cvx Users' Guide

for cvx version 1.21 *

Michael Grant
mcgrant@stanford.edu

Stephen Boyd
boyd@stanford.edu

October, 2010

*code commit 803, 2010-10-14 12:30:37; doc commit 795, 2010-05-31 08:26:19

Contents

1 Introduction

1.1 What is cvx?

cvx is a modeling system for *disciplined convex programming*. Disciplined convex programs, or DCPs, are convex optimization problems that are described using a limited set of construction rules, which enables them to be analyzed and solved efficiently. cvx can solve standard problems such as linear programs (LPs), quadratic programs (QPs), second-order cone programs (SOCPs), and semidefinite programs (SDPs); but compared to directly using a solver for one or these types of problems, cvx can greatly simplify the task of specifying the problem. cvx can also solve much more complex convex optimization problems, including many involving nondifferentiable functions, such as ℓ_1 norms. You can use cvx to conveniently formulate and solve constrained norm minimization, entropy maximization, determinant maximization, and many other problems.

To use cvx effectively, you need to know at least a bit about convex optimization. For background on convex optimization, see the book *Convex Optimization* [BV04], available on-line at `www.stanford.edu/~boyd/cvxbook/`, or the Stanford course EE364A, available at `www.stanford.edu/class/ee364a/`.

cvx is implemented in Matlab [Mat04], effectively turning Matlab into an optimization modeling language. Model specifications are constructed using common Matlab operations and functions, and standard Matlab code can be freely mixed with these specifications. This combination makes it simple to perform the calculations needed to form optimization problems, or to process the results obtained from their solution. For example, it is easy to compute an optimal trade-off curve by forming and solving a family of optimization problems by varying the constraints. As another example, cvx can be used as a component of a larger system that uses convex optimization, such as a branch and bound method, or an engineering design framework.

cvx also provides special modes to simplify the construction of problems from two specific problem classes. In *SDP mode*, cvx applies a matrix interpretation to the inequality operator, so that *linear matrix inequalities* (LMIs) and SDPs may be expressed in a more natural form. In *GP mode*, cvx accepts all of the special functions and combination rules of geometric programming, including monomials, posynomials, and generalized posynomials, and transforms such problems into convex form so that they can be solved efficiently. For background on geometric programming, see the tutorial paper [BKVH05], available at `www.stanford.edu/~boyd/papers/gp_tutorial.html`.

cvx was designed by Michael Grant and Stephen Boyd, with input from Yinyu Ye; and was implemented by Michael Grant [GBY06]. It incorporates ideas from earlier work by Löfberg [Löf05], Dahl and Vandenberghe [DV05], Crusius [Cru02], Wu and Boyd [WB00], and many others. The modeling language follows the spirit of AMPL [FGK99] or GAMS [BKMR98]; unlike these packages, however, cvx was designed from the beginning to fully exploit convexity. The specific method for implementing cvx in Matlab draws heavily from YALMIP [Löf05]. We also hope to develop versions of cvx for other platforms in the future.

1.2 What is disciplined convex programming?

Disciplined convex programming is a methodology for constructing convex optimization problems proposed by Michael Grant, Stephen Boyd, and Yinyu Ye [GBY06, Gra04]. It is meant to support the formulation and construction of optimization problems that the user intends *from the outset* to be convex. Disciplined convex programming imposes a set of conventions or rules, which we call the *DCP ruleset*. Problems which adhere to the ruleset can be rapidly and automatically verified as convex and converted to solvable form. Problems that violate the ruleset are rejected, even when the problem is convex. That is not to say that such problems cannot be solved using DCP; they just need to be rewritten in a way that conforms to the DCP ruleset.

A detailed description of the DCP ruleset is given in §4, and it is important for anyone who intends to actively use `cvx` to understand it. The ruleset is simple to learn, and is drawn from basic principles of convex analysis. In return for accepting the restrictions imposed by the ruleset, we obtain considerable benefits, such as automatic conversion of problems to solvable form, and full support for nondifferentiable functions. In practice, we have found that disciplined convex programs closely resemble their natural mathematical forms.

1.3 About this version

Supported solvers. This version of `cvx` supports two core solvers, SeDuMi [Stu99] and SDPT3 [TTT06], which is the default. Future versions of `cvx` may support other solvers, such as MOSEK [MOS05] or CVXOPT [DV05]. SeDuMi and SDPT3 are open-source interior-point solvers written in Matlab for LPs, SOCPs, SDPs, and combinations thereof.

Problems handled exactly. `cvx` will convert the specified problem to an LP, SOCP, or SDP, when all the functions in the problem specification can be represented in these forms. This includes a wide variety of functions, such as minimum and maximum, absolute value, quadratic forms, the minimum and maximum eigenvalues of a symmetric matrix, power functions x^p, and ℓ_p norms (both for p rational).

Problems handled with (good) approximations. For a few functions, `cvx` will make a (good) approximation to transform the specified problem to one that can be handled by a combined LP, SOCP, and SDP solver. For example, when a power function or ℓ_p-norm is used, with non-rational exponent p, `cvx` replaces p with a nearby rational. The log of the normal cumulative distribution $\log \Phi(x)$ is replaced with an SDP-compatible approximation.

Problems handled with successive approximation. This version of `cvx` adds support for a number of functions that cannot be exactly represented via LP, SOCP, or SDP, including log, exp, log-sum-exp $\log(\exp x_1 + \cdots + \exp x_n)$, entropy, and Kullback-Leibler divergence. These problems are handled by solving a sequence (typically just

a handful) of SDPs, which yields the solution to the full accuracy of the core solver. On the other hand, this technique can be *substantially slower* than if the core solver directly handled such functions. The successive approximation method is briefly described in Appendix D.1. Geometric problems are now solved in this manner as well; in previous versions, an approximation was made.

Ultimately, we will interface `cvx` to a solver with native support for such functions, which result in a large speedup in solving problems with these functions. Until then, users should be aware that problems involving these functions can be slow to solve using the current version of `cvx`. For this reason, when one of these functions is used, the user will be warned that the successive approximate technique will be used.

We emphasize that most users do not need to know how `cvx` handles their problem; what matters is what functions and operations can be handled. For a full list of functions supported by `cvx`, see Appendix B, or use the online help function by typing `help cvx/builtins` (for functions already in Matlab, such as `sqrt` or `log`) or `help cvx/functions` (for functions not in Matlab, such as `lambda_max`).

1.4 Feedback

Please contact Michael Grant (`mcgrant@stanford.edu`) or Stephen Boyd (`boyd@stanford.edu`) with your comments. If you discover what you think is a bug, please include the following in your communication, so we can reproduce and fix the problem:

- the `cvx` model and supporting data that caused the error
- a copy of any error messages that it produced
- the `cvx` version number and build number
- the version number of Matlab that you are running
- the name and version of the operating system you are using

The latter three items can all be discovered by typing

```
cvx_version
```

at the MATLAB command prompt; simply copy its output into your email message.

1.5 What `cvx` is *not*

`cvx` is *not* meant to be a tool for checking if your problem is convex. You need to know a bit about convex optimization to effectively use `cvx`; otherwise you are the proverbial monkey at the typewriter, hoping to (accidentally) type in a valid disciplined convex program.

On the other hand, if `cvx` accepts your problem, you can be sure it is convex. In conjunction with a course on (or self study of) convex optimization, `cvx` (especially, its error messages) can be very helpful in learning some basic convex analysis. While

`cvx` will attempt to give helpful error messages when you violate the DCP ruleset, it can sometimes give quite obscure error messages.

`cvx` is *not* meant for very large problems, so if your problem is very large (for example, a large image processing problem), `cvx` is unlikely to work well (or at all). For such problems you will likely need to directly call a solver, or to develop your own methods, to get the efficiency you need.

For such problems `cvx` can play an important role, however. Before starting to develop a specialized large-scale method, you can use `cvx` to solve scaled-down or simplified versions of the problem, to rapidly experiment with exactly what problem you want to solve. For image reconstruction, for example, you might use `cvx` to experiment with different problem formulations on 50×50 pixel images.

`cvx` *will* solve many medium and large scale problems, provided they have exploitable structure (such as sparsity), and you avoid `for` loops, which can be slow in Matlab, and functions like log and exp that require successive approximation. If you encounter difficulties in solving large problem instances, please do contact us; we may be able to suggest an equivalent formulation that `cvx` can process more efficiently.

2 A quick start

Once you have installed `cvx` (see §A), you can start using it by entering a `cvx` *specification* into a Matlab script or function, or directly from the command prompt. To delineate `cvx` specifications from surrounding Matlab code, they are preceded with the statement `cvx_begin` and followed with the statement `cvx_end`. A specification can include any ordinary Matlab statements, as well as special `cvx`-specific commands for declaring primal and dual optimization variables and specifying constraints and objective functions.

Within a `cvx` specification, optimization variables have no numerical value; instead, they are special Matlab objects. This enables Matlab to distinguish between ordinary commands and `cvx` objective functions and constraints. As Matlab reads a `cvx` specification, it builds an internal representation of the optimization problem. If it encounters a violation of the rules of disciplined convex programming (such as an invalid use of a composition rule or an invalid constraint), an error message is generated. When Matlab reaches the `cvx_end` command, it completes the conversion of the `cvx` specification to a canonical form, and calls the underlying core solver to solve it.

If the optimization is successful, the optimization variables declared in the `cvx` specification are converted from objects to ordinary Matlab numerical values that can be used in any further Matlab calculations. In addition, `cvx` also assigns a few other related Matlab variables. One, for example, gives the status of the problem (*i.e.*, whether an optimal solution was found, or the problem was determined to be infeasible or unbounded). Another gives the optimal value of the problem. Dual variables can also be assigned.

This processing flow will become more clear as we introduce a number of simple examples. We invite the reader to actually follow along with these examples in Matlab, by running the `quickstart` script found in the `examples` subdirectory of the `cvx` distribution. For example, if you are on Windows, and you have installed the `cvx` distribution in the directory `D:\Matlab\cvx`, then you would type

```
cd D:\Matlab\cvx\examples
quickstart
```

at the Matlab command prompt. The script will automatically print key excerpts of its code, and pause periodically so you can examine its output. (Pressing "Enter" or "Return" resumes progress.) The line numbers accompanying the code excerpts in this document correspond to the line numbers in the file `quickstart.m`.

