(y_1, \dots, y_\ell)_i,$$

holds for all $(x_1, \dots, x_k) \in \mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_k}$ and $(y_1, \dots, y_\ell) \in \mathbf{R}^{m_1} \times \cdots \times \mathbf{R}^{m_\ell}$. Here $f(x_1, \dots, x_k)_i$ and $f^*(y_1, \dots, y_\ell)_i$ refer to the ith output of f and f^*, respectively.

Sum and Copy The function **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 $f(x_1, \dots, x_k) = x_1 + \cdots + x_k$. The adjoint f^* is the function **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 $n = km$.

Vstack and Split The function **vstack** : $\mathbf{R}^{m_1} \times \cdots \times \mathbf{R}^{m_k} \to \mathbf{R}^n$ is represented by the FAO $\Gamma = (f, \Phi_f, \Phi_{f^*})$, where $f(x_1, \dots, x_k)$ concatenates its k inputs into a single vector output. The adjoint f^* is the function **split** : $\mathbf{R}^n \to \mathbf{R}^{m_1} \times \cdots \times \mathbf{R}^{m_k}$, which divides a single vector into k separate components. The algorithms Φ_f and Φ_{f^*} simply reinterpret their input as a differently sized output in $O(1)$ flops, using only $O(1)$ bytes of data to store the dimensions of f's input and output. Here $n = m = m_1 + \cdots + m_k$.

2.4 Additional Examples

The literature on fast linear transforms goes far beyond the preceding examples. In this section we highlight a few notable omissions. Many methods have been developed for matrices derived from physical systems. The multigrid [62] and algebraic multigrid [18] methods efficiently apply the inverse of a matrix representing discretized partial differential equations (PDEs). The fast multipole method accelerates multiplication by matrices representing pairwise interactions [21, 59], much like the fast Gauss transform [60]. Hierarchical matrices are a matrix format that allows fast multiplication by the matrix and its inverse, with applications to discretized integral operators and PDEs [14, 63, 64].

Many approaches exist for factoring an invertible sparse matrix into a product of components whose inverses can be applied efficiently, yielding a fast method for applying the inverse of the matrix [34, 41]. A sparse LU factorization, for instance, decomposes an invertible sparse matrix $A \in \mathbf{R}^{n\times n}$ into the product $A = LU$ of a lower triangular matrix $L \in \mathbf{R}^{n\times n}$ and an upper triangular matrix $U \in \mathbf{R}^{n\times n}$. The relationship between $\mathbf{nnz}(A)$, $\mathbf{nnz}(L)$, and $\mathbf{nnz}(U)$ is complex and depends on the factorization algorithm [34, Chap. 6].

We only discussed 1-D and 2-D DFTs and convolutions, but these and related transforms can be extended to arbitrarily many dimensions [40, 82]. Similarly, many wavelet transforms naturally operate on data indexed by more than two dimensions [76, 84, 116].

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 sparse matrix mappings. 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*.

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^*}.
- A list E_{in} of incoming edges.
- A list E_{out} of outgoing edges.

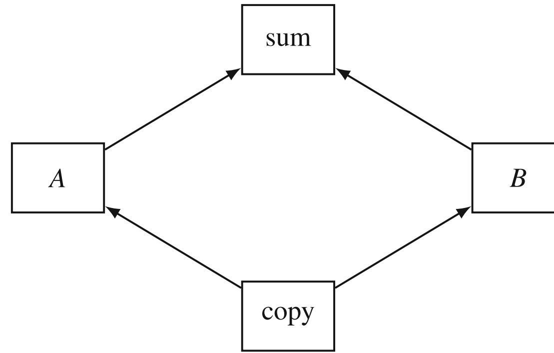

Fig. 1 The FAO DAG for $f(x) = Ax + Bx$

Each edge has an associated array. The incoming edges to a node store the arguments to the node's FAO. When the FAO is evaluated, it writes the result to the node's outgoing edges. Matrix arguments and outputs are stored in column-major order on the edge arrays.

As an example, Fig. 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 **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 **sum** node sums two vectors together. The **copy** node is the start node, and the **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 transposes. The edge arrays also require $O(mn)$ bytes of memory.

3.1 Forward Evaluation

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

The general procedure for evaluating an FAO DAG is given in Algorithm 7. 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.

3.2 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}.

Algorithm 7 Evaluate an FAO DAG

Precondition: $G = (V, E)$ 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.

Create edges whose arrays are the elements of I and save them as the list of incoming edges for the start node.
Create edges whose arrays are the elements of O and save them as the list of outgoing edges for the end node.
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**
  $u \leftarrow$ pop the front node of Q.
  Evaluate u's algorithm Φ_f on u's incoming edges, writing the result to u's outgoing edges.
  Add u to S.
  for each edge $e = (u, v)$ in u's E_{out} **do**
    if for all edges (p, v) in v's E_{in}, p is in S **then**
      Add v to the end of Q.
    end if
  end for
end while

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

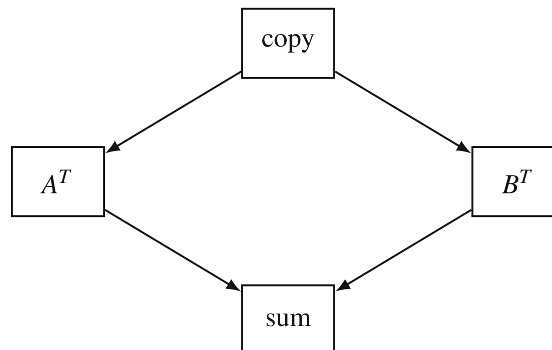

Fig. 2 The FAO DAG for $f^*(u) = A^T u + B^T u$ obtained by transforming the FAO DAG in Fig. 1

We also reverse the orientation of each edge in G. We can apply Algorithm 7 to the resulting graph G^* to evaluate f^*. Figure 2 shows the FAO DAG in Fig. 1 transformed into an FAO DAG for the adjoint.

3.3 Parallelism

Algorithm 7 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 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 have simple parallel variants.

The extent to which parallelism speeds up evaluation of an FAO DAG is difficult to predict. Naive parallel evaluation may be slower than serial evaluation due to communication costs and other overhead. Achieving a perfect parallel speed-up would require sophisticated analysis of the DAG to determine which aspects of the algorithm to parallelize, and may only be possible for highly structured DAGs like one describing a block matrix [54].

3.4 Optimizing the DAG

The FAO DAG can often be transformed so that the output of Algorithm 7 is the same but the algorithm is executed more efficiently. Such optimizations are especially important when the FAO DAG will be evaluated on many different inputs (as will be the case for matrix-free solvers, to be discussed later). For example, the FAO DAG representing $f(x) = ABx + ACx$ where $A, B, C \in \mathbf{R}^{n \times n}$, shown in Fig. 3, can be transformed into the FAO DAG in Fig. 4, which requires one fewer multiplication by A. The transformation is equivalent to rewriting $f(x) = ABx+ACx$ as $f(x) = A(Bx+Cx)$. Many other useful graph transformations can be derived from the rewriting rules used in program analysis and code generation [3].

Sometimes graph transformations will involve pre-computation. For example, if two nodes representing the composition $f(x) = b^T cx$, where $b, c \in \mathbf{R}^n$, appear in an FAO DAG, the DAG can be made more efficient by evaluating $\alpha = b^T c$ and replacing the two nodes with a single node for scalar multiplication by α.

The optimal rewriting of a DAG will depend on the hardware and overall architecture on which the multiplication algorithm is being run. For example, if the

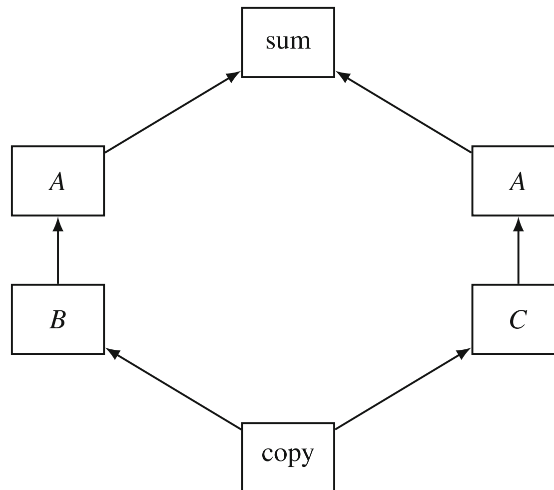

Fig. 3 The FAO DAG for $f(x) = ABx + ACx$

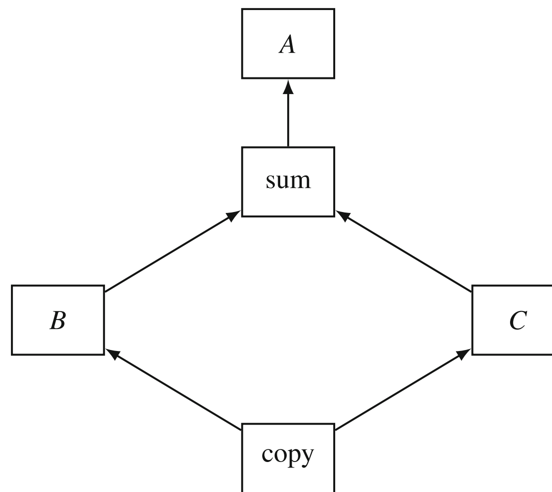

Fig. 4 The FAO DAG for $f(x) = A(Bx + Cx)$

algorithm is being run on a distributed computing cluster then a node representing multiplication by a large matrix

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

could be split into separate nodes for each block, with the nodes stored on different computers. This rewriting would be necessary if the matrix A is so large it cannot be stored on a single machine. The literature on optimizing compilers suggests many approaches to optimizing an FAO DAG for evaluation on a particular architecture [3].

