Disciplined Convex Programming

Michael Grant[1], Stephen Boyd[1], and Yinyu Ye[1 2]

[1] Department of Electrical Engineering, Stanford University
{mcgrant,boyd,yyye}@stanford.edu
[2] Department of Management Science and Engineering, Stanford University

Summary. A new methodology for constructing convex optimization models called *disciplined convex programming* is introduced. The methodology enforces a set of conventions upon the models constructed, in turn allowing much of the work required to analyze and solve the models to be automated.

Key words: Convex programming, automatic verification, symbolic computation, modelling language.

1 Introduction

Convex programming is a subclass of nonlinear programming (NLP) that unifies and generalizes least squares (LS), linear programming (LP), and convex quadratic programming (QP). This generalization is achieved while maintaining many of the important, attractive theoretical properties of these predecessors. Numerical algorithms for solving convex programs are maturing rapidly, providing reliability, accuracy, and efficiency. A large number of applications have been discovered for convex programming in a wide variety of scientific and non-scientific fields, and it seems clear that even more remain to be discovered. For these reasons, convex programming arguably has the potential to become a ubiquitous modeling technology alongside LS, LP, and QP. Indeed, efforts are underway to develop and teach it as a distinct discipline [29, 21, 115].

Nevertheless, there remains a significant impediment to the more widespread adoption of convex programming: the high level of expertise required to use it. With mature technologies such as LS, LP, and QP, problems can be specified and solved with relatively little effort, and with at most a very basic understanding of the computations involved. This is not the case with general convex programming. That a user must understand the basics of convex analysis is both reasonable and unavoidable; but in fact, a much deeper understanding is required. Furthermore, a user must find a way to transform

his problem into one of the many limited standard forms; or, failing that, develop a custom solver. For potential users whose focus is the application, these requirements can form a formidable "expertise barrier"—especially if it is not yet certain that the outcome will be any better than with other methods. The purpose of the work presented here is to lower this barrier.

In this article, we introduce a new modeling methodology called *disciplined convex programming*. As the term "disciplined" suggests, the methodology imposes a set of conventions that one must follow when constructing convex programs. The conventions are simple and teachable, taken from basic principles of convex analysis, and inspired by the practices of those who regularly study and apply convex optimization today. Conforming problems are called, appropriately, *disciplined convex programs*. The conventions do not limit generality; but they *do* allow much of the manipulation and transformation required to analyze and solve convex programs to be automated. For example, the task of determining if an arbitrary NLP is convex is both theoretically and practically intractable; the task of determining if it is a *disciplined* convex program is straightforward. In addition, the transformations necessary to convert disciplined convex programs into solvable form can be fully automated.

A novel aspect of this work is a new way to define a function in a modeling framework: as the solution of a disciplined convex program. We call such a definition a *graph implementation*, so named because it exploits the properties of epigraphs and hypographs of convex and concave functions, respectively. The benefits of graph implementations to are significant, because they provide a means to support nondifferentiable functions without the loss of reliability or performance typically associated with them.

We have created a modeling framework called `cvx` that implements the principles of the disciplined convex programming methodology. The system is built around a specification language that allows disciplined convex programs to be specified in a natural mathematical form, and addresses key tasks such as verification, conversion to solvable form, and numerical solution. The development of `cvx` is ongoing, and an initial version is near release. We will be disseminating `cvx` freely to encourage its use in coursework, research, and applications.

The remainder of this article begins with some motivation for this work, by examining how current numerical methods can be used to solve a simple norm minimization problem. In §3, we provide a brief overview of convex programming technology; and in §4, we discuss the benefits of modeling frameworks in general, and `cvx` in particular. Finally, we introduce disciplined convex programming in detail in §5-§10.

2 Motivation

To illustrate the complexities of practical convex optimization, let us consider how one might solve a basic and yet common problem: the unconstrained

norm minimization

$$\text{minimize } f(x) \triangleq \|Ax - b\| \qquad A \in \mathbb{R}^{m \times n} \quad b \in \mathbb{R}^m \tag{1}$$

The norm $\|\cdot\|$ has not yet been specified; we will be examining several choices of that norm. For simplicity, we will assume that $n \leq m$ and that A has full rank; and for convenience, we shall partition A and b into rows,

$$A^T \triangleq \begin{bmatrix} a_1 & a_2 & \cdots & a_m \end{bmatrix} \qquad b^T \triangleq \begin{bmatrix} b_1 & b_2 & \cdots & b_m \end{bmatrix} \tag{2}$$

2.1 The norms

The Euclidean norm

The most common choice of the norm in (1) is certainly the ℓ_2 or Euclidean norm,

$$f(x) \triangleq \|Ax - b\|_2 = \sqrt{\textstyle\sum_{i=1}^n (a_i^T x - b_i)^2} \tag{3}$$

In this case, (1) is easily recognized as a least squares problem, which as its name implies is often presented in an equivalent quadratic form,

$$\text{minimize } f(x)^2 = \|Ax - b\|_2^2 = \textstyle\sum_{i=1}^m (a_i^T x - b_i)^2 \tag{4}$$

This problem has an analytical solution $x = (A^T A)^{-1} A^T b$, which can be computed using a Cholesky factorization of $A^T A$, or more accurately using a QR or SVD factorization of A [76]. A number of software packages to solve least squares problems are readily available; *e.g.*, [1, 106].

The Chebyshev norm

For the ℓ_∞ or Chebyshev norm,

$$f(x) \triangleq \|Ax - b\|_\infty = \max_{i=1,2,\ldots,m} |a_i^T x - b_i|, \tag{5}$$

there is no analytical solution. But a solution can be obtained by solving the linear program

$$\begin{array}{l} \text{minimize } q \\ \text{subject to } -q \leq a_i^T x - b_i \leq q, \; i = 1, 2, \ldots, m \end{array} \tag{6}$$

Despite the absence of an analytical solution, numerical solutions for this problem are not difficult to obtain. A number of efficient and reliable LP solvers are readily available; in fact, a basic LP solver is included with virtually every piece of spreadsheet software sold [63].

The Manhattan norm

Similarly, for the ℓ_1 or Manhattan norm

$$f(x) \triangleq \|Ax - b\|_1 = \sum_{i=1}^{m} |a_i^T x - b_i|, \tag{7}$$

a solution can also be determined by solving an appropriate LP:

$$\begin{array}{l} \text{minimize } \sum_{i=1}^m v_i \\ \text{subject to } -v_i \leq a_i^T x - b_i \leq v_i, \ i = 1, 2, \ldots, m \end{array} \tag{8}$$

So again, this problem can be easily solved with readily available software, even though an analytical solution does not exist.

The Hölder norm, part 1

Now consider the Hölder or ℓ_p norm

$$f(x) \triangleq \|Ax - b\|_p = \left(\sum_{i=1}^m |a_i^T x - b_i|^p\right)^{1/p} \tag{9}$$

for $p \geq 2$. We may consider solving (1) for this norm using Newton's method. For simplicity we will in fact apply the method to the related function

$$g(x) \triangleq f(x)^p = \sum_{i=1}^m |a_i^T x - b_i|^p \tag{10}$$

which yields the same solution x but is twice differentiable everywhere. The iterates produced by Newton's method are

$$\begin{aligned} x_{k+1} &= x_k - \alpha_k \left(\nabla^2 g(x_k)\right)^{-1} \nabla g(x_k) \\ &= x_k - \frac{\alpha_k}{p-1} (A^T W_k A)^{-1} A^T W_k (A x_k - b) \\ &= \frac{p - 1 - \alpha_k}{p-1} x_k + \frac{\alpha_k}{p-1} \arg\min_{\bar{w}} \left\| W_k^{1/2} (A\bar{w} - b) \right\|_2 \end{aligned} \tag{11}$$

where $x_0 = 0$, $k = 1, 2, 3, \ldots$, and we have defined

$$W_k \triangleq \mathbf{diag}(|a_1^T x_k - b_1|^{p-2}, |a_2^T x_k - b_2|^{p-2}, \ldots, |a_m^T x_k - b_m|^{p-2}). \tag{12}$$

and $\alpha_k \in [0, 1)$ is either fixed or determined at each iteration using a line search technique. Notice how the Newton computation involves a (weighted) least squares problem; In fact, if $p = 2$, then $W_k \equiv I$, and a single Newton iteration produces the correct least squares solution. So the more "complex" ℓ_p case simply involves solving a series of similar least squares problems. This resemblance to least squares turns up quite often in numerical methods for convex programming.

An important technical detail must be mentioned here. When the residual vector $Ax_k - b$ has any zero entries, the matrix W_k (12) will be singular. If $m \gg n$, this will not necessarily render the Hessian $\nabla^2 g(x)$ itself singular, but care must be taken nonetheless to guard for this possibility. A variety of methods can be considered, including the introduction of a slight dampening factor to W_k; *i.e.*, $W_k + \epsilon I$.

Newton's method is itself a relatively straightforward algorithm, and a number of implementations have been developed; *e.g.*, [113]. These methods do require that code be created to compute the computation of the gradient and Hessian of the function involved. This task is eased somewhat by using an automatic differentiation package such ADIC [81] or ADIFOR [10], which can generate derivative code from code that simply computes a function's value.

The Hölder norm, part 2

For $1 < p < 2$, Newton's method cannot reliably be employed, because neither $f(x)$ nor $g(x) \triangleq f(x)^p$ is twice differentiable whenever the residual vector $Ax - b$ has any zero entries. An alternative that works for all $p \in [1, +\infty)$ is to apply a barrier method to the problem. A full introduction to barrier methods is beyond the scope of this text, so we will highlight only key details. The reader is invited to consult [118] for a truly exhaustive development of barrier methods, or [29] for a gentler introduction.

To begin, we note that the solution to (1) can be obtained by solving

$$\begin{array}{l} \text{minimize } \mathbf{1}^T v \\ \text{subject to } |a_i^T x - b_i|^p \leq v_i \; i = 1, 2, \ldots, m \end{array} \tag{13}$$

To solve (13), we construct a *barrier function* $\phi : (\mathbb{R}^n \times \mathbb{R}^m) \rightarrow (\mathbb{R} \cup +\infty)$ to represent the inequality constraints [118]:

$$\phi(x, v) \triangleq \begin{cases} \sum_{i=1}^m -\log(v_i^{2/p} - (a_i^T x - b_i)^2) - 2\log v_i & (x, v) \in S \\ +\infty & (x, v) \notin S \end{cases} \tag{14}$$

$$S \triangleq \left\{ (x, v) \in \mathbb{R}^n \times \mathbb{R} \mid |a_i^T x - b_i|^p < v_i, \; i = 1, 2, \ldots, m \right\}$$

The barrier function is finite and twice differentiable whenever the inequality constraints in (13) are strictly satisfied, and $+\infty$ otherwise. This barrier is used to create a family of functions g_t parameterized over a quantity $t > 0$:

$$g_t : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R} \cup +\infty, \quad g_t(x, v) \triangleq \mathbf{1}^T v + t\phi(x, v) \tag{15}$$

It can be shown that, as $t \rightarrow 0$, the minimizing values for $g_t(x, v)$ converge to the solution to the original problem. A practical barrier method takes Newton steps to minimize $g_t(x, v)$, decreasing the value of t between iterations in a manner chosen to insure convergence and acceptable performance.

This approach is obviously significantly more challenging than the previous efforts. As with the Newton method for the $p \geq 2$ case, code must be written

(or automatic differentiation employed) to compute the gradient and Hessian of $g_t(x, v)$. Furthermore, the authors are unaware of any readily available software implementing a general purpose barrier methods such as this, so one would be forced to write their own.

An uncommon choice

Given a vector $w \in \mathbb{R}^m$, let $w_{[|k|]}$ be the k-th element of the vector after it has been sorted from largest to smallest in absolute value:

$$|w_{[|1|]}| \geq |w_{[|2|]}| \geq \cdots \geq |w_{[|m|]}| \tag{16}$$

Then let us define the *largest-L* norm as follows:

$$\|w\|_{[|L|]} \triangleq \sum_{k=1}^{L} |w_{[|k|]}| \quad (L \in \{1, 2, \ldots, m\}). \tag{17}$$

Solving (1) using this norm produces a vector x that minimizes the sum of the L largest residuals of $Ax - b$. This is equivalent to the ℓ_∞ case for $L = 1$ and the ℓ_1 case for $L = m$, but for $1 < L < m$ this norm produces novel results.

While it may not be obvious that (17) is a norm or even a convex function, it is indeed both. Even less obvious is how to solve this problem—but in fact, it turns out that it can be solved as an LP!

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{m} v_i + Lq \\ \text{subject to} & -v_i - q \leq a_i^T x - b_i \leq v_i + q, \quad i = 1, 2, \ldots, m \\ & v_i \geq 0, \quad i = 1, 2, \ldots, m \end{array} \tag{18}$$

This LP is only slightly larger than the one used for the ℓ_1 case. The result is known—see, for example [124]—but not widely so, even among those who actively study optimization. Thus it is likely that someone wishing to solve this problem would consider a far more difficult approach such as a barrier method or a subgradient method (or not even try).

2.2 The expertise barrier

The conceptual similarity of these problems is obvious, but the methods employed to solve them differ significantly. A variety of numerical algorithms are represented: least squares, linear programming, Newton's method, and a barrier method. In most cases, transformations are required to produce an equivalent problem suitable for numerical solution. These transformations are not likely to be obvious to an applications-oriented user whose primary expertise is not optimization.

As a result of this complexity, those wishing to solve a norm minimization problem may, out of ignorance or practicality, restrict their view to norms for which solution methods are widely known, such as ℓ_2 or ℓ_∞— even if doing

so compromises the accuracy of their models. This might be understandable for cases like, say, $\ell_{1.5}$, where the computational method employed is quite complex. But the true computational complexity may be far less severe than it seems on the surface, as is the case with the largest-L norm.

Even this simple example illustrates the high level of expertise needed to solve even basic convex optimization problems. Of course, the situation worsens if more complex problems are considered. For example, adding simple bounds on x (*e.g.*, $l \leq x \leq u$) eliminates analytical solutions for the ℓ_2 case, and prevents the use of a simple Newton's method for the ℓ_p case.

2.3 Lowering the barrier

In Figure 1, `cvx` specifications for three of the norm minimization problems presented here are given. In each case, the problem is given in its original form; no transformations described above have been applied in advance to convert them into "solvable" form. Instead, the models utilize the functions `norm_inf`, `norm_p`, and `norm_largest` for the ℓ_∞, ℓ_p, and largest-L norms, respectively. The definitions of these functions have been stored in a separate file `norms.cvx`, and referred to by an `include` command.

```
minimize norm_inf( A x - b );
parameters A[m,n], b[n];
variable x[n];
include "norms.cvx";

minimize norm_p( A x - b, p );
parameters A[m,n], b[n], p >= 1;
variable x[n];
include "norms.cvx";

minimize norm_largest( A x - b, L );
parameters A[m,n], b[n], L in #{1,2,...,n};
variable x[n];
include "norms.cvx";
```

Fig. 1. `cvx` specifications for the Chebyshev, Hölder, and largest-L cases, respectively.

In most modeling frameworks, function definitions consist of computer code to compute their values and derivatives. This method is not useful for these functions, because they are not differentiable. Graph implementations, which we describe in detail in §10, solve this problem. For now, it is enough to know that they effectively describe the very transformations illustrated in §2.1 above. For example, the definitions for the `norm_inf` and `norm_largest` provide the information necessary to convert their respective problems into

LPs. The definition for `norm_p` includes a barrier function for its epigraph, which can be used to apply a barrier method to the third problem.

So neither disciplined convex programming nor the cvx framework eliminates the transformations needed to solve any of these convex programs. Rather, they allow the transformations to be *encapsulated*: that is, hidden from the user, and performed without that user's intervention. A function definition can be used in multiple models and shared with many users. A natural, collaborative environment is suggested, where the work of those with advanced expertise in convex programming can share their knowledge with less experienced modelers in a practical way, by creating libraries of function definitions. The task of *solving* convex programs is appropriately returned to experts, freeing applications-oriented users to confidently focus on modeling.

3 Convex programming

A *mathematical program* is an optimization problem of the form

$$\begin{array}{l} \text{minimize } f(x) \\ \text{subject to } g_i(x) \leq 0 \;\; i = 1, 2, \ldots, n_g \\ \qquad\qquad h_j(x) = 0 \;\; j = 1, 2, \ldots, n_h \end{array} \tag{19}$$

or one that can be readily converted into this form. The vector x is the *problem variable*; the quantity the quantity $f(x)$ is the *objective function*, and the relations $g_i(x) \leq 0$ and $h_j(x) = 0$ are the *inequality* and *equality constraints*, respectively. The study of mathematical programs focuses almost exclusively on special cases of (19). The most popular is certainly the LP, for which the functions f, g_i, h_j are all affine. Least squares problems, QPs, and NLPs can all be represented by this form (19) as well.