2.1 Least-squares

We first consider the most basic convex optimization problem, least-squares. In a least-squares problem, we seek $x \in \mathbf{R}^n$ that minimizes $\|Ax - b\|_2$, where $A \in \mathbf{R}^{m \times n}$ is skinny and full rank (*i.e.*, $m \geq n$ and $\mathbf{Rank}(A) = n$). Let us create some test problem data for m, n, A, and b in Matlab:

```
15 m = 16; n = 8;
16 A = randn(m,n);
17 b = randn(m,1);
```

(We chose small values of m and n to keep the output readable.) Then the least-squares solution $x = (A^T A)^{-1} A^T b$ is easily computed using the backslash operator:

```
20 x_ls = A \ b;
```

Using `cvx`, the same problem can be solved as follows:

```
23 cvx_begin
24     variable x(n);
25     minimize( norm(A*x-b) );
26 cvx_end
```

(The indentation is used for purely stylistic reasons and is optional.) Let us examine this specification line by line:

- Line 23 creates a placeholder for the new `cvx` specification, and prepares Matlab to accept variable declarations, constraints, an objective function, and so forth.
- Line 24 declares `x` to be an optimization variable of dimension n. `cvx` requires that all problem variables be declared before they are used in an objective function or constraints.
- Line 25 specifies an objective function to be minimized; in this case, the Euclidean or ℓ_2-norm of $Ax - b$.
- Line 26 signals the end of the `cvx` specification, and causes the problem to be solved.

The backslash form is clearly simpler—there is no reason to use `cvx` to solve a simple least-squares problem. But this example serves as sort of a "Hello world!" program in `cvx`; *i.e.*, the simplest code segment that actually does something useful.

If you were to type `x` at the Matlab prompt after line 24 but before the `cvx_end` command, you would see something like this:

```
x =
    cvx affine expression (8x1 vector)
```

That is because within a specification, variables have no numeric value; rather, they are Matlab objects designed to represent problem variables and expressions involving them. Similarly, because the objective function `norm(A*x-b)` involves a `cvx` variable, it does not have a numeric value either; it is also represented by a Matlab object.

When Matlab reaches the `cvx_end` command, the least-squares problem is solved, and the Matlab variable `x` is overwritten with the solution of the least-squares problem, *i.e.*, $(A^T A)^{-1} A^T b$. Now `x` is an ordinary length-n numerical vector, identical to what would be obtained in the traditional approach, at least to within the accuracy of the solver. In addition, two additional Matlab variables are created:

- `cvx_optval`, which contains the value of the objective function; *i.e.*, $\|Ax-b\|_2$;
- `cvx_status`, which contains a string describing the status of the calculation. In this case, `cvx_status` would contain the string `Solved`. See Appendix C for a list of the possible values of `cvx_status` and their meaning.
- `cvx_slvtol`: the tolerance level achieved by the solver.
- `cvx_slvitr`: the number of iterations taken by the solver.

All of these quantities, `x`, `cvx_optval`, and `cvx_status`, *etc.* may now be freely used in other Matlab statements, just like any other numeric or string values.[1]

There is not much room for error in specifying a simple least-squares problem, but if you make one, you will get an error or warning message. For example, if you replace line 25 with

```
maximize( norm(A*x-b) );
```

which asks for the norm to be maximized, you will get an error message stating that a convex function cannot be maximized (at least in disciplined convex programming):

```
??? Error using ==> maximize
Disciplined convex programming error:
Objective function in a maximization must be concave.
```

2.2 Bound-constrained least-squares

Suppose we wish to add some simple upper and lower bounds to the least-squares problem above: *i.e.*, we wish to solve

$$\begin{array}{ll} \text{minimize} & \|Ax-b\|_2 \\ \text{subject to} & l \preceq x \preceq u, \end{array} \tag{1}$$

where l and u are given data, vectors with the same dimension as the variable x. The vector inequality $u \preceq v$ means componentwise, *i.e.*, $u_i \leq v_i$ for all i. We can no longer use the simple backslash notation to solve this problem, but it can be transformed into a quadratic program (QP), which can be solved without difficulty if you have some form of QP software available.

Let us provide some numeric values for `l` and `u`:

```
47 bnds = randn(n,2);
48 l = min( bnds, [], 2 );
49 u = max( bnds, [], 2 );
```

Then if you have the Matlab Optimization Toolbox [Mat05], you can use the `quadprog` function to solve the problem as follows:

[1] If you type `who` or `whos` at the command prompt, you may see other, unfamiliar variables as well. Any variable that begins with the prefix `cvx_` is reserved for internal use by `cvx` itself, and should not be changed.

```
53 x_qp = quadprog( 2*A'*A, -2*A'*b, [], [], [], [], l, u );
```

This actually minimizes the square of the norm, which is the same as minimizing the norm itself. In contrast, the `cvx` specification is given by

```
59 cvx_begin
60     variable x(n);
61     minimize( norm(A*x-b) );
62     subject to
63         x >= l;
64         x <= u;
65 cvx_end
```

Three new lines of `cvx` code have been added to the `cvx` specification:

- The `subject to` statement on line 62 does nothing—`cvx` provides this statement simply to make specifications more readable. It is entirely optional.
- Lines 63 and 64 represent the $2n$ inequality constraints $l \preceq x \preceq u$.

As before, when the `cvx_end` command is reached, the problem is solved, and the numerical solution is assigned to the variable `x`. Incidentally, `cvx` will *not* transform this problem into a QP by squaring the objective; instead, it will transform it into an SOCP. The result is the same, and the transformation is done automatically.

In this example, as in our first, the `cvx` specification is longer than the Matlab alternative. On the other hand, it is easier to read the `cvx` version and relate it to the original problem. In contrast, the `quadprog` version requires us to know in advance the transformation to QP form, including the calculations such as `2*A'*A` and `-2*A'*b`. For all but the simplest cases, a `cvx` specification is simpler, more readable, and more compact than equivalent Matlab code to solve the same problem.

2.3 Other norms and functions

Now let us consider some alternatives to the least-squares problem. Norm minimization problems involving the ℓ_∞ or ℓ_1 norms can be reformulated as LPs, and solved using a linear programming solver such as `linprog` in the Matlab Optimization Toolbox (see, *e.g.*, [BV04, §6.1]). However, because these norms are part of `cvx`'s base library of functions, `cvx` can handle these problems directly.

For example, to find the value of x that minimizes the Chebyshev norm $\|Ax-b\|_\infty$, we can employ the `linprog` command from the Matlab Optimization Toolbox:

```
97  f    = [ zeros(n,1); 1          ];
98  Ane  = [ +A,         -ones(m,1)  ; ...
99           -A,         -ones(m,1) ];
100 bne  = [ +b;         -b         ];
101 xt   = linprog(f,Ane,bne);
102 x_cheb = xt(1:n,:);
```

With `cvx`, the same problem is specified as follows:

```
cvx_begin
    variable x(n);
    minimize( norm(A*x-b,Inf) );
cvx_end
```

The code based on `linprog`, and the `cvx` specification above will both solve the Chebyshev norm minimization problem, *i.e.*, each will produce an x that minimizes $\|Ax-b\|_\infty$. Chebyshev norm minimization problems can have multiple optimal points, however, so the particular x's produced by the two methods can be different. The two points, however, must have the same value of $\|Ax-b\|_\infty$.

Similarly, to minimize the ℓ_1 norm $\|\cdot\|_1$, we can use `linprog` as follows:

```
f    = [ zeros(n,1); ones(m,1);  ones(m,1)  ];
Aeq  = [ A,          -eye(m),    +eye(m)    ];
lb   = [ -Inf(n,1);  zeros(m,1); zeros(m,1) ];
xzz  = linprog(f,[],[],Aeq,b,lb,[]);
x_l1 = xzz(1:n,:);
```

The `cvx` version is, not surprisingly,

```
cvx_begin
    variable x(n);
    minimize( norm(A*x-b,1) );
cvx_end
```

`cvx` automatically transforms both of these problems into LPs, not unlike those generated manually for `linprog`.

The advantage that automatic transformation provides is magnified if we consider functions (and their resulting transformations) that are less well-known than the ℓ_∞ and ℓ_1 norms. For example, consider the norm

$$\|Ax-b\|_{\mathrm{lgst},k} = |Ax-b|_{[1]} + \cdots + |Ax-b|_{[k]},$$

where $|Ax-b|_{[i]}$ denotes the ith largest element of the absolute values of the entries of $Ax-b$. This is indeed a norm, albeit a fairly esoteric one. (When $k=1$, it reduces to the ℓ_∞ norm; when $k=m$, the dimension of $Ax-b$, it reduces to the ℓ_1 norm.) The problem of minimizing $\|Ax-b\|_{\mathrm{lgst},k}$ over x can be cast as an LP, but the transformation is by no means obvious so we will omit it here. But this norm is provided in the base `cvx` library, and has the name `norm_largest`, so to specify and solve the problem using `cvx` is easy:

```
k = 5;
cvx_begin
    variable x(n);
    minimize( norm_largest(A*x-b,k) );
cvx_end
```

Unlike the ℓ_1, ℓ_2, or ℓ_∞ norms, this norm is not part of the standard Matlab distribution. Once you have installed `cvx`, though, the norm is available as an ordinary Matlab function outside a `cvx` specification. For example, once the code above is processed, `x` is a numerical vector, so we can type

```
cvx_optval
norm_largest(A*x-b,k)
```

The first line displays the optimal value as determined by `cvx`; the second recomputes the same value from the optimal vector `x` as determined by `cvx`.

The list of supported nonlinear functions in `cvx` goes well beyond `norm` and `norm_largest`. For example, consider the Huber penalty minimization problem

$$\text{minimize} \quad \sum_{i=1}^m \phi((Ax-b)_i),$$

with variable $x \in \mathbf{R}^n$, where ϕ is the Huber penalty function

$$\phi(z) = \begin{cases} |z|^2 & |z| \leq 1 \\ 2|z|-1 & |z| \geq 1. \end{cases}$$

The Huber penalty function is convex, and has been provided in the `cvx` function library. So solving the Huber penalty minimization problem in `cvx` is simple:

```
cvx_begin
    variable x(n);
    minimize( sum(huber(A*x-b)) );
cvx_end
```

`cvx` automatically transforms this problem into an SOCP, which the core solver then solves. (The `cvx` user, however, does not need to know how the transformation is carried out.)