3.5 Reducing the Memory Footprint

In a naive implementation, the total bytes needed to represent an FAO DAG G, with node set V and edge set E, is the sum of the bytes of data on each node $u \in V$ and the bytes of memory needed for the array on each edge $e \in E$. A more sophisticated approach can substantially reduce the memory needed. For example, when the same FAO occurs more than once in V, duplicate nodes can share data.

We can also reuse memory across edge arrays. The key is determining which arrays can never be in use at the same time during Algorithm 7. An array for an edge (u, v) is in use if node u has been evaluated but node v has not been evaluated. The arrays for edges (u_1, v_1) and (u_2, v_2) can never be in use at the same time if and only if there is a directed path from v_1 to u_2 or from v_2 to u_1. If the sequence in which the nodes will be evaluated is fixed, rather than following an unknown topological ordering, then we can say precisely which arrays will be in use at the same time.

After we determine which edge arrays may be in use at the same time, the next step is to map the edge arrays onto a global array, keeping the global array as small as possible. Let $L(e)$ denote the length of edge e's array and $U \subseteq E \times E$ denote the set of pairs of edges whose arrays may be in use at the same time. Formally, we want to solve the optimization problem

$$\begin{array}{ll} \text{minimize} & \max_{e \in E}\{z_e + L(e)\} \\ \text{subject to} & [z_e, z_e + L(e) - 1] \cap [z_f, z_f + L(f) - 1] = \emptyset, \quad (e,f) \in U \\ & z_e \in \{1, 2, \ldots\}, \quad e \in E, \end{array} \tag{6}$$

where the z_e are the optimization variables and represent the index in the global array where edge e's array begins.

When all the edge arrays are the same length, problem (6) is equivalent to finding the chromatic number of the graph with vertices E and edges U. Problem (6) is thus NP-hard in general [72]. A reasonable heuristic for problem (6) is to first find a graph coloring of (E, U) using one of the many efficient algorithms for finding graph colorings that use a small number of colors; see, e.g., [19, 65]. We then have a mapping ϕ from colors to sets of edges assigned to the same color. We order the colors arbitrarily as $c_1, \ldots, c_k$ and assign the z_e as follows:

$$z_e = \begin{cases} 1, & e \in \phi(c_1) \\ \max_{f \in \phi(c_{i-1})}\{z_f + L(f)\}, & e \in \phi(c_i), \quad i > 1. \end{cases}$$

Additional optimizations can be made based on the unique characteristics of different FAOs. For example, the outgoing edges from a **copy** node can share the incoming edge's array until the outgoing edges' arrays are written to (i.e., copy-on-write). Another example is that the outgoing edges from a **split** node can point to segments of the array on the incoming edge. Similarly, the incoming edges on a **vstack** node can point to segments of the array on the outgoing edge.

3.6 Software Implementations

Several software packages have been developed for constructing and evaluating compositions of linear functions. The MATLAB toolbox SPOT allows users to construct expressions involving both fast transforms, like convolution and the DFT, and standard matrix multiplication [66]. TFOCS, a framework in MATLAB for solving convex problems using a variety of first-order algorithms, provides functionality for constructing and composing FAOs [11]. The Python package `linop` provides methods for constructing FAOs and combining them into linear expressions [107]. Halide is a domain specific language for image processing that makes it easy to optimize compositions of fast transforms for a variety of architectures [98].

Our approach to representing and evaluating compositions of functions is similar to the approach taken by autodifferentiation tools. These tools represent a composite function $f : \mathbf{R}^n \to \mathbf{R}^m$ as a DAG [61], and multiply by the Jacobian $J \in \mathbf{R}^{m \times n}$ and its adjoint efficiently through graph traversal. Forward mode autodifferentiation computes $x \to Jx$ efficiently by traversing the DAG in topological order. Reverse mode autodifferentiation, or backpropagation, computes $u \to J^T u$ efficiently by traversing the DAG once in topological order and once in reverse topological order [8]. An enormous variety of software packages have been developed for autodifferentiation; see [8] for a survey. Autodifferentiation in the form of backpropagation plays a central role in deep learning frameworks such as TensorFlow [1], Theano [7, 13], Caffe [71], and Torch [29].

4 Cone Programs and Solvers

In this section we describe cone programs, the standard intermediate-form representation of a convex problem, and solvers for cone programs.

4.1 Cone Programs

A cone program is a convex optimization problem of the form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax + b \in \mathcal{K}, \end{array} \tag{7}$$

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 [16, 92]. 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: $\mathcal{K}_{\text{psd}} = \{\mathbf{vec}(X) \mid X \in \mathbf{S}^n,\ z^T X z \geq 0 \text{ for all } z \in \mathbf{R}^n\}$.
- Exponential cone ([95, Sect. 6.3.4]):

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

- Power cone [68, 90, 100]:

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

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). Note that all the cones are subsets of $\mathbf{R}^n$, i.e., real vectors. It might be more natural to view the elements of a cone as matrices or tuples, but viewing the elements as vectors simplifies the matrix-free canonicalization algorithm in Sect. 5.

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.

4.2 Cone Solvers

Many methods have been developed to solve cone programs, the most widely used being interior-point methods; see, e.g., [16, 91, 93, 112, 115].

Interior-Point A large number of interior-point cone solvers have been implemented. Most support symmetric cone programs. SDPT3 [105] and SeDuMi [103] are open-source solvers implemented in MATLAB; CVXOPT [6] is an open-source solver implemented in Python; MOSEK [89] 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 [39]; Akle extended ECOS to support the exponential cone [4]. DSDP5 [12] and SDPA [47] 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 [26] is a first-order cone solver based on the alternating direction method of multipliers (ADMM) [15]. PDOS supports second-order cone programs. POGS [45] 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 [94]. Many other first-order algorithms can be applied to cone programs (e.g., [22, 78, 96]), 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 [75]. 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. Many other matrix-free algorithms for solving SDPs have been proposed (e.g., [25, 48, 104, 118]). CVXOPT can be used as a matrix-free cone solver, as it allows users to specify linear functions as Python functions for evaluating matrix-vector products, rather than as explicit matrices [5].

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 [52, 53, 67]. PDCO is a matrix-free interior-point solver that can solve QPs [99], using LSMR to solve linear systems [43].

5 Matrix-Free 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 [83], CVX [57], CVXPY [36], and Convex.jl [106]. The same technique is used in the code generators CVXGEN [87] and QCML [27].

The downside of canonicalization's generality is that special structure in the original problem may be lost during the transformation into a cone program. In particular, 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*.

The remainder of this section has the following outline: In Sect. 5.1 we give an informal overview of the matrix-free canonicalization algorithm. In Sect. 5.2 we define the expression DAG data structure, which is used throughout the matrix-free canonicalization algorithm. In Sect. 5.3 we define the data structure used to represent convex optimization problems as input to the algorithm. In Sect. 5.4 we define the representation of a cone program output by the matrix-free canonicalization algorithm. In Sect. 5.5 we present the matrix-free canonicalization algorithm itself.

For clarity, we move some details of canonicalization to the appendix. In "Equivalence of the Cone Program" we give a precise definition of the equivalence between the cone program output by the canonicalization algorithm and the original convex optimization problem given as input. In "Sparse Matrix Representation" we explain how the standard canonicalization algorithm generates a sparse matrix representation of a cone program.

5.1 Informal Overview

In this section we give an informal overview of the matrix-free canonicalization algorithm. Later sections define the data structures used in the algorithm and make the procedure described in this section formal and explicit.

We are given an optimization problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\ & h_i(x) + d_i = 0, \quad i = 1, \ldots, q, \end{array} \tag{8}$$

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* [55, 58]. 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

$$\begin{array}{lll} f_i(x) = & \text{minimize (over } t^{(i)}) & g_0^{(i)}(x, t^{(i)}) + e_0^{(i)} \\ & \text{subject to} & g_j^{(i)}(x, t^{(i)}) + e_j^{(i)} \in \mathcal{K}_j^{(i)}, \quad j = 1, \ldots, r^{(i)}, \end{array}$$

where $t^{(i)} \in \mathbf{R}^{s^{(i)}}$ is the optimization variable, $g_0^{(i)}, \ldots, g_{r^{(i)}}^{(i)}$ 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.