A *convex program* (CP) is yet another special case of (19), one in which the objective function f and inequality constraint functions g_i are convex, and the equality constraint functions h_j are affine. The set of CPs is a strict subset of the set of NLPs, and includes all least squares problems, LPs, and convex QPs. Several other classes of CPs have been identified recently as standard forms. These include *semidefinite programs* (SDPs) [155], *second-order cone programs* (SOCPs) [104], and *geometric programs* (GPs) [48, 3, 138, 52, 92]. The work we present her applies to all of these special cases as well as to the general class of CPs.

The practice of modeling, analyzing, and solving CPs is known as *convex programming*. In this section we provide a survey of convex programming, including its theoretical properties, numerical algorithms, and applications.

3.1 Theoretical properties

A number of powerful and practical theoretical conclusions can be drawn once it can be established that a mathematical program is convex. A comprehensive

theory of convex analysis was developed by the 1970s [137, 142], and advances have continued since [83, 84, 21, 29].

The most fundamental distinction between CPs and general NLPs is that, for the former, local optima are guaranteed to be global. Put another way, if local optimality can somehow be demonstrated (say, using KKT conditions), then global optimality is assured. Except in certain special cases, no similar guarantees can be made for nonconvex NLPs. Such problems might exhibit multiple local optima, so an exhaustive search would be required to prove global optimality—an intractable task.

Convex programming also has a rich duality theory that is very similar to the duality theory that accompanies linear programming, though it is a bit more complex. The dual of a CP is itself a CP, and its solution often provides interesting and useful information about the original problem. For example, if the dual problem is unbounded, then the original must be infeasible. Under certain conditions, the reverse implication is also true: if a problem is infeasible, then its dual must be unbounded. These and other consequences of duality facilitate the construction of numerical algorithms with definitive stopping criteria for detecting infeasibility, unboundedness, and near-optimality. For a more complete development of convex duality, see [137, 102].

Another important property of CPs is the provable existence of efficient algorithms for solving them. Nesterov and Nemirovsky proved that a polynomial-time barrier method can be constructed for any CP that meets certain technical conditions [117]. Other authors have shown that problems which do not meet those conditions can be embedded into larger problems that do—effectively making barrier methods universal [169, 102, 171].

Finally, we note that the theoretical properties discussed here, including the existence of efficient solution methods, hold even if a CP is nondifferentiable—that is, if one or more of the constraint or objective functions is nondifferentiable. The practical ramifications of this fact are discussed in §3.4.

3.2 Numerical algorithms

The existence of efficient algorithms for solving CPs has been known since the 1970s, but it is only through advances in the last two decades that this promise has been realized in practice. Much of the modern work in numerical algorithms has focused on *interior-point methods* [166, 37, 163]. Initially such work was limited to LPs [88, 133, 73, 89, 109, 105, 62], but was soon extended to encompass other CPs as well [117, 118, 7, 87, 119, 162, 15, 120]. Now a number of excellent solvers are readily available, both commercial and freely distributed.

Below we provide a brief survey of solvers for convex optimization. For the purposes of our discussion, we have separated them into two classes: those that rely on *standard forms*, and those that rely on *custom code*.

Standard forms

Most solvers for convex programming are designed to handle certain prototypical CPs known as *standard forms*. In other words, such solvers handle a limited family of problems with a very specific structure, or obeying certain conventions. The least squares problem and the LP are two common examples of standard forms; we list several others below. These solvers trade generality for ease of use and performance.

It is instructive to think of the collection of standard form solvers as a "toolchest" of sorts. This toolchest is reasonably complete, in that most CPs that one might encounter can be transformed into one (or more) of these standard forms. However, the required transformations are often far from obvious, particularly for an applications-oriented user.

Smooth convex programs

A number of solvers have been designed to solve CPs in standard NLP form (19), under the added condition that the objective function f and inequality constraint functions g_i are *smooth*—that is, twice continuously differentiable—at least over the region that the algorithm wishes to search. We will call such problems *smooth* CPs; conversely, we will label CPs that do not fit this categorization as *nonsmooth* CPs.

Software packages that solve smooth CPs include LOQO [154], which employs a primal/dual method, and the commercial package MOSEK [110], which implements the homogeneous algorithm. These solvers generally perform quite well in practice. Many systems designed for smooth *nonconvex* NLPs will often solve smooth CPs efficiently as well [32, 69, 112, 70, 41, 31, 156, 122, 14, 33, 13, 71, 61, 153]. This is not surprising when one considers that these algorithms typically exploit *local* convexity when computing search directions.

One practical difficulty in the use of smooth CP or NLP solvers is that the solver must be able to calculate the gradient and Hessian of the objective and inequality constraint functions at points of its choosing. In some cases, this may require the writing of custom code to perform these computations. Many modeling frameworks simplify this process greatly in most cases by allowing functions to be expressed in natural mathematical form and compute derivatives automatically (*e.g.*, [56, 18]).

Conic programs

An entire family of standard forms that have become quite common are the primal and dual *conic forms*

$$\begin{array}{c} \text{minimize } c^T x \\ \text{subject to } Ax = b \\ x \in \mathcal{K} \end{array} \quad \text{or} \quad \begin{array}{c} \text{minimize } b^T y \\ \text{subject to } A^T y + z = c \\ z \in \mathcal{K}^* \end{array} \tag{20}$$

in which the sets $\mathcal{K}$ and $\mathcal{K}^*$ are closed, convex *cones* (*i.e.*, they satisfy $\alpha\mathcal{K} \equiv \mathcal{K}$ and $\alpha\mathcal{K}^* \equiv \mathcal{K}^*$ for all $\alpha > 0$). The most common conic form is the LP, for which $\mathcal{K} = \mathcal{K}^*$ is the nonnegative orthant

$$\mathcal{K} = \mathcal{K}^* = \mathbb{R}^n_+ \triangleq \{ x \in \mathbb{R}^n \mid x_i \geq 0,\ i = 1, \ldots, n \} \tag{21}$$

It can be shown that virtually any CP can be represented in conic form, with appropriate choice of $\mathcal{K}$ or $\mathcal{K}^*$ [118]. In practice, two conic forms (besides LP) dominate all recent study and implementation. One is the *semidefinite program* (SDP), for which $\mathcal{K} = \mathcal{K}^*$ is an isomorphism of the cone of positive semidefinite matrices

$$\mathcal{S}^n_+ \triangleq \left\{ X = X^T \in \mathbb{R}^{n \times n} \mid \lambda_{\min}(X) \geq 0 \right\}. \tag{22}$$

The second is the *second-order cone program* (SOCP), for which $\mathcal{K} = \mathcal{K}^*$ is the Cartesian product of one or more second-order or Lorentz cones,

$$\mathcal{K} = \mathcal{Q}^{n_1} \times \cdots \times \mathcal{Q}^{n_K}, \quad \mathcal{Q}^n \triangleq \{ (x, y) \in \mathbb{R}^n \times \mathbb{R} \mid \|x\|_2 \leq y \}. \tag{23}$$

SDP and SOCP receive this focused attention because many applications have been discovered for them, and because their geometry admits certain useful algorithmic optimizations [121, 66, 149, 53, 68, 102, 126]. Publicly available solvers for SDP and SOCP include SeDuMi [140], CDSP [22], SDPA [58], SDPT3 [152], and DSDP [12]. These solvers are generally quite efficient, reliable, and are entirely data-driven: that is, they require no external code to perform function calculations.

Geometric programs

Another standard form that has been studied for some time, but which has generated renewed interest recently, is the *geometric program* (GP). The GP is actually a bit of an unique case, in that it is in fact not convex—but a simple transformation produces an equivalent problem that is convex. In convex form, the objective and inequality constraint functions obtain a so-called "log-sum-exp" structure; for example,

$$f(x) \triangleq \log \sum_{k=1}^{M} e^{a_k^T x + b_k} \qquad a_k \in \mathbb{R}^n,\ b_k \in \mathbb{R},\ k = 1, 2, \ldots, M \tag{24}$$

GPs have been used in various fields since the late 1960s [48]. In convex form they are smooth CPs, but recent advances in specialized algorithms have greatly improved the efficiency of their solution [92].

Custom code

There are instances where a CP cannot be transformed into one of the standard forms above—or perhaps the transformations cannot be determined. An

alternative is to use one of the methods that we list here, which are *universal* in the sense that they can, in theory, be applied to *any* CP. The cost of this universality is that the user must determine certain mathematical constructions, and write custom code to implement them.

Barrier methods

A barrier method replaces the inequality constraint set

$$\mathcal{S} \triangleq \{ x \in \mathbb{R}^n \mid g_i(x) \leq 0,\ i = 1, \ldots, n_g \} \tag{25}$$

with a twice differentiable convex barrier function $\phi : \mathbb{R}^n \to \mathbb{R}$ satisfying $\mathbf{dom}\,\phi = \operatorname{Int}\mathcal{S}$, producing a modified problem

$$\begin{array}{l} \text{minimize } f(x) + t\phi(x) \\ \text{subject to } h_j(x) = 0 \quad j = 1, 2, \ldots, n_h \end{array} \tag{26}$$

Under mild conditions, the solution to this modified problem converges to that of the original problem as $t \to 0$. Each iteration of a barrier method effectively performs Newton minimization steps on (26) for steadily decreasing values of t. A complete development of barrier methods, including proofs of universality, convergence, and performance, as well as a number of complete algorithms, is given in [118].

There are several practical roadblocks to the use of a barrier method. First of all, this author knows of no publicly-available, *general purpose* barrier solver; someone wishing to use this technique would have to write their own. Even if a barrier solver is found, the user must supply code to compute the value and derivatives of the barrier function. Furthermore, determining a valid barrier function is not always trivial, particularly if the inequality constraints are nondifferentiable.

Cutting plane methods

Localization or cutting-plane methods such as ACCPM [131] require no derivative information for the functions f and g_i, instead relying solely on cutting planes to restrict the search set. The user is expected to supply code to compute subgradients or cutting planes. The performance of these methods is usually inferior to the others mentioned here, but they are ideal for use when second derivative information is not available or difficult to compute. In addition, they often lend themselves to distributed methods for solution.

3.3 Applications

A wide variety of practical applications for convex programming have already been discovered, and the list is steadily growing. Perhaps the field in which the application of convex programming is the most mature and pervasive is control theory; see [44, 8, 11, 40, 114] for a sample of these applications. Other fields where applications for convex optimization are known include, but are not limited to,

- robotics [132, 35, 82];
- pattern analysis and data mining, including support vector machines [141, 172, 134, 164, 91];
- combinatorial optimization and graph theory [101, 5, 28, 77, 57, 55, 90, 170, 174, 111, 42, 60, 59, 85];
- structural optimization [2, 23, 9, 25, 26, 173, 86, 27, 24, 94];
- algebraic geometry [125, 95, 97],[96];
- signal processing [79, 161, 65, 139, 150, 165, 49, 151, 6, 47, 103, 143, 54, 144, 64, 146, 108, 107, 128, 50, 129, 45, 17, 72, 168, 160, 145, 175];
- communications and information theory [19, 98, 43, 135, 75, 148, 4];
- networking [20, 30];
- circuit design [157, 158, 80, 39, 78, 16];
- quantum computation [46];
- neural networks [127];
- chemical engineering [136];
- economics and finance [67, 167].

This list, while large, is certainly incomplete, and excludes applications where only LP or QP is employed. Such a list would be significantly larger; and yet convex programming is of course a generalization of these technologies.

One promising source of new applications for convex programming is the extension and enhancement of existing applications for linear programming. An example of this is *robust linear programming*, which allows uncertainties in the coefficients of an LP model to be accounted for in the solution of the problem, by transforming it into a nonlinear CP [104]. This approach produces robust solutions more quickly, and arguably more reliably, than using Monte Carlo methods. Presumably, robust linear programming would find application anywhere linear programming is currently employed, and where uncertainty in the model poses a significant concern.

Some may argue that our prognosis of the usefulness of convex programming is optimistic, but there is good reason to believe the number of applications is in fact being underestimated. We can appeal to the history of linear programming as precedent. George Dantzig first published his invention of the simplex method for linear programming in the 1947; and while a number of military applications were soon found, it was not until 1955-1960 that the field enjoyed robust growth [38]. Certainly, this delay was in large part due to the dearth of adequate computational resources; but that is the point: the discovery of new applications accelerated only once hardware and software advances made it truly practical to solve LPs.

Similarly, then, there is good reason to believe that the number of known applications for convex programming will rise dramatically if it can be made easier for people to create, analyze, and solve CPs.

3.4 Convexity and differentiability

As mentioned in §3.2, many solvers for smooth (nonconvex) NLPs can be used to effectively solve many smooth CPs. An arguable case can be made that the advance knowledge of convexity is not critical in such cases. Dedicated solvers for smooth CPs do provide some advantages, such as the ability to reliably detect infeasibility and degeneracy; see, for example, [171]. But such advantages may not immediately seem compelling to those accustomed to traditional nonlinear programming.

In the nonsmooth case, the situation is markedly different. Nondifferentiability poses a significant problem for traditional nonlinear programming. The best methods available to solve nondifferentiable NLPs are far less accurate, reliable, or efficient than their smooth counterparts. The documentation for the GAMS modeling framework "strongly discourages" the specification of nonsmooth problems, instead recommending that points of nondifferentiability be eliminated by replacing them with Boolean variables and expressions [18]. But doing so is not always straightforward, and introduces significant practical complexities of a different sort.

In contrast, there is nothing in *theory* that prevents a nonsmooth CP from being solved as efficiently as a smooth CP. For example, as mentioned in §3.1, the proof provided by Nesterov and Nemirovsky of the existence of barrier functions for CPs does not depend on smoothness considerations. And nonsmooth CPs can often be converted to an equivalent smooth problem with a carefully chosen transformation—consider the ℓ_∞, ℓ_1, and largest-L norm minimization problems presented in §2. Of course, neither the construction of a valid barrier function nor the smoothing transformation is always (or even often) obvious.

One might ask: just how often are the CPs encountered in practice nonsmooth? We claim that it is quite often. Most non-trivial SDPs and SOCPs, for example, are nonsmooth. Common convex functions such as the absolute value and most norms are nonsmooth. Examining the current inventory of applications for convex programming, and excluding those that immediately present themselves as LPs and QPs, smoothness is the exception, not the rule.

Thus a convex programming methodology that provides truly practical support for nonsmooth problems is of genuine practical benefit. If such a solution can be achieved, then the *a priori* distinction between convexity and nonconvexity becomes far more important, because the need to avoid nondifferentiability remains only in the nonconvex case.

3.5 Convexity verification

Given the benefits of advance knowledge of convexity, it would be genuinely useful to perform automatic *convexity verification*: that is, to determine whether or not a given mathematical program is convex. Unfortunately, the task of determining whether or not a general mathematical program is convex

is at least as difficult as solving nonconvex problems: that is, it is theoretically intractable. Practical attempts have achieved various degrees of success, as we survey here.

Perhaps the most computationally ambitious approach to convexity verification has been independently developed by Crusius [36] and Orban and Fourer [123]. The first of these has been refined and integrated into a commercial offering [116, 63]. These systems combine interval methods, symbolic or automatic differentiation, and other methods to determine if the Hessians of key subexpressions in the objective and constraints are positive semidefinite over an estimate of the feasible region. The efforts are impressive, but these systems do fail to make conclusive determinations in many cases—that is, some problems can neither be proven convex nor nonconvex. Furthermore, these systems are limited to smooth NLPs, due to their reliance on derivative information.

Limiting the scope to one or more standard forms produces more reliable results. For example, many modeling frameworks automatically determine if a model is an LP, enabling specialized algorithms to be selected for them [56, 18]. Similar approaches are employed by modeling tools such as SDPSOL and LMITOOL to automatically verify SDPs [159, 51]. These approaches are effective because these particular standard forms can be recognized entirely through an analysis of their *textual* structure. They are perfectly reliable, making conclusive determinations in every case: *e.g.*, a model is proven to be an LP, or proven otherwise. But of course, generality is significantly compromised. And these systems do not attempt to recognize problems that are *transformable* into the supported standard form. For example, the ℓ_1 norm minimization in §2.1 would have to be manually converted to an LP before it would be recognized as such by these systems.

Yet another alternative is provided by the MPROBE [34] system, which employs numerical sampling to *empirically* determine the shapes of constraint and objective functions. It will often conclusively *disprove* linearity or convexity in many cases, but it can never conclusively prove convexity, because doing so would require an exhaustive search. To be fair, its author makes no claims to that effect, instead promoting MPROBE as a useful tool for interactively assisting the user to make his own decisions.

These practical approaches to automatic convexity verification compromises generality, whether due to limitations of the algorithms or by deliberate restrictions in scope. As we will see below, disciplined convex programming makes a compromise of a different sort, recovering generality by incorporating knowledge provided by its users.

