Graph Implementations for Nonsmooth Convex Programs

Michael C. Grant[1] and Stephen P. Boyd[2]

[1] Stanford University
mcgrant@stanford.edu
[2] Stanford University
boyd@stanford.edu

Summary. We describe *graph implementations*, a generic method for representing a convex function via its epigraph, described in a disciplined convex programming framework. This simple and natural idea allows a very wide variety of smooth and nonsmooth convex programs to be easily specified and efficiently solved, using interior-point methods for smooth or cone convex programs.

Keywords: Convex optimization, nonsmooth optimization, disciplined convex programming, optimization modeling languages, semidefinite programming, second-order cone programming, conic optimization, nondifferentiable functions.

1 Introduction

It is well known that convex programs have many attractive properties, including the proven existence of efficient methods to solve them. What is not as widely appreciated, however, is that nonsmooth convex programs—*i.e.*, models with nondifferentiable constraints or objectives—can, in theory, be solved just as efficiently as their smooth counterparts. But here, as is often the case, theory and practice do not coincide. Methods that are theoretically efficient for general nondifferentiable problems, such as the ellipsoid method [3], are notoriously slow in practice.

In contrast, there are many solvers available for smooth convex programs, as well as for certain standard forms such as semidefinite programs (SDPs), that are efficient in both theory and practice; *e.g.*, [13, 16, 17, 18]. These solvers can often be used to solve a general nonsmooth problem as well—not directly of course, but by first transforming it into an equivalent form supported by the solver. The equivalent problem is usually larger than the original, but the superior efficiency of the solver more than compensates for the increase in size, especially if problem structure is taken into account.

The transformation approach dates back to the very first days of linear programing [7]. It is usually taught as a collection of tricks that a modeler can use to (hopefully) reformulate problems by hand. The versatility of the approach, of course, depends upon the variety of transformations known by the modeler. But while some transformations are fairly obvious and widely known, others are

V.D. Blondel et al. (Eds.) Recent Advances in Learning and Control, LNCIS 371, pp. 95–110, 2008.
springerlink.com

neither obvious nor well known, even to some experts in convex optimization. Furthermore, even if a transformation is identified, the reformulation process is often time consuming and error prone, for both experts and applications-oriented modelers alike.

We propose to enable modeling frameworks to largely *automate* the process of identifying and applying these transformations, so that a much wider variety of models—smooth and nonsmooth alike—can be both easily specified and efficiently solved. Our approach depends upon two distinct but interrelated developments. The first is a methodology for constructing convex optimization models called *disciplined convex programming*. The methodology imposes a set of rules or conventions that must be followed when constructing convex programs. The rules are simple and teachable, drawn from basic principles of convex analysis, and follow the practices of those who regularly use convex optimization. Conforming problems are called, appropriately, *disciplined convex programs*, or *DCPs*. The DCP ruleset does not limit generality, but it does require that the modeler explicitly provide just enough structure to allow further analysis and solution of a problem to be automated.

The second development is a new method for defining or implementing a function in a modeling framework, as as the optimal value of a parameterized convex program (specifically, a DCP). We call such a function definition a *graph implementation* because it exploits the relationship between convex and concave functions and their epigraphs and hypographs, respectively. A graph implementation encapsulates a method for transforming instances of a specific function in a constraint or objective into a form compatible with the underlying solver's standard form. The conditions imposed by the DCP ruleset ensure that these transformations always preserve equivalence and convexity. The most significant benefit of graph implementations is their ability to efficiently implement nondifferentiable functions. But in fact, graph implementations can also be used to implement many *smooth* functions as well when the target standard form is nonsmooth (*e.g.*, an SDP).

We have created a modeling framework called `cvx` [8] that supports disciplined convex programming and graph implementations. `cvx` uses the object-oriented features of MATLAB® to turn it into an optimization modeling language: optimization variables can be declared and constraints and objectives specified using natural MATLAB® syntax. `cvx` verifies compliance with the DCP ruleset, transforms conforming models to solvable form, calls an appropriate numerical solver, and translates the numerical results back to the original problem—all without user intervention. The framework includes a large library of common convex and concave functions, both smooth and nonsmooth, and more can be added.

To an applications-oriented user, the conceptual model presented by `cvx` is very simple: `cvx` solves any problem (up to some practical size limits, of course) constructed according to the DCP ruleset from functions found in the `cvx` library. The modeler need not know what transformations are taking place, or even that a transformation is necessary. That is, graph implementations are entirely *opaque* or hidden from a standard `cvx` user. On the other hand, expert users

can use graph implementations to add new transformations to the system that general users can exploit simply by calling the new functions in their models. This division of labor allows modelers to focus on building convex programs, and experts to focus on solving them.