We rewrite problem (8) as the equivalent cone program

$$\begin{array}{ll} \text{minimize} & g_0^{(0)}(x, t^{(0)}) + e_0^{(0)} \\ \text{subject to} & -g_0^{(i)}(x, t^{(i)}) - e_0^{(i)} \in \mathcal{K}_+, \quad i = 1, \ldots, p, \\ & g_j^{(i)}(x, t^{(i)}) + e_j^{(i)} \in \mathcal{K}_j^{(i)} \quad i = 1, \ldots, p, \quad j = 1, \ldots, r^{(i)} \\ & h_i(x) + d_i \in \mathcal{K}_0^{m_i}, \quad i = 1, \ldots, q. \end{array} \tag{9}$$

We convert problem (9) into the standard form for a matrix-free cone program given in (7) by representing $g_0^{(0)}$ as the inner product with a vector $c \in \mathbf{R}^{n+s^{(0)}}$,

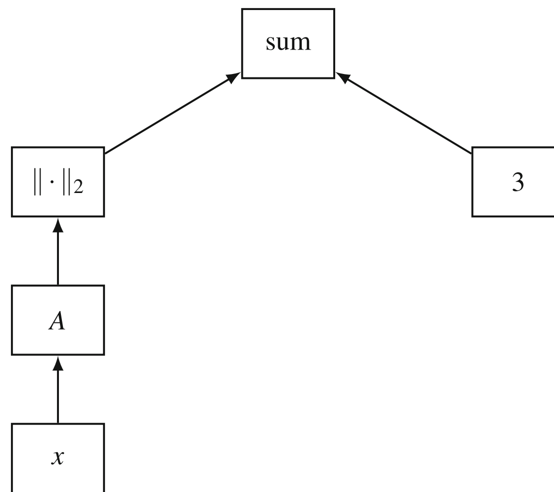

Fig. 5 The expression DAG for $f(x) = \|Ax\|_2 + 3$

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.

5.2 *Expression DAGs*

The canonicalization algorithm uses a data structure called an *expression DAG* to represent functions in an optimization problem. Like the FAO DAG defined in Sect. 3, an expression DAG encodes a composition of functions as a DAG where a node represents a function and an edge from a node u to a node v signifies that an output of u is an input to v. Figure 5 shows an expression DAG for the composition $f(x) = \|Ax\|_2 + 3$, where $x \in \mathbf{R}^n$ and $A \in \mathbf{R}^{m\times n}$.

Formally, an expression DAG is a connected DAG with one node with no outgoing edges (the end node) and one or more nodes with no incoming edges (start nodes). Each node in an expression DAG has the following attributes:

- A symbol representing a function f.
- The data needed to parameterize the function, such as the power p for the function $f(x) = x^p$.
- A list E_{in} of incoming edges.
- A list E_{out} of outgoing edges.

Each start node in an expression DAG is either a constant function or a variable. A variable is a symbol that labels a node input. If two nodes u and v both have incoming edges from variable nodes with symbol t, then the inputs to u and v are the same.

We say an expression DAG is affine if every non-start node represents a linear function. If in addition every start node is a variable, we say the expression DAG is linear. We say an expression DAG is constant if it contains no variables, i.e., every start node is a constant.

5.3 Optimization Problem Representation

An *optimization problem representation* (OPR) is a data structure that represents a convex optimization problem. The input to the matrix-free canonicalization algorithm is an OPR. An OPR can encode any mathematical optimization problem of the form

$$\begin{array}{ll} \text{minimize (over } y \text{ w.r.t. } \mathcal{K}_0) & f_0(x,y) \\ \text{subject to} & f_i(x,y) \in \mathcal{K}_i, \quad i = 1,\dots,\ell, \end{array} \tag{10}$$

where $x \in \mathbf{R}^n$ and $y \in \mathbf{R}^m$ are the optimization variables, $\mathcal{K}_0$ is a proper cone, $\mathcal{K}_1,\dots,\mathcal{K}_\ell$ are convex cones, and for $i = 0,\dots,\ell$, we have $f_i : \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}^{m_i}$ where $\mathcal{K}_i \subseteq \mathbf{R}^{m_i}$. (For background on convex optimization with respect to a cone, see, e.g., [16, Sect. 4.7].)

Problem (10) is more complicated than the standard definition of a convex optimization problem given in (8). The additional complexity is necessary so that OPRs can encode partial minimization over cone programs, which can involve minimization with respect to a cone and constraints other than equalities and inequalities. These partial minimization problems play a major role in the canonicalization algorithm. Note that we can easily represent equality and inequality constraints using the zero and nonnegative cones.

Concretely, an OPR is a tuple (s, o, C) where

- The element s is a tuple $(V, \mathcal{K})$ representing the problem's objective sense. The element V is a set of symbols encoding the variables being minimized over. The element $\mathcal{K}$ is a symbol encoding the proper cone the problem objective is being minimized with respect to.
- The element o is an expression DAG representing the problem's objective function.
- The element C is a set representing the problem's constraints. Each element $c_i \in C$ is a tuple $(e_i, \mathcal{K}_i)$ representing a constraint of the form $f(x,y) \in \mathcal{K}$. The element e_i is an expression DAG representing the function f and $\mathcal{K}_i$ is a symbol encoding the convex cone $\mathcal{K}$.

The matrix-free canonicalization algorithm can only operate on OPRs that satisfy the two DCP requirements [55, 58]. The first requirement is that each nonlinear function in the OPR have a known representation as partial minimization over a cone program. See [56] for many examples of such representations.

The second requirement is that the objective o be verifiable as convex with respect to the cone $\mathcal{K}$ in the objective sense s by the DCP composition rule. Similarly, for each element $(e_i, \mathcal{K}_i) \in C$, the constraint that the function represented by e_i lie in the convex cone represented by $\mathcal{K}_i$ must be verifiable as convex by the composition rule. The DCP composition rule determines the curvature of a composition $f(g_1(x),\dots,g_k(x))$ from the curvatures and ranges of the arguments $g_1,\dots,g_k$, the curvature of the function f, and the monotonicity of f on the range of

its arguments. See [55] and [106] for a full discussion of the DCP composition rule. Additional rules are used to determine the range of a composition from the range of its arguments.

Note that it is not enough for the objective and constraints to be convex. They must also be structured so that the DCP composition rule can verify their convexity. Otherwise the cone program output by the matrix-free canonicalization algorithm is not guaranteed to be equivalent to the original problem.

To simplify the exposition of the canonicalization algorithm, we will also require that the objective sense s represent minimization over all the variables in the problem with respect to the nonnegative cone, i.e., the standard definition of minimization. The most general implementation of canonicalization would also accept OPRs that can be transformed into an equivalent OPR with an objective sense that meets this requirement.

5.4 Cone Program Representation

The matrix-free canonicalization algorithm outputs a tuple $(c_{\mathrm{arr}}, d_{\mathrm{arr}}, b_{\mathrm{arr}}, G, \mathcal{K}_{\mathrm{list}})$ where

- The element c_{arr} is a length n array representing a vector $c \in \mathbf{R}^n$.
- The element d_{arr} is a length one array representing a scalar $d \in \mathbf{R}$.
- The element b_{arr} is a length m array representing a vector $b \in \mathbf{R}^m$.
- The element G is an FAO DAG representing a linear function $f(x) = Ax$, where $A \in \mathbf{R}^{m\times n}$.
- The element $\mathcal{K}_{\mathrm{list}}$ is a list of symbols representing the convex cones $(\mathcal{K}_1, \ldots, \mathcal{K}_\ell)$.

The tuple represents the matrix-free cone program

$$\begin{array}{ll} \text{minimize} & c^T x + d \\ \text{subject to} & Ax + b \in \mathcal{K}, \end{array} \tag{11}$$

where $\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_\ell$.

We can use the FAO DAG G and Algorithm 7 to represent A as an FAO, i.e., export methods for multiplying by A and A^T. These two methods are all a matrix-free cone solver needs to efficiently solve problem (11).

5.5 Algorithm

The matrix-free canonicalization algorithm can be broken down into subroutines. We describe these subroutines before presenting the overall algorithm.

`Conic-Form` The `Conic-Form` subroutine takes an OPR as input and returns an equivalent OPR where every non-start node in the objective and constraint expression DAGs represents a linear function. The output of the `Conic-Form` subroutine represents a cone program, but the output must still be transformed into a data structure that a cone solver can use, e.g., the cone program representation described in Sect. 5.4.

The general idea of the `Conic-Form` algorithm is to replace each nonlinear function in the OPR with an OPR representing partial minimization over a cone program. Recall that the canonicalization algorithm requires that all nonlinear functions in the problem be representable as partial minimization over a cone program. The OPR for each nonlinear function is spliced into the full OPR. We refer the reader to [56] and [106] for a full discussion of the `Conic-Form` algorithm.

The `Conic-Form` subroutine preserves fast linear transforms in the problem. All linear functions in the original OPR are present in the OPR output by `Conic-Form`. The only linear functions added are ones like **sum** and scalar multiplication that are very efficient to evaluate. Thus, evaluating the FAO DAG representing the final cone program will be as efficient as evaluating all the linear functions in the original problem (8).

`Linear` and `Constant` The `Linear` and `Constant` subroutines take an affine expression DAG as input and return the DAG's linear and constant components, respectively. Concretely, the `Linear` subroutine returns a copy of the input DAG where every constant start node is replaced with a variable start node and a node mapping the variable output to a vector (or matrix) of zeros with the same dimensions as the constant. The `Constant` subroutine returns a copy of the input DAG where every variable start node is replaced with a zero-valued constant node of the same dimensions. Figures 7 and 8 show the results of applying the `Linear` and `Constant` subroutines to an expression DAG representing $f(x) = x + 2$, as depicted in Fig. 6.

`Evaluate` The `Evaluate` subroutine takes a constant expression DAG as input and returns an array. The array contains the value of the function represented by the expression DAG. If the DAG evaluates to a matrix $A \in \mathbf{R}^{m \times n}$, the array represents $\mathbf{vec}(A)$. Similarly, if the DAG evaluates to multiple output vectors $(b_1, \ldots, b_k) \in \mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_k}$, the array represents $\mathbf{vstack}(b_1, \ldots, b_k)$. For example, the output of the `Evaluate` subroutine on the expression DAG in Fig. 8 is a length one array with first entry equal to 2.