4 Modeling frameworks

The purpose of a *modeling framework* is to enable someone to become a proficient *user* of a particular mathematical technology (*e.g.*, convex programming)

without requiring that they may become an expert in it (*e.g.*, interior-point methods). It accomplishes this by providing a convenient interface for specifying problems, and then by automating many of the underlying mathematical and computational steps for analyzing and solving them.

A number of excellent modeling frameworks for LP, QP, and NLP are in widespread use and have had a broad and positive impact on the use of optimization in many application areas, including AMPL [56], GAMS [18], LINGO [99], and Frontline [63]. These frameworks are well-suited for solving smooth CPs as well. More recently, a number of modeling frameworks for semidefinite programming have been developed, including SDPSOL [159], LMITool [51], MATLAB's LMI Control Toolbox [147], YALMIP [100], and SOSTOOLS [130]. These tools are used by thousands in control design, analysis, and research, and in other fields as well.

We are developing a modeling framework called `cvx` to support the disciplined convex programming methodology. The framework addresses a number of the challenges already addressed in this article, including support for non-differentiable problems, convexity verification, and automatic conversion to solvable form. We have implemented a simple barrier solver for the framework; but in fact, any numerical method currently used to solve CPs can be used to solve DCPs. So we intend to work to create interfaces between `cvx` and other well-known solvers.

```
maximize      entropy( x1, x2, x3, x4 );
subject to    a11 x1 + a12 x2 + a13 x3 + a14 x4 = b1;
              a21 x1 + a22 x2 + a23 x3 + a24 x4 = b2;
                  x1 +     x2 +     x3 +     x4 = 1;
parameters    a11, a12, a13, a14, b1,
              a21, a22, a23, a24, b2;
variables     x1 >= 0, x2 >= 0, x3 >= 0, x4 >= 0;
function entropy( ... ) concave;
```

Fig. 2. An example CP in the `cvx` modeling language.

The `cvx` framework is built around a modeling language that allows optimization problems to be expressed using a relatively obvious mathematical syntax. The language shares a number of basic features with other modeling languages such as AMPL or GAMS, such as parameter and variable declarations, common mathematical operations, and so forth. See Figure 2 for an example of a simple entropy maximization problem expressed in the `cvx` syntax.

Throughout this article, we will illustrate various concepts using examples rendered in the `cvx` modeling language, using a fixed-width font (`example`) to distinguish them. However, because `cvx` is still in development, it is possible that future versions of the language will use a slightly different syntax; and a

few examples use features of the language that have not yet been implemented. So the examples should not be treated as definitive references. The reader is invited to visit the Web site `http://www.stanford.edu/~boyd/cvx` to find the most recent information about `cvx`.

5 Disciplined convex programming

Disciplined convex programming is inspired by the practices of those who regularly study and use convex optimization in research and applications. They do not simply construct constraints and objective functions without advance regard for convexity; rather, they draw from a mental library of functions and sets whose convexity properties are already known, and combine and manipulate them in ways which convex analysis insures will produce convex results. When it proves necessary to determine the convexity properties of a new function or set from basic principles, that function or set is added to the mental library to be reused in other models.

Disciplined convex programming formalizes this strategy, and includes two key components:

- An *atom library*: an *extensible* collection of functions and sets, or *atoms*, whose properties of curvature/shape (convex/concave/affine), monotonicity, and range are explicitly declared.
- A *convexity ruleset*, drawn from basic principles of convex analysis, that governs how atoms, variables, parameters, and numeric values can be combined to produce convex results.

A valid *disciplined convex program*, or DCP, is simply a mathematical program built in accordance with the convexity ruleset using elements from the atom library. This methodology provides a teachable conceptual framework for people to use when studying and using convex programming, as well as an effective platform for building software to analyze and solve CPs.

The convexity ruleset, introduced in §6 below, has been designed to be easy to learn and understand. The rules constitute a set of sufficient conditions to guarantee convexity. In other words, any mathematical program constructed in accordance with the convexity ruleset is guaranteed to be convex. The converse, however, is not true: it is possible to construct problems which do not obey the rules, but which are convex nonetheless. Such problems are not valid DCPs, and the methodology does not attempt to accommodate them. This does not mean that they cannot be solved, but it does mean that they will have to be rewritten to comply with the convexity ruleset.

Because the convexity ruleset limits the variety of CPs that can be constructed from a fixed atom library, it follows that the generality of disciplined convex programming depends upon that library being extensible. Each atom must be given a declaration of information about its curvature or shape, monotonicity, and range, information which is referred to when verifying that the

atom is used in accordance with the convexity ruleset. We introduce the atom library in detail in §7.

In §9, we examine the consequences of a restricted approach such as this. In particular, we provide some examples of some common and useful CPs that, in their native form, are not *disciplined* convex programs. We show that the these limitations are readily remedied using the extensibility of the atom library. We argue that the remedies are, in fact, consistent with the very thought process that disciplined convex programming is attempting to formalize.

Finally, in §10, we discuss how the elements in the atom library can be *implemented*—that is, how they can be represented in a form usable by numerical algorithms. In addition to some very traditional forms, such as barrier functions, cutting planes, derivatives for Newton's method, and so forth. we introduce the concept of *graph implementations*. Graph implementations allow functions and sets to be defined in terms of other DCPs, and provide such benefits as support for nondifferentiable functions.

Before we proceed, let us address a notational issue. We have chosen to follow the lead of [137] and adopt the extended-valued approach to defining convex and concave functions with limited domains; *e.g.*,

$$f : \mathbb{R} \to (\mathbb{R} \cup +\infty), \qquad f(x) = \begin{cases} +\infty & x < 0 \\ x^{1.5} & x \geq 0 \end{cases} \tag{27}$$

$$g : \mathbb{R} \to (\mathbb{R} \cup -\infty), \qquad g(x) = \begin{cases} -\infty & x \leq 0 \\ \log x & x > 0 \end{cases} \tag{28}$$

Using extended-valued functions simplifies many of the derivations and proofs. Still, we will on occasion use the **dom** operator to refer to the set of domain values that yield finite results:

$$\mathbf{dom}\, f = \{\, x \mid f(x) < +\infty \,\} = [0, +\infty), \tag{29}$$

$$\mathbf{dom}\, g = \{\, x \mid g(x) > -\infty \,\} = (0, +\infty) \tag{30}$$

6 The convexity ruleset

The convexity ruleset governs how variables, parameters, and atoms (functions and sets) may be combined to form DCPs. DCPs are a strict subset of general CPs, so another way to say this is that the ruleset imposes a set of conventions or restrictions on CPs. The ruleset can be separated into four categories: *top-level rules*, *product-free rules*, *sign rules*, and *composition rules*.

6.1 Top-level rules

As the name implies, top-level rules govern the top-level structure of DCPs. These rules are more descriptive than they are restrictive, in the sense that

nearly all *general* CPs follow these conventions anyway. But for completeness they must be explicitly stated.

Problem types. A valid DCP can either be:

T1 a *minimization*: a convex objective and zero or more convex constraints;

T2 a *maximization*: a concave objective and zero or more convex constraints; or

T3 a *feasibility problem*: no objective, and one or more convex constraints.

A valid DCP may also include any number of *assertions*; see rule **T9**.

At the moment, support for multiobjective problems and games has not been developed, but both are certainly reasonable choices for future work.

affine `==` *affine*	**(T4)**
convex `<=` *concave* or *convex* `<` *concave*	**(T5)**
concave `>=` *convex* or *concave* `>` *convex*	**(T6)**
(*affine*, *affine*, ..., *affine*) `in` *convex set*	**(T7)**

Fig. 3. Valid constraints.

Constraints. See Figure 3. Valid constraints include:

T4 an equality constraint with affine left- and right-hand expressions.

T5 a less than ($<$,$\leq$) inequality, with a convex left-hand expression and a concave right-hand expression;

T6 a greater than ($>$,$\geq$) inequality, with a concave left-hand expression and a convex right-hand expression; or

T7 a set membership constraint $(lexp_1, \ldots, lexp_m) \in cset$, where $m \geq 1$, $lexp_1, \ldots, lexp_m$ are affine expressions, and $cset$ is a convex set.

Non-equality ($\neq$) constraints and set non-membership ($\notin$) constraints are not permitted, because they are convex only in exceptional cases—and support for exceptional cases is anathema to the philosophy behind disciplined convex programming.

Constant expressions and assertions.

T8 Any well-posed numeric expression consisting only of numeric values and parameters is a valid constant expression.

T9 Any Boolean expression performing tests or comparisons on valid constant expressions is a valid assertion.

T10 If a function or set is parameterized, then those parameters must be valid constant expressions.

A *constant expression* a numeric expression involving only numeric values and/or parameters; a *non-constant* expression depends on the value of at least one problem variable. Obviously a constant expression is trivially affine,

convex, and concave. Constant expressions must be well-posed: which, for our purposes, means that they produce well-defined results for any set of parameter values that satisfy a problem's assertions.

An *assertion* resembles a constraint, but involves only constant expressions. As such, they are not true constraints *per se*, because their truth or falsity is determined entirely by the numerical values supplied for a model's parameters, *before* the commencement of any numerical optimization algorithm. Assertions are not restricted in the manner that true constraints are; for example, non-equality ($\neq$) and set non-membership ($\notin$) operations may be freely employed. Assertions serve as *preconditions*, guaranteeing that a problem is numerically valid or physically meaningful. There are several reasons why an assertion may be wanted or needed; for example:

- to represent physical limits dictated by the model. For example, if a parameter w represents the physical weight of an object, an assertion $w > 0$ enforces the fact that the weight must be positive.
- to insure numeric well-posedness. For example, if x, y, and z are variables and a, b, and c are parameters, then the inequality constraint $ax + by + z/c \leq 1$ is well-posed only if c is nonzero; this can be insured by an assertion such as $c \neq 0$ or $c > 0$.
- to guarantee compliance with the preconditions attached to a function or set in the atom library. For example, a function $f_p(x) = \|x\|_p$ is parameterized by a value $p \geq 1$. If p is supplied as a parameter, then an assertion such as $p \geq 1$ would be required to guarantee that the function is being properly used. See §7.1 for information on how such preconditions are supplied in the atom library.
- to insure compliance with the sign rules §6.3 or composition rules §6.4 below; see those sections and §8 for more details.

The final rule **T10** refers to functions or sets that are parameterized; *e.g.*,

$$f_p : \mathbb{R}^n \to \mathbb{R}, \quad f_p(x) = \|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \tag{31}$$

$$B_p = \{ x \in \mathbb{R}^n \mid \|x\|_p \leq 1 \} \tag{32}$$

and simply states that parameters such as p above must be constant. Of course this is generally assumed, but we must make it explicit for the purposes of computer implementation.

6.2 The product-free rules

Some of the most basic principles of convex analysis govern the sums and scaling of convex, concave, and affine expressions; for example:

- The sum of two or more convex (concave, affine) expressions is convex (concave, affine).

- The product of a convex (concave) expression and a nonnegative constant expression is convex (concave).
- The product of a convex (concave) expression and a nonpositive constant expression, or the simple negation of the former, is concave (convex).
- The product of an affine expression and any constant is affine.

Conspicuously absent from these principles is any mention of the *product* of convex or concave expressions. The reason for this is simple: there is no simple, general principle that can identify the curvature in such cases. For instance, suppose that x is a scalar variable; then:

- The expression $x \cdot x$, a product of two affine expressions, is convex.
- The expression $x \cdot \log x$, a product between and affine and a concave expression, is convex.
- The expression $x \cdot e^x$, a product between an affine and a convex expression, is neither convex nor concave.

For this reason, the most prominent structural convention enforced by disciplined convex programming is the prohibition of products (and related operations, like exponentiation) between non-constant expressions. The result is a set of rules appropriately called the *product-free rules*:

> *Product-free rule for numeric expressions:* All valid numeric expressions must be product-free; such expressions include:
> PN1 A simple variable reference.
> PN2 A *constant* expression.
> PN3 A call to a function in the atom library. Each argument of the function must be a product-free expression.
> PN4 The sum of two or more product-free expressions.
> PN5 The difference of product-free expressions.
> PN6 The negation of a product-free expression.
> PN7 The product of a product-free expression and a constant expression.
> PN8 The division of a product-free expression by a constant expression.
> We assume in each of these rules that the results are well-posed; for example, that dimensions are compatible.

In the scalar case, a compact way of restating these rules is to say that a valid numeric expression can be reduced to the form

$$a + \sum_{i=1}^{n} b_i x_i + \sum_{j=1}^{L} c_j f_j(arg_{j,1}, arg_{j,2}, \ldots, arg_{j,m_j}) \tag{33}$$

where a, b_i, c_j are constants; x_i are the problem variables; and $f_j : \mathbb{R}^{m_j} \to \mathbb{R}$ are functions from the atom library, and their arguments $arg_{j,k}$ are product-free expressions themselves. Certain special cases of (33) are notable: if $L = 0$,

then (33) is a simple affine expression; if, in addition, $b_1 = b_2 = \cdots = b_n = 0$, then (33) is a constant expression.

For an illustration of the use of these product-free rules, suppose that a, b, and c are parameters; x, y, and z are variables; and $f(\cdot)$, $g(\cdot,\cdot)$, and $h(\cdot,\cdot,\cdot)$ are functions from the atom library. Then the expression

$$af(x) + y + (h(x, bg(y,z), c) - z + b)/a \tag{34}$$

satisfies the product-free rule, which can be seen by rewriting it as follows:

$$(b/a) + y + (-1/a)z + af(x) + (1/a)h(x, bg(y,z)c) \tag{35}$$

On the other hand, the following expression does not obey the product-free rule:

$$axy/2 - f(x)g(y,z) + h(x, y^b, z, c) \tag{36}$$

Now certainly, because these rules prohibit *all* products between non-constant expressions, some genuinely useful expressions such as quadratic forms like $x \cdot x$ are prohibited; see §9 for further discussion on this point.

For set expressions, a similar set of product-free rules apply:

Product-free rules for set expressions: All valid set expressions used in constraints must be product-free; such expressions include:

PS1 A call to a convex set in the atom library.

PS2 A call to a function in the atom library. Each argument of the function must be a product-free set expression or a constant (numeric) expression.

PS3 The sum of product-free expressions, or of a product-free expression and a constant expression, or vice versa.

PS4 The difference of two product-free set expressions, or of of a product-free set expression and a constant expression, or vice versa.

PS5 The negation of a product-free set expression.

PS6 The product of a product-free set expression and a constant expression.

PS7 The division of a product-free set expression by a constant expression.

PS8 The intersection of two or more product-free set expressions.

PS9 The Cartesian product of two or more product-free set expressions.

We also assume in each of these rules that the results are well-posed; for example, that dimensions are compatible.

In other words, valid set expressions are reducible to the form

$$a + \sum_{i=1}^{n} b_i S_i + \sum_{j=1}^{L} c_j f_j(arg_{j,1}, arg_{j,2}, \ldots, arg_{j,m_j}) \tag{37}$$

where a, b_i, c_j are constants, and the quantities S_k are either sets from the atom library, or intersections and/or Cartesian products of valid set expressions. The functions $f_j : \mathbb{R}^{m_j} \to \mathbb{R}$ are functions in the atom library, and their arguments $arg_{j,k}$ are product-free set expressions themselves. As we will see in §6.3 below, set expressions are more constrained than numerical expressions, in that the functions f_j must be affine.

It is well understood that the intersection of convex sets is convex, as is the direct product of convex sets; and that unions and set differences generally are not convex. What may not be clear is why **PS8-PS9** are considered "product-free" rules. By examining these rules in terms of indicator functions, the link becomes clear. Consider, for example, the problem

$$\begin{array}{l} \text{minimize } ax + by \\ \text{subject to } (x, y) \in (S_1 \times S_1) \cup S_2 \end{array} \tag{38}$$

If $\phi_1 : \mathbb{R} \to \mathbb{R}$ and $\phi_2 : (\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$ are convex indicator functions for the sets S_1 and S_2, respectively, then the problem can be reduced to

$$\text{minimize } ax + by + (\phi_1(x) + \phi_2(y))\phi_2(x, y) \tag{39}$$

and the objective function now violates the product-free rule. What has occurred, of course, is that the union operation became a forbidden product.

6.3 The sign rules

Once the product-free conventions are established, the sum and scaling principles of convex analysis can be used to construct a simple set of sufficient conditions to establish whether or not expressions are convex, concave, or affine. These conditions form what we call the *sign rules*, so named because their consequence is to govern the signs of the quantities $c_1, \ldots, c_L$ in (33). We can concisely state the sign rules for numeric expressions in the following manner.