2.4 Other constraints

We hope that, by now, it is not surprising that adding the simple bounds $l \preceq x \preceq u$ to the problems in §2.3 above is as simple as inserting the lines

```
x >= l;
x <= u;
```

before the `cvx_end` statement in each `cvx` specification. In fact, `cvx` supports more complex constraints as well. For example, let us define new matrices `C` and `d` in Matlab as follows,

```
p = 4;
C = randn(p,n);
d = randn(p,1);
```

Now let us add an equality constraint and a nonlinear inequality constraint to the original least-squares problem:

```
232 cvx_begin
233     variable x(n);
234     minimize( norm(A*x-b) );
235     subject to
236         C*x == d;
237         norm(x,Inf) <= 1;
238 cvx_end
```

Both of the added constraints conform to the DCP rules, and so are accepted by `cvx`. After the `cvx_end` command, `cvx` converts this problem to an SOCP, and solves it.

Expressions using comparison operators (`==`, `>=`, *etc.*) behave quite differently when they involve `cvx` optimization variables, or expressions constructed from `cvx` optimization variables, than when they involve simple numeric values. For example, because `x` is a declared variable, the expression `C*x==d` in line 236 above causes a constraint to be included in the `cvx` specification, and returns no value at all. On the other hand, outside of a `cvx` specification, if `x` has an appropriate numeric value—for example immediately after the `cvx_end` command—that same expression would return a vector of `1`s and `0`s, corresponding to the truth or falsity of each equality.[2] Likewise, within a `cvx` specification, the statement `norm(x,Inf)<=1` adds a nonlinear constraint to the specification; outside of it, it returns a `1` or a `0` depending on the numeric value of `x` (specifically, whether its ℓ_∞-norm is less than or equal to, or more than, 1).

Because `cvx` is designed to support convex optimization, it must be able to verify that problems are convex. To that end, `cvx` adopts certain construction rules that govern how constraint and objective expressions are constructed. For example, `cvx` requires that the left- and right- hand sides of an equality constraint be affine. So a constraint such as

```
norm(x,Inf) == 1;
```

results in the following error:

```
??? Error using ==> cvx.eq
Disciplined convex programming error:
Both sides of an equality constraint must be affine.
```

Inequality constraints of the form $f(x) \leq g(x)$ or $g(x) \geq f(x)$ are accepted only if f can be verified as convex and g verified as concave. So a constraint such as

```
norm(x,Inf) >= 1;
```

[2] In fact, immediately after the `cvx_end` command above, you would likely find that most if not all of the values returned would be `0`. This is because, as is the case with many numerical algorithms, solutions are determined only to within some nonzero numeric tolerance. So the equality constraints will be satisfied closely, but often not exactly.

results in the following error:

```
??? Error using ==> cvx.ge
Disciplined convex programming error:
The left-hand side of a ">=" inequality must be concave.
```

The specifics of the construction rules are discussed in more detail in §4 below. These rules are relatively intuitive if you know the basics of convex analysis and convex optimization.

2.5 An optimal trade-off curve

For our final example in this section, let us show how traditional Matlab code and `cvx` specifications can be mixed to form and solve multiple optimization problems. The following code solves the problem of minimizing $\|Ax - b\|_2 + \gamma\|x\|_1$, for a logarithmically spaced vector of (positive) values of γ. This gives us points on the optimal trade-off curve between $\|Ax - b\|_2$ and $\|x\|_1$. An example of this curve is given in Figure 1.

```
gamma = logspace( -2, 2, 20 );
l2norm = zeros(size(gamma));
l1norm = zeros(size(gamma));
fprintf( 1, '   gamma       norm(x,1)    norm(A*x-b)\n' );
fprintf( 1, '---------------------------------------\n' );
for k = 1:length(gamma),
    fprintf( 1, '%8.4e', gamma(k) );
    cvx_begin
        variable x(n);
        minimize( norm(A*x-b)+gamma(k)*norm(x,1) );
    cvx_end
    l1norm(k) = norm(x,1);
    l2norm(k) = norm(A*x-b);
    fprintf( 1, '   %8.4e   %8.4e\n', l1norm(k), l2norm(k) );
end
plot( l1norm, l2norm );
xlabel( 'norm(x,1)' );
ylabel( 'norm(A*x-b)' );
grid
```

Line 277 of this code segment illustrates one of the construction rules to be discussed in §4 below. A basic principle of convex analysis is that a convex function can be multiplied by a nonnegative scalar, or added to another convex function, and the result is then convex. `cvx` recognizes such combinations and allows them to be used anywhere a simple convex function can be—such as an objective function to be minimized, or on the appropriate side of an inequality constraint. So in our example, the expression

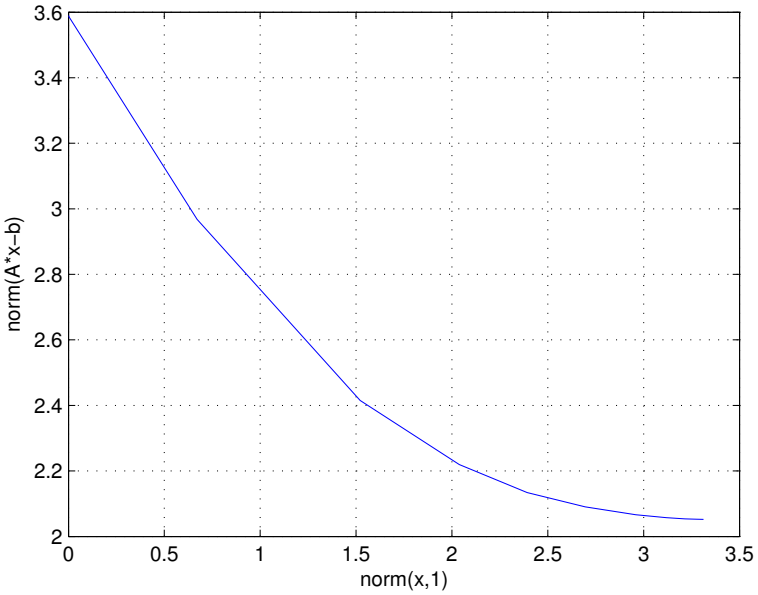

Figure 1: An example trade-off curve from the `quickstart` demo, lines 268-286.

```
norm(A*x-b)+gamma(k)*norm(x,1)
```

on line 277 is recognized as convex by `cvx`, as long as `gamma(k)` is positive or zero. If `gamma(k)` were negative, then this expression becomes the sum of a convex term and a concave term, which causes `cvx` to generate the following error:

```
??? Error using ==> cvx.plus
Disciplined convex programming error:
Addition of convex and concave terms is forbidden.
```

3 The basics

3.1 `cvx_begin` and `cvx_end`

All `cvx` models must be preceded by the command `cvx_begin` and terminated with the command `cvx_end`. All variable declarations, objective functions, and constraints should fall in between.

The `cvx_begin` command accepts several modifiers that you may find useful. For instance, `cvx_begin quiet` prevents the model from producing any screen output while it is being solved. Additionally, `cvx_begin sdp` and `cvx_begin gp` invoke special semidefinite and geometric programming modes, discussed in §6 and §7, respectively. These modifiers may be combined when appropriate; for instance, `cvx_begin sdp quiet` invokes SDP mode and silences the solver output.

3.2 Data types for variables

As mentioned above, all variables must be declared using the `variable` command (or the `variables` command; see below) before they can be used in constraints or an objective function.

Variables can be real or complex; and scalar, vector, matrix, or n-dimensional arrays. In addition, matrices can have *structure* as well, such as symmetry or bandedness. The structure of a variable is given by supplying a list of descriptive keywords after the name and size of the variable. For example, the code segment

```
variable w(50) complex;
variable X(20,10);
variable Y(50,50) symmetric;
variable Z(100,100) hermitian toeplitz;
```

(inside a `cvx` specification) declares that `w` is a complex 50-element vector variable, `X` is a real 20×10 matrix variable, `Y` is a real 50×50 symmetric matrix variable, and `Z` is a complex 100×100 Hermitian Toeplitz matrix variable. The structure keywords can be applied to n-dimensional arrays as well: each 2-dimensional "slice" of the array is given the stated structure. The currently supported structure keywords are:

```
banded(lb,ub)  complex  diagonal  hankel  hermitian  lower_bidiagonal
 lower_hessenberg  lower_triangular  scaled_identity  skew_symmetric
  symmetric  toeplitz  tridiagonal  upper_bidiagonal  upper_hankel
                upper_hessenberg  upper_triangular
```

With a couple of exceptions, the structure keywords are self-explanatory:

- `banded(lb,ub)`: the matrix is banded with a lower bandwidth `lb` and an upper bandwidth `ub`. If both `lb` and `ub` are zero, then a diagonal matrix results. `ub` can be omitted, in which case it is set equal to `lb`. For example, `banded(1,1)` (or `banded(1)`) is a tridiagonal matrix.

- `scaled_identity`: the matrix is a (variable) multiple of the identity matrix. This is the same as declaring it to be diagonal and Toeplitz.

- `upper_hankel`: The matrix is Hankel (*i.e.*, constant along antidiagonals), and zero below the central antidiagonal, *i.e.*, for $i + j > n + 1$.

When multiple keywords are supplied, the resulting matrix structure is determined by intersection; if the keywords conflict, then an error will result.

A `variable` statement can be used to declare only a single variable, which can be a bit inconvenient if you have a lot of variables to declare. For this reason, the `variables` statement is provided which allows you to declare multiple variables; *i.e.*,

```
variables x1 x2 x3 y1(10) y2(10,10,10);
```

The one limitation of the `variables` command is that it cannot declare complex or structured arrays (*e.g.*, `symmetric`, *etc.*). These must be declared one at a time, using the singular `variable` command.

3.3 Objective functions

Declaring an objective function requires the use of the `minimize` or `maximize` function, as appropriate. (For the benefit of our users with a British influence, the synonyms `minimise` and `maximise` are provided as well.) The objective function in a call to `minimize` must be convex; the objective function in a call to `maximize` must be concave. At most one objective function may be declared in a given `cvx` specification, and the objective function must have a scalar value. (For the only exception to this rule, see the section on defining new functions in §5).

If no objective function is specified, the problem is interpreted as a feasibility problem, which is the same as performing a minimization with the objective function set to zero. In this case, `cvx_optval` is either `0`, if a feasible point is found, or `+Inf`, if the constraints are not feasible.