In what follows, we will describe disciplined convex programming and graph implementations in detail. Both are abstract, language-independent concepts; nevertheless, it will be easier to explain both using examples from an actual modeling framework such as `cvx`. So we begin by introducing `cvx` with a few simple examples. A basic familiarity with MATLAB® is assumed throughout.

2 A Brief Introduction to `cvx`

To begin, consider the simple linear program

$$\begin{array}{l} \text{minimize} \quad c^T x \\ \text{subject to } Ax \leq b, \end{array} \tag{1}$$

with variable $x \in \mathbf{R}^n$ and data $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$, and $c \in \mathbf{R}^n$. The following (MATLAB®/`cvx`) code generates and solves a random instance of (1):

```
m = 16; n = 8;
A = randn(m,n); b = randn(m,1); c = randn(n,1);
cvx_begin
    variable x(n)
    minimize( c' * x )
    subject to
        A * x <= b;
cvx_end
```

The indentation is purely for stylistic reasons and is optional. The code is relatively self-explanatory, but a few notes are in order:

- The `cvx_begin` and `cvx_end` commands mark the beginning and end, respectively, of any `cvx` model.
- Variables must be declared before their first use. For example, the `variable` statement above declares `x` to be a vector of length `n`.
- The `subject to` statement is optional—it is provided by `cvx` only to make models more readable and has no mathematical effect.
- Objectives and constraints may be placed in any order.

When `cvx_end` is reached, `cvx` will complete the conversion of the model to solvable form and call the underlying numerical solver. It will replace the MATLAB® variable `x`, which up to that point was a special `cvx` variable object, with a numeric vector representing an optimal value.

Now consider a norm minimization problem with box constraints:

$$\begin{array}{l} \text{minimize} \quad \|Ay - b\|_2 \\ \text{subject to } \ell \leq y \leq u \end{array} \tag{2}$$

The following cvx/MATLAB® code constructs and solves a version of (2), reusing A and b from above and generating random bounds:

```
l = -abs(randn(n,2)); u = +abs(randn(n,2));
cvx_begin
    variable y(n)
    minimize( norm(A*y-b,2) )
    subject to
        y <= u;
        y >= l;
cvx_end
```

It is well known that (2) can be reformulated as a (convex) quadratic program (QP) or a second-order cone program (SOCP). (cvx, in fact, converts this problem to an SOCP.) The transformation in this case is not particularly complicated; still, it is nice to have it completely automated.

cvx supports a variety of other norms and penalties for this model simply by replacing the objective function; for example:

```
minimize( norm(A*y-b,2) + 0.1*norm(y,1) )
minimize( norm(A*y-b,3.5) )
minimize( sum(huber(A*y-b)) )
```

All of these examples can be reformulated as SOCPs as well. Here, however, the transformations are not at all obvious, even to experts; and in all three cases working out the transformation by hand would be tedious and error prone. cvx, however, can solve all three problems automatically.

As a final example, consider the task of determining the minimum volume ellipsoid (also known as the Löwner-John ellipsoid) $\mathcal{E}$ containing a finite set of points $z_1, z_2, \ldots, z_n \in \mathbf{R}^d$:

$$\begin{array}{ll} \text{minimize} & \text{vol}(\mathcal{E}) \\ \text{subject to} & z_i \in \mathcal{E}, \quad i = 1, \ldots, n. \end{array} \tag{3}$$

The parameterization we will use for $\mathcal{E}$ is

$$\mathcal{E} \triangleq \{\, u \mid \|Pu + q\|_2 \leq 1 \,\}, \tag{4}$$

where $(P, q) \in \mathbf{R}^{d \times d} \times \mathbf{R}^d$, and P is symmetric positive semidefinite. In this case, $\text{vol}(\mathcal{E})$ is proportional to $\det(P^{-1})$ (see [6, §8.4]). With this parametrization we can cast the problem above as

$$\begin{array}{ll} \text{minimize} & \det P^{-1} \\ \text{subject to} & \|Pz_i + q\|_2 \leq 1, \quad i = 1, 2, \ldots, n, \end{array} \tag{5}$$

with variables $P = P^T \in \mathbf{R}^{d \times d}$ and $q \in \mathbf{R}^d$. We have written the objective informally as $\det P^{-1}$; a more precise description is $f_{\texttt{det\_inv}}(P)$, where

$$f_{\texttt{det\_inv}}(P) \triangleq \begin{cases} \det(P)^{-1} & P = P^T \succ 0 \\ +\infty & \text{otherwise.} \end{cases} \tag{6}$$

This function implicitly constrains P to be symmetric and positive definite. The function $f_{\texttt{det\_inv}}$ is convex, so the problem above is a convex problem.