Sign rules for numeric expressions. Given a product-free expression, the following must be true of its reduced form (33):

SN1 If the expression is expected to be convex, then each term $c_j f_j(\ldots)$ must be convex; hence of the following must be true:
- $f_j(arg_{j,1}, arg_{j,2}, \ldots, arg_{j,m_j})$ is affine;
- $f_j(arg_{j,1}, arg_{j,2}, \ldots, arg_{j,m_j})$ is convex and $c_j \geq 0$;
- $f_j(arg_{j,1}, arg_{j,2}, \ldots, arg_{j,m_j})$ is concave and $c_j \leq 0$.

SN2 If the expression is expected to be concave, then each term $c_j f_j(\ldots)$ must be concave; hence one of the following must be true:
- $f_j(arg_{j,1}, arg_{j,2}, \ldots, arg_{j,m_j})$ is affine;
- $f_j(arg_{j,1}, arg_{j,2}, \ldots, arg_{j,m_j})$ is concave and $c_j \geq 0$;
- $f_j(arg_{j,1}, arg_{j,2}, \ldots, arg_{j,m_j})$ is convex and $c_j \leq 0$.

SN3 If the expression is expected to be affine, then each function f_j must be affine, as must each of its arguments $arg_{j,1}, arg_{j,2}, \ldots, arg_{j,m_j}$.
SN4 If the expression is expected to be constant, then it must be true that $L = 0$ and $b_1 = b_2 = \cdots = b_n = 0$.
All function arguments must obey these rules as well, with their expected curvature dictated by the composition rules (§6.4).

For example, suppose that that the expression (34) is expected to be convex, and that the atom library indicates that the function $f(\cdot)$ is convex, $g(\cdot,\cdot)$ is concave, and $h(\cdot,\cdot,\cdot)$ is convex. Then the sign rule dictates that

$$af(x) \text{ convex} \implies a \geq 0 \tag{40}$$

$$(1/a)h(x, bg(y,z), c) \text{ convex} \implies 1/a \geq 0 \tag{41}$$

Function arguments must obey the sign rule as well, and their curvature is dictated by the composition rules discussed in the next section. So, for example, if the second argument of h is required to be convex, then

$$bg(y,z) \text{ convex} \implies b \leq 0 \tag{42}$$

It is the responsibility of the modeler to insure that the values of the coefficients $c_1, \ldots, c_L$ obey the sign rule; that is, that conditions such as those generated in (40)-(42) are satisfied. This can be accomplished by adding appropriate assertions to the model; see §8 for an example of this.

There is only one "sign rule" for set expressions:

The sign rule for set expressions. Given a product-free set expression, the following must be true of its reduced form (37):
SS1 Each function f_j, and any functions used in their arguments, must be affine.

Unlike the product-free rule for numerical expressions, functions involved in set expressions are *required* to be affine. To understand why this must be the case, one must understand how an expression of the form (37) is interpreted. A simple example should suffice; for the $L = 0$ case,

$$x \in a + \sum_{i=1}^{n} b_i S_i \iff \exists (t_1, t_2, \ldots, t_n) \in S_1 \times S_2 \times \cdots \times S_n \quad x = a + \sum_{i=1}^{n} b_i t_i \tag{43}$$

When $L > 0$, similar substitutions are made recursively in the function arguments as well, producing a similar result: a series of simple set membership constraints of the form $t_k \in S_k$, and a single equality constraint. Thus in order to insure that this implied equality constraint is convex, set expressions (specifically, those used in constraints) must reduce to affine combinations of sets. (Of course, set expressions used in assertions are not constrained in this manner.)

6.4 The composition rules

A basic principle of convex analysis is that the composition of a convex function with an affine mapping remains convex. In fact, under certain conditions, similar guarantees can be made for compositions with nonlinear mappings as well. The ruleset incorporates a number of these conditions, and we have called them the *composition rules.*

Designing the composition rules required a balance between simplicity and expressiveness. In [29], a relatively simple composition rule for convex functions is presented:

Lemma 1. *If* $f : \mathbb{R} \to (\mathbb{R} \cup +\infty)$ *is convex and nondecreasing and* $g : \mathbb{R}^n \to (\mathbb{R} \cup +\infty)$ *is convex, then* $h = f \circ g$ *is convex.*

So, for example, if $f(y) = e^y$ and $g(x) = x^2$, then the conditions of the lemma are satisfied, and $h(x) = f(g(x)) = e^{x^2}$ is convex. Similar composition rules are given for concave and/or nonincreasing functions as well:

- If $f : \mathbb{R} \to (\mathbb{R} \cup +\infty)$ is convex and nonincreasing and $g : \mathbb{R}^n \to (\mathbb{R} \cup -\infty)$ is concave, then $f \circ g$ is convex.
- If $f : \mathbb{R} \to (\mathbb{R} \cup -\infty)$ is concave and nondecreasing and $g : \mathbb{R}^n \to (\mathbb{R} \cup -\infty)$ is concave, then $f \circ g$ is concave.
- If $f : \mathbb{R} \to (\mathbb{R} \cup -\infty)$ is concave and nonincreasing and $g : \mathbb{R}^n \to (\mathbb{R} \cup +\infty)$ is convex, then $f \circ g$ is concave.

In addition, similar rules are described for functions with multiple arguments.

One way to interpret these composition rules is that they only allow those nonlinear compositions that can be to be *separated* or *decomposed.* To explain, consider a nonlinear inequality $f(g(x)) \leq y$, where f is convex and nondecreasing and g is convex, thus satisfying the conditions of Lemma 1. Then it can be shown that

$$f(g(x)) \leq y \quad \Longleftrightarrow \quad \exists z \; f(z) \leq y, \; g(x) \leq z \tag{44}$$

Similar decompositions can be constructed for the other composition rules as well. Decompositions serve as an important component in the conversion of DCPs into solvable form. Thus the composition rules guarantee that equivalence is reserved when these decompositions are performed.

Now the composition rules suggested by Lemma 1 and its related corollaries are a good start. But despite their apparent simplicity, they require a surprising amount of care to apply. In particular, the use of extended-valued functions is a necessary part of the lemma and has subtle impact. For example, consider the functions

$$f(y) = y^2, \quad g(x) = \|x\|_2 \tag{45}$$

Certainly, $h(x) = f(g(x)) = \|x\|_2^2$ is convex; but Lemma 1 would not predict this, because $f(y)$ is not monotonic. A sensible attempt to rectify the problem

would be to restrict the domain of the function (in the real-valued sense) to the nonnegative orthant, where it is nondecreasing:

$$\bar{f}(y) = \begin{cases} y^2 & y \geq 0 \\ +\infty & y < 0 \end{cases} \tag{46}$$

But while $\bar{f}$ is nondecreasing over its domain, it is *nonmonotonic* in the extended-valued sense, so the lemma does not apply. The only way to reconcile Lemma 1 with this example is to introduce a far less intuitive version of f which extends it in a nondecreasing fashion:

$$\bar{\bar{f}}(y) = \begin{cases} y^2 & y \geq 0 \\ 0 & y < 0 \end{cases} \tag{47}$$

Figure 4 provides a graph of each function. Forcing users of disciplined con-

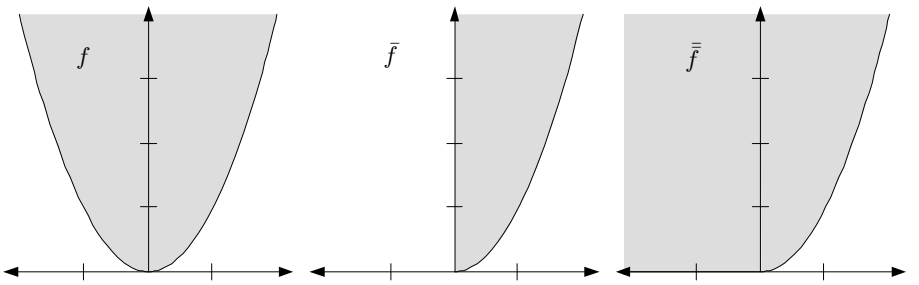

Fig. 4. Three different "versions" of $f(y) = y^2$.

vex programming to consider such technical conditions seems an unnecessary complication, particularly when the goal is to *simplify* the construction of CPs.

To simplify the use of these composition rules, we begin by recognizing something that seems intuitively obvious: f need only be nondecreasing over the range of g. We can formalize this intuition as follows:

Lemma 2. *Let $f : \mathbb{R} \to (\mathbb{R} \cup +\infty)$ and $g : \mathbb{R}^n \to (\mathbb{R} \cup +\infty)$ be two convex functions. If f is nondecreasing over the range of g—i.e., the interval $g(\mathbb{R}^n)$—then $h = f \circ g$ is convex.*

Proof. Let $x_1, x_2 \in \mathbb{R}^n$ and let $\theta \in [0, 1]$. Because g is convex,

$$g(\theta x_1 + (1-\theta)x_2) \leq \theta g(x_1) + (1-\theta)g(x_2) \leq \max\{g(x_1), g(x_2)\} \tag{48}$$

The right-hand inequality has been added to establish that

$$[g(\theta x_1 + (1-\theta)x_2), \theta g(x_1) + (1-\theta)g(x_2)] \subseteq \\ [g(\theta x_1 + (1-\theta)x_2), \max\{g(x_1), g(x_2)\}] \subseteq g(\mathbb{R}^n) \tag{49}$$

f is therefore nondecreasing over this interval; so

$$\begin{aligned} f(g(\theta x_1 + (1-\theta)x_2)) &\leq f(\theta g(x_1) + (1-\theta)g(x_2)) && \textit{(f nondecreasing)} \\ &\leq \theta f(g(x_1)) + (1-\theta)f(g(x_2)) && \textit{(f convex)} \end{aligned} \tag{50}$$

establishing that $h = f \circ g$ is convex.

This lemma does indeed predict the convexity of (45): f is nondecreasing over the interval $[0, +\infty)$, and $g(\mathbb{R}^n) = [0, +\infty)$, which coincide perfectly; hence, $f \circ g$ is convex.

And yet this revised lemma, while more inclusive, presents its own challenge. A critical goal for these composition rules is that adherence can be quickly, reliably, and automatically verified; see §8. The simple composition rules such as Lemma 1 plainly satisfy this condition; but can we be sure to accomplish this with these more complex rules? We claim that it is simpler than it may first appear. Note that our example function $f(x) = x^2$ is nondecreasing over a *half-line*; specifically, for all $x \in [0, +\infty)$. This will actually be true for *any* nonmonotonic scalar function:

Lemma 3. *Let $f : \mathbb{R} \to \mathbb{R} \cup +\infty$ be a convex function which is nondecreasing over some interval $\bar{F} \subset \mathbb{R}$, $\operatorname{Int} \bar{F} \neq \emptyset$. Then it is, in fact, nonincreasing over the entire half-line $F = \bar{F} + [0, +\infty)$; that is,*

$$F = (F_{\min}, +\infty) \quad \textit{or} \quad F = [F_{\min}, +\infty). \tag{51}$$

Proof. If $\bar{F}$ already extends to $+\infty$, we are done. Otherwise, select any two points $x_1, x_2 \in \bar{F}$ and a third point $x_3 > x_2$. Then

$$\begin{aligned} & f(x_2) \leq \alpha f(x_1) + (1-\alpha)f(x_3), \quad \alpha \triangleq (x_2 - x_1)/(x_3 - x_1) \\ \implies \quad & f(x_2) \leq \alpha f(x_2) + (1-\alpha)f(x_3) \quad \implies \quad f(x_2) \leq f(x_3). \end{aligned} \tag{52}$$

So $f(x_3) \geq f(x_2)$ for all $x_3 > x_2$. Now consider another point $x_4 > x_3$; then

$$\begin{aligned} & f(x_3) \leq \bar{\alpha} f(x_2) + (1-\bar{\alpha})f(x_4), \quad \bar{\alpha} \triangleq (x_3 - x_2)/(x_4 - x_2) \\ \implies \quad & f(x_3) \leq \bar{\alpha} f(x_3) + (1-\bar{\alpha})f(x_4) \quad \implies \quad f(x_4) \leq f(x_3) \end{aligned} \tag{53}$$

So $f(x_4) \geq f(x_3)$ for all $x_4 > x_3 > x_2$; that is, f is nondecreasing for all $x > x_2 \in \bar{F}$.

In other words, *any* scalar convex function which is nondecreasing between two points is so over an entire half-line. So determining whether f is nondecreasing over $g(\mathbf{dom}\, g)$ reduces to a single comparison between $F_{\min}$ and $\inf_x g(x)$. For concave or nonincreasing functions, similar intervals can be constructed:

- f convex, nonincreasing: $F = (-\infty, F_{\max}]$ or $F = (-\infty, F_{\max})$

- f concave, nondecreasing: $F = (-\infty, F_{\max}]$ or $F = (-\infty, F_{\max})$
- f concave, nonincreasing: $F = [F_{\min}, +\infty)$ or $F = (F_{\min}, +\infty)$

As we show in §7, it is straightforward to include such intervals in the atom library, so that they are readily available to verify compositions.

The task of determining $\inf_x g(x)$ or $\sup_x g(x)$ remains. In §7, we show that function ranges are *included* in the atom library for just this purpose, alongside information about their curvature and monotonicity properties. But often the inner expression $g(x)$ is not a simple function call, but an expression reducible to the form (33):

$$g(x) = \inf_x a + \sum_{i=1}^{n} b_i x_i + \sum_{j=1}^{L} c_j f_j(\dots) \tag{54}$$

We propose the following heuristic in such cases. Let $X_i \subseteq \mathbb{R}$, $i = 1, 2, \dots, n$, be simple interval bounds on the variables, retrieved from any simple bounds present in the model. In addition, let $F_j = f_j(\mathbb{R}) \subseteq \mathbb{R}$, $j = 1, 2, \dots, L$, be the range bounds retrieved from the atom library. Then

$$g(\mathbb{R}) \subseteq a + \sum_{i=1}^{n} b_i X_i + \sum_{j=1}^{L} c_j F_j \tag{55}$$

So (55) provides a conservative bound on $\inf_x g(x)$ or $\sup_x g(x)$, as needed. In practice, this heuristic proves sufficient for supporting the composition rules in nearly all circumstances. In those exceptional circumstances where it the bound is too conservative, and the heuristic fails to detect a valid composition, a model may have to be rewritten slightly—say, by manually performing the decomposition (44) above. It is a small price to pay for added expressiveness in the vast majority of cases.

Generalizing these composition rules to functions with multiple arguments is straightforward, but requires a bit of technical care. The result is as follows:

The composition rules. Consider a numerical expression of the form

$$f(arg_1, arg_2, \dots, arg_m) \tag{56}$$

where f is a function from the atom library. For each argument arg_k, construct a bound $G_k \subseteq \mathbb{R}$ on the range using the heuristic described above, so that

$$(arg_1, arg_2, \dots, arg_m) \in G = G_1 \times G_2 \times \cdots \times G_m \tag{57}$$

Given these definitions, (56) must satisfy exactly one of the following rules:

C1-C3 If the expression is expected to be convex, then f must be affine or convex, and one of the following must be true for each $k = 1, \dots, m$:

C1 f is nondecreasing in argument k over G, and arg_k is convex; or

C2 f is nonincreasing in argument k over G, and arg_k is concave; or

C3 arg_k is affine.

C4-C6 If the expression is expected to be concave, then f must be affine or concave, and one of the following must be true for each $k = 1, \ldots, m$:

C4 f is nondecreasing in argument k over G, and arg_k is concave; or

C5 f is nonincreasing in argument k over G, and arg_k is convex; or

C6 arg_k is affine.

C7 If the expression is expected to be affine, then f must be affine, and each arg_k is affine for all $k = 1, \ldots, m$.

7 The atom library

The second component of disciplined convex programming is the *atom library*. As a concept, the atom library is relatively simple: an extensible list of functions and sets whose properties of curvature/shape, monotonicity, and range are known. The description of the convexity ruleset in §6 shows just how this information is utilized.

As a tangible entity, the atom library requires a bit more explanation. In cvx, the library is a collection of text files containing descriptions of functions and sets. Each entry is divided into two sections: the *declaration* and the *implementation*. The declaration is divided further into two components:

- the *prototype*: the name of the function or set, the number and structure of its inputs, and so forth.
- the *attribute list*: a list of descriptive statements concerning the curvature, monotonicity, and range of the function; or the shape of the set.

The *implementation* is a computer-friendly description of the function or set that enables it to be used in numerical solution algorithms. What is important to note here is that the implementation section is *not used* to determine whether or not a particular problem is a DCP. Instead, it comes into play only *after* a DCP has been verified, and one wishes to compute a numerical solution. For this reason, we will postpone the description of the implementation until §10.

7.1 The prototype

A function prototype models the usage syntax for that function, and in the process lists the number and dimensions of its arguments; *e.g.*,

```
function sq( x );
```
$$f(x) = x^2 \tag{58}$$

```
function max( x, y );
```
$$g(x,y) = \max\{x,y\} \tag{59}$$

Some functions are parameterized; *e.g.*, $h_p(x) = \|x\|_p$. In `cvx`, parameters are included in the argument list along with the arguments; *e.g.*,

```
function norm_p( p, x[n] ) convex( x ) if p >= 1;
```
$$h_p(x) = \|x\|_p \quad (p \geq 1) \tag{60}$$

Parameters are then distinguished from variables through the curvature attribute; see §7.2 below.