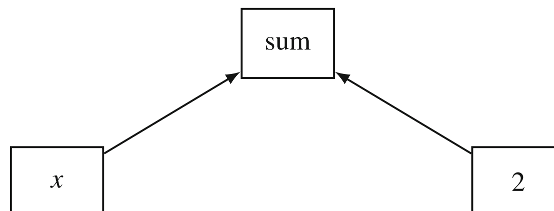

Fig. 6 The expression DAG for $f(x) = x + 2$

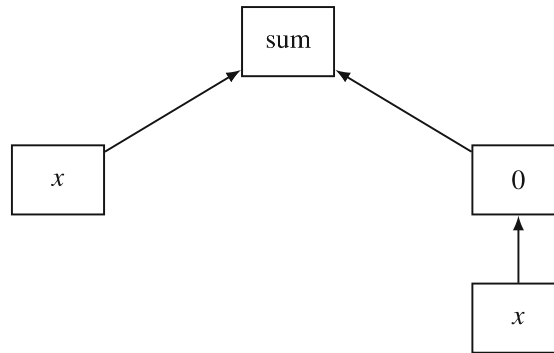

Fig. 7 The `Linear` subroutine applied to the expression DAG in Fig. 6

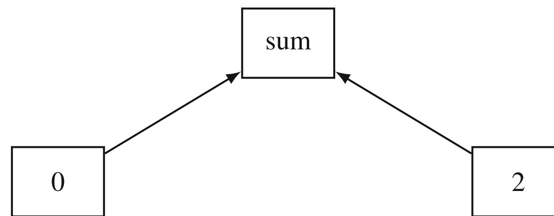

Fig. 8 The `Constant` subroutine applied to the expression DAG in Fig. 6

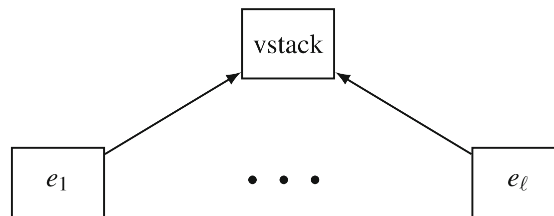

Fig. 9 The expression DAG for $\mathbf{vstack}(e_1, \ldots, e_\ell)$

`Graph-Repr` The `Graph-Repr` subroutine takes a list of linear expression DAGs, $(e_1, \ldots, e_\ell)$, and an ordering over the variables in the expression DAGs, $<_V$, as input and outputs an FAO DAG G. We require that the end node of each expression DAG represent a function with a single vector as output.

We construct the FAO DAG G in three steps. In the first step, we combine the expression DAGs into a single expression DAG $H^{(1)}$ by creating a **vstack** node and adding an edge from the end node of each expression DAG to the new node. The expression DAG $H^{(1)}$ is shown in Fig. 9.

In the second step, we transform $H^{(1)}$ into an expression DAG $H^{(2)}$ with a single start node. Let $x_1, \ldots, x_k$ be the variables in $(e_1, \ldots, e_\ell)$ ordered by $<_V$. Let n_i be the length of x_i if the variable is a vector and of $\mathbf{vec}(x_i)$ if the variable is a matrix, for $i = 1, \ldots, k$. We create a start node representing the function $\mathbf{split} : \mathbf{R}^n \to \mathbf{R}^{n_1} \times \cdots \times \mathbf{R}^{n_k}$. For each variable x_i, we add an edge from output i of the start node to a **copy** node and edges from that **copy** node to all the nodes representing x_i. If x_i is a vector, we replace all the nodes representing x_i with nodes representing the identity function. If x_i is a matrix, we replace all the nodes representing x_i with **mat** nodes. The transformation from $H^{(1)}$ to $H^{(2)}$ when $\ell = 1$ and e_1 represents $f(x) = x + A(x + y)$, where $x, y \in \mathbf{R}^n$ and $A \in \mathbf{R}^{n \times n}$, are depicted in Figs. 10 and 11.

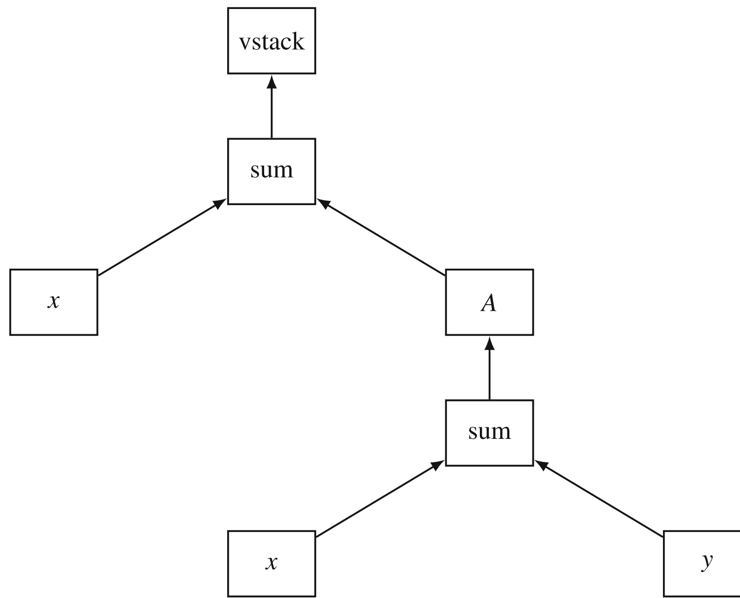

Fig. 10 The expression DAG $H^{(1)}$ when $\ell = 1$ and e_1 represents $f(x, y) = x + A(x + y)$

In the third and final step, we transform $H^{(2)}$ from an expression DAG into an FAO DAG G. $H^{(2)}$ is almost an FAO DAG, since each node represents a linear function and the DAG has a single start and end node. To obtain G we simply add the node and edge attributes needed in an FAO DAG. For each node u in $H^{(2)}$ representing the function f, we add to u an FAO (f, Φ_f, Φ_{f^*}) and the data needed to evaluate Φ_f and Φ_{f^*}. The node already has the required lists of incoming and outgoing edges. We also add an array to each of $H^{(2)}$'s edges.

`Optimize-Graph` The `Optimize-Graph` subroutine takes an FAO DAG G as input and outputs an equivalent FAO DAG G^{opt}, meaning that the output of Algorithm 7 is the same for G and G^{opt}. We choose G^{opt} by optimizing G so that the runtime of Algorithm 7 is as short as possible (see Sect. 3.4). We also compress the FAO data and edge arrays to reduce the graph's memory footprint (see Sect. 3.5). We could optimize the graph for the adjoint, G^*, as well, but asymptotically at least the flop count and memory footprint for G^* will be the same as for G, meaning optimizing G is the same as jointly optimizing G and G^*.

`Matrix-Repr` The `Matrix-Repr` subroutine takes a list of linear expression DAGs, $(e_1, \ldots, e_\ell)$, and an ordering over the variables in the expression DAGs, $<_V$, as input and outputs a sparse matrix. Note that the input types are the same as in the `Graph-Repr` subroutine. In fact, for a given input the sparse matrix output by `Matrix-Repr` represents the same linear function as the FAO DAG output by `Graph-Repr`. The `Matrix-Repr` subroutine is used by the standard canonicalization algorithm to produce a sparse matrix representation of a cone program. The implementation of `Matrix-Repr` is described in the appendix in "Sparse Matrix Representation".

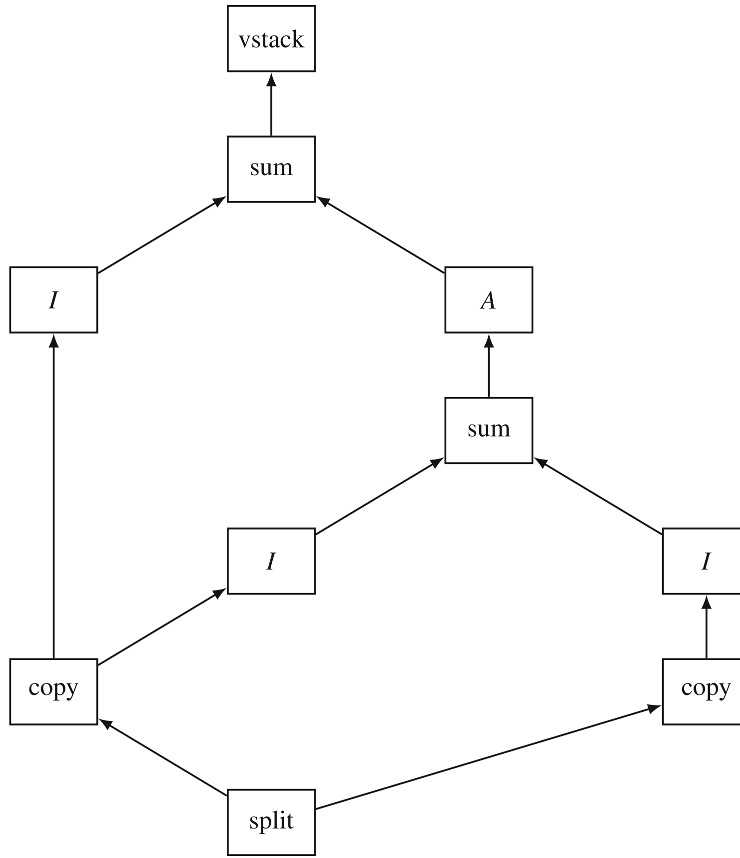

Fig. 11 The expression DAG $H^{(2)}$ obtained by transforming $H^{(1)}$ in Fig. 10

Overall Algorithm With all the subroutines in place, the matrix-free canonicalization algorithm is straightforward. The implementation is given in Algorithm 8.