The following cvx/MATLAB® code generates a random set of points and computes the optimal ellipsoid by solving (5):

```
d = 2;
z = randn(d,n);
cvx_begin
    variables P(d,d) q(d)
    minimize( det_inv(P) )
    subject to
        for i = 1 : n,
            norm( P*z(:,i)+q,2 ) <= 1;
        end
cvx_end
```

The function `det_inv` represents $f_{\texttt{det\_inv}}(\cdot)$, including the implicit constraint that its argument be symmetric and positive definite. It is known that this problem can be cast as a semidefinite program (SDP), but the required conversion is quite complicated. Fortunately, that conversion is buried inside `cvx`'s definition of `det_inv` and performed automatically.

This is, of course, a considerably abbreviated introduction to `cvx`, intended only to give the reader an idea of the basic syntax and structure of `cvx` models. The reader is encouraged to read the user's guide [8] for a more thorough treatment, or to download the software and try it. The examples presented here can be entered exactly as listed.

3 Disciplined Convex Programming

Disciplined convex programming was first named and described by Grant, Boyd, and Ye in [9] and Grant in [10]. It was modeled on the methods used by those who regularly construct convex optimization models. Such modelers do not simply construct arbitrary nonlinear programs and attempt to verify convexity after the fact; rather, they begin with a mental library of functions and sets with known geometries, and combine them in ways which convex analysis guarantees will preserve convexity.

Disciplined convex programming is an attempt to formalize and this practice and codify its techniques. It consists of two key components:

- an *atom library*—a collection of functions or sets with known properties of curvature (convexity and concavity) and monotonicity; and
- the *DCP ruleset*—a finite enumeration of ways in which atoms may be combined in objectives and constraints while preserving convexity.

The rules are drawn from basic principles of convex analysis, and are easy to learn, once you have had an exposure to convex analysis and convex optimization. They constitute a set of sufficient but not necessary conditions for convexity,

which means that it is possible to build models that violate the rules but are still convex. We will provide examples of such violations and their resolution later in this section.

3.1 Preliminaries

The rules of disciplined convex programming depend primarily upon the *curvature* of numeric expressions. The four categories of curvature considered are *constant*, *affine*, *convex*, and *concave*. The usual definitions apply here; for example, a function $f : \mathbf{R}^n \to \mathbf{R}$ is convex if its domain is a convex set, and

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y) \quad \forall x, y \in \mathbf{R}^n,\ \alpha \in [0,1]. \tag{7}$$

Of course, there is significant overlap in these categories: constant expressions are affine, and real affine expressions are both convex and concave. Complex constant and affine expressions are considered as well, but of course convex and concave expressions are real by definition.

Functions in the atom library are chracterized not just by curvature but by *monotonicity* as well. Three categories of monotonicity are considered: *nondecreasing*, *nonincreasing*, and *nonmonotonic*. Again, the usual mathematical definitions apply; for example, a function $f : \mathbf{R} \to \mathbf{R}$ is nondecreasing if

$$x \geq y \quad \Longrightarrow \quad f(x) \geq f(y). \tag{8}$$

Two technical clarifications are worth making here. First, monotonicity is considered in a global, extended-valued sense. For example, the MATLAB® square root function `sqrt` is interpreted in `cvx` as follows:

$$f_{\texttt{sqrt}} : \mathbf{R} \to (\mathbf{R} \cup -\infty), \quad f_{\texttt{sqrt}}(x) \triangleq \begin{cases} \sqrt{x} & x \geq 0 \\ -\infty & x < 0 \end{cases} \tag{9}$$

Under this interpretation, it is concave and nondecreasing. Secondly, for functions with multiple arguments, curvature is considered *jointly*, while montonicity is considered *separately* for each argument. For example, the function `quad_over_lin` in `cvx`, given by

$$f_{\texttt{qol}}(x,y) : (\mathbf{R}^n \times \mathbf{R}) \to (\mathbf{R} \cup +\infty), \quad f_{\texttt{qol}}(x,y) \triangleq \begin{cases} x^T x/y & y > 0 \\ +\infty & y \leq 0 \end{cases} \tag{10}$$

is jointly convex in x and y, but nonincreasing in y alone.