The `norm_p` example also illustrates another feature of `cvx`, which is to allow *conditions* to be placed on the a function's parameters using an `if` construct. In order to use a function with preconditions, they must be enforced somehow; if necessary, by using an assertion. For example, `norm_p( 2.5, x )` would be verified as valid; but if `b` is a parameter, `norm_p( b, x )` would not be, unless the value of `b` could somehow be guaranteed to be greater than 1; for example, unless an assertion like `b > 1` was provided in the model.

Set prototypes look identical to that of functions:

```
set integers( x );
```
$$A = \mathbb{Z} = \{\ldots,-2,-1,0,1,2,\ldots\} \tag{61}$$

```
set simplex( x[n] );
```
$$B = \{\, x \in \mathbb{R}^n \mid x \geq 0,\ \textstyle\sum_i x_i = 1 \,\} \tag{62}$$

```
set less_than( x, y );
```
$$C = \{\, (x,y) \in \mathbb{R} \times \mathbb{R} \mid x < y \,\} \tag{63}$$

Unlike functions, the actual *usage* of a set differs from its prototype—the arguments are in fact the components of the set, and therefore appear to the left of a set membership expression: *e.g.*,

```
x in integers;
```
$$x \in A \tag{64}$$

```
y in simplex;
```
$$y \in B \tag{65}$$

```
(x,y) in less_than;
```
$$(x,y) \in C \tag{66}$$

For parameterized sets, there is yet another difference: the parameters are supplied in a *separate* parameter list, preceding the argument list: *e.g.*,

```
set ball_p( p )( x[n] ) if p >= 1;
```
$$B_p(x) = \{\, x \in \mathbb{R}^n \mid \|x\|_p \leq 1 \,\} \quad (p \geq 1) \tag{67}$$

This parameter list remains on the right-hand side of the constraint along with the name of the set:

```
z in ball_p( q );
```
$$z \in B_q \tag{68}$$

7.2 Attributes

As we have seen in §6, the convexity ruleset depends upon one or more of the following pieces of information about each function and set utilized in a DCP. For sets, it utilizes just one piece of information:

- *shape*: specifically, whether or not the set is convex.

For functions, a bit more information is used:

- *curvature*: whether the function is convex, concave, affine, or otherwise.
- *monotonicity*: whether the functions are nonincreasing or nondecreasing; and over what subsets of their domain they are so.
- *range*: the minimum of convex functions and the maximum of concave functions.

The `cvx` framework allows this information to be provided through the use of *attributes*: simple text tags that allow each of the above properties to be identified as appropriate.

Shape

For sets, only one attribute is recognized: `convex`. A set is either convex, in which case this attribute is applied, or it is not. Given the above four examples, only `integers` is not convex:

```
set integers( x );                                    (69)
set simplex( x[n] ) convex;                           (70)
set less_than( x, y ) convex;                         (71)
set ball_p( p )( x[n] ) convex if p >= 1;             (72)
```

Sets which are not convex are obviously of primary interest for DCP, but non-convex sets may be genuinely useful, for example, for restricting the values of parameters to realistic values.

Curvature

Functions can be declared as `convex`, `concave`, or `affine`, or none of the above. Clearly, this last option is the least useful; but such functions can be used in constant expressions or assertions. No more than one curvature keyword can be supplied. For example:

```
function max( x, y ) convex;          g(x,y) = max{x,y}     (73)
function min( x, y ) concave;         f(x,y) = min{x,y}     (74)
function plus( x, y ) affine;         p(x,y) = x + y        (75)
function sin( x );                    g(x) = sin x          (76)
```

By default, a function declared as convex, concave, or affine is assumed to be *jointly* so over all of its arguments. It is possible to specify that it is so only over a subset of its arguments by listing those arguments after the curvature keyword; for example,

```
function norm_p( p, x[n] ) convex( x ) if p >= 1;
```
$$h_p(x) = \|x\|_p \quad (p \geq 1) \tag{77}$$

In effect, this convention allows parameterized functions to be declared: arguments omitted from the list are treated as the parameters of the function and are expected to be constant.

Disciplined convex programming allows only functions which are *globally* convex or concave to be specified as such. Functions which are "sometimes" convex or concave—that is, over a subset of their domains—are not permitted. For example, the simple inverse function

$$f : \mathbb{R} \to \mathbb{R}, \quad f(x) \triangleq 1/x \qquad (x \neq 0) \tag{78}$$

is neither convex nor concave, and so cannot be used to construct a DCP. However, we commonly think of f as convex *if x is known to be positive.* In disciplined convex programming, this understanding must be realized by defining a *different function*

$$f_{\text{cvx}} : \mathbb{R} \to \mathbb{R}, \quad f(x) \triangleq \begin{cases} 1/x & x > 0 \\ +\infty & x \leq 0 \end{cases} \tag{79}$$

which is globally convex, and can therefore be used in DCPs. Similarly, the power function

$$g : \mathbb{R}^2 \to \mathbb{R}, \quad g(x, y) \triangleq x^y \qquad \text{(when defined)} \tag{80}$$

is convex or concave on certain subsets of $\mathbb{R}^2$, such as:

- convex for $x \in [0, \infty)$ and fixed $y \in [1, \infty)$.
- concave for $x \in [0, \infty)$ and fixed $y \in (0, 1]$;
- convex for fixed $x \in (0, \infty)$ and $y \in \mathbb{R}$

In order to introduce nonlinearities such as $x^{2.5}$ or $x^{0.25}$ into a DCP, then, there must be appropriate definitions of these "restricted" versions of the power function:

$$f_y : \mathbb{R} \to \mathbb{R}, \quad f_y(x) = \begin{cases} x^y & x \geq 0 \\ +\infty & x < 0 \end{cases} \quad (y \geq 1) \tag{81}$$

$$g_y : \mathbb{R} \to \mathbb{R}, \quad g_y(x) = \begin{cases} x^y & x \geq 0 \\ -\infty & x < 0 \end{cases} \quad (0 < y < 1) \tag{82}$$

$$h_y : \mathbb{R} \to \mathbb{R}, \quad h_y(x) = y^x \quad (y > 0) \tag{83}$$

Thus the disciplined convex programming approach forces the user to consider convexity more carefully. We consider this added rigor an advantage, not a liability.

Monotonicity

The monotonicity of a function with respect to its arguments proves to be a key property exploited by the ruleset. For this reason, the cvx atom library provides the keywords `increasing`, `nondecreasing`, `nonincreasing`, or `decreasing` in each of its arguments. Each argument can be given a separate declaration:

```
function exp( x ) convex increasing;
```
$$f(x) = e^x \tag{84}$$

As far as the convexity ruleset is concerned, strict monotonicity is irrelevant; so, for example, `increasing` and `nondecreasing` are effectively synonymous, as are `decreasing` and `nonincreasing`.

There is one somewhat technical but critical detail that must be adhered to when declaring a function to be monotonic. Specifically, monotonicity must be judged in the *extended-valued* sense. For example, given $p \geq 1$, the function

```
function pow_p( p, x )
    convex( x ) if p >= 1;
```
$$f_p(x) = \begin{cases} x^p & x \geq 0 \\ +\infty & x < 0 \end{cases} \tag{85}$$

is increasing (and, therefore, nondecreasing) over its domain. However, in the extended-valued sense, the function is *nonmonotonic*, so f_p cannot be declared as globally nondecreasing.

As suggested in §6.4, the cvx atom library allows *conditions* to be placed on monotonicity. So, for example, $\tilde{f}_p(x)$ is, of course, nondecreasing over $x \in [0, +\infty)$, suggesting the following declaration:

```
function pow_p( p, x ) convex( x ) if p >= 1,
                       increasing( x ) if x >= 0;
```
(86)

Multiple declarations are possible: for example, the function $f(x) = x^2$ is both nonincreasing over $x \in (-\infty, 0]$ and nondecreasing over $x \in [0, +\infty)$:

```
function sq( x ) convex, decreasing if x <= 0,
                         increasing( x ) if x >= 0;
```
(87)

Each argument of a function with multiple inputs can be given a separate, independent monotonicity declaration. For example, $f(x, y) = x - y$ is increasing in x and decreasing in y:

```
function minus( x, y )
        affine increasing( x ) decreasing( y );
```
(88)

Range

Each function definition can include a declaration of its its *range*, using a simple inequality providing a lower or upper bound for the function. For example,

```
function exp( x ) convex, increasing, >= 0;
```
$$f(x) = e^x \tag{89}$$

As with the monotonicity operations, the range must indeed be specified in the extended-valued sense, so it will inevitably be one-sided: that is, all convex functions are unbounded above, and all concave functions are unbounded below.

8 Verification

In order to solve a problem as a DCP, one must first establish that it is indeed a valid DCP—that is, that it involves only functions and sets present in the atom library, and combines them in a manner compliant with the complexity ruleset. A proof of validity is necessarily hierarchical in nature, reflecting the structure of the problem and its expressions. To illustrate the process, consider the simple optimization problem

$$\begin{array}{c} \text{minimize } cx \\ \text{subject to } \exp(y) \leq \log(a\sqrt{x} + b) \\ ax + by = d \end{array} \tag{90}$$

where a, b, c, d are parameters, and x, y are variables. A `cvx` version of this model is given in Figure 5. Note in particular the explicit declarations of the three atoms `exp`, `log`, and `sqrt`. Usually these declarations will reside in an external file, but we include them here to emphasize that every atom used in a model must be accounted for in the atom library.

```
minimize    c x;
subject to  exp( y ) <= log( a sqrt( x ) + b );
            a x + b y = d;
variables   x, y;
parameters  a, b, c, d;
function    exp( x )  convex  increasing >= 0;
function    sqrt( x ) concave nondecreasing;
function    log( x )  concave increasing;
```

Fig. 5. The `cvx` specification for (90).

It is helpful to divide the proof into two stages. The first stage verifies that each of expressions involved is product-free. Below is a textual description of this stage. Each line has been indented to represent the hierarchy present in the proof, and includes the rule employed to establish that line of the proof:

cx is product-free, because (**PN6**)
  c is a constant expression(**T8**)
  x is product-free (**PN1**)

$\exp(y)$ is product-free, because (**PN3**)
y is product-free (**PN2**)
$\log(a\sqrt{x}+b)$ is product-free, because (**PN3**)
$a\sqrt{x}+b$ is product-free, because (**PN4**)
$a\sqrt{x}$ is product-free, because (**PN6**)
a is a constant expression (**PN2**)
$\sqrt{x}$ is product-free, because (**PN3**)
x is product-free (**PN2**)
b is product-free (**PN2**,**T8**)
$ax+by$ is product-free, because(**PN4**)
ax is product-free, because (**PN6**)
a is a constant expression (**T8**)
x is product-free (**PN1**)
by is product-free, because (**PN6**)
b is a constant expression (**T8**)
y is product-free (**PN1**)
d is product-free (**PN2**)

The second stage proceeds by verifying that the top-level, sign, and composition rules in a similarly hierarchical fashion:

The minimization problem is valid if $a \geq 0$, because (**T1**)
The objective function is valid, because (**T1**)
cx is convex (**SN1**)
The first constraint is valid if $a \geq 0$, because (**T1**)
$\exp(y) \leq \log(a\sqrt{x}+b)$ is convex if $a \geq 0$, because (**T5**)
$\exp(y)$ is convex, because (**SN1**)
$\exp(y)$ is convex, because (**C1**)
$\exp(\cdot)$ is convex and nondecreasing (atom library)
y is convex (**SN1**)
$\log(a\sqrt{x}+b)$ is concave if $a \geq 0$, because (**SN2**)
$\log(a\sqrt{x}+b)$ is concave if $a \geq 0$, because (**C4**)
$\log(\cdot)$ is concave and nondecreasing (atom library)
$a\sqrt{x}+b$ is concave if $a \geq 0$, because (**SN2**)
$\sqrt{x}$ is concave, because (**C4**)
$\sqrt{\cdot}$ is concave and nondecreasing (atom library)
x is concave (**SN2**)
The second constraint is valid, because (**T1**)
$ax+by=d$ is convex, because (**T4**)
$ax+by$ is affine (**SN3**)
d is affine (**SN3**)

It can be quite helpful to examine the structure of the problem and its validity proof graphically. Figure 6 presents an *expression tree* of the problem (90), annotated with the relevant rules verified at each position in the tree.

The verification process is guaranteed to yield one of three conclusions:

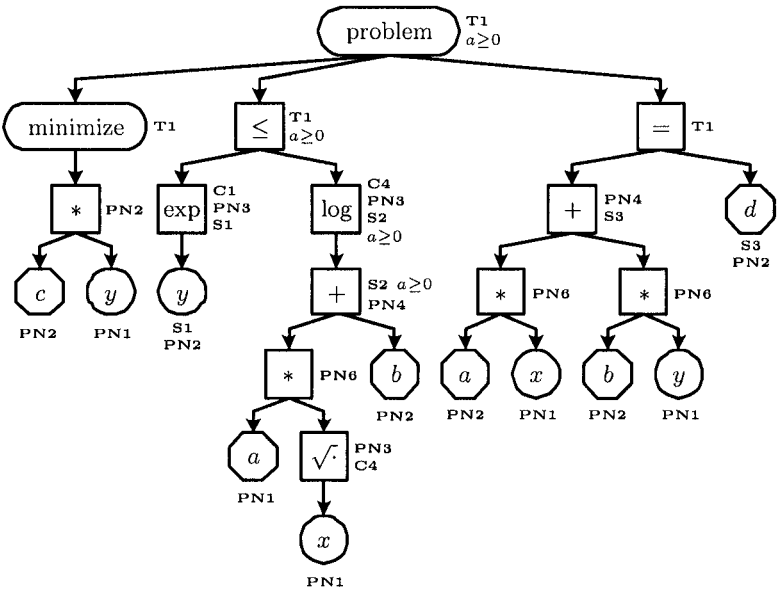

Fig. 6. An expression tree for (90), annotated with applicable convexity rules.

- *valid*: the rules are fully satisfied.
- *conditionally valid*: the rules will be fully satisfied if one or more additional preconditions on the parameters are satisfied.
- *invalid*: one or more of the convexity rules has been violated.

In this case, a conclusion of *conditionally valid* has been reached: the analysis has revealed that an additional condition $a \geq 0$ must satisfied. If this precondition were somehow assured, then the proof would have conclusively determined that the problem is a valid DCP. One simple way to accomplish this would be to add it as an assertion; *i.e.*, by adding the assertion `a >= 0` to the list of constraints. If, on the other hand, we were to do the opposite and add an assertion `a < 0`, the sign rule **SN2** would be violated; in fact, the expression $a\sqrt{x} + b$ would be verifiably convex.

The task of verifying DCPs comprises yet another approach to the challenge of automatic convexity verification described in §3.5. Like the methods used to verify LPs and SDPs, a certain amount of structure is assumed via the convexity rules that enables the verification process to proceed in a reliable and deterministic fashion. However, unlike these more limited methods, disciplined convex programming maintains generality by allowing new functions and sets to be added to the atom library. Thus disciplined convex programming provides a sort of *knowledgebase* environment for convex programming, in which human-supplied information about functions and sets is used to expand the body of problems that can be recognized as convex.

9 Creating disciplined convex programs

As mentioned previously, adherence to the convexity ruleset is sufficient but not necessary to insure convexity. It is possible to construct mathematical programs that are indeed convex, but which fail to be DCPs, because one or more of the expressions involved violates the convexity ruleset.

It is actually quite simple to construct examples of such violations. For example, consider the entropy maximization problem

$$\begin{array}{ll} \text{maximize} & -\sum_{i=1}^{n} x_i \log x_i \\ \text{subject to} & Ax = b \\ & \mathbf{1}^T x = 1 \\ & x \geq 0 \end{array} \tag{91}$$

where $x \in \mathbb{R}^n$ is the problem variable and $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ are parameters; and $\log(\cdot)$ is defined in the atom library. The expression $x_i \log x_i$ violates the product-free rule **PS6**—and as a result, (91) is not a DCP, even though it is a well-known CP.