Algorithm 8 Matrix-free canonicalization

Precondition: p is an OPR that satisfies the requirements of DCP.

$(s, o, C) \leftarrow$ `Conic-Form`(p).
Choose any ordering $<_V$ on the variables in (s, o, C).
Choose any ordering $<_C$ on the constraints in C.
$((e_1, \mathcal{K}_1), \ldots, (e_\ell, \mathcal{K}_\ell)) \leftarrow$ the constraints in C ordered according to $<_C$.
$c_{\text{mat}} \leftarrow$ `Matrix-Repr`((`Linear`(o)), $<_V$).
Convert c_{mat} from a 1-by-n sparse matrix into a length n array c_{arr}.
$d_{\text{arr}} \leftarrow$ `Evaluate`(`Constant`(o)).
$b_{\text{arr}} \leftarrow$ **vstack**(`Evaluate`(`Constant`(e_1)), ..., `Evaluate`(`Constant`(e_ℓ))).
$G \leftarrow$ `Graph-Repr`((`Linear`(e_1), ..., `Linear`(e_ℓ)), $<_V$).
$G^{\text{opt}} \leftarrow$ `Optimize-Graph`(G)
$\mathcal{K}_{\text{list}} \leftarrow (\mathcal{K}_1, \ldots, \mathcal{K}_\ell)$.
return $(c_{\text{arr}}, d_{\text{arr}}, b_{\text{arr}}, G^{\text{opt}}, \mathcal{K}_{\text{list}})$.

---

6 Numerical Results

We have implemented the matrix-free canonicalization algorithm as an extension of CVXPY [36], available at

https://github.com/mfopt/mf_cvxpy.

To solve the resulting matrix-free cone programs, we implemented modified versions of SCS [94] and POGS [45] that are truly matrix-free, available at

https://github.com/mfopt/mf_scs,
https://github.com/mfopt/mf_pogs.

(The main modification was using the matrix-free equilibration described in [37].) 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.

In this section we benchmark our implementation of matrix-free canonicalization and of matrix-free SCS and POGS on several convex optimization problems involving fast linear transforms. We compare the performance of our matrix-free convex optimization modeling system with that of the current CVXPY modeling system, which represents the matrix A in a cone program as a sparse matrix and uses standard cone solvers. The standard cone solvers and matrix-free SCS were run serially on a single Intel Xeon processor, while matrix-free POGS was run on a Titan X GPU.

6.1 Nonnegative Deconvolution

We applied our matrix-free convex optimization modeling system to the nonnegative deconvolution problem (1). 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 = norm2(conv(c, x) - b)
prob = Problem(Minimize(cost),
               [x >= 0])
# Solve using matrix-free SCS.
prob.solve(solver=MF_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

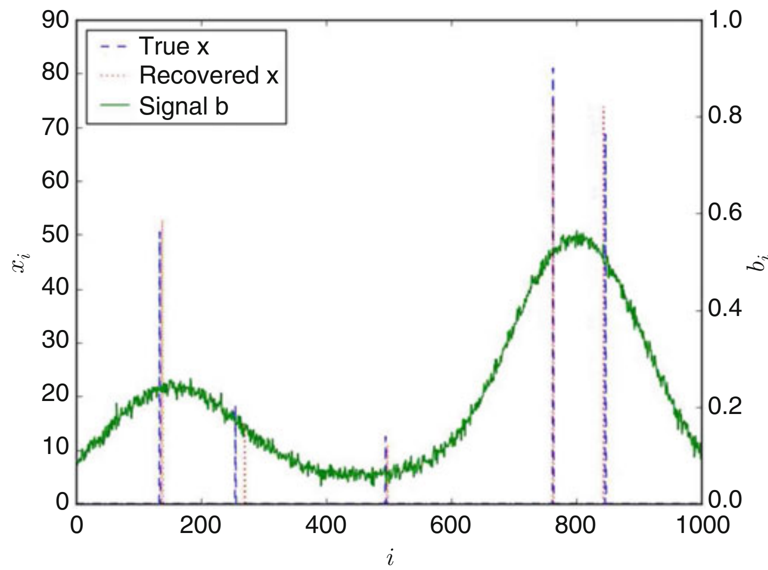

Fig. 12 Results for a problem instance with $n = 1000$

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 five 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 vector $\tilde{x}$. Figure 12 shows the solution recovered by ECOS [39] 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 Fig. 12 shows only the largest entry in each cluster, with value set to the sum of the cluster's entries.

Results Figure 13 compares the performance on problem (1) of the interior-point solver ECOS [39] and matrix-free versions of SCS and POGS as the size n of the optimization variable increases. We limited the solvers to 10^4 s.

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}.

For each solver, we plot the solve times and the least-squares linear fit to those solve times (the dotted line). 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

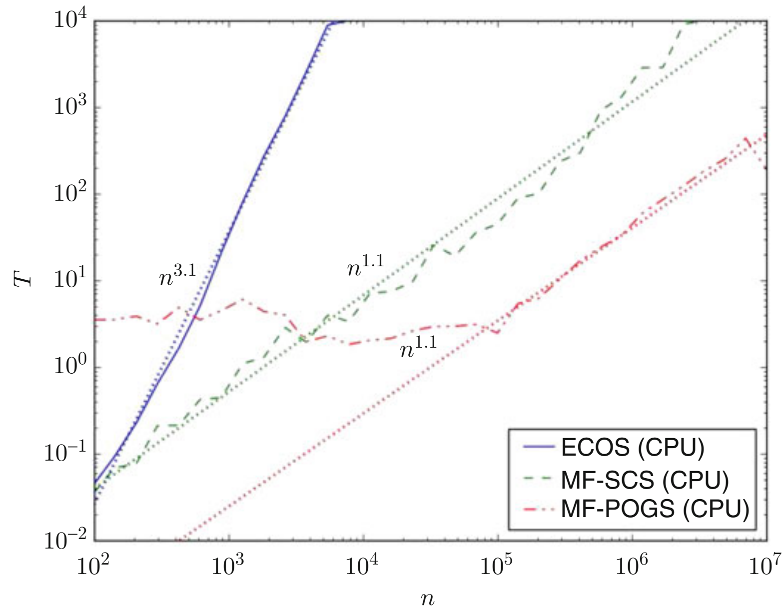

Fig. 13 Solve time in seconds T versus variable size n

the solve time scales like n^3, as expected. The least-squares linear fit for the matrix-free SCS solve times has slope 1.1, 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 is not saturated, so increasing n barely increases the solve time.

6.2 Sylvester LP

We applied our matrix-free convex optimization modeling system to Sylvester LPs, or convex optimization problems of the form

$$\begin{array}{ll} \text{minimize} & \mathbf{Tr}(D^T X) \\ \text{subject to} & AXB \leq C \\ & X \geq 0, \end{array} \tag{12}$$

where $X \in \mathbf{R}^{p \times q}$ is the optimization variable, and $A \in \mathbf{R}^{p \times p}$, $B \in \mathbf{R}^{q \times q}$, $C \in \mathbf{R}^{p \times q}$, and $D \in \mathbf{R}^{p \times q}$ are problem data. The inequality $AXB \leq C$ is a variant of the Sylvester equation $AXB = C$ [49].

Existing convex optimization modeling systems will convert problem (12) into the vectorized format

$$\begin{array}{ll} \text{minimize} & \mathbf{vec}(D)^T \mathbf{vec}(X) \\ \text{subject to} & (B^T \otimes A)\, \mathbf{vec}(X) \leq \mathbf{vec}(C) \\ & \mathbf{vec}(X) \geq 0, \end{array} \tag{13}$$

where $B^T \otimes A \in \mathbf{R}^{pq \times pq}$ is the Kronecker product of B^T and A. Let $p = kq$ for some fixed k, and let $n = kq^2$ denote the size of the optimization variable. A standard interior-point solver will take $O(n^3)$ flops and $O(n^2)$ bytes of memory to solve problem (13). A specialized matrix-free solver that exploits the matrix product AXB, by contrast, can solve problem (12) in $O(n^{1.5})$ flops using $O(n)$ bytes of memory [110].

Problem Instances We used the following procedure to generate interesting (nontrivial) instances of problem (12). We fixed $p = 5q$ and generated $\tilde{A}$ and $\tilde{B}$ by drawing entries i.i.d. from the folded standard normal distribution (i.e., the absolute value of the standard normal distribution). We then set

$$A = \tilde{A}/\|\tilde{A}\|_2 + I, \qquad B = \tilde{B}/\|\tilde{B}\|_2 + I,$$

so that A and B had positive entries and bounded condition number. We generated D by drawing entries i.i.d. from a standard normal distribution. We fixed $C = 11^T$. Our method of generating the problem data ensured the problem was feasible and bounded.