With terminology defined, we now proceed to the ruleset itself.

3.2 Constraints and Objectives

A disciplined convex program may either be an *optimization* problem consisting of a single objective and zero or more constraints, or a *feasibility* problem

consisting of one or more constraints and no objective. The rules for objectives and constraints are as follows:

- A valid objective is
  - the minimization of a convex expression;
  - the maximization of a concave expression.
- A valid constraint is
  - a set membership relation ($\in$) in which the left-hand side (LHS) is affine and the right-hand side (RHS) is a convex set.
  - an equality ($=$) with an affine LHS and an affine RHS.
  - a less-than inequality ($\leq$) with a convex LHS and a concave RHS.
  - a greater-than inequality ($\geq$) with a concave LHS and a convex RHS.

For any problem that conforms to these rules, the constraint set is convex. These rules, however, require more than just convexity of the constraint set: They constrain *how* the constraint set is described. For example, the constraint `square(x)==0`, where `x` is a scalar variable, defines the convex set $\{0\}$. But it is rejected by the rules above, since the LHS of this equality constraint is not affine. When the constraint is written in the equivalent form `x==0`, however, which is accepted by the rules above, since both sides are affine.

3.3 Simple Expressions

Disciplined convex programming determines the curvature of numeric and set expressions by recursively applying the following rules. This list may seem long, but it is for the most part an enumeration of basic rules of convex analysis for combining convex, concave, and affine forms: sums, multiplication by scalars, and so forth. For the basics of convex analysis, see, *e.g.*, [2, 4, 6, 15, 20].

- A valid affine expression is
  - a valid constant expression;
  - a declared variable;
  - a valid call to a function with an affine result;
  - the sum or difference of affine expressions;
  - the product of an affine expression and a constant.
- A valid convex expression is
  - a valid constant or affine expression;
  - a valid call to a function with a convex result;
  - the sum of two or more convex expressions;
  - the difference between a convex expression and a concave expression;
  - the product of a convex expression and a nonnegative constant;
  - the product of a concave expression and a nonpositive constant;
  - the negation of a concave expression.
- A valid concave expression is
  - a valid constant or affine expression;
  - a valid call to a function in the atom library with a concave result;
  - the sum of two or more concave expressions;

  - the difference between a concave expression and a convex expression;
  - the product of a concave expression and a nonnegative constant;
  - the product of a convex expression and a nonpositive constant;
  - the negation of a convex expression.
- A valid convex set expression is
  - a valid reference to a convex set in the atom library;
  - the intersection of two or more convex set expressions;
  - the sum or difference of convex set expressions;
  - the sum or difference of a convex set expression and a constant;
  - the product of a convex set expression and constant.

If an expression cannot be categorized by this ruleset, then it is rejected by `cvx`. For matrix and array expressions, these rules are applied on an elementwise basis. We note that the set of rules listed above is redundant; there are much smaller, equivalent sets of rules.

Of particular note is that these expression rules forbid *products* between non-constant expressions. We call this the *no-product rule* for obvious reasons. For example, the expression $x\sqrt{x}$, written in `cvx` as `x*sqrt(x)`, is convex (at least when x is positive) but is rejected by `cvx` as violating the above rules. Fortunately, `cvx` provides a function called `pow_pos(x,p)` that implements the convex and concave branches of `x^p`, so this expression can be written as `pow(x,3/2)`.

3.4 Compositions

A basic rule of convex analysis is that convexity is closed under composition with an affine mapping. This is part of the DCP ruleset as well:

- A convex, concave, or affine function may accept as an argument an affine expression (assuming it is of compatible size).

For example, consider the function `square`, which is provided in the `cvx` atom library. This function squares its argument; *i.e.*, it computes `x.*x`. (For array arguments, it squares each element independently.) It is known to be convex, provided its argument is real. So if `x` is a real variable, then

```
square( x )
```

is accepted by `cvx`; and, thanks to the above rule, so is

```
square( A * x + b )
```

if `A` and `b` are constant matrices of compatible size.

The DCP ruleset also provides for certain nonlinear compositions as well. The four composition rules are:

- If a convex function is nondecreasing in a given argument, then that argument may be convex.
- If a convex function is nonincreasing in a given argument, then that argument may be concave.

- If a concave function is nondecreasing in a given argument, then that argument may be concave.
- If a concave function is nonincreasing in a given argument, then that argument may be convex.

(In each case, we assume that the argument is of compatible size.) In fact, nearly every rule in the DCP ruleset can be derived from these composition rules.