Alternatively, consider the GP in convex form,

$$\begin{array}{ll} \text{minimize} & \log \sum_{k=1}^{K_0} \exp(a_{0k}^T x + b_{0k}) \\ \text{subject to} & \log \sum_{k=1}^{K_i} \exp(a_{ik}^T x + b_{ik}) \leq 0 \qquad i = 1, 2, \ldots, m \\ & A^{(m+1)} x + b^{(m+1)} = 0 \end{array} \tag{92}$$

where $x \in \mathbb{R}^n$ is the problem variable,

$$\begin{array}{l} A^{(i)} = \begin{bmatrix} a_{i1} & a_{i2} & \ldots & a_{im_i} \end{bmatrix}^T \in \mathbb{R}^m n m_i n \\ b^{(i)} = \begin{bmatrix} b_{i1} & b_{i2} & \ldots & b_{im_i} \end{bmatrix}^T \in \mathbb{R}^{m_i} \end{array} \qquad i = 1, 2, \ldots, m+1 \tag{93}$$

are parameters; and both $\log(\cdot)$ and $\exp(\cdot)$ are defined in the atom library. This problem satisfies the product-free rules, but the objective function and inequality constraints fail either the sign rule **SN1** or composition rule **C4**, depending on how you verify them. But of course, (92) is a CP.

It is important to note that these violations do not mean that the problems cannot solved in the disciplined convex programming framework; it simply means that they must be rewritten in a compliant manner. In both of these cases, the simplest way to do so is to add new functions to the atom library that encapsulate the offending nonlinearities. By adding the two functions

$$f_{\text{entr}}(x) = \begin{cases} -x \log x & x > 0 \\ 0 & x = 0 \\ -\infty & x < 0 \end{cases} \qquad f_{\text{lse}}(x) = \log \sum_{i=1}^{n} e^x \tag{94}$$

to the atom library, both problems can be rewritten as valid DCPs; (91) as

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^{n} f_{\text{entr}}(x_i) \\ \text{subject to} & Ax = b \\ & \mathbf{1}^T x = 1 \\ & x \geq 0 \end{array} \tag{95}$$

and the GP (92) as

$$\begin{array}{ll} \text{minimize} & f_{\text{lse}}(A^{(0)}x + b^{(0)}) \\ \text{subject to} & f_{\text{lse}}(A^{(i)}x + b^{(i)}) \leq 0, \quad i = 1, 2, \ldots, m \\ & A^{(m+1)}x + b^{(m+1)} = 0 \end{array} \tag{96}$$

The ability to extend the atom library as needed has the potential to be taken to an inelegant extreme. For example, consider the problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_m(x) \leq 0, \quad k = 1, 2, \ldots, m \end{array} \tag{97}$$

where the functions $f_0, f_1, \ldots, f_m$ are all convex. One way to cast this problem as a DCP would simply be to add all $m + 1$ of the functions to the atom library. The convexity rules would then be satisfied rather trivially; and yet this would likely require *more*, not *less*, effort than a more traditional NLP modeling method. In practice, however, the functions f_k are rarely monolithic, opaque objects. Rather, they will be constructed from components such as affine forms, norms, and other known functions, combined in ways consistent with the basic principles of convex analysis, as captured in the convexity ruleset. It is *those* functions that are ideal candidates for addition into the atom library.

We should add that once an atom is defined and implemented, it can be freely reused across many DCPs. The atoms can be shared with other users as well. The effort involved in adding a new function to the atom library, then, is significantly amortized. A collaborative hierarchy is naturally suggested, wherein more advanced users can create new atoms for application-specific purposes, while novice users can take them and employ them in their models without regard for how they were constructed.

We argue, therefore, that (91) and (92) are ideal examples of the kinds of problems that disciplined convex programming is intended to support, so that the convexity ruleset poses little practical burden in these cases. While it is true that the term $-x_i \log x_i$ violates the product-free rules, someone interested in entropy maximization does not consider this expression as a product of nonlinearities but rather as a single, encapsulated nonlinearity—as represented by the function f_{expr}. In a similar manner, those studying geometric programming treat the function $f_{\text{expr}}(y) = \log \sum \exp(y_i)$ as a monolithic convex function; it is irrelevant that it happens to be the composition of a concave function and a convex function. Thus the addition of these functions to the atom library coincides with the intuitive understanding of the problems that employ them.

Still, the purity of the convexity rules prevent even the use of obviously convex quadratic forms such as $x^2 + 2xy + y^2$ in a model. It could be argued that this is impractically restrictive, since quadratic forms are so common. And indeed, we are considering extending the relaxation of the product-free rules to include quadratic forms. However, in many cases, a generic quadratic form may in fact represent a quantity with more structure or meaning. For example, traditionally, the square of a Euclidean norm $\|Ax + b\|_2^2$ would be converted to a quadratic form

$$\|Ax + b\|_2 = x^T P x + q^T x + r, \quad P \triangleq A^T A, \quad q \triangleq A^T b, \quad r \triangleq b^T b \tag{98}$$

But within a DCP, this term can instead be expressed as a composition

$$\|Ax + b\|_2 = f(g(Ax + b)), \quad f(y) \triangleq y^2, \quad g(z) \triangleq \|z\|_2 \tag{99}$$

In disciplined convex programming, there is no natural bias against (99), so it should be preferred over the converted form (98) simply because it reflects the original intent of the problem. So we argue that support for generic quadratic forms would be at least somewhat less useful than in a more traditional modeling framework. Furthermore, we can easily support quadratic forms with the judicious addition of functions to the atom library, such as the function $f(y) = y^2$ above, or a more complex quadratic form such as

$$f_Q : \mathbb{R}^n \to \mathbb{R}, \quad f_Q(x) = x^T Q x \qquad (Q \succeq 0). \tag{100}$$

Thus support for quadratic forms is a matter of convenience, not necessity.

10 Implementing atoms

As enumerated in §3.2, there are a variety of methods that can be employed to solve CPs: primal/dual methods, barrier methods, cutting-plane methods, and so forth. All of these methods can be adapted to disciplined convex programming with minimal effort. Limited space prohibits us from examining these methods in detail; please see [74] for a more thorough treatment of the topic. It is sufficient here to say this: that each of these methods will need to perform certain computations involving each of the atoms, each of the functions and sets, employed in the problems they solve. The purpose of the *implementation* of an atom, first introduced in §7 above, is to provide these solvers with the means to perform these calculations.

Disciplined convex programming and the `cvx` modeling framework distinguish between two different types of implementations:

- a *simple* implementation, which provides traditional calculations such as derivatives, subgradients and supergradients for functions; and indicator functions, barrier functions, and cutting planes for sets; and

- a *graph* implementation, in which the function or set is defined *as the solution to another DCP.*

The computations supported by simple implementations should be quite familiar to anyone who studies the numerical solution of optimization problems. To the best of our knowledge, however, the concept of graph implementations is new, and proves to be an important part of the power and expressiveness of disciplined convex programming.

In `cvx`, an implementation is surrounded by curly braces, and consists of a list of key/value pairs with the syntax *key* `:=` *value*. See Figures 7 and 8 for examples. It is also possible for an implementation to be constructed in a lower-level language like C, but we will not consider that feature here.

10.1 Simple implementations

Any continuous function can have a simple implementation. Simple function implementations use the following *key* `:=` *value* entries:

- `value`: the value of the function.
- `domain_point`: a point on the interior of the domain of the function. If omitted, then the origin is assumed to be in the domain of the function.
- For differentiable functions:
  - `gradient`: the first derivative.
  - `Hessian` (if twice differentiable): the second derivative.
- For nondifferentiable functions:
  - `subgradient` (if convex): a subgradient of a function f at point $x \in \mathbf{dom}\, f$ is any vector $v \in \mathbb{R}^n$ satisfying

$$f(y) \geq f(x) + v^T(y - x) \quad \forall y \in \mathbb{R}^n \tag{101}$$

  - `supergradient` (if concave): a supergradient of a function g at point $x \in \mathbf{dom}\, g$ is any vector $v \in \mathbb{R}^n$ satisfying

$$g(y) \leq g(x) + v^T(y - x) \quad \forall y \in \mathbb{R}^n \tag{102}$$

It is not difficult to see how different algorithms might utilize this information. Most every method would use `value` and `domain_point`, for example. A smooth CP method would depend on the entries `gradient` and `Hessian` to calculate Newton search directions. A localization method would use the entries `gradient`, `subgradient`, and `supergradient` to compute cutting planes.

Any set with a non-empty interior can have a simple implementation. Simple set implementations use the following *key* `:=` *value* entries:

- `interior_point`: a point on the interior of the set.
- `indicator`: an expression that is 0 for points inside the set, and $+\infty$ for points outside the set.
- At least one, but ideally both, of the following:

- `barrier`: a reference to a convex, twice differentiable *barrier function* for the set, declared separately as a function atom with a direct implementation.
- `oracle`: a cutting plane oracle for the set. Given a set $S \subset \mathbb{R}^n$, the cutting plane oracle accepts as input a point $x \in \mathbb{R}^n$; and, if $x \notin S$, returns a separating hyperplane; that is, a pair $(a, b) \in \mathbb{R}^n \times \mathbb{R}$ satisfying

$$a^T x > b, \quad S \subseteq \{ y \mid a^T y \leq b \} \tag{103}$$

If $x \in S$, then the oracle returns $(a, b) = (0, 0)$.

```
function min( x, y ) concave, nondecreasing {
    value := x < y ? x : y;
    supergradient := x < y ? ( 1, 0 ) : ( 0, 1 );
}
set pos( x ) convex {
    interior_point := 1.0;
    indicator := x < 0 ? +Inf : 0;
    oracle := x < 0 ? ( 1, 0 ) : ( 0, 0 );
    barrier := neglog( x );
}
function neglog( x ) convex {
    domain_point := 1.0;
    value := x <= 0 ? +Inf : - log( x );
    gradient := - 1 / x;
    Hessian := 1 / x^2;
}
```

Fig. 7. Simple implementations.

Figure 7 presents several examples of simple implementations. Again, we do not wish to document the `cvx` syntax here, only illustrate key the feasibility of this approach. Note that the set `pos` has been given both a barrier function and a cutting plane generator, allowing it to be used in both types of algorithms.

10.2 Graph implementations

A fundamental principle in convex analysis is the very close relationship between convex and concave functions and convex sets. A function $f : \mathbb{R}^n \to \mathbb{R} \cup +\infty$ is convex if and only if its *epigraph*

$$F = \operatorname{epi} f = \{ (x, y) \in \mathbb{R}^n \times \mathbb{R} \mid f(x) \leq y \} \tag{104}$$

is a convex set. Likewise, a function $g : \mathbb{R}^n \to \mathbb{R} \cup +\infty$ is concave if and only if its *hypograph*

$$G = \operatorname{hypo} g = \{\, (x,y) \in \mathbb{R}^n \times \mathbb{R} \mid g(x) \geq y \,\} \tag{105}$$

is a convex set. These relationships can be expressed in reverse fashion as well:

$$f(x) = \inf \{\, y \mid (x,y) \in F \,\} \tag{106}$$

$$g(x) = \sup \{\, y \mid (x,y) \in G \,\} \tag{107}$$

A *graph implementation* of a function is effectively a representation of the epigraph or hypograph of a function, as appropriate, as a disciplined convex feasibility problem. The cvx framework supports this approach using the following *key* := *value* pairs:

- `epigraph` (if convex): the epigraph of the function.
- `hypograph` (if concave): the hypograph of the function.

A simple example is the absolute value function $f(x) = |x|$. The epigraph of this function is

$$\operatorname{epi} f = \{\, (x,y) \mid |x| \leq y \,\} = \{\, (x,y) \mid -y \leq x \leq y \,\} \tag{108}$$

In Figure 8, we show how this epigraph is represented in cvx. Notice that the

```
function abs( x ) convex, >= 0 {
    value := x < 0 ? -x : x;
    epigraph := { -abs <= x <= +abs; }
}
function min( x, y ) concave, nondecreasing {
    value := x < y ? x : y;
    supergradient := x < y ? ( 1, 0 ) : ( 0, 1 );
    hypograph := { min <= x; min <= y; }
}
function entropy( x ) concave {
    value := x < 0 ? -Inf : x = 0 ? 0 : - x log( x );
    hypograph := { ( x, y ) in hypo_entropy; }
}
set simplex( x[n] ) convex {
    constraints := { sum( x ) = 1; x >= 0; }
}
```

Fig. 8. Graph implementations.

name of the function, `abs` is used to represent the epigraph variable.

The primary benefit of graph implementations is that they provide an elegant means to define nondifferentiable functions. The absolute value function above is one such example; another is the two-argument minimum $g(x,y) = \min\{x,y\}$. This function is concave, and its hypograph is

$$\operatorname{hypo} g = \{\, (x,y,z) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R} \mid z \leq x,\ z \leq y \,\} \tag{109}$$

Figure 8 shows how the function `min` represents this hypograph in `cvx`. Notice that this function has a simple implementation as well—allowing the underlying solver to decide which it prefers to use. Of course, both `abs` and `min` are rather simple, but far more complex functions are possible. For example, graph implementations can be constructed for each of the norms examined in §2.

More subtle but important instances of nondifferentiability occur in functions that are discontinuous at the boundaries of their domains. These functions require special care as well, and graph implementations provide that. For example, consider the scalar entropy function

$$f : \mathbb{R} \to \mathbb{R}, \quad f(x) \triangleq \begin{cases} -x \log x & x > 0 \\ 0 & x = 0 \\ -\infty & x < 0 \end{cases} \tag{110}$$

This function is smooth over the positive interval, but it is discontinuous at the origin, and its derivative is unbounded near the origin. Both of these features cause problems for some numerical methods [93]. Using the hypograph

$$\operatorname{hypo} f = \mathbf{cl}\,\{\,(x,y) \in \mathbb{R} \times \mathbb{R} \mid x > 0,\ -x \log x > y\,\} \tag{111}$$

can solve these problems. In Figure 8, we show the definition of a function `entropy` that refers to a set `hypo_entropy` representing this hypograph. We have chosen to omit the implementation of this set here, but it would certainly contain a definition of the barrier function

$$\begin{aligned} &\phi : \mathbb{R} \times \mathbb{R} \to (\mathbb{R} \cup +\infty), \\ &\phi(x,y) = \begin{cases} -\log(-y - x \log x) - \log x & (x,y) \in \operatorname{Int} \operatorname{epi} f \\ +\infty & \text{otherwise} \end{cases} \end{aligned} \tag{112}$$

[118], as well as an oracle to compute cutting planes $a_1 x + a_2 y \leq b$, where

$$(a_1, a_2, b) \triangleq \begin{cases} (0,0,0) & (x,y) \in \operatorname{hypo} f \\ (-1,0,0) & x < 0 \\ (\log(y/2) + 1, 1, y/2) & x = 0,\ y > 0 \\ (\log x + 1, 1, x) & x > 0 \end{cases} \tag{113}$$

Graph implementations can also be used to unify traditional, inequality-based nonlinear programming with conic programming. For example, consider the maximum singular value function

$$f : \mathbb{R}^{m \times n} \to \mathbb{R}, \quad f(X) = \sigma_{\max}(X) = \sqrt{\lambda_{\max}(X^T X)} \tag{114}$$

This function is convex, and in theory could be used in a disciplined convex programming framework. The epigraph of f is

$$\begin{aligned}\operatorname{epi} f &= \left\{ (X,y) \in \mathbb{R}^{m\times n} \times \mathbb{R} \mid \sigma_{\max}(X) \leq y \right\} \\ &= \left\{ (X,y) \;\middle|\; \begin{bmatrix} yI & X \\ X^T & yI \end{bmatrix} \in \mathcal{S}^n_+,\ y \geq 0 \right\}\end{aligned} \tag{115}$$

where $\mathcal{S}^n_+$ is the set of positive semidefinite matrices. For someone who is familiar with semidefinite programming, (115) is likely quite familiar. By burying this construct within the implementation in the atom library, however, it enables people who are *not* comfortable with semidefinite programming to take advantage of its benefits in a traditional NLP-style problem.

Graph implementations are possible for sets as well, through the use of a a single *key* := *value* pair:

- `constraints`: a list of constraints representing the set.

What this means is that a set can be described in terms of a disciplined convex feasibility problem. There are several reasons why this might be used. For example, graph implementations can be used to represent sets with non-empty interiors, such as the set of n-element probability distributions

$$S = \left\{ x \in \mathbb{R}^n \mid x \geq 0,\ \mathbf{1}^T x = 1 \right\} \tag{116}$$

A `cvx` version of this set is given in Figure 8. Graph implementations can also be used to represent sets using a sequence of smooth inequalities so that smooth CP solvers can support them. For example, the second-order cone

$$\mathcal{Q}^n = \left\{ (x,y) \in \mathbb{R}^n \times \mathbb{R} \mid \|x\|_2 \leq y \right\} \tag{117}$$

can be represented by smooth inequalities as follows:

$$\mathcal{Q}^n = \left\{ (x,y) \in \mathbb{R}^n \times \mathbb{R} \mid x^T x/y - y \leq 0,\ y \geq 0 \right\} \tag{118}$$

10.3 Using graph implementations

To solve a DCP involving functions or sets with graph implementations, those transformations must be applied through a process we call *graph expansion*, in which the DCP that describes a given atom is incorporated into the problem. To illustrate what this entails, consider the problem

$$\begin{aligned} &\text{maximize } \min\{c_1^T x + d_1, c_2^T x + d_2\} \\ &\text{subject to } Ax = b \\ &\qquad\qquad x \geq 0 \end{aligned} \tag{119}$$

employing the function $\min\{\cdot,\cdot\}$. The hypograph of this function, presented in (109) above, allows this problem to be rewritten as

$$\begin{aligned} &\text{maximize } \sup \left\{ y \mid y \leq c_1^T x + d_1,\ y \leq c_2^T x + d_2 \right\} \\ &\text{subject to } Ax = b \\ &\qquad\qquad x \geq 0 \end{aligned} \tag{120}$$

Incorporating the variable y into the model itself yields expanded result

$$\begin{array}{ll} \text{maximize} & y \\ \text{subject to} & y \leq c_1^T x + d_1 \\ & y \leq c_2^T x + d_2 \\ & Ax = b \\ & x \geq 0 \end{array} \tag{121}$$

It is not difficult to see that this problem is equivalent to the original, and yet now it is a simple LP.