Results Figure 14 compares the performance on problem (12) of the interior-point solver ECOS [39] and matrix-free versions of SCS and POGS as the size $n = 5q^2$ of

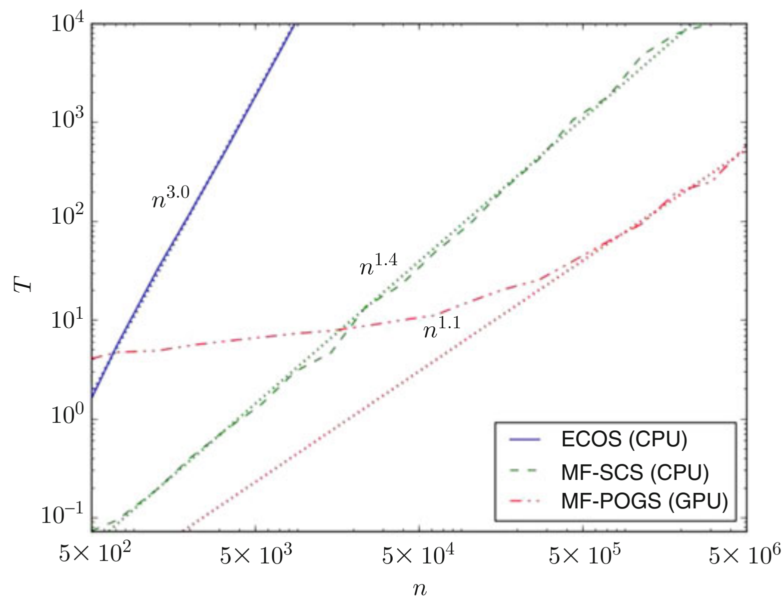

Fig. 14 Solve time in seconds T versus variable size n

the optimization variable increases. We limited the solvers to 10^4 s. 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}.

For each solver, we plot the solve times and the least-squares linear fit to those solve times (the dotted line). The slopes of the lines show how the solvers scale. The least-squares linear fit for the ECOS solve times has slope 3.0, 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 1.4, which indicates that the solve time scales like the expected $n^{1.5}$. The least-squares linear fit for the matrix-free POGS solve times in the range $n \in [5 \times 10^5, 5 \times 10^6]$ has slope 1.1. The solve time scales more slowly than the expected $n^{1.5}$, likely because the GPU was not fully saturated even on the largest problem instances. For $n < 5 \times 10^5$, the GPU was far from saturated, so increasing n barely increases the solve time.

Acknowledgements We thank Eric Chu, Michal Kočvara, and Alex Aiken for helpful comments on earlier versions of this work, and Chris Fougner, John Miller, Jack Zhu, and Paul Quigley for their work on the POGS cone solver and CVXcanon [88], which both contributed to the implementation of matrix-free CVXPY. We also thank the anonymous reviewers for useful feedback. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-114747 and by the DARPA XDATA program.

Appendix

Equivalence of the Cone Program

In this section we explain the precise sense in which the cone program output by the matrix-free canonicalization algorithm is equivalent to the original convex optimization problem.

Theorem 1. *Let p be a convex optimization problem whose OPR is a valid input to the matrix-free canonicalization algorithm. Let $\Phi(p)$ be the cone program represented by the output of the algorithm given p's OPR as input. All the variables in p are present in $\Phi(p)$, along with new variables introduced during the canonicalization process [55, 106]. Let $x \in \mathbf{R}^n$ represent the variables in p stacked into a vector and $t \in \mathbf{R}^m$ represent the new variables in $\Phi(p)$ stacked into a vector.*

The problems p and $\Phi(p)$ are equivalent in the following sense:

1. *For all x feasible in p, there exists $t^\star$ such that $(x, t^\star)$ is feasible in $\Phi(p)$ and $p(x) = \Phi(p)(x, t^\star)$.*
2. *For all (x, t) feasible in $\Phi(p)$, x is feasible in p and $p(x) \leq \Phi(p)(x, t)$.*

For a point x feasible in p, by $p(x)$ we mean the value of p's objective evaluated at x. The notation $\Phi(p)(x, t)$ is similarly defined.

Proof. See [55].

Theorem 1 implies that p and $\Phi(p)$ have the same optimal value. Moreover, p is infeasible if and only if $\Phi(p)$ is infeasible, and p is unbounded if and only if $\Phi(p)$ is unbounded. The theorem also implies that any solution $x^\star$ to p is part of a solution $(x^\star, t^\star)$ to $\Phi(p)$ and vice versa.

A similar equivalence holds between the Lagrange duals of p and $\Phi(p)$, but the details are beyond the scope of this paper. See [55] for a discussion of the dual of the cone program output by the canonicalization algorithm.

Sparse Matrix Representation

In this section we explain the `Matrix-Repr` subroutine used in the standard canonicalization algorithm to obtain a sparse matrix representation of a cone program. Recall that the subroutine takes a list of linear expression DAGs, $(e_1, \ldots, e_\ell)$, and an ordering over the variables in the expression DAGs, $<_V$, as input and outputs a sparse matrix A.

The algorithm to carry out the subroutine is not discussed anywhere in the literature, so we present here the version used by CVXPY [36]. The algorithm first converts each expression DAG into a map from variables to sparse matrices, representing a sum of terms. For example, if the map ϕ maps the variable $x \in \mathbf{R}^n$ to the sparse matrix coefficient $B \in \mathbf{R}^{m\times n}$ and the variable $y \in \mathbf{R}^n$ to the sparse matrix coefficient $C \in \mathbf{R}^{m\times n}$, then ϕ represents the sum $Bx + Cy$.

The conversion from expression DAG to map of variables to sparse matrices is done using Algorithm 9. The algorithm uses the subroutine `Matrix-Coeff`, which takes a node representing a linear function f and indices i and j as inputs and outputs a sparse matrix D. Let $\tilde{f}$ be a function defined on the range of f's ith input such that $\tilde{f}(x)$ is equal to f's jth output when f is evaluated on ith input x and zero-valued matrices (of the appropriate dimensions) for all other inputs. The output of `Matrix-Coeff` is the sparse matrix D such that for any value x in the domain of $\tilde{f}$,

$$D\,\mathbf{vec}(x) = \mathbf{vec}(\tilde{f}(x)).$$

The sparse matrix coefficients in the maps of variables to sparse matrices are assembled into a single sparse matrix A, as follows: Let $x_1, \ldots, x_k$ be the variables in the expression DAGs, ordered according to $<_V$. Let n_i be the length of x_i if the variable is a vector and of $\mathbf{vec}(x_i)$ if the variable is a matrix, for $i = 1, \ldots, k$. Let m_j be the length of expression DAG e_j's output, for $j = 1, \ldots, \ell$. The coefficients for x_1 are placed in the first n_1 columns in A, the coefficients for x_2 in the next n_2 columns, etc. Similarly, the coefficients from e_1 are placed in the first m_1 rows of A, the coefficients from e_2 in the next m_2 rows, etc.

Algorithm 9 Convert an expression DAG into a map from variables to sparse matrices

Precondition: e is a linear expression DAG that outputs a single vector.

Create an empty queue Q for nodes that are ready to evaluate.
Create an empty set S for nodes that have been evaluated.
Create a map M from (node, output index) tuples to maps of variables to sparse matrices.
for every start node u in e **do**
  $x \leftarrow$ the variable represented by node u.
  $n \leftarrow$ the length of x if the variable is a vector and of $\mathbf{vec}(x)$ if the variable is a matrix.
  $M[(u,1)] \leftarrow$ a map with key x and value the n-by-n identity matrix.
  Add u to S.
end for
Add all nodes in e to Q whose only incoming edges are from start nodes.
while Q is not empty **do**
  $u \leftarrow$ pop the front node of Q.
  Add u to S.
  for edge (u,p) in u's E_{out}, with index j **do**
    Create an empty map M_j from variables to sparse matrices.
    for edge (v,u) in u's E_{in}, with index i **do**
      $A^{(ij)} \leftarrow \texttt{Matrix-Coeff}(u,i,j)$.
      $k \leftarrow$ the index of (v,u) in v's E_{out}.
      for key x and value C in $M[(v,k)]$ **do**
        if M_j has an entry for x **then**
          $M_j[x] \leftarrow M_j[x] + A^{(ij)}C$.
        else
          $M_j[x] \leftarrow A^{(ij)}C$.
        end if
      end for
    end for
    $M[(u,j)] \leftarrow M_j$.
    if for all edges (q,p) in p's E_{in}, q is in S **then**
      Add p to the end of Q.
    end if
  end for
end while
$u_{\text{end}} \leftarrow$ the end node of e.
return $M[(u_{\text{end}},1)]$.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: large-scale machine learning on heterogeneous systems. (2015) http://tensorflow.org/. Cited 2 March 2016