For example, the pointwise maximum of convex functions is convex, because the maximum function is convex and nondecreasing. Thus if `x` is a vector variable then

```
max( abs( x ) )
```

obeys the first of the four composition rules and is therefore accepted by `cvx`. In fact, the infinity-norm function `norm( x, Inf )` is defined in exactly this manner. Affine functions must obey these composition rules as well; but because they are both convex and concave, they prove a bit more flexible. So, for example, the expressions

```
sum( square( x ) )
sum( sqrt( x ) )
```

are both valid nonlinear compositions in `cvx` since the rules for both the convex-nondecreasing and convex-nonincreasing cases apply to `sum`.

3.5 The Ruleset in Practice

As we stated in the introduction to this section, the DCP rules are sufficient but not necessary conditions for the convexity (or concavity) of an expression, constraint, or objective. Some expressions which are obviously convex or concave will fail to satisfy them. For example, if `x` is a `cvx` vector variable, then the expression

```
sqrt( sum( square( x ) ) )
```

is rejected by `cvx`, because there is no rule governing the composition of a concave nondecreasing function with a convex function. Fortunately, there is a simple workaround in this case: use `norm( x )` instead, since `norm` is in the atom library and is known by `cvx` to be convex.

This is an example of what is meant by our statement in the introduction that disciplined convex programming requires the modeler to supply "just enough" structure to enable the automation of the solution process. Obviously, both `norm` and the longer, non-compliant version are equivalent numerically, but the latter form enables `cvx` to complete the verification and conversion process. Of course, because the library is finite, there will inevitably be instances where a simple substitution is not possible. Thus to insure generality, the atom library must be expandable.

4 Graph Implementations

Any modeling framework for optimization must provide a computational description of the functions its supports to the underlying solver. For a smooth function, this traditionally consists of code to compute the value an derivatives of the function at requested points. In cvx, it is possible to define a convex or a concave function as the solution of a parameterized DCP. We call such a definition a *graph implementation*, a term first coined in [10] and inspired by the properties of epigraphs and hypographs of convex and concave functions, respectively.

4.1 The Basics

Recall the definition of the *epigraph* of a function $f : \mathbf{R}^n \to (\mathbf{R} \cup +\infty)$:

$$\mathbf{epi}\, f \triangleq \{\, (x, y) \in \mathbf{R}^n \times \mathbf{R} \mid f(x) \leq y \,\}. \tag{11}$$

A fundamental principle of convex analysis states that f is a convex function if and only if $\mathbf{epi}\, f$ is a convex set. The relationship between the two can be expresed in a reverse fashion as well:

$$f(x) \equiv \inf \{\, y \mid (x, y) \in \mathbf{epi}\, f \,\}. \tag{12}$$

(We adopt the convention that the infimum of an empty set is $+\infty$.) Equation (12) expresses f as the solution to a convex optimization problem—or, more accurately, a family of such problems, parameterized by the argument x.

A *graph implementation* of f takes the relationship in (12) and makes it concrete, by expressing $\mathbf{epi}\, f$ in a solvable manner—that is, with an equivalent collection of constraints in x and y that are compatible with the target solver. For example, consider the real absolute value function $f_{\mathtt{abs}}(x) = |x|$. Its epigraph can be represented as an intersection of two linear inequalities:

$$\mathbf{epi}\, f_{\mathtt{abs}} = \{\, (x, y) \mid |x| \leq y \,\} = \{\, (x, y) \mid x \leq y, -x \leq y \,\} \tag{13}$$

A graph implementation is just a description or *encapsulation* of that transformation, justified mathematically through a simple equivalency of sets.

In cvx, graph implementations can be specified using the same syntax as other cvx models, and are subject to the same DCP ruleset as well. The following cvx/MATLAB® code is a representation of $f_{\mathtt{abs}}$:

```
function y = f_abs(x)
cvx_begin
    variable y
    minimize( y )
    subject to
         x <= y;
        -x <= y;
cvx_end
```

(The absolute value function `abs` in `cvx` is actually implemented a bit differently; for example, it supports complex values and vector-valued arguments, in an elementwise fashion.)

If `f_abs` is called with a numeric value of `x`, then the `cvx` specification it contains will construct a linear program with a single variable and two inequalities. Upon reaching `cvx_end`, `cvx` will call the underlying solver and compute the correct result—at least to within the tolerances of the solver. This is, of course, a rather impractical way to compute the absolute value; in the real implementation of `abs` in `cvx` we avoid this inefficiency. But it is, at least, technically correct, and it is also a useful way to debug a graph implementation.