Thus, as stated in §2.3, cvx allows the transformations required to convert DCPs into solvable form to be encapsulated—and graph implementations are how this is accomplished. Indeed, consider once again the ℓ_∞, ℓ_1, ℓ_p, and largest-L minimization problems described in §2. The transformations used to solve those problems in that section are, in fact, the very transformations that cvx would use to solve them as well (or at least, very nearly so).

Because graph implementations are expanded before a numerical algorithm is deployed, they require no adjustment on the part of those algorithms to support them. Thus graph implementations are *algorithm agnostic*: any algorithm which can successfully support simply implemented functions and sets—by computing derivatives, sub/supergradients, barrier functions, *etc.*—can solve problems with functions and sets with graph implementations as well. Put another way, algorithms which previously could not support nondifferentiable functions are enabled to do so through cvx.

The concept of graph implementations is based on relatively basic principles of convex analysis; and yet, an applications-oriented user—someone who is not expert in convex optimization—is not likely to be constructing new atoms with graph implementations. Indeed they are not likely to be constructing new simple implementations either. Thankfully, they do not need to *build* them to *use* them. The implementations themselves can be built by those with more expertise in such details, and shared with applications-oriented users. As the development of the cvx framework continues, the authors will build a library of common and useful functions; and we hope that others will do so and share them with the community of users.

11 Conclusion

In this article, we have introduced a new methodology for convex optimization called *disciplined convex programming*. Disciplined convex programming simplifies the specification, analysis, and solution of convex programs by imposing certain restrictions on their construction. These restrictions are simple and teachable; they are inspired by the basic principles of convex analysis; and they formalize the intuitive practices of many who use convex optimization today. Despite the restrictions, generality is preserved through the expandability of the atom library.

We have enumerated a number of the benefits that disciplined convex programing obtains for practical convex optimization. Verifying that a model is a valid DCP is a straightforward and reliable process. Nondifferentiable functions may be freely employed without fear of sacrificing numerical performance. And while we did not explore in detail how DCPs are solved, we did discussed how the implementation of functions and sets enables a variety of numerical methods to be used—methods whose performance and reliability are well-known. We refer the reader to [74] for more development on this topic.

An overarching goal of the development of disciplined convex programming is *unification*. There are no less than *seven* standard forms for convex programming being studied and used today: LS, LP, QP, SDP, SOCP, GP, and smooth CP. Deciding which form best suits a given application is not always obvious; and for many problems, a custom solver is the only appropriate choice. Unification allows modelers to freely consider all of these problem types simultaneously—because they need not think of them as separate types at all.

The principles of disciplined convex programming have been implemented in the `cvx` modeling framework. The current version employs a simple barrier solver, but we intend to develop a more powerful solver in the future, and we hope to convince other developers to provide a link to `cvx` for their own solvers. We will be disseminating `cvx` freely with BSD-like licensing, and it is our hope that it will be used widely in coursework, research, and applications.

The reader is invited to visit the Web site `http://www.stanford.edu/~boyd/cvx` to monitor the development of `cvx`, to download the latest versions, and to read the accompanying documentation.

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.
2. W. Achtziger, M. Bendsoe, A. Ben-Tal, and J. Zowe. Equivalent displacement based formulations for maximum strength truss topology design. *Impact of Computing in Science and Engineering*, 4(4):315–45, December 1992.
3. M. Avriel, R. Dembo, and U. Passy. Solution of generalized geometric programs. *International Journal for Numerical Methods in Engineering*, 9:149–168, 1975.
4. M. Abdi, H. El Nahas, A. Jard, and E. Moulines. Semidefinite positive relaxation of the maximum-likelihood criterion applied to multiuser detection in a CDMA context. *IEEE Signal Processing Letters*, 9(6):165–167, June 2002.
5. F. Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization*, 5(1):13–51, February 1995.

6. B. Alkire and L. Vandenberghe. Convex optimization problems involving finite autocorrelation sequences. *Mathematical Programming*, Series A, 93:331–359, 2002.
7. E. Andersen and Y. Ye. On a homogeneous algorithm for the monotone complementarity problem. *Mathematical Programming*, 84:375–400, 1999.
8. S. Boyd and C. Barratt. *Linear Controller Design: Limits of Performance.* Prentice-Hall, 1991.
9. M. Bendsoe, A. Ben-Tal, and J. Zowe. Optimization methods for truss geometry and topology design. *Structural Optimization*, 7:141–159, 1994.
10. C. Bischof, A. Carle, G. Corliss, A. Grienwank, and P. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, pages 1–29, December 1991.
11. S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan. *Linear Matrix Inequalities in System and Control Theory.* SIAM, 1994.
12. S. Benson. DSDP 4.5: A daul scaling algorithm for semidefinite programming. Web site: `http://www-unix.mcs.anl.gov/~benson/dsdp/`, March 2002.
13. D. Bertsekas. *Nonlinear Programming.* Athena Scientific, Belmont, Massachusetts, 1995.
14. R. Byrd, N. Gould, J. Norcedal, and R. Waltz. An active-set algorithm for nonlinear programming using linear programming and equality constrained subproblems. Technical Report OTC 2002/4, Optimization Technology Center, Northwestern University, October 2002.
15. O. Bahn, J. Goffin, J. Vial, and O. Du Merle. Implementation and behavior of an interior point cutting plane algorithm for convex programming: An application to geometric programming. Working Paper, University of Geneva, Geneva, Switzerland, 1991.
16. S. Boyd, M. Hershenson, and T. Lee. Optimal analog circuit design via geometric programming, 1997. Preliminary Patent Filing, Stanford Docket S97-122.
17. R. Banavar and A. Kalele. A mixed norm performance measure for the design of multirate filterbanks. *IEEE Transactions on Signal Processing*, 49(2):354–359, February 2001.
18. A. Brooke, D. Kendrick, A. Meeraus, and R. Raman. *GAMS: A User's Guide.* The Scientific Press, South San Francisco, 1998. Web site: `http://www.gams.com/docs/gams/GAMSUsersGuide.pdf`.
19. J. Borwein and A. Lewis. Duality relationships for entropy-like minimization problems. *SIAM J. Control and Optimization*, 29(2):325–338, March 1991.
20. D. Bertsimas and J. Nino-Mora. Optimization of multiclass queuing networks with changeover times via the achievable region approach: part ii, the multistation case. *Mathematics of Operations Research*, 24(2), May 1999.
21. D. Bertsekas, A. Nedic, and A. Ozdaglar. *Convex Analysis and Optimization.* Athena Scientific, Nashua, New Hampshire, 2004.
22. B. Borchers. CDSP, a C library for semidefinite programming. *Optimization Methods and Software*, 11:613–623, 1999.
23. A. Ben-Tal and M. Bendsoe. A new method for optimal truss topology design. *SIAM J. Optim.*, 13(2), 1993.
24. A. Ben-Tal, M. Kocvara, A. Nemirovski, and J. Zowe. Free material optimization via semidefinite programming: the multiload case with contact conditions. *SIAM Review*, 42(4):695–715, 2000.

25. A. Ben-Tal and A. Nemirovski. Interior point polynomial time method for truss topology design. *SIAM Journal on Optimization*, 4(3):596–612, August 1994.
26. A. Ben-Tal and A. Nemirovski. Robust truss topology design via semidefinite programming. *SIAM J. Optim.*, 7(4):991–1016, 1997.
27. A. Ben-Tal and A. Nemirovski. Structural design via semidefinite programming. In *Handbook on Semidefinite Programming*, pages 443–467. Kluwer, Boston, 2000.
28. S. Boyd and L. Vandenberghe. Semidefinite programming relaxations of nonconvex problems in control and combinatorial optimization. In A. Paulraj, V. Roychowdhuri, , and C. Schaper, editors, *Communications, Computation, Control and Signal Processing: a Tribute to Thomas Kailath*, chapter 15, pages 279–288. Kluwer Academic Publishers, 1997.
29. S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.
30. P. Biswas and Y. Ye. Semidefinite programming for ad hoc wireless sensor network localization. Technical report, Stanford University, April 2004. Web site: `http://www.stanford.edu/~yyye/adhocn4.pdf`.
31. A. Conn, N. Gould, D. Orban, and Ph. Toint. A primal-dual trust-region algorithm for non-convex nonlinear programming. *Mathematical Programming*, 87:215–249, 2000.
32. A. Conn, N. Gould, and Ph. Toint. *LANCELOT: a Fortran Package for Large-Scale Nonlinear Optimization (Release A)*, volume 17 of *Springer Series in Computational Mathematics.* Springer Verlag, 1992.
33. A. Conn, N. Gould, and Ph. Toint. *Trust-Region Methods.* Series on Optimization. SIAM/MPS, Philadelphia, 2000.
34. J. Chinneck. MProbe 5.0 (software package). Web site: `http://www.sce.carleton.ca/faculty/chinneck/mprobe.html`, December 2003.
35. G. Calafiore and M. Indri. Robust calibration and control of robotic manipulators. In *American Control Conference*, pages 2003–2007, 2000.
36. C. Crusius. *A parser/solver for convex optimization problems.* PhD thesis, Stanford University, 2002.
37. T. Terlaky C. Roos and J.-Ph. Vial. *Interior Point Approach to Linear Optimization: Theory and Algorithms.* John Wiley & Sons, New York, NY, 1997.
38. G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.
39. J. Dawson, S. Boyd, M. Hershenson, and T. Lee. Optimal allocation of local feedback in multistage amplifiers via geometric programming. *IEEE Journal of Circuits and Systems I*, 48(1):1–11, January 2001.
40. M. Dahleh and I. Diaz-Bobillo. *Control of Uncertain Systems. A Linear Programming Approach.* Prentice Hall, 1995.
41. S. Dirkse and M. Ferris. The PATH solver: A non-monotone stabilzation scheme for mixed complementarity problems. *Optimization Methods and Software*, 5:123–156, 1995.
42. Y. Doids, V. Guruswami, and S. Khanna. The 2-catalog segmentation problem. In *Proceedings of SODA*, pages 378–380, 1999.
43. T. Davidson, Z. Luo, and K. Wong. Design of orthogonal pulse shapes for communications via semidefinite programming. *IEEE Transactions on Communications*, 48(5):1433–1445, May 2000.

44. G. Dullerud and F. Paganini. *A Course in Robust Control Theory*, volume 36 of *Texts in Applied Mathematics*. Springer-Verlag, 2000.
45. C. de Souza, R. Palhares, and P. Peres. Robust H_∞ filter design for uncertain linear systems with multiple time-varying state delays. *IEEE Transactions on Signal Processing*, 49(3):569–575, March 2001.
46. A. Doherty, P. Parrilo, and F. Spedalieri. Distinguishing separable and entangled states. *Physical Review Letters*, 88(18), 2002.
47. B. Dumitrescu, I. Tabus, and P. Stoica. On the parameterization of positive real sequences and MA parameter estimation. *IEEE Transactions on Signal Processing*, 49(11):2630–2639, November 2001.
48. R. Duffin. Linearizing geometric programs. *SIAM Review*, 12:211–227, 1970.
49. C. Du, L. Xie, and Y. Soh. H_∞ filtering of 2-D discrete systems. *IEEE Transactions on Signal Processing*, 48(6):1760–1768, June 2000.
50. H. Du, L. Xie, and Y. Soh. H_∞ reduced-order approximation of 2-D digital filters. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 48(6):688–698, June 2001.
51. Laurent El Ghaoui, Jean-Luc Commeau, Francois Delebecque, and Ramine Nikoukhah. LMITOOL 2.1 (software package). Web site: `http://robotics.eecs.berkeley.edu/~elghaoui/lmitool/lmitool.html`, March 1999.
52. J. Ecker. Geometric programming: methods, computations and applications. *SIAM Rev.*, 22(3):338–362, 1980.
53. J.-P. A. Haeberly F. Alizadeh and M. Overton. Primal-dual interior-point methods for semidefinite programming: Convergence rates, stability and numerical results. *SIAM J. Optimization*, 8:46–76, 1998.
54. M. Fu, C. de Souza, and Z. Luo. Finite-horizon robust Kalman filter design. *IEEE Transactions on Signal Processing*, 49(9):2103–2112, September 2001.
55. U. Feige and M. Goemans. Approximating the value of two prover proof systems, with applications to max 2sat and max dicut. In *Proceedings of the 3nd Israel Symposium on Theory and Computing Systems*, pages 182–189, 1995.
56. R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, December 1999.
57. A. Frieze and M. Jerrum. Improved approximation algorithms for max k-cut and max bisection. *Algorithmica*, 18:67–81, 1997.
58. K. Fujisawa, M. Kojima, K. Nakata, and M. Yamashita. SDPA (Semi-Definite Programming Algorithm) user's manual—version 6.00. Technical report, Tokyo Insitute of Technology, July 2002.
59. U. Feige and M. Langberg. Approximation algorithms for maximization problems arising in graph partitioning. *Journal of Algorithms*, 41:174–211, 2001.
60. U. Feige and M. Langberg. The rpr^2 rounding technique for semidefinte programs. In *ICALP*, Lecture Notes in Computer Science. Springer, Berlin, 2001.
61. R. Fourer. Nonlinear programming frequently asked questions. Web site: `http://www-unix.mcs.anl.gov/otc/Guide/faq/nonlinear-programming%-faq.html`, 2000.
62. R. Freund. Polynomial-time algorithms for linear programming based only on primal scaling and projected gradients of a potential function. *Mathematical Programming*, 51:203–222, 1991.
63. Frontline Systems, Inc. Premium Solver Platform (software package). Web site: `http://www.solver.com`, September 2004.
64. E. Fridman and U. Shaked. A new H_∞ filter design for linear time delay systems. *IEEE Transactions on Signal Processing*, 49(11):2839–2843, July 2001.

65. J. Geromel. Optimal linear filtering under parameter uncertainty. *IEEE Transactions on Signal Processing*, 47(1):168–175, January 1999.
66. O. Güler and R. Hauser. Self-scaled barrier functions on symmetric cones and their classification. *Foundations of Computational Mathematics*, 2:121–143, 2002.
67. D. Goldfarb and G. Iyengar. Robust portfolio selection problems. Technical report, Computational Optimization Research Center, Columbia University, March 2002. Web site: `http://www.corc.ieor.columbia.edu/reports/techreports/tr-2002-03.pdf`.
68. D. Goldfarb and G. Iyengar. Robust quadratically constrained problems program. Technical Report TR-2002-04, Department of IEOR, Columbia University, New York, NY USA, 2002.
69. P. Gill, W. Murray, and M. Saunders. SNOPT: An sqp algorithm for large-scale constrained optimization. *SIAM Journal on Optimization*, 12:979–1006, 2002.
70. P. Gill, W. Murray, M. Saunders, and M. Wright. User's guide for NPSOL 5.0: A FORTRAN package for nonlinear programming. Technical Report SOL 86-1, Systems Optimization Laboratory, Stanford University, July 1998. Web site: `http://www.sbsi-sol-optimize.com/manuals/NPSOL%205-0%20Manual.p%df`.
71. P. Gill, W. Murray, and M. Wright. *Practical Optimization*. Academic Press, London, 1981.
72. J. Geromel and M. De Oliveira. H_2/H_∞ robust filtering for convex bounded uncertain systems. *IEEE Transactions on Automatic Control*, 46(1):100–107, January 2001.
73. C. Gonzaga. Path following methods for linear programming. *SIAM Review*, 34(2):167–227, 1992.
74. M. Grant. *Disciplined Convex Programming*. PhD thesis, Department of Electrical Engineering, Stanford University, December 2004.
75. D. Guo, L. Rasmussen, S. Sun, and T. Lim. A matrix-algebraic approach to linear parallel interference cancellation in CDMA. *IEEE Transactions on Communications*, 48(1):152–161, January 2000.
76. G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins Univ. Press, Baltimore, second edition, 1989.
77. M. Goemans and D. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42:1115–1145, 1995.
78. M. Hershenson, S. Boyd, and T. Lee. Optimal design of a CMOS op-amp via geometric programming. *IEEE Transactions on Computer-Aided Design*, January 2001.
79. L. Huaizhong and M. Fu. A linear matrix inequality approach to robust H_∞ filtering. *IEEE Transactions on Signal Processing*, 45(9):2338–2350, September 1997.
80. M. Hershenson, S. Mohan, S. Boyd, and T. Lee. Optimization of inductor circuits via geometric programming. In *Proceedings 36th IEEE/ACM Integrated Circuit Design Automation Conference*, 1999.
81. P. Hovland, B. Norris, and C. Bischof. ADIC (software package), November 2003. `http://www-fp.mcs.anl.gov/adic/`.