3.4 Constraints

The following constraint types are supported in `cvx`:

- Equality `==` constraints, where both the left- and right-hand sides are affine functions of the optimization variables.

- Less-than `<=`, `<` inequality constraints, where the left-hand expression is convex, and the right-hand expression is concave.

- Greater-than `>=`, `>` constraints, where the left-hand expression is concave, and the right-hand expression is convex.

In `cvx`, the strict inequalities `<` and `>` are accepted, but interpreted as the associated nonstrict inequalities, `<=` and `>=`, respectively. We encourage you to use the nonstrict

forms `<=` and `>=`, since they are mathematically correct. (Future versions of `cvx` might assign a slightly different meaning to strict inequalities.)

These equality and inequality operators work for arrays. When both sides of the constraint are arrays of the same size, the constraint is imposed elementwise. For example, if `a` and `b` are $m \times n$ matrices, then `a<=b` is interpreted by `cvx` as mn (scalar) inequalities, *i.e.*, each entry of `a` must be less than or equal to the corresponding entry of `b`. `cvx` also handles cases where one side is a scalar and the other is an array. This is interpreted as a constraint for each element of the array, with the (same) scalar appearing on the other side. As an example, if `a` is an $m \times n$ matrix, then `a>=0` is interpreted as mn inequalities: each element of the matrix must be nonnegative.

Note also the important distinction between `=`, which is an assignment, and `==`, which imposes an equality constraint (inside a `cvx` specification); for more on this distinction, see §8.4. Also note that the non-equality operator `~=` may *not* be used in a constraint; in any case, such constraints are rarely convex. Inequalities cannot be used if either side is complex.

`cvx` also supports a *set membership* constraint; see §3.6.

3.5 Functions

The base `cvx` function library includes a variety of convex, concave, and affine functions which accept `cvx` variables or expressions as arguments. Many are common Matlab functions such as `sum`, `trace`, `diag`, `sqrt`, `max`, and `min`, re-implemented as needed to support `cvx`; others are new functions not found in Matlab. A complete list of the functions in the base library can be found in §B. It's also possible to add your own new functions; see §5.

An example of a function in the base library is `quad_over_lin`, which represents the quadratic-over-linear function, defined as $f(x, y) = x^T x/y$, with domain $\mathbf{R}^n \times \mathbf{R}_{++}$, *i.e.*, x is an arbitrary vector in $\mathbf{R}^n$, and y is a positive scalar. (The function also accepts complex x, but we'll consider real x to keep things simple.) The quadratic-over-linear function is convex in x and y, and so can be used as an objective, in an appropriate constraint, or in a more complicated expression. We can, for example, minimize the quadratic-over-linear function of $(Ax - b, c^T x + d)$ using

```
minimize( quad_over_lin( A*x-b, c'*x+d ) );
```

inside a `cvx` specification, assuming `x` is a vector optimization variable, `A` is a matrix, `b` and `c` are vectors, and `d` is a scalar. `cvx` recognizes this objective expression as a convex function, since it is the composition of a convex function (the quadratic-over-linear function) with an affine function.

You can also use the function `quad_over_lin` *outside* a `cvx` specification. In this case, it just computes its (numerical) value, given (numerical) arguments. It's not quite the same as the expression `((A*x-b)'*(A*x-b))/(c'*x+d)`, however. This expression makes sense, and returns a real number, when $c^T x + d$ is negative; but `quad_over_lin(A*x-b,c'*x+d)` returns `+Inf` if $c^T x + d \not> 0$.

3.6 Sets

`cvx` supports the definition and use of convex sets. The base library includes the cone of positive semidefinite $n \times n$ matrices, the second-order or Lorentz cone, and various norm balls. A complete list of sets supplied in the base library is given in §B.

Unfortunately, the Matlab language does not have a set membership operator, such as `x in S`, to denote $x \in S$. So in `cvx`, we use a slightly different syntax to require that an expression is in a set. To represent a set we use a *function* that returns an unnamed variable that is required to be in the set. Consider, for example, $\mathbf{S}_+^n$, the cone of symmetric positive semidefinite $n \times n$ matrices. In `cvx`, we represent this by the function `semidefinite(n)`, which returns an unnamed new variable, that is constrained to be positive semidefinite. To require that the matrix expression `X` be symmetric positive semidefinite, we use the syntax `X == semidefinite(n)`. The literal meaning of this is that `X` is constrained to be equal to some unnamed variable, which is required to be an $n \times n$ symmetric positive semidefinite matrix. This is, of course, equivalent to saying that `X` must be symmetric positive semidefinite.

As an example, consider the constraint that a (matrix) variable `X` is a correlation matrix, *i.e.*, it is symmetric, has unit diagonal elements, and is positive semidefinite. In `cvx` we can declare such a variable and impose such constraints using

```
variable X(n,n) symmetric;
X == semidefinite(n);
diag(X) == ones(n,1);
```

The second line here imposes the constraint that `X` be positive semidefinite. (You can read '`==`' here as 'is', so the second line can be read as '`X` is positive semidefinite'.) The lefthand side of the third line is a vector containing the diagonal elements of `X`, whose elements we require to be equal to one. Incidentally, `cvx` allows us to simplify the third line to

```
diag(X) == 1;
```

because `cvx` follows the Matlab convention of handling array/scalar comparisons by comparing each element of the array independently with the scalar.

If this use of equality constraints to represent set membership remains confusing or simply aesthetically displeasing, we have create a "pseudo-operator" `<In>` that you can use in its place. So, for example, the semidefinite constraint above can be replaced by

```
X <In> semidefinite(n);
```

This is exactly equivalent to using the equality constraint operator, but if you find it more pleasing, feel free to use it. Implementing this operator required some Matlab trickery, so don't expect to be able to use it outside of `cvx` models.

Sets can be combined in affine expressions, and we can constrain an affine expression to be in a convex set. For example, we can impose constraints of the form

```
A*X*A'-X == B*semidefinite(n)*B';
```

where `X` is an $n \times n$ symmetric variable matrix, and `A` and `B` are $n \times n$ constant matrices. This constraint requires that $AXA^T - X = BYB^T$, for some $Y \in \mathbf{S}^n_+$.

`cvx` also supports sets whose elements are ordered lists of quantities. As an example, consider the second-order or Lorentz cone,

$$\mathbf{Q}^m = \{\, (x, y) \in \mathbf{R}^m \times \mathbf{R} \;\mid\; \|x\|_2 \leq y \,\} = \mathbf{epi}\, \|\cdot\|_2, \tag{2}$$

where **epi** denotes the epigraph of a function. An element of $\mathbf{Q}^m$ is an ordered list, with two elements: the first is an m-vector, and the second is a scalar. We can use this cone to express the simple least-squares problem from §2.1 (in a fairly complicated way) as follows:

$$\begin{array}{ll} \text{minimize} & y \\ \text{subject to} & (Ax - b, y) \in \mathbf{Q}^m. \end{array} \tag{3}$$

`cvx` uses Matlab's cell array facility to mimic this notation:

```
cvx_begin
    variables x(n) y;
    minimize( y );
    subject to
        { A*x-b, y } <In> lorentz(m);
cvx_end
```

The function call `lorentz(m)` returns an unnamed variable (*i.e.*, a pair consisting of a vector and a scalar variable), constrained to lie in the Lorentz cone of length `m`. So the constraint in this specification requires that the pair `{ A*x-b, y }` lies in the appropriately-sized Lorentz cone.

3.7 Dual variables

When a disciplined convex program is solved, the associated *dual problem* is also solved. (In this context, the original problem is called the *primal problem*.) The optimal dual variables, each of which is associated with a constraint in the original problem, give valuable information about the original problem, such as the sensitivities with respect to perturbing the constraints [BV04, Ch.5]. To get access to the optimal dual variables in `cvx`, you simply declare them, and associate them with the constraints. Consider, for example, the LP

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \preceq b, \end{array}$$

with variable $x \in \mathbf{R}^n$, and m inequality constraints. The dual of this problem is

$$\begin{array}{ll} \text{maximize} & -b^T y \\ \text{subject to} & c + A^T y = 0 \\ & y \succeq 0, \end{array}$$

where the dual variable y is associated with the inequality constraint $Ax \preceq b$ in the original LP. To represent the primal problem and this dual variable in `cvx`, we use the following syntax:

```
n = size(A,2);
cvx_begin
    variable x(n);
    dual variable y;
    minimize( c' * x );
    subject to
        y : A * x <= b;
cvx_end
```

The line

```
dual variable y
```

tells `cvx` that `y` will represent the dual variable, and the line

```
y : A * x <= b;
```

associates it with the inequality constraint. Notice how the colon `:` operator is being used in a different manner than in standard Matlab, where it is used to construct numeric sequences like `1:10`. This new behavior is in effect only when a dual variable is present, so there should be no confusion or conflict. No dimensions are given for `y`; they are automatically determined from the constraint with which it is associated. For example, if $m = 20$, typing `y` at the Matlab command prompt immediately before `cvx_end` yields

```
y =
    cvx dual variable (20x1 vector)
```

It is not necessary to place the dual variable on the left side of the constraint; for example, the line above can also be written in this way:

```
A * x <= b : y;
```

In addition, dual variables for inequality constraints will always be nonnegative, which means that the sense of the inequality can be reversed without changing the dual variable's value; *i.e.*,

```
b >= A * x : y;
```

yields an identical result. For *equality* constraints, on the other hand, swapping the left- and right- hand sides of an equality constraint will *negate* the optimal value of the dual variable.

After the `cvx_end` statement is processed, and assuming the optimization was successful, `cvx` assigns numerical values to `x` and `y`—the optimal primal and dual variable values, respectively. Optimal primal and dual variables for this LP must satisfy the *complementary slackness conditions*

$$y_i(b - Ax)_i = 0, \quad i = 1, \ldots, m. \tag{4}$$

You can check this in Matlab with the line

```
y .* (b-A*x)
```

which prints out the products of the entries of `y` and `b-A*x`, which should be nearly zero. This line must be executed *after* the `cvx_end` command (which assigns numerical values to `x` and `y`); it will generate an error if it is executed inside the `cvx` specification, where `y` and `b-A*x` are still just abstract expressions.