The more interesting case is when `f_abs` is used within a `cvx` model, with an affine `cvx` expression for an argument. In this case, the `cvx` specification will be incomplete, because the value of `x` is not yet known. What `cvx` does in this circumstance is to incorporate the specification *into the surrounding model* itself, in a manner not unlike the expansion of a `inline` function in C++. For example, if `z` is a scalar `cvx` variable, then the constraint

```
f_abs(z-3) <= 1;
```

will be translated internally by `cvx` as follows:

```
y <= 1;
x == z-3;
x <= y;
-x <= y;
```

(Steps are taken as needed to avoid name conflicts with existing variables.) The constraint is now in a form compatible with an efficient solver. Of course, two new variables and several new constraints have been added, but in the long run the added costs of expansions like this are far outweighed by the fact that a much more efficient solver can now be used, because the nondifferentiability has been eliminated.

Of course, the transformation of the absolute value function into an efficiently solvable form is relatively well known. But while it may be obvious to some, it is certainly not to everyone; and it is certainly convenient to have the transformation automated. For more advanced functions, the benefits should be more clear.

4.2 Advanced Usage

Graph implementations of convex functions are not, in fact, limited to strict epigraph representations. Suppose that $S \subset \mathbf{R}^n \times \mathbf{R}^m$ is a convex set and $\bar{f} : (\mathbf{R}^n \times \mathbf{R}^m) \to (\mathbf{R} \cup +\infty)$ is jointly convex in x and y; then

$$f : \mathbf{R}^n \to (\mathbf{R} \cup +\infty), \quad f(x) \triangleq \inf \left\{ \bar{f}(x,y) \;\middle|\; (x,y) \in S \right\} \tag{14}$$

is a convex function of x. If $m = 1$ and $\bar{f}(x,y) \triangleq y$, then the epigraph form (12) is recovered; but `cvx` fully supports this more general form.

For example, consider the unit-halfwidth Huber penalty function $h(x)$:

$$h : \mathbf{R} \to \mathbf{R}, \quad h(x) \triangleq \begin{cases} x^2 & |x| \leq 1 \\ 2|x| - 1 & |x| \geq 1 \end{cases} \tag{15}$$

This function cannot be used in an optimization algorithm utilizing Newton's method, because its Hessian is discontiuous at $x = \pm 1$, and zero for $|x| \geq 1$. However, it can be expressed in the form (14) in this manner:

$$h(x) \triangleq \inf \left\{ 2v + w^2 \mid |x| \leq v + w, \ w \leq 1 \right\} \tag{16}$$

We can implement the Huber penalty function in `cvx` as follows:

```
function cvx_optval = huber( x )
cvx_begin
    variables w v;
    minimize( 2 * v + square( w ) );
    subject to
        abs( x ) <= w + v;
        w <= 1;
cvx_end
```

If `huber` is called with a numeric value of `x`, then `cvx` will solve the resulting QP and return the numeric result. (As with `f_abs`, there is a simpler way to compute the Huber penalty when its argument is a numeric constant.) But if `huber` is called from within a larger `cvx` specification, then `cvx` will use this implementation to transform the call into a form compatible with the underlying solver. Note that the precise transformation depends on how `square` and `abs` are themselves implemented; multilevel transformations like this are quite typical.

There is a corresponding development for concave functions as well. Given the set S above and a concave function $\bar{g} : (\mathbf{R}^n \times \mathbf{R}^m) \to (\mathbf{R} \cup +\infty)$ is concave, the function

$$\bar{f} : \mathbf{R} \to (\mathbf{R} \cup +\infty), \quad \bar{f}(x) \triangleq \sup \{ g(x, y) \mid (x, y) \in S \} \tag{17}$$

is also a concave function. If $\bar{g}(x, y) \triangleq y$, then

$$\bar{f}(x) \triangleq \sup \{ y \mid (x, y) \in S \} \tag{18}$$

gives the *hypograph* representation of $\bar{f}$; that is, $S = \mathbf{hypo}\, f$. In `cvx`, a concave incomplete specification is simply one that uses a `maximize` objective instead of a `minimize` objective.