82. L. Han, J. Trinkle, and Z. Li. Grasp analysis as linear matrix inequality problems. *IEEE Transactions on Robotics and Automation*, 16(6):663–674, December 2000.
83. J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms I*, volume 305 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, New York, 1993.
84. J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms II: Advanced Theory and Bundle Methods*, volume 306 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, New York, 1993.
85. Q. Han, Y. Ye, and J. Zhang. An improved rounding method and semidefinite programming relaxation for graph partition. *Math. Programming*, 92:509–535, 2002.
86. F. Jarre, M. Kocvara, and J. Zowe. Optimal truss design by interior point methods. *SIAM J. Optim.*, 8(4):1084–1107, 1998.
87. F. Jarre and M. Saunders. A practical interior-point method for convex programming. *SIAM Journal on Optimization*, 5:149–171, 1995.
88. N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.
89. M. Kojima, S. Mizuno, and A. Yoshise. An $O(\sqrt{n}L)$-iteration potential reduction algorithm for linear complementarity problems. *Mathematical Programming*, 50:331–342, 1991.
90. J. Kleinberg, C. Papadimitriou, and P. Raghavan. Segmentation problems. In *Proceedings of the 30th Symposium on Theory of Computation*, pages 473–482, 1998.
91. J. Keuchel, C. Schnörr, C. Schellewald, and D. Cremers. Binary partitioning, perceptual grouping, and restoration with semidefinite programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(11):1364–1379, November 2003.
92. K. Kortanek, X. Xu, and Y. Ye. An infeasible interior-point algorithm for solving primal and dual geometric progams. *Mathematical Programming*, 1:155–181, 1997.
93. K. Kortanek, X. Xu, and Y. Ye. An infeasible interior-point algorithm for solving primal and dual geometric programs. *Mathematical Programming*, 76:155–182, 1997.
94. M. Kocvara, J. Zowe, and A. Nemirovski. Cascading—an approach to robust material optimization. *Computers and Structures*, 76:431–442, 2000.
95. J. Lasserre. Global optimization with polynomials and the problem of moments. *SIAM Journal of Optimization*, 11:796–817, 2001.
96. J. Lasserre. Bounds on measures satisfying moment conditions. *Annals of Applied Probability*, 12:1114–1137, 2002.
97. J. Lasserre. Semidefinite programming vs. LP relaxation for polynomial programming. *Mathematics of Operations Research*, 27(2):347–360, May 2002.
98. H. Lebret and S. Boyd. Antenna array pattern synthesis via convex optimization. *IEEE Transactions on Signal Processing*, 45(3):526–532, March 1997.
99. Lindo Systems, Inc. LINGO version 8.0 (software package). Web site: `http://www.lindo.com`, September 2004.
100. J. Löfberg. YALMIP verison 2.1 (software package). Web site: `http://www.control.isy.liu.se/~johanl/yalmip.html`, September 2001.

101. L. Lovasz. *An Algorithmic Theory of Numbers, Graphs and Convexity*, volume 50 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 1986.
102. Z.-Q. Luo, J. Sturm, and S. Zhang. Duality and self-duality for conic convex programming. Technical report, Department of Electrical and Computer Engineering, McMaster University, 1996.
103. W. Lu. A unified approach for the design of 2-D digital filters via semidefinite programming. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 49(6):814–826, June 2002.
104. M. Lobo, L. Vandenberghe, S. Boyd, and H. Lebret. Applications of second-order cone programming. *Linear Algebra and its Applications*, 284:193–228, November 1998. Special issue on Signals and Image Processing.
105. R. Monteiro and I. Adler. Interior path following primal-dual algorithms: Part I: Linear programming. *Mathematical Programming*, 44:27–41, 1989.
106. The MathWorks, Inc. *PRO-MATLAB User's Guide*. The MathWorks, Inc., 1990.
107. M. Mahmoud and A. Boujarwah. Robust H_∞ filtering for a class of linear parameter-varying systems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 48(9):1131–1138, September 2001.
108. G. Millerioux and J. Daafouz. Global chaos synchronization and robust filtering in noisy context. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 48(10):1170–1176, October 2001.
109. N. Megiddo. Pathways to the optimal set in linear programming. In N. Megiddo, editor, *Progress in Mathematical Programming: Interior Point and Related Methods*, pages 131–158. Springer Verlag, New York, 1989. Identical version in: *Proceedings of the 6th Mathematical Programming Symposium of Japan, Nagoya, Japan, 1-35, 1986*.
110. MOSEK ApS. Mosek (software package). Web site: `http://www.mosek.com`, July 2001.
111. S. Mahajan and H. Ramesh. Derandomizing semidefinite programming based approximation algorithms. *SIAM J. of Computing*, 28:1641–1663, 1999.
112. B. Murtaugh and M. Saunders. MINOS 5.5 user's guide. Technical report, Systems Optimizaiton Laboratory, Stanford University, July 1998. Web site: `http://www.sbsi-sol-optimize.com/manuals/Minos%205-5%20Manual.p%df`.
113. J. Moré and D. Sorensen. NMTR (software package), March 2000. Web site: `http://www-unix.mcs.anl.gov/~more/nmtr/`.
114. M. Milanese and A. Vicino. Optimal estimation theory for dynamic systems with set membership uncertainty: An overview. *Automatica*, 27(6):997–1009, November 1991.
115. Y. Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*, volume 87 of *Applied Optimization*. Kluwer, Boston, 2004.
116. I. Nenov, D. Fylstra, and L. Kolev. Convexity determination in the microsoft excel solver using automatic differentiation techniques. In *The 4th Internation Conference on Automatic Differentiation*, 2004.
117. Yu. Nesterov and A. Nemirovsky. A general approach to polynomial-time algorithms design for convex programming. Technical report, Centr. Econ. & Math. Inst., USSR Acad. Sci., Moscow, USSR, 1988.

118. Yu. Nesterov and A. Nemirovsky. *Interior-Point Polynomial Algorithms in Convex Programming: Theory and Algorithms*, volume 13 of *Studies in Applied Mathematics*. Society of Industrial and Applied Mathematics (SIAM) Publications, Philadelphia, PA 19101, USA, 1993.
119. Yu. Nesterov, O. Pèton, and J.-Ph. Vial. Homogeneous analytic center cutting plane methods with approximate centers. In F. Potra, C. Roos, and T. Terlaky, editors, *Optimization Methods and Software*, pages 243–273, November 1999. Special Issue on Interior Point Methods.
120. S. Nash and A. Sofer. A barrier method for large-scale constrained optimization. *ORSA Journal on Computing*, 5:40–53, 1993.
121. Yu. Nesterov and M. Todd. Self-scaled barriers and interior-point methods for convex programming. *Mathematics of Operations Research*, 22:1–42, 1997.
122. J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, New York, 1999.
123. D. Orban and R. Fourer. DrAmpl: a meta-solver for optimization. Technical report, Ecole Poytechnique de Montreal, 2004.
124. M. Overton and R. Womersley. On the sum of the largest eigenvalues of a symmetric matrix. *SIAM Journal on Matrix Analysis and Applications*, 13(1):41–45, January 1992.
125. P. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, Series B, 96(2):293–320, 2003.
126. G. Pataki. Geometry of cone-optimization problems and semi-definite programs. Technical report, GSIA Carnegie Mellon University, Pittsburgh, PA, 1994.
127. J. Park, H. Cho, and D. Park. Design of GBSB neural associative memories using semidefinite programming. *IEEE Transactions on Neural Networks*, 10(4):946–950, July 1999.
128. R. Palhares, C. de Souza, and P. Dias Peres. Robust H_∞ filtering for uncertain discrete-time state-delayed systems. *IEEE Transactions on Signal Processing*, 48(8):1696–1703, August 2001.
129. R. Palhares and P. Peres. LMI approach to the mixed H_2/H_∞ filtering design for discrete-time uncertain systems. *IEEE Transactions on Aerospace and Electronic Systems*, 37(1):292–296, January 2001.
130. S. Prajna, A. Papachristodoulou, and P. Parrilo. *SOSTOOLS: Sum of squares optimization toolbox for MATLAB*, 2002. Available from `http://www.cds.caltech.edu/sostools` and `http://www.aut.ee.ethz.ch/parrilo/sostools`.
131. O. Pèton and J.-P. Vial. A tutorial on ACCPM: User's guide for version 2.01. Technical Report 2000.5, HEC/Logilab, University of Geneva, March 2001. See also the `http://ecolu-info.unige.ch/~logilab/software/accpm/accpm.html`.
132. E. Rimon and S. Boyd. Obstacle collision detection using best ellipsoid fit. *Journal of Intelligent and Robotic Systems*, 18:105–126, 1997.
133. J. Renegar. A polynomial-time algorithm, based on Newton's method, for linear programming. *Mathematical Programming*, 40:59–93, 1988.
134. B. Radig and S. Florczyk. *Evaluation of Convex Optimization Techniques for the Weighted Graph-Matching Problem in Computer Vision*, pages 361–368. Springer, December 2001.

135. L. Rasmussen, T. Lim, and A. Johansson. A matrix-algebraic approach to successive interference cancellation in CDMA. *IEEE Transactions on Communications*, 48(1):145–151, January 2000.
136. M. Rijckaert and X. Martens. Analysis and optimization of the williams-otto process by geometric programming. *AIChE Journal*, 20(4):742–750, July 1974.
137. R. Rockafellar. *Convex Analysis*. Princeton Univ. Press, Princeton, New Jersey, second edition, 1970.
138. E. Rosenberg. *Globally Convergent Algorithms for Convex Programming with Applications to Geometric Programming*. PhD thesis, Department of Operations Research, Stanford University, 1979.
139. P. Stoica, T. McKelvey, and J. Mari. Ma estimation in polynomial time. *IEEE Transactions on Signal Processing*, 48(7):1999–2012, July 2000.
140. J. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11:625–653, 1999.
141. J. A. K. Suykens, T. Van Gestel, J. De Brabanter, B. De Moor, and J. Vandewalle. Least squares support vector machines, 2002.
142. J. Stoer and C. Witzgall. *Convexity and Optimization in Finite Dimensions I*. Springer-Verlag, 1970.
143. U. Shaked, L. Xie, and Y. Soh. New approaches to robust minimum variance filter design. *IEEE Transactions on Signal Processing*, 49(11):2620–2629, November 2001.
144. H. Tuan, P. Apkarian, and T. Nguyen. Robust and reduced-order filtering: new LMI-based characterizations and methods. *IEEE Transactions on Signal Processing*, 49(12):2975–2984, December 2001.
145. H. Tuan, P. Apkarian, T. Nguyen, and T. Narikiyo. Robust mixed H_2/H_∞ filtering of 2-D systems. *IEEE Transactions on Signal Processing*, 50(7):1759–1771, July 2002.
146. C. Tseng and B. Chen. H_∞ fuzzy estimation for a class of nonlinear discrete-time dynamic systems. *IEEE Transactions on Signal Processing*, 49(11):2605–2619, November 2001.
147. The Mathworks, Inc. LMI control toolbox 1.0.8 (software package). Web site: `http://www.mathworks.com/products/lmi`, August 2002.
148. H. Tan and L. Rasmussen. The application of semidefinite programming for detection in CDMA. *IEEE Journal on Selected Areas in Communications*, 19(8):1442–1449, August 2001.
149. T. Tsuchiya. A polynomial primal-dual path-following algorithm for second-order cone programming. Technical report, The Institute of Statistical Mathematics, Tokyo, Japan, October 1997.
150. Z. Tan, Y. Soh, and L. Xie. Envelope-constrained H_∞ filter design: an LMI optimization approach. *IEEE Transactions on Signal Processing*, 48(10):2960–2963, October 2000.
151. Z. Tan, Y. Soh, and L. Xie. Envelope-constrained H_∞ FIR filter design. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(1):79–82, January 2000.
152. R. Tütüncü, K. Toh, and M. Todd. SDPT3—a MATLAB software package for semidefinite-quadratic-linear programming, version 3.0. Technical report, Carnegie Mello University, August 2001.
153. R. Tapia, Y. Zhang, and L. Velazquez. On convergence of minimization methods: Attraction, repulsion and selection. *Journal of Optimization Theory and Applications*, 107:529–546, 2000.

154. R. Vanderbei. LOQO user's manual—version 4.05. Technical report, Operations Research and Financial Engineering, Princeton University, October 2000.
155. L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, March 1996.
156. R. Vanderbei and H. Benson. On forumulating semidefinite programming problems as smooth convex nonlinear optimization problems. Technical Report ORFE-99-01, Operations Research and Financial Engineering, Princeton University, January 2000.
157. L. Vandenberghe, S. Boyd, and A. El Gamal. Optimal wire and transistor sizing for circuits with non-tree topology. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer Aided Design*, pages 252–259, 1997.
158. L. Vandenberghe, S. Boyd, and A. El Gamal. Optimizing dominant time constant in RC circuits. *IEEE Transactions on Computer-Aided Design*, 2(2):110–125, February 1998.
159. S.-P. Wu and S. Boyd. SDPSOL: A parser/solver for semidefinite programs with matrix structure. In L. El Ghaoui and S.-I. Niculescu, editors, *Recent Advances in LMI Methods for Control*, chapter 4, pages 79–91. SIAM, 2000.
160. F. Wang and V. Balakrishnan. Robust Kalman filters for linear time-varying systems with stochastic parametric uncertainties. *IEEE Transactions on Signal Processing*, 50(4):803–813, April 2002.
161. S.-P. Wu, S. Boyd, and L. Vandenberghe. FIR filter design via spectral factorization and convex optimization. In B. Datta, editor, *Applied and Computational Control, Signals, and Circuits*, volume 1, pages 215–245. Birkhauser, 1998.
162. M. Wright. Some properties of the Hessian of the logarithmic barrier function. *Mathematical Programming*, 67:265–295, 1994.
163. S. Wright. *Primal Dual Interior Point Methods*. Society of Industrial and Applied Mathematics (SIAM) Publications, Philadelphia, PA 19101, USA, 1999.
164. J. Weickert and Christoph Schnörr. A theoretical framework for convex regularizers in pde-based computation of image motion. *International Journal of Computer Vision, Band 45*, 3:245–264, 2001.
165. S. Wang, L. Xie, and C. Zhang. H_2 optimal inverse of periodic FIR digital filters. *IEEE Transactions on Signal Processing*, 48(9):2696–2700, September 2000.
166. Y. Ye. *Interior-point algorithms: Theory and practice*. John Wiley & Sons, New York, NY, 1997.
167. Y. Ye. A path to the arrow-debreu competitive market equilibrium. Technical report, Stanford University, February 2004. Web site: `http://www.stanford.edu/~yyye/arrow-debreu2.pdf`.
168. F. Yang and Y. Hung. Robust mixed H_2/H_∞ filtering with regional pole assignment for uncertain discrete-time systems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 49(8):1236–1241, August 2002.
169. Y.Ye, M. Todd, and S. Mizuno. An $O(\sqrt{n}L)$-iteration homogeneous and self-dual linear programming algorithm. *Mathematics of Operations Research*, 19(1):53–67, 1994.
170. Y. Ye and J. Zhang. Approximation for dense-n/2-subgraph and the complement of min-bisection. *Manuscript*, 1999.

171. S. Zhang. A new self-dual embedding method for convex programming. Technical report, Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, October 2001.
172. M. Zibulevsky. Pattern recognition via support vector machine with computationally efficient nonlinear transform. Technical report, The University of New Mexico, Computer Science Department, 1998. Web site: `http://iew3.technion.ac.il/~mcib/nipspsvm.ps.gz`.
173. J. Zowe, M. Kocvara, and M. Bendsoe. Free material optimization via mathematical programming. *Mathematical Programming*, 9:445–466, 1997.
174. U. Zwick. Outward rotations: a tool for rounding solutions of semidefinite programming relaxations, with applications to max cut and other problems. In *Proceedings of the 31th Symposium on Theory of Computation*, pages 679–687, 1999.
175. H. Zhou, L. Xie, and C. Zhang. A direct approach to H_2 optimal deconvolution of periodic digital channels. *IEEE Transactions on Signal Processing*, 50(7):1685–1698, July 2002.