If the optimization is *not* successful, because either the problem is infeasible or unbounded, then `x` and `y` will have different values. In the unbounded case, `x` will contain an *unbounded direction*; *i.e.*, a point x satisfying

$$c^T x = -1, \quad Ax \preceq 0, \tag{5}$$

and `y` will be filled with `NaN` values, reflecting the fact that the dual problem is infeasible. In the infeasible case, `x` is filled with `NaN` values, while `y` contains an *unbounded dual direction*; *i.e.*, a point y satisfying

$$b^T y = -1, \quad A^T y = 0, \quad y \succeq 0 \tag{6}$$

Of course, the precise interpretation of primal and dual points and/or directions depends on the structure of the problem. See references such as [BV04] for more on the interpretation of dual information.

`cvx` also supports the declaration of *indexed* dual variables. These prove useful when the *number* of constraints in a model (and, therefore, the number of dual variables) depends upon the parameters themselves. For more information on indexed dual variables, see §8.5.

3.8 Expression holders

Sometimes it is useful to store a `cvx` expression into a Matlab variable for future use. For instance, consider the following `cvx` script:

```
variables x y
z = 2 * x - y;
square( z ) <= 3;
quad_over_lin( x, z ) <= 1;
```

The construction `z = 2 * x - y` is *not* an equality constraint; it is an assignment. It is storing an intermediate calculation `2 * x - y`, which is an affine expression, which is then used later in two different constraints. We call `z` an *expression holder* to differentiate it from a formally declared `cvx` variable. For more on the critical differences between assignment and equality, see Section §8.4.

Often it will be useful to accumulate an array of expressions into a single Matlab variable. Unfortunately, a somewhat technical detail of the Matlab object model can cause problems in such cases. Consider this construction:

```
variable u(9);
x(1) = 1;
```

```
for k = 1 : 9,
    x(k+1) = sqrt( x(k) + u(k) );
end
```

This seems reasonable enough: `x` should be a vector whose first value is `1`, and whose subsequent values are concave `cvx` expressions. But if you try this in a `cvx` model, Matlab will give you a rather cryptic error:

```
??? The following error occurred converting from cvx to double:
Error using ==> double
Conversion to double from cvx is not possible.
```

The reason this occurs is that the Matlab variable `x` is initialized as a numeric array when the assignment `x(1)=1` is made; and Matlab will not permit `cvx` objects to be subsequently inserted into numeric arrays.

The solution is to explicitly *declare* `x` to be an expression holder before assigning values to it. We have provided keywords `expression` and `expressions` for just this purpose, for declaring a single or multiple expression holders for future assignment. Once an expression holder has been declared, you may freely insert both numeric and `cvx` expressions into it. For example, the previous example can be corrected as follows:

```
variable u(9);
expression x(10);
x(1) = 1;
for k = 1 : 9,
    x(k+1) = sqrt( x(k) + u(k) );
end
```

`cvx` will accept this construction without error. You can then use the concave expressions `x(1)`, ..., `x(10)` in any appropriate ways; for example, you could maximize `x(10)`.

The differences between a `variable` object and an `expression` object are quite significant. A `variable` object holds an optimization variable, and cannot be overwritten or assigned in the `cvx` specification. (After solving the problem, however, `cvx` will overwrite optimization variables with optimal values.) An `expression` object, on the other hand, is initialized to zero, and should be thought of as a temporary place to store `cvx` expressions; it can be assigned to, freely re-assigned, and overwritten in a `cvx` specification.

Of course, as our first example shows, it is not always *necessary* to declare an expression holder before it is created or used. But doing so provides an extra measure of clarity to models, so we strongly recommend it.

4 The DCP ruleset

`cvx` enforces the conventions dictated by the disciplined convex programming ruleset, or *DCP ruleset* for short. `cvx` will issue an error message whenever it encounters a violation of any of the rules, so it is important to understand them before beginning to build models. The rules are drawn from basic principles of convex analysis, and are easy to learn, once you've had an exposure to convex analysis and convex optimization.

The DCP ruleset is a set of sufficient, but not necessary, conditions for convexity. So it is possible to construct expressions that violate the ruleset but are in fact convex. As an example consider the entropy function, $-\sum_{i=1}^{n} x_i \log x_i$, defined for $x > 0$, which is concave. If it is expressed as

```
- sum( x .* log( x ) )
```

`cvx` will reject it, because its concavity does not follow from any of the composition rules. (Specifically, it violates the no-product rule described in §4.4.) Problems involving entropy, however, can be solved, by explicitly using the entropy function,

```
sum(entr( x ))
```

which is in the base `cvx` library, and thus recognized as concave by `cvx`. If a convex (or concave) function is not recognized as convex or concave by `cvx`, it can be added as a new atom; see §5.

As another example consider the function $\sqrt{x^2+1} = \|[x\ 1]\|_2$, which is convex. If it is written as

```
norm([x 1])
```

(assuming `x` is a scalar variable or affine expression) it will be recognized by `cvx` as a convex expression, and therefore can be used in (appropriate) constraints and objectives. But if it is written as

```
sqrt(x^2+1)
```

`cvx` will reject it, since convexity of this function does not follow from the `cvx` ruleset.

4.1 A taxonomy of curvature

In disciplined convex programming, a scalar expression is classified by its *curvature*. There are four categories of curvature: *constant*, *affine*, *convex*, and *concave*. For a function $f : \mathbf{R}^n \to \mathbf{R}$ defined on all $\mathbf{R}^n$, the categories have the following meanings:

constant:	$f(\alpha x + (1-\alpha)y) = f(x)$	$\forall x, y \in \mathbf{R}^n,\ \alpha \in \mathbf{R}$
affine:	$f(\alpha x + (1-\alpha)y) = \alpha f(x) + (1-\alpha) f(y)$	$\forall x, y \in \mathbf{R}^n,\ \alpha \in \mathbf{R}$
convex:	$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha) f(y)$	$\forall x, y \in \mathbf{R}^n,\ \alpha \in [0,1]$
concave:	$f(\alpha x + (1-\alpha)y) \geq \alpha f(x) + (1-\alpha) f(y)$	$\forall x, y \in \mathbf{R}^n,\ \alpha \in [0,1]$

Of course, there is significant overlap in these categories. For example, constant expressions are also affine, and (real) affine expressions are both convex and concave.

Convex and concave expressions are real by definition. Complex constant and affine expressions can be constructed, but their usage is more limited; for example, they cannot appear as the left- or right-hand side of an inequality constraint.

4.2 Top-level rules

`cvx` supports three different types of disciplined convex programs:

- A *minimization problem*, consisting of a convex objective function and zero or more constraints.
- A *maximization problem*, consisting of a concave objective function and zero or more constraints.
- A *feasibility problem*, consisting of one or more constraints.

4.3 Constraints

Three types of constraints may be specified in disciplined convex programs:

- An *equality constraint*, constructed using `==`, where both sides are affine.
- A *less-than inequality constraint*, using either `<=` or `<`, where the left side is convex and the right side is concave.
- A *greater-than inequality constraint*, using either `>=` or `>`, where the left side is concave and the right side is convex.

Non-equality constraints, constructed using `~=`, are never allowed. (Such constraints are not convex.)

One or both sides of an equality constraint may be complex; inequality constraints, on the other hand, must be real. A complex equality constraint is equivalent to two real equality constraints, one for the real part and one for the imaginary part. An equality constraint with a real side and a complex side has the effect of constraining the imaginary part of the complex side to be zero.

As discussed in §3.6 above, `cvx` enforces set membership constraints (*e.g.*, $x \in S$) using equality constraints. The rule that both sides of an equality constraint must be affine applies to set membership constraints as well. In fact, the returned value of set atoms like `semidefinite()` and `lorentz()` is affine, so it is sufficient to simply verify the remaining portion of the set membership constraint. For composite values like `{ x, y }`, each element must be affine.

In this version, strict inequalities `<`, `>` are interpreted identically to nonstrict inequalities `>=`, `<=`. Eventually `cvx` will flag strict inequalities so that they can be verified after the optimization is carried out.

4.4 Expression rules

So far, the rules as stated are not particularly restrictive, in that all convex programs (disciplined or otherwise) typically adhere to them. What distinguishes disciplined convex programming from more general convex programming are the rules governing the construction of the expressions used in objective functions and constraints.

Disciplined convex programming determines the curvature of scalar expressions by recursively applying the following rules. While this list may seem long, 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.

- A valid constant expression is
  - any well-formed Matlab expression that evaluates to a finite value.
- A valid affine expression is
  - a valid constant expression;
  - a declared variable;
  - a valid call to a function in the atom library 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 in the atom library with a convex result;
  - an affine scalar raised to a constant power $p \geq 1$, $p \neq 3, 5, 7, 9, ...$;
  - a convex scalar quadratic form (§4.8);
  - 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;
  - a concave scalar raised to a power $p \in (0, 1)$;
  - a concave scalar quadratic form (§4.8);
  - 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.

If an expression cannot be categorized by this ruleset, 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 generally forbid *products* between nonconstant expressions, with the exception of scalar quadratic forms (see §4.8 below). For example, the expression `x*sqrt(x)` happens to be a convex function of `x`, but its convexity cannot be verified using the `cvx` ruleset, and so is rejected. (It can be expressed as `x^(3/2)` or `pow_p(x,3/2)`, however.) We call this the *no-product rule*, and paying close attention to it will go a long way to insuring that the expressions you construct are valid.

4.5 Functions

In `cvx`, functions are categorized in two attributes: *curvature* (*constant*, *affine*, *convex*, or *concave*) and *monotonicity* (*nondecreasing*, *nonincreasing*, or *nonmonotonic*). Curvature determines the conditions under which they can appear in expressions according to the expression rules given in §4.4 above. Monotonicity determines how they can be used in function compositions, as we shall see in §4.6 below.

For functions with only one argument, the categorization is straightforward. Some examples are given in the table below.

Function	Meaning	Curvature	Monotonicity
`sum( x )`	$\sum_i x_i$	affine	nondecreasing
`abs( x )`	$\lvert x \rvert$	convex	nonmonotonic
`sqrt( x )`	$\sqrt{x}$	concave	nondecreasing

Following standard practice in convex analysis, convex functions are interpreted as $+\infty$ when the argument is outside the domain of the function, and concave functions are interpreted as $-\infty$ when the argument is outside its domain. In other words, convex and concave functions in `cvx` are interpreted as their *extended-valued extensions*.