Some functions are not thought of as nondifferentiable in a casual setting but are technically so, and must be dealt with as such in an optimization algorithm. Consider, for example, the real square root function (9) above. This function is concave, and is smooth for positive x, but not at $x = 0$. Its hypograph, however, is

$$\mathbf{hypo}\, f_{\mathtt{sqrt}} \triangleq \left\{ (x, y) \mid x \geq 0, \ \sqrt{x} \geq y \right\} = \left\{ (x, y) \mid \max\{y, 0\}^2 \leq x \right\} \tag{19}$$

Thus a graph implementation can solve the nondifferentiability problem. In cvx, this function can be implemented as follows:

```
function y = f_sqrt(x)
cvx_begin
    variable y
    maximize( y )
    subject to
        square( y ) <= x
cvx_end
```

This particular type of nondifferentiability also occurs in the concave entropy function; it can be eliminated with a similar transformation.

4.3 Conic Solver Support

The most obvious benefit of graph implementations is their ability to describe nonsmooth functions in a computationally efficient manner. But the solvers used in the first publicly released versions of cvx posed a different challenge: they did not support smooth functions either. Rather, these solvers solved *semidefinite-quadratic-linear programs* (SQLPs)—problems of the form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & \mathcal{A}x = b \\ & x \in \mathcal{K}_1 \times \mathcal{K}_2 \times \ldots \mathcal{K}_L \end{array} \tag{20}$$

where x is the optimization variable, $\mathcal{A}$ is a linear operator, b and c are vectors, and the sets $\mathcal{K}_i$ are convex cones from a select list: the nonnegative orthant $\mathbf{R}^n_+$, the second-order cone $\mathcal{Q}^n$, and the semidefinite cone $\mathcal{S}^n_+$:

$$\mathcal{Q}^n \triangleq \{ (x, y) \in \mathbf{R}^n \times \mathbf{R} \mid \|x\|_2 \leq y \} \tag{21}$$

$$\mathcal{S}^n_+ \triangleq \left\{ X \in \mathbf{R}^{n \times n} \;\middle|\; X = X^T,\ \lambda_{\min}(X) \geq 0 \right\} \tag{22}$$

Clearly, SQLPs are very closely related to SDPs; in fact, all SQLPs can be solved as SDPs. For more information about these problems, consult [12, 19], or the documentation on the solvers themselves [16, 17].

In practice, few application-driven models naturally present themselves as SQLPs; rather, modelers have simply recognized that their problems can be transformed into that form. In fact, as is known to readers of certain well-known texts on the subject [1, 5, 12, 14], SQLPs are *very* versatile, and can represent a wide variety of smooth and nonsmooth convex programs. The challenge, then, lies in finding an SQLP representation of of a given convex program—assuming one exists.

Using graph implementations, a variety of both smooth and nonsmooth functions were added to the cvx atom library for SQLP solvers, including minimums and maximums, absolute values, quadratic forms, convex and concave branches of the power function x^p, ℓ_p norms, convex and concave polynomials, geometric

means, eigenvalue and singular value functions, and determinants. Key omissions include logarithms, exponentials, and entropy; such functions simply cannot be exactly represented in an SQLP solver. (On the other hand, smooth solvers cannot solve many of the eigenvalue and determinant problems for which SQLP solvers excel.)

For a simple example, consider the function $f_{\mathtt{sq}}(x) \triangleq x^2$; its epigraph form (12) can be expressed using a single semidefinite cone:

$$f_{\mathtt{sq}}(x) \triangleq \inf \left\{ y \;\middle|\; \left[\begin{smallmatrix} y & x \\ x & 1 \end{smallmatrix}\right] \in \mathcal{S}^2_+ \right\}. \tag{23}$$

The `cvx` version of this function is

```
function y = f_sq(x)
cvx_begin
    variable y
    minimize( y )
    [ y, x ; x, 1 ] == semidefinite(2);
cvx_end
```

(Since MATLAB® does not have a set membership $\in$ operator, `cvx` uses equality constraints and functions like `semidefinite` to accomplish the same result.)

For a somewhat more complex example, consider the matrix fractional function $f_{\mathtt{mf}} : (\mathbf{R}^n \times \mathbf{R}^{n\times n}) \to (\mathbf{R} \cup +\infty)$, where

$$f_{\mathtt{mf}}(x,Y) = \begin{cases} \frac{1}{2} x^T Y^{-1} x & Y = Y^T \succ 0 \\ +\infty & \text{otherwise} \end{cases} \tag{24}$$

This function is convex in both arguments, and implicitly constrains `Y` to be both symmetric and positive definite. Its epigraph representation is

$$f_{\mathtt{mf}}(x,Y) \triangleq \sup \left\{ z \;\middle|\; \left[\begin{smallmatrix} Y & x \\ x^T & z \end{smallmatrix}\right] \in \mathcal{S}^{n+1}_+ \right\} \tag{25}$$

so it may be implemented in `cvx` as follows:

```
function cvx_optval = f_mf( x, Y )
n = length( x );
cvx_begin
    variable z;
    minimize( z );
    subject to
        [ Y, x ; x', z ] == semidefinite(n+1);
cvx_end
```