2. Ahmed, N., Natarajan, T., Rao, K.: Discrete cosine transform. IEEE Trans. Comput. **C-23**(1), 90–93 (1974)
3. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley Longman, Boston (2006)
4. Akle, S.: Algorithms for unsymmetric cone optimization and an implementation for problems with the exponential cone. Ph.D. thesis, Stanford University (2015)
5. Andersen, M., Dahl, J., Liu, Z., Vandenberghe, L.: Interior-point methods for large-scale cone programming. In: Sra, S., Nowozin, S., Wright, S. (eds.) Optimization for Machine Learning, pp. 55–83. MIT Press, Cambridge (2012)
6. Andersen, M., Dahl, J., Vandenberghe, L.: CVXOPT: Python software for convex optimization, version 1.1 (2015). http://cvxopt.org/. Cited 2 March 2016
7. Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., Bouchard, N., Bengio, Y.: Theano: new features and speed improvements. In: Deep Learning and Unsupervised Feature Learning, Neural Information Processing Systems Workshop (2012)
8. Baydin, A., Pearlmutter, B., Radul, A., Siskind, J.: Automatic differentiation in machine learning: a survey. Preprint (2015). http://arxiv.org/abs/1502.05767. Cited 2 March 2016
9. Beck, A., Teboulle, M.: Fast gradient-based algorithms for constrained total variation image denoising and deblurring problems. IEEE Trans. Image Process. **18**(11), 2419–2434 (2009)
10. Beck, A., Teboulle, M.: A fast iterative shrinkage-thresholding algorithm for linear inverse problems. SIAM J. Imag. Sci. **2**(1), 183–202 (2009)
11. Becker, S., Candès, E., Grant, M.: Templates for convex cone problems with applications to sparse signal recovery. Math. Program. Comput. **3**(3), 165–218 (2011)
12. Benson, S., Ye, Y.: Algorithm 875: DSDP5—software for semidefinite programming. ACM Trans. Math. Software **34**(3), (2008)
13. Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., Bengio, Y.: Theano: a CPU and GPU math expression compiler. In: Proceedings of the Python for Scientific Computing Conference (2010)
14. Börm, S., Grasedyck, L., Hackbusch, W.: Introduction to hierarchical matrices with applications. Eng. Anal. Bound. Elem. **27**(5), 405–422 (2003)
15. Boyd, S., Parikh, N., Chu, E., Peleato, B., Eckstein, J.: Distributed optimization and statistical learning via the alternating direction method of multipliers. Found. Trends Mach. Learn. **3**, 1–122 (2011)
16. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press, Cambridge (2004)
17. Bracewell, R.: The fast Hartley transform. In: Proceedings of the IEEE, vol. 72, pp. 1010–1018 (1984)
18. Brandt, A., McCormick, S., Ruge, J.: Algebraic multigrid (AMG) for sparse matrix equations. In: D. Evans (ed.) Sparsity and its Applications, pp. 257–284. Cambridge University Press, Cambridge (1985)
19. Brélaz, D.: New methods to color the vertices of a graph. Commun. ACM **22**(4), 251–256 (1979)
20. Candès, E., Demanet, L., Donoho, D., Ying, L.: Fast discrete curvelet transforms. Multiscale Model. Simul. **5**(3), 861–899 (2006)
21. Carrier, J., Greengard, L., Rokhlin, V.: A fast adaptive multipole algorithm for particle simulations. SIAM J. Sci. Stat. Comput. **9**(4), 669–686 (1988)
22. Chambolle, A., Pock, T.: A first-order primal-dual algorithm for convex problems with applications to imaging. J. Math. Imaging Vision **40**(1), 120–145 (2011)
23. Chan, T., Esedoglu, S., Nikolova, M.: Algorithms for finding global minimizers of image segmentation and denoising models. SIAM J. Appl. Math. **66**(5), 1632–1648 (2006)
24. Chen, S., Donoho, D., Saunders, M.: Atomic decomposition by basis pursuit. SIAM J. Sci. Comput. **20**(1), 33–61 (1998)
25. Choi, C., Ye, Y.: Solving sparse semidefinite programs using the dual scaling algorithm with an iterative solver. Working paper, Department of Management Sciences, University of Iowa (2000)

26. Chu, E., O'Donoghue, B., Parikh, N., Boyd, S.: A primal-dual operator splitting method for conic optimization. Preprint (2013). http://stanford.edu/~boyd/papers/pdf/pdos.pdf. Cited 2 March 2016
27. Chu, E., Parikh, N., Domahidi, A., Boyd, S.: Code generation for embedded second-order cone programming. In: Proceedings of the European Control Conference, pp. 1547–1552 (2013)
28. Cohen, A., Daubechies, I., Feauveau, J.C.: Biorthogonal bases of compactly supported wavelets. Commun. Pure Appl. Math. **45**(5), 485–560 (1992)
29. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: a MATLAB-like environment for machine learning. In: BigLearn, Neural Information Processing Systems Workshop (2011)
30. Cooley, J., Lewis, P., Welch, P.: The fast Fourier transform and its applications. IEEE Trans. Educ. **12**(1), 27–34 (1969)
31. Cooley, J., Tukey, J.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**(90), 297–301 (1965)
32. Daubechies, I.: Orthonormal bases of compactly supported wavelets. Commun. Pure Appl. Math. **41**(7), 909–996 (1988)
33. Daubechies, I.: Ten lectures on wavelets. SIAM, Philadelphia (1992)
34. Davis, T.: Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2). SIAM, Philadelphia (2006)
35. Diamond, S., Boyd, S.: Convex optimization with abstract linear operators. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 675–683 (2015)
36. Diamond, S., Boyd, S.: CVXPY: A Python-embedded modeling language for convex optimization. J. Mach. Learn. Res. **17**(83), 1–5 (2016)
37. Diamond, S., Boyd, S.: Stochastic matrix-free equilibration. J. Optim. Theory Appl. (2016, to appear)
38. Do, M., Vetterli, M.: The finite ridgelet transform for image representation. IEEE Trans. Image Process. **12**(1), 16–28 (2003)
39. Domahidi, A., Chu, E., Boyd, S.: ECOS: an SOCP solver for embedded systems. In: Proceedings of the European Control Conference, pp. 3071–3076 (2013)
40. Dudgeon, D., Mersereau, R.: Multidimensional Digital Signal Processing. Prentice-Hall, Englewood Cliffs (1984)
41. Duff, I., Erisman, A., Reid, J.: Direct Methods for Sparse Matrices. Oxford University Press, New York (1986)
42. Figueiredo, M., Nowak, R., Wright, S.: Gradient projection for sparse reconstruction: application to compressed sensing and other inverse problems. IEEE J. Sel. Top. Signal Process. **1**(4), 586–597 (2007)
43. Fong, D., Saunders, M.: LSMR: an iterative algorithm for sparse least-squares problems. SIAM J. Sci. Comput. **33**(5), 2950–2971 (2011)
44. Forsyth, D., Ponce, J.: Computer Vision: A Modern Approach. Prentice Hall, Upper Saddle River (2002)
45. Fougner, C., Boyd, S.: Parameter selection and pre-conditioning for a graph form solver. (2015, preprint). http://arxiv.org/pdf/1503.08366v1.pdf. Cited 2 March 2016
46. Fountoulakis, K., Gondzio, J., Zhlobich, P.: Matrix-free interior point method for compressed sensing problems. Math. Program. Comput. **6**(1), 1–31 (2013)
47. Fujisawa, K., Fukuda, M., Kobayashi, K., Kojima, M., Nakata, K., Nakata, M., Yamashita, M.: SDPA (semidefinite programming algorithm) user's manual – version 7.0.5. Tech. rep. (2008)
48. Fukuda, M., Kojima, M., Shida, M.: Lagrangian dual interior-point methods for semidefinite programs. SIAM J. Optim. **12**(4), 1007–1031 (2002)
49. Gardiner, J., Laub, A., Amato, J., Moler, C.: Solution of the Sylvester matrix equation $AXB^T + CXD^T = E$. ACM Trans. Math. Software **18**(2), 223–231 (1992)
50. Gilbert, A., Strauss, M., Tropp, J., Vershynin, R.: One sketch for all: fast algorithms for compressed sensing. In: Proceedings of the ACM Symposium on Theory of Computing, pp. 237–246 (2007)