This has the effect of automatically constraining the argument of a function to be in the function's domain. For example, if we form `sqrt(x+1)` in a `cvx` specification, where `x` is a variable, then `x` will automatically be constrained to be larger than or equal to -1. There is no need to add a separate constraint, `x>=-1`, to enforce this.

Monotonicity of a function is determined in the extended sense, *i.e.*, *including the values of the argument outside its domain*. For example, `sqrt(x)` is determined to be nondecreasing since its value is constant ($-\infty$) for negative values of its argument; then jumps *up* to 0 for argument zero, and increases for positive values of its argument.

`cvx` does *not* consider a function to be convex or concave if it is so only over a portion of its domain, even if the argument is constrained to lie in one of these portions. As an example, consider the function $1/x$. This function is convex for $x > 0$, and concave for $x < 0$. But you can never write `1/x` in `cvx` (unless `x` is constant), even if you have imposed a constraint such as `x>=1`, which restricts `x` to lie in the convex portion of function $1/x$. You can use the `cvx` function `inv_pos(x)`, defined as $1/x$ for $x > 0$ and ∞ otherwise, for the convex portion of $1/x$; `cvx` recognizes this function as convex and nonincreasing. In `cvx`, you can express the concave portion of $1/x$, where x is negative, using `-inv_pos(-x)`, which will be correctly recognized as concave and nonincreasing.

For functions with multiple arguments, curvature is always considered *jointly*, but monotonicity can be considered on an *argument-by-argument* basis. For example,

$$\texttt{quad\_over\_lin(x, y)} \quad \begin{cases} |x|^2/y & y > 0 \\ +\infty & y \leq 0 \end{cases} \quad \text{convex, nonincreasing in } y$$

is jointly convex in both arguments, but it is monotonic only in its second argument.

In addition, some functions are convex, concave, or affine only for a *subset* of its arguments. For example, the function

$$\texttt{norm(x, p)} \quad \|x\|_p \ \ (1 \leq p) \quad \text{convex in } x, \text{ nonmonotonic}$$

is convex only in its first argument. Whenever this function is used in a `cvx` specification, then, the remaining arguments must be constant, or `cvx` will issue an error message. Such arguments correspond to a function's parameters in mathematical terminology; *e.g.*,

$$f_p(x) : \mathbf{R}^n \rightarrow \mathbf{R}, \quad f_p(x) \triangleq \|x\|_p$$

So it seems fitting that we should refer to such arguments as *parameters* in this context as well. Henceforth, whenever we speak of a `cvx` function as being convex, concave, or affine, we will assume that its parameters are known and have been given appropriate, constant values.

4.6 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 an affine expression (of compatible size) as an argument. The result is convex, concave, or affine, respectively.

For example, consider the function `square( x )`, 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 in the `cvx` atom library, and known to be convex, provided its argument is real. So if `x` is a real variable of dimension n, `a` is a constant n-vector, and `b` is a constant, the expression

```
square( a' * x + b )
```

is accepted by `cvx`, which knows that it is convex.

The affine composition rule above is a special case of a more sophisticated composition rule, which we describe now. We consider a function, of known curvature and monotonicity, that accepts multiple arguments. For *convex* functions, the rules are:

- If the function is nondecreasing in an argument, that argument must be convex.
- If the function is nonincreasing in an argument, that argument must be concave.
- If the function is neither nondecreasing or nonincreasing in an argument, that argument must be affine.

If each argument of the function satisfies these rules, then the expression is accepted by `cvx`, and is classified as convex. Recall that a constant or affine expression is both convex and concave, so any argument can be affine, including as a special case, constant.

The corresponding rules for a concave function are as follows:

- If the function is nondecreasing in an argument, that argument must be concave.
- If the function is nonincreasing in an argument, that argument must be convex.
- If the function is neither nondecreasing or nonincreasing in an argument, that argument must be affine.

In this case, the expression is accepted by `cvx`, and classified as concave.

For more background on these composition rules, see [BV04, §3.2.4]. In fact, with the exception of scalar quadratic expressions, the entire DCP ruleset can be thought of as special cases of these six rules.

Let us examine some examples. The maximum function is convex and nondecreasing in every argument, so it can accept any convex expressions as arguments. For example, if `x` is a vector variable, then

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

obeys the first of the six composition rules and is therefore accepted by `cvx`, and classified as convex.

As another example, consider the sum function, which is both convex and concave (since it is affine), and nondecreasing in each argument. Therefore the expressions

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

are recognized as valid in `cvx`, and classified as convex and concave, respectively. The first one follows from the first rule for convex functions; and the second one follows from the first rule for concave functions.

Most people who know basic convex analysis like to think of these examples in terms of the more specific rules: a maximum of convex functions is convex, and a sum of convex (concave) functions is convex (concave). But these rules are just special

cases of the general composition rules above. Some other well known basic rules that follow from the general composition rules are: a nonnegative multiple of a convex (concave) function is convex (concave); a nonpositive multiple of a convex (concave) function is concave (convex).

Now we consider a more complex example in depth. Suppose `x` is a vector variable, and `A`, `b`, and `f` are constants with appropriate dimensions. `cvx` recognizes the expression

```
sqrt(f'*x) + min(4,1.3-norm(A*x-b))
```

as concave. Consider the term `sqrt(f'*x)`. `cvx` recognizes that `sqrt` is concave and `f'*x` is affine, so it concludes that `sqrt(f'*x)` is concave. Now consider the second term `min(4,1.3-norm(A*x-b))`. `cvx` recognizes that `min` is concave and nondecreasing, so it can accept concave arguments. `cvx` recognizes that `1.3-norm(A*x-b)` is concave, since it is the difference of a constant and a convex function. So `cvx` concludes that the second term is also concave. The whole expression is then recognized as concave, since it is the sum of two concave functions.

The composition rules are sufficient but not necessary for the classification to be correct, so some expressions which are in fact convex or concave will fail to satisfy them, and so will be rejected by `cvx`. For example, if `x` is a vector variable, 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. Of course, the workaround is simple in this case: use `norm( x )` instead, since `norm` is in the atom library and known by `cvx` to be convex.

4.7 Monotonicity in nonlinear compositions

Monotonicity is a critical aspect of the rules for nonlinear compositions. This has some consequences that are not so obvious, as we shall demonstrate here by example. Consider the expression

```
square( square( x ) + 1 )
```

where `x` is a scalar variable. This expression is in fact convex, since $(x^2+1)^2 = x^4 + 2x^2 + 1$ is convex. But `cvx` will reject the expression, because the outer `square` cannot accept a convex argument. Indeed, the square of a convex function is not, in general, convex: for example, $(x^2-1)^2 = x^4 - 2x^2 + 1$ is not convex.

There are several ways to modify the expression above to comply with the ruleset. One way is to write it as `x^4 + 2*x^2 + 1`, which `cvx` recognizes as convex, since `cvx` allows positive even integer powers using the `^` operator. (Note that the same technique, applied to the function $(x^2-1)^2$, will fail, since its second term is concave.)

Another approach is to use the alternate outer function `square_pos`, included in the `cvx` library, which represents the function $(x_+)^2$, where $x_+ = \max\{0, x\}$. Obviously, `square` and `square_pos` coincide when their arguments are nonnegative. But

`square_pos` is nondecreasing, so it can accept a convex argument. Thus, the expression

```
square_pos( square( x ) + 1 )
```

is mathematically equivalent to the rejected version above (since the argument to the outer function is always positive), but it satisfies the DCP ruleset and is therefore accepted by `cvx`.

This is the reason several functions in the `cvx` atom library come in two forms: the "natural" form, and one that is modified in such a way that it is monotonic, and can therefore be used in compositions. Other such "monotonic extensions" include `sum_square_pos` and `quad_pos_over_lin`. If you are implementing a new function yourself, you might wish to consider if a monotonic extension of that function would also be useful.

4.8 Scalar quadratic forms

In its original form described in [Gra04, GBY06], the DCP ruleset forbids even the use of simple quadratic expressions such as `x * x` (assuming `x` is a scalar variable). For practical reasons, we have chosen to make an exception to the ruleset to allow for the recognition of certain specific quadratic forms that map directly to certain convex quadratic functions (or their concave negatives) in the `cvx` atom library:

`conj( x ) .* x`	is replaced with	`square( x )`
`y' * y`	is replaced with	`sum_square( y )`
`(A*x-b)'*Q*(Ax-b)`	is replaced with	`quad_form( A * x - b, Q )`

`cvx` detects the quadratic expressions such as those on the left above, and determines whether or not they are convex or concave; and if so, translates them to an equivalent function call, such as those on the right above.

`cvx` examines each *single* product of affine expressions, and each *single* squaring of an affine expression, checking for convexity; it will not check, for example, sums of products of affine expressions. For example, given scalar variables `x` and `y`, the expression

```
x ^ 2 + 2 * x * y + y ^2
```

will cause an error in `cvx`, because the second of the three terms `2 * x * y`, is neither convex nor concave. But the equivalent expressions

```
( x + y ) ^ 2
( x + y ) * ( x + y )
```

will be accepted. `cvx` actually completes the square when it comes across a scalar quadratic form, so the form need not be symmetric. For example, if `z` is a vector variable, `a`, `b` are constants, and `Q` is positive definite, then

```
( z + a )' * Q * ( z + b )
```

will be recognized as convex. Once a quadratic form has been verified by `cvx`, it can be freely used in any way that a normal convex or concave expression can be, as described in §4.4.

Quadratic forms should actually be used less frequently in disciplined convex programming than in a more traditional mathematical programming framework, where a quadratic form is often a smooth substitute for a nonsmooth form that one truly wishes to use. In `cvx`, such substitutions are rarely necessary, because of its support for nonsmooth functions. For example, the constraint

```
sum( ( A * x - b ) .^ 2 ) <= 1
```

is equivalently represented using the Euclidean norm:

```
norm( A * x - b ) <= 1
```

With modern solvers, the second form can be represented using a second-order cone constraint—so the second form may actually be more efficient. So we encourage you to re-evaluate the use of quadratic forms in your models, in light of the new capabilities afforded by disciplined convex programming.

5 Adding new functions to the cvx atom library

cvx allows new convex and concave functions to be defined and added to the atom library, in two ways, described in this section. The first method is simple, and can (and should) be used by many users of cvx, since it requires only a knowledge of the basic DCP ruleset. The second method is very powerful, but a bit complicated, and should be considered an advanced technique, to be attempted only by those who are truly comfortable with convex analysis, disciplined convex programming, and cvx in its current state.

Please do let us know if you have implemented a convex or concave function that you think would be useful to other users; we will be happy to incorporate it in a future release.

5.1 New functions via the DCP ruleset

The simplest way to construct a new function that works within cvx is to construct it using expressions that fully conform to the DCP ruleset. To illustrate this, consider the convex *deadzone* function, defined as

$$f(x) = \max\{|x| - 1, 0\} = \begin{cases} 0 & |x| \leq 1 \\ x - 1 & x > 1 \\ -1 - x & x < -1 \end{cases}$$

To implement this function in cvx, simply create a file `deadzone.m` containing

```
function y = deadzone( x )
y = max( abs( x ) - 1, 0 )
```

This function works just as you expect it would outside of cvx—*i.e.*, when its argument is numerical. But thanks to Matlab's operator overloading capability, it will also work within cvx if called with an affine argument. cvx will properly conclude that the function is convex, because all of the operations carried out conform to the rules of DCP: abs is recognized as a convex function; we can subtract a constant from it, and we can take the maximum of the result and 0, which yields a convex function. So we are free to use deadzone anywhere in a cvx specification that we might use abs, for example, because cvx knows that it is a convex function.

Let us emphasize that when defining a function this way, the expressions you use *must* conform to the DCP ruleset, just as they would if they had been inserted directly into a cvx model. For example, if we replace max with min above; *e.g.*,

```
function y = deadzone_bad( x )
y = min( abs( x ) - 1, 0 )
```

then the modified function fails to meet the DCP ruleset. The function will work *outside* of a cvx specification, happily computing the value $\min\{|x| - 1, 0\}$ for a *numerical* argument x. But inside a cvx specification, invoked with a nonconstant argument, it will not work, because it doesn't follow the DCP composition rules.

5.2 New functions via partially specified problems

A more advanced method for defining new functions in `cvx` relies on the following basic result of convex analysis. Suppose that $S \subset \mathbf{R}^n \times \mathbf{R}^m$ is a convex set and $g : (\mathbf{R}^n \times \mathbf{R}^m) \to (\mathbf{R} \cup +\infty)$ is a convex function. Then

$$f : \mathbf{R}^n \to (\mathbf{R} \cup +\infty), \quad f(x) \triangleq \inf \{\, g(x,y) \mid \exists y,\ (x,y) \in S \,\} \tag{7}$$

is also a convex function. (This rule is sometimes called the *partial minimization rule*.) We can think of the convex function f as the optimal value of a family of convex optimization problems, indexed or parametrized by x,

$$\begin{array}{ll} \text{minimize} & g(x,y) \\ \text{subject to} & (x,y) \in S \end{array}$$

with optimization variable y.

One special case should be very familar: if $m = 1$ and $g(x,y) \triangleq y$, then

$$f(x) \triangleq \inf \{\, y \mid \exists y,\ (x,y) \in S \,\}$$

gives the classic *epigraph* representation of f:

$$\mathbf{epi}\, f = S + (\{0\} \times \mathbf{R}_+)\,,$$

where $0 \in \mathbf{R}^n$.

In `cvx` you can define a convex function in this very manner, that is, as the optimal value of a parameterized family of disciplined convex programs. We call the underlying convex program in such cases an *incomplete specification*—so named because the parameters (that is, the function inputs) are unknown when the specification is constructed. The concept of incomplete specifications can at first seem a bit complicated, but it is very powerful mechanism that allows `cvx` to support a wide variety of functions.

Let us look at an example to see how this works. Consider the unit-halfwidth Huber penalty function $h(x)$:

$$h : \mathbf{R} \to \mathbf{R}, \quad h(x) \triangleq \begin{cases} x^2 & |x| \leq 1 \\ 2|x| - 1 & |x| \geq 1. \end{cases} \tag{8}$$

We can express the Huber function in terms of the following family of convex QPs, parameterized by x:

$$\begin{array}{ll} \text{minimize} & 2v + w^2 \\ \text{subject to} & |x| \leq v + w \\ & w \leq 1,\ v \geq 0 \end{array} \tag{9}$$

with scalar variables v and w. The optimal value of this simple QP is equal to the Huber penalty function of x. We note that the objective and constraint functions in this QP are (jointly) convex in v, w *and* x.

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

```
function cvx_optval = huber( x )
cvx_begin
    variables w v;
    minimize( w^2 + 2 * v );
    subject to
        abs( x ) <= w + v;
        w <= 1; v >= 0;
cvx_end
```

If `huber` is called with a numeric value of `x`, then upon reaching the `cvx_end` statement, `cvx` will find a complete specification, and solve the problem to compute the result. `cvx` places the optimal objective function value into the variable `cvx_optval`, and function returns that value as its output. Of course, it's very inefficient to compute the Huber function of a numeric value x by solving a QP. But it does give the correct value (up to the core solver accuracy).

What is most important, however, is that if `huber` is used within a `cvx` specification, with an affine `cvx` expression for its argument, then `cvx` will do the right thing. In particular, `cvx` will recognize the Huber function, called with affine argument, as a valid convex expression. In this case, the function `huber` will contain a special Matlab object that represents the function call in constraints and objectives. Thus the function `huber` can be used anywhere a traditional convex function can be used, in constraints or objective functions, in accordance with the DCP ruleset.

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

$$f : \mathbf{R} \to (\mathbf{R} \cup -\infty), \quad f(x) \triangleq \sup \{ g(x,y) \mid \exists y,\ (x,y) \in S \} \tag{10}$$

is concave. If $g(x,y) \triangleq y$, then

$$f(x) \triangleq \sup \{ y \mid \exists y,\ (x,y) \in S \} \tag{11}$$

gives the *hypograph* representation of f:

$$\mathbf{hypo}\, f = S - \mathbf{R}^n_+.$$

In `cvx`, a concave incomplete specification is simply one that uses a `maximize` objective instead of a `minimize` objective; and if properly constructed, it can be used anywhere a traditional concave function can be used within a `cvx` specification.

For an example of a concave incomplete specification, consider the function

$$f : \mathbf{R}^{n \times n} \to \mathbf{R}, \quad f(X) = \lambda_{\min}(X + X^T) \tag{12}$$

Its hypograph can be represented using a single linear matrix inequality:

$$\mathbf{hypo}\, f = \{ (X,t) \mid f(X) \geq t \} = \left\{ (X,t) \mid X + X^T - tI \succeq 0 \right\} \tag{13}$$

So we can implement this function in `cvx` as follows:

```
function cvx_optval = lambda_min_symm( X )
n = size( X, 1 );
cvx_begin
    variable y;
    maximize( y );
    subject to
        X + X' - y * eye( n ) == semidefinite( n );
cvx_end
```

If a numeric value of `X` is supplied, this function will return `min(eig(X+X'))` (to within numerical tolerances). However, this function can also be used in `cvx` constraints and objectives, just like any other concave function in the atom library.

There are two practical issues that arise when defining functions using incomplete specifications, both of which we will illustrate using our `huber` example above. First of all, as written the function works only with scalar values. To apply it (elementwise) to a vector requires that we iterate through the elements in a `for` loop—a *very* inefficient enterprise, particularly in `cvx`. A far better approach is to extend the `huber` function to handle vector inputs. This is, in fact, rather simple to do: we simply create a *multiobjective* version of the problem:

```
function cvx_optval = huber( x )
sx = size( x );
cvx_begin
    variables w( sx ) v( sx );
    minimize( w .^ 2 + 2 * v );
    subject to
        abs( x ) <= w + v;
        w <= 1; v >= 0;
cvx_end
```

This version of `huber` will in effect create `sx` "instances" of the problem in parallel; and when used in a `cvx` specification, will be handled correctly.

The second issue is that if the input to `huber` is numeric, then direct computation is a far more efficient way to compute the result than solving a QP. (What is more, the multiobjective version cannot be used with numeric inputs.) One solution is to place both versions in one file, with an appropriate test to select the proper version to use:

```
function cvx_optval = huber( x )
if isnumeric( x ),
    xa   = abs( x );
    flag = xa < 1;
    cvx_optval = flag .* xa.^2 + (~flag) * (2*xa-1);
else,
    sx = size( x );
    cvx_begin
```

```
            variables w( sx ) v( sx );
            minimize( w .^ 2 + 2 * v );
            subject to
                abs( x ) <= w + v;
                w <= 1; v >= 0;
        cvx_end
    end
```

Alternatively, you can create two separate versions of the function, one for numeric input and one for `cvx` expressions, and place the `cvx` version in a subdirectory called `@cvx`. (Do not include this directory in your Matlab `path`; only include its parent.) Matlab will automatically call the version in the `@cvx` directory when one of the arguments is a `cvx` variable. This is the approach taken for the version of `huber` found in the `cvx` atom library.

One good way to learn more about using incomplete specifications is to examine some of the examples already in the `cvx` atom library. Good choices include `huber`, `inv_pos`, `lambda_min`, `lambda_max`, `matrix_frac`, `quad_over_lin`, `sum_largest`, and others. Some are a bit difficult to read because of diagnostic or error-checking code, but these are relatively simple.

6 Semidefinite programming using cvx

Those who are familiar with *semidefinite programming* (SDP) know that the constraints that utilize the set `semidefinite(n)` in §3.6 above are, in practice, typically expressed using *linear matrix inequality* (LMI) notation. For example, given $X = X^T \in \mathbf{R}^{n\times n}$, the constraint $X \succeq 0$ denotes that $X \in \mathbf{S}^n_+$; that is, that X is positive semidefinite.

`cvx` provides a special *SDP mode* which allows this LMI convention to be employed inside `cvx` models using Matlab's standard inequality operators `>=`, `<=`, *etc.*. In order to use it, one must simply begin a model with the statement `cvx_begin sdp` or `cvx_begin SDP` instead of simply `cvx_begin`. When SDP mode is engaged, `cvx` interprets certain inequality constraints in a different manner. To be specific:

- Equality constraints are interpreted the same (*i.e.*, elementwise).
- Inequality constraints involving vectors and scalars are interpreted the same; *i.e.*, elementwise.
- Inequality constraints involving non-square matrices are *disallowed*; attempting to use them causes an error. If you wish to do true elementwise comparison of matrices `X` and `Y`, use a vectorization operation `X(:) <= Y(:)` or `vec( X ) <= vec( Y )`. (`vec` is a function provided by `cvx` that is equivalent to the colon operation.)
- Inequality constraints involving real, square matrices are interpreted as follows:

`X >= Y`	and	`X > Y`	become	`X - Y == semidefinite(n)`
`X <= Y`	and	`X < Y`	become	`Y - X == semidefinite(n)`

  If either side is complex, then the inequalities are interpreted as follows:

`X >= Y`	and	`X > Y`	become	`X - Y == hermitian_semidefinite(n)`
`X <= Y`	and	`X < Y`	become	`Y - X == hermitian_semidefinite(n)`

  In the above, `n=max(size(X,1),size(Y,1))`.

- There is one additional restriction: both `X` and `Y` must be the same size, or one must be the scalar zero. For example, if `X` and `Y` are matrices of size `n`,

`X >= 1`	or	`1 >= Y`	*illegal*
`X >= ones(n,n)`	or	`ones(n,n) >= Y`	*legal*
`X >= 0`	or	`0 >= Y`	*legal*

  In effect, `cvx` enforces a stricter interpretation of the inequality operators for LMI constraints.

- Note that LMI constraints enforce symmetry (real or Hermitian, as appropriate) on their inputs. Unlike SDPSOL [WB00], `cvx` does not extract the symmetric part for you: you must take care to insure symmetry yourself. Since `cvx` supports the declaration of symmetric matrices, this is reasonably straightforward. If `cvx` cannot determine that an LMI is symmetric, a warning will be issued.

- A dual variable, if supplied, will be applied to the converted equality constraint. It will be given a positive semidefinite value if an optimal point is found.

So, for example, the `cvx` model found in the file `examples/closest_toeplitz_sdp.m`,

```
cvx_begin
    variable Z(n,n) hermitian toeplitz
    dual variable Q
    minimize( norm( Z - P, 'fro' ) )
    Z == hermitian_semidefinite( n ) : Q;
cvx_end
```

can also be written as follows:

```
cvx_begin sdp
    variable Z(n,n) hermitian toeplitz
    dual variable Q
    minimize( norm( Z - P, 'fro' ) )
    Z >= 0 : Q;
cvx_end
```

Many other examples in the `cvx` example library utilize semidefinite constraints; and all of them use SDP mode. To find them, simply search for the text `cvx_begin sdp` in the `examples/` subdirectory tree using your favorite file search tool. One of these examples is reproduced in §8.5.

Since semidefinite programming is popular, some may wonder why SDP mode is not the default behavior. The reason for this is that we place a strong emphasis on maintaining consistency between Matlab's native behavior and that of `cvx`; and the use of the `>=`, `<=`, `>`, `<` operators to create LMIs represents a deviation from that ideal. For example, the expression `Z >= 0` in the example above constrains the variable `Z` to be positive semidefinite. But after the model has been solved and `Z` has been replaced with a numeric value, the expression `Z >= 0` will test for the *elementwise* nonnegativity of `Z`. To verify that the numeric value of `Z` is, in fact, positive semidefinite, you must perform a test like `min(eig(Z)) >= 0`.

7 Geometric programming using cvx

Geometric programs (GPs) are special mathematical programs that can be converted to convex form using a change of variables. The convex form of GPs can be expressed as DCPs, but cvx also provides a special mode that allows a GP to be specified in its native form. cvx will automatically perform the necessary conversion, compute a numerical solution, and translate the results back to the original problem. For a tutorial on geometric programming, we refer the reader to [BKVH05].

To utilize GP mode, you must begin your cvx specification with the command cvx_begin gp or cvx_begin GP instead of simply cvx_begin. For example, the following code, found in the example library at gp/max_volume_box.m, determines the maximum volume box subject to various area and ratio constraints:

```
cvx_begin gp
    variables w h d
    maximize( w * h * d )
    subject to
        2*(h*w+h*d) <= Awall;
        w*d <= Afloor;
        h/w >= alpha;
        h/w <= beta;
        d/w >= gamma;
        d/w <= delta;
cvx_end
```

As the example illustrates, cvx supports the construction of monomials and posynomials using addition, multiplication, division (when appropriate), and powers. In addition, cvx supports the construction of *generalized geometric programs* (GGPs), by permitting the use of *generalized posynomials* wherever posynomials are permitted in standard GP [BKVH05].

The solvers used in this version of cvx do not support geometric programming natively. Instead, they are solved using the successive approximation technique described in Appendix D.1. This means that solving GPs can be slow, but for small and medium sized problems, the method works well.

In the remainder of this section, we will describe specific rules that apply when constructing models in GP mode.

7.1 Top-level rules

cvx supports three types of geometric programs:

- A *minimization problem*, consisting of a generalized posynomial objective and zero or more constraints.
- A *maximization problem*, consisting of a *monomial* objective and zero or more constraints.

- A *feasibility problem*, consisting of one or more constraints.

The asymmetry between minimizations and maximizations—specifically, that only monomial objectives are allowed in the latter—is an unavoidable artifact of the geometry of GPs and GGPs.

7.2 Constraints

Three types of constraints may be specified in geometric programs:

- An *equality constraint*, consrtucted using `==`, where both sides are monomials.
- A *less-than inequality constraint* `<=`, `<` where the left side is a generalized posynomial and the right side is a monomial.
- A *greater-than inequality constraint* `>=`, `>` where the left side is a monomial and the right side is a generalized posynomial.

As with DCPs, non-equality constraints are not permitted.

7.3 Expressions

The basic building blocks of generalized geometric programming are monomials, posynomials, and generalized posynomials. A valid monomial is

- a declared variable;
- the product of two or more monomials;
- the ratio of two monomials;
- a monomial raised to a real power; or
- a call to one of the following functions with monomial arguments: `prod`, `cumprod`, `geo_mean`, `sqrt`.

A valid posynomial expression is

- a valid monomial;
- the sum of two or more posynomials;
- the product of two or more posynomials;
- the ratio of a posynomial and a monomial;
- a posynomial raised to a positive integral power; or
- a call to one of the following functions with posynomial arguments: `sum`, `cumsum`, `mean`, `prod`, `cumprod`.

A valid generalized posynomial expression is

- a valid posynomial;
- the sum of two or more generalized posynomials;
- the product of two or more generalized posynomials;
- the ratio of a generalized posynomial and a monomial;
- a generalized posynomial raised to a positive real power; or
- a call to one of the following functions with arguments that are generalized posynomials: `sum`, `cumsum`, `mean`, `prod`, `cumprod`, `geo_mean`, `sqrt`, `norm`, `sum_largest`, `norm_largest`.

It is entirely possible to create and manipulate arrays of monomials, posynomials, and/or generalized posynomials in `cvx`, in which case these rules extend in an obvious manner. For example, the product of two monomial matrices produces either a posynomial matrix or a monomial matrix, depending upon the structure of said matrices.

8 Advanced topics

In this section we describe a number of the more advanced capabilities of `cvx`. We recommend that you *skip* this section at first, until you are comfortable with the basic capabilities described above.

8.1 Solver selection

`cvx` currently supports two solvers: SeDuMi and SDPT3. With an exception discussed below, SeDuMi is used by default. We have found that SeDuMi is faster and more reliable for most problems, but for your application you may find otherwise. To select SDPT3 as your default solver instead, simply type

```
12 cvx_solver sdpt3
```

at the command line. To revert to SDPT3, type

```
13 cvx_solver sedumi
```

To see which solver is currently selected, simply type

```
14 cvx_solver
```

If you issue this command inside a model, it will change the solver being used *only* for that model; the default will be preserved. If you issue this command *outside* of a model, it will change the solver used for all future models.

Both solvers are being actively maintained by their authors. If you are having difficulty with one solver, try the other. If you encounter a problem that one solver can handle but the other cannot, please send us a bug report and we will forward the results to the authors of SeDuMI or SDPT3.

8.2 Controlling solver precision

Numerical methods for convex optimization are not exact; they compute their results to within a predefined numerical precision or tolerance. Upon solution of your model, the tolerance level the solver has achieved is returned in the `cvx_slvtol` variable. Attempts to interpret this tolerance level in any absolute sense are not recommended. For one thing, each solver computes it differently. For another, it depends heavily on the considerable transformations that `cvx` applies to your model before delivering it to the solver. So while you may find its value interesting we strongly discourage dependence upon it within your applications.

The tolerance levels that `cvx` selects by default have been inherited from the underlying solvers being used, with minor modifications. `cvx` actually considers *three* different tolerance levels $\epsilon_{\text{solver}} \leq \epsilon_{\text{standard}} \leq \epsilon_{\text{reduced}}$ when solving a model:

- The *solver tolerance* ϵ_{solver} is the level requested of the solver. The solver will stop as soon as it achieves this level, or until no further progress is possible.

- The *standard tolerance* $\epsilon_{\text{standard}}$ is the level at which `cvx` considers the model solved to full precision.
- The *reduced tolerance* $\epsilon_{\text{reduced}}$ is the level at which `cvx` considers the model "inaccurately" sovled, returning a status with the `Inaccurate/` prefix. If this tolerance cannot be achieved, `cvx` returns a status of `Failed`, and the values of the variables should not be considered reliable.

(See Appendix C for more information about the status messages.) Typically, $\epsilon_{\text{solver}} = \epsilon_{\text{standard}}$, but setting $\epsilon_{\text{standard}} < \epsilon_{\text{solver}}$ has a useful interpretation: it allows the solver to search for more accurate solutions without causing an `Inaccurate/` or `Failed` condition if it cannot do so. The default values of $[\epsilon_{\text{solver}}, \epsilon_{\text{standard}}, \epsilon_{\text{reduced}}]$ are set to $[\epsilon^{1/2}, \epsilon^{1/2}, \epsilon^{1/4}]$, where $\epsilon = 2.22 \times 10^{-16}$ is the machine precision. This should be quite sufficient for most applications.

If you wish to modify the tolerances, you may do so using the `cvx_precision` command. There are three ways to invoke this command. Called with no arguments it will return the current tolerance levels as a 3-element row vector.

Calling `cvx_precision` with a string argument allows you to select from a set of predefined precision modes:

- `cvx_precision low`: $[\epsilon^{3/8}, \epsilon^{1/4}, \epsilon