Both `f_sq` and `f_mf` are relatively simple examples in comparison to other functions in the `cvx` library. The complexity of some SQLP implementations is in some cases quite striking. For example, the ℓ_p norm can be represented exactly in an SQLP whenever $p = n/d$ is rational. The number of cone constraints required to represent it, however, depends not only on the size of the vector involved, but also in the pattern of bits in a binary representation of n and d! Needless to say, performing such transformations by hand is quite impractical—but once implemented, quite reasonable for a computer.

5 Final Words

We believe that disciplined convex programming closes a significant gap between the theory and practice of convex optimization. A large fraction of useful convex programs are nonsmooth; and until now, those who wished to solve them were faced with unattractive options: transform them by hand to a different, more easily solved form; develop a custom solver; utilize a poorly-performing subgradient-based method; or approximate. A modeling framework that supports disciplined convex programming provides a truly attractive alternative in most of these cases.

References

1. Alizadeh, F., Goldfarb, D.: Second-order cone programming (January 2004)
2. Bertsekas, D.P.: Convex Analysis and Optimization. In: Nedić, A., Ozdaglar, A.E. (eds.) Athena Scientific (2003)
3. Bland, R., Goldfarb, D., Todd, M.: The ellipsoid method: A survey. Operations Research 29(6), 1039–1091 (1981)
4. Borwein, J., Lewis, A.: Convex Analysis and Nonlinear Optimization: Theory and Examples. Springer, Heidelberg (2000)
5. Ben-Tal, A., Nemirovski, A.: Lectures on Modern Convex Optimization: Analysis, Algorithms and Engineering Applications. MPS/SIAM Series on Optimization. SIAM, Philadelphia (2001)
6. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge Univ. Press, Cambridge (2004), `http://www.stanford.edu/~boyd/cvxbook.html`
7. Dantzig, G.: Linear Programming and Extensions. Princeton University Press, Princeton (1963)
8. Grant, M., Boyd, S.: CVX: MATLAB® software for disciplined convex programming, version 1.1 (September 2007), `http://www.stanford.edu/~boyd/cvx/`
9. Grant, M., Boyd, S., Ye, Y.: Disciplined convex programming. In: Liberti, L., Maculan, N. (eds.) Global Optimization: from Theory to Implementation, Nonconvex Optimization and Its Applications, pp. 155–210. Springer Science+Business Media, Inc., New York (2006)
10. Grant, M.: Disciplined Convex Programming. PhD thesis, Department of Electrical Engineering, Stanford University (December 2004)
11. Löfberg, J.: YALMIP: A toolbox for modeling and optimization in MATLAB®. In: Proceedings of the CACSD Conference, Taipei, Taiwan (2004), `http://control.ee.ethz.ch/~joloef/yalmip.php`
12. Lobo, M., Vandenberghe, L., Boyd, S., Lebret, H.: Applications of second-order cone programming. Linear Algebra and its Applications 284, 193–228 (1998) (special issue on Signals and Image Processing)
13. MOSEK ApS. Mosek (software package) (September 2007), `http://www.mosek.com`
14. Nesterov, Yu., Nemirovsky, A.: Interior-Point Polynomial Algorithms in Convex Programming : Theory and Algorithms of Studies in Applied Mathematics, vol. 13. SIAM Publications, Philadelphia, PA (1993)
15. Rockafellar, R.T.: Convex Analysis. Princeton University Press, Princeton (1970)

16. Sturm, J.: Using SeDuMi 1.02, a MATLAB® toolbox for optimization over symmetric cones. Optimization Methods and Software 11, 625–653 (1999), Updated version available at `http://sedumi.mcmaster.ca`
17. Toh, K., Todd, M., Tutuncu, R.: SDPT3 — a MATLAB® software package for semidefinite programming. Optimization Methods and Software 11, 545–581 (1999)
18. Vanderbei, R.: LOQO: An interior point code for quadratic programming. Optimization Methods and Software 11, 451–484 (1999)
19. Vandenberghe, L., Boyd, S.: Semidefinite programming. SIAM Review 38(1), 49–95 (1996)
20. van Tiel, J.: Convex Analysis. An Introductory Text. John Wiley & Sons, Chichester (1984)