51. Goldstein, T., Osher, S.: The split Bregman method for ℓ_1-regularized problems. SIAM J. Imag. Sci. **2**(2), 323–343 (2009)
52. Gondzio, J.: Matrix-free interior point method. Comput. Optim. Appl. **51**(2), 457–480 (2012)
53. Gondzio, J.: Convergence analysis of an inexact feasible interior point method for convex quadratic programming. SIAM J. Optim. **23**(3), 1510–1527 (2013)
54. Gondzio, J., Grothey, A.: Parallel interior-point solver for structured quadratic programs: application to financial planning problems. Ann. Oper. Res. **152**(1), 319–339 (2007)
55. Grant, M.: Disciplined convex programming. Ph.D. thesis, Stanford University (2004)
56. Grant, M., Boyd, S.: Graph implementations for nonsmooth convex programs. In: Blondel, V., Boyd, S., Kimura, H. (eds.) Recent Advances in Learning and Control. Lecture Notes in Control and Information Sciences, pp. 95–110. Springer, London (2008)
57. Grant, M., Boyd, S.: CVX: MATLAB software for disciplined convex programming, version 2.1 (2014). http://cvxr.com/cvx. Cited 2 March 2016
58. Grant, M., Boyd, S., Ye, Y.: Disciplined convex programming. In: Liberti, L., Maculan, N. (eds.) Global Optimization: From Theory to Implementation, Nonconvex Optimization and its Applications, pp. 155–210. Springer, New York (2006)
59. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. J. Comput. Phys. **73**(2), 325–348 (1987)
60. Greengard, L., Strain, J.: The fast Gauss transform. SIAM J. Sci. Stat. Comput. **12**(1), 79–94 (1991)
61. Griewank, A.: On automatic differentiation. In: Iri, M., Tanabe, K. (eds.) Mathematical Programming: Recent Developments and Applications, pp. 83–108. Kluwer Academic, Tokyo (1989)
62. Hackbusch, W.: Multi-Grid Methods and Applications. Springer, Heidelberg (1985)
63. Hackbusch, W.: A sparse matrix arithmetic based on $\mathscr{H}$-matrices. Part I: introduction to $\mathscr{H}$-matrices. Computing **62**(2), 89–108 (1999)
64. Hackbusch, W., Khoromskij, B., Sauter, S.: On $\mathscr{H}^2$-matrices. In: Bungartz, H.J., Hoppe, R., Zenger, C. (eds.) Lectures on Applied Mathematics, pp. 9–29. Springer, Heidelberg (2000)
65. Halldórsson, M.: A still better performance guarantee for approximate graph coloring. Inf. Process. Lett. **45**(1), 19–23 (1993)
66. Hennenfent, G., Herrmann, F., Saab, R., Yilmaz, O., Pajean, C.: SPOT: a linear operator toolbox, version 1.2 (2014). http://www.cs.ubc.ca/labs/scl/spot/index.html. Cited 2 March 2016
67. Hestenes, M., Stiefel, E.: Methods of conjugate gradients for solving linear systems. J. Res. Nat. Bur. Stand. **49**(6), 409–436 (1952)
68. Hien, L.: Differential properties of Euclidean projection onto power cone. Math. Methods Oper. Res. **82**(3), 265–284 (2015)
69. Jacques, L., Duval, L., Chaux, C., Peyré, G.: A panorama on multiscale geometric representations, intertwining spatial, directional and frequency selectivity. IEEE Trans. Signal Process. **91**(12), 2699–2730 (2011)
70. Jensen, A., la Cour-Harbo, A.: Ripples in Mathematics. Springer, Berlin (2001)
71. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. (2014, preprint). http://arxiv.org/abs/1408.5093. Cited 2 March 2016
72. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J., Bohlinger, J. (eds.) Complexity of Computer Computations, The IBM Research Symposia Series, pp. 85–103. Springer, New York (1972)
73. Kelner, J., Orecchia, L., Sidford, A., Zhu, A.: A simple, combinatorial algorithm for solving SDD systems in nearly-linear time. In: Proceedings of the ACM Symposium on Theory of Computing, pp. 911–920 (2013)
74. Kim, S.J., Koh, K., Lustig, M., Boyd, S., Gorinevsky, D.: An interior-point method for large-scale ℓ_1-regularized least squares. IEEE J. Sel. Top. Signal Process. **1**(4), 606–617 (2007)
75. Kočvara, M., Stingl, M.: On the solution of large-scale SDP problems by the modified barrier method using iterative solvers. Math. Program. **120**(1), 285–287 (2009)

76. Kovacevic, J., Vetterli, M.: Nonseparable multidimensional perfect reconstruction filter banks and wavelet bases for $\mathscr{R}^n$. IEEE Trans. Inf. Theory **38**(2), 533–555 (1992)
77. Krishnaprasad, P., Barakat, R.: A descent approach to a class of inverse problems. J. Comput. Phys. **24**(4), 339–347 (1977)
78. Lan, G., Lu, Z., Monteiro, R.: Primal-dual first-order methods with $O(1/\epsilon)$ iteration-complexity for cone programming. Math. Program. **126**(1), 1–29 (2011)
79. Liberty, E.: Simple and deterministic matrix sketching. In: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 581–588 (2013)
80. Lim, J.: Two-dimensional Signal and Image Processing. Prentice-Hall, Upper Saddle River (1990)
81. Lin, Y., Lee, D., Saul, L.: Nonnegative deconvolution for time of arrival estimation. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 2, pp. 377–380 (2004)
82. Loan, C.V.: Computational Frameworks for the Fast Fourier Transform. SIAM, Philadelphia (1992)
83. Lofberg, J.: YALMIP: A toolbox for modeling and optimization in MATLAB. In: Proceedings of the IEEE International Symposium on Computed Aided Control Systems Design, pp. 294–289 (2004)
84. Lu, Y., Do, M.: Multidimensional directional filter banks and surfacelets. IEEE Trans. Image Process. **16**(4), 918–931 (2007)
85. Mallat, S.: A theory for multiresolution signal decomposition: the wavelet representation. IEEE Trans. Pattern Anal. Mach. Intell. **11**(7), 674–693 (1989)
86. Martucci, S.: Symmetric convolution and the discrete sine and cosine transforms. IEEE Trans. Signal Process. **42**(5), 1038–1051 (1994)
87. Mattingley, J., Boyd, S.: CVXGEN: A code generator for embedded convex optimization. Optim. Eng. **13**(1), 1–27 (2012)
88. Miller, J., Zhu, J., Quigley, P.: CVXcanon, version 0.0.22 (2015). https://github.com/cvxgrp/CVXcanon. Cited 2 March 2016
89. MOSEK optimization software, version 7 (2015). https://mosek.com/. Cited 2 March 2016
90. Nesterov, Y.: Towards nonsymmetric conic optimization. Optim. Methods Software **27**(4–5), 893–917 (2012)
91. Nesterov, Y., Nemirovskii, A.: Interior-Point Polynomial Algorithms in Convex Programming. SIAM, Philadelphia (1994)
92. Nesterov, Y., Nemirovsky, A.: Conic formulation of a convex programming problem and duality. Optim. Methods Softw. **1**(2), 95–115 (1992)
93. Nocedal, J., Wright, S.: Numerical Optimization. Springer, New York (2006)
94. O'Donoghue, B., Chu, E., Parikh, N., Boyd, S.: Conic optimization via operator splitting and homogeneous self-dual embedding. J. Optim. Theory Appl. **169**(3), 1042–1068 (2016)
95. Parikh, N., Boyd, S.: Proximal algorithms. Found. Trends Optim. **1**(3), 123–231 (2014)
96. Pock, T., Chambolle, A.: Diagonal preconditioning for first order primal-dual algorithms in convex optimization. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1762–1769 (2011)
97. Pock, T., Cremers, D., Bischof, H., Chambolle, A.: An algorithm for minimizing the Mumford-Shah functional. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1133–1140 (2009)
98. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 519–530 (2013)
99. Saunders, M., Kim, B., Maes, C., Akle, S., Zahr, M.: PDCO: Primal-dual interior method for convex objectives (2013). http://web.stanford.edu/group/SOL/software/pdco/. Cited 2 March 2016
100. Skajaa, A., Ye, Y.: A homogeneous interior-point algorithm for nonsymmetric convex conic optimization. Math. Program. **150**(2), 391–422 (2014)

101. Spielman, D., Teng, S.H.: Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In: Proceedings of the ACM Symposium on Theory of Computing, pp. 81–90 (2004)
102. Starck, J.L., Candès, E., Donoho, D.: The curvelet transform for image denoising. IEEE Trans. Image Process. **11**(6), 670–684 (2002)
103. Sturm, J.: Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. Optim. Methods Softw. **11**(1–4), 625–653 (1999)
104. Toh, K.C.: Solving large scale semidefinite programs via an iterative solver on the augmented systems. SIAM J. Optim. **14**(3), 670–698 (2004)
105. Toh, K.C., Todd, M., Tütüncü, R.: SDPT3 — a MATLAB software package for semidefinite programming, version 4.0. Optim. Methods Softw. **11**, 545–581 (1999)
106. Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S., Boyd, S.: Convex optimization in Julia. In: Proceedings of the Workshop for High Performance Technical Computing in Dynamic Languages, pp. 18–28 (2014)
107. Vaillant, G.: linop, version 0.7 (2013). http://pythonhosted.org//linop/. Cited 2 March 2016
108. van den Berg, E., Friedlander, M.: Probing the Pareto frontier for basis pursuit solutions. SIAM J. Sci. Comput. **31**(2), 890–912 (2009)
109. Vandenberghe, L., Boyd, S.: A polynomial-time algorithm for determining quadratic Lyapunov functions for nonlinear systems. In: Proceedings of the European Conference on Circuit Theory and Design, pp. 1065–1068 (1993)
110. Vandenberghe, L., Boyd, S.: A primal-dual potential reduction method for problems involving matrix inequalities. Math. Program. **69**(1–3), 205–236 (1995)
111. Vishnoi, K.: Laplacian solvers and their algorithmic applications. Theor. Comput. Sci. **8**(1–2), 1–141 (2012)
112. Wright, S.: Primal-Dual Interior-Point Methods. SIAM, Philadelphia (1987)
113. Yang, C., Duraiswami, R., Davis, L.: Efficient kernel machines using the improved fast Gauss transform. In: Saul, L., Weiss, Y., Bottou, L. (eds.) Advances in Neural Information Processing Systems 17, pp. 1561–1568. MIT Press, Cambridge (2005)
114. Yang, C., Duraiswami, R., Gumerov, N., Davis, L.: Improved fast Gauss transform and efficient kernel density estimation. In: Proceedings of the IEEE International Conference on Computer Vision, vol. 1, pp. 664–671 (2003)
115. Ye, Y.: Interior Point Algorithms: Theory and Analysis. Wiley-Interscience, New York (2011)
116. Ying, L., Demanet, L., Candès, E.: 3D discrete curvelet transform. In: Proceedings of SPIE: Wavelets XI, vol. 5914, pp. 351–361 (2005)
117. Zach, C., Pock, T., Bischof, H.: A duality based approach for realtime TV-ℓ_1 optical flow. In: Hamprecht, F., Schnörr, C., Jähne, B. (eds.) Pattern Recognition. Lecture Notes in Computer Science, vol. 4713, pp. 214–223. Springer, Heidelberg (2007)
118. Zhao, X.Y., Sun, D., Toh, K.C.: A Newton-CG augmented Lagrangian method for semidefinite programming. SIAM J. Optim. **20**(4), 1737–1765 (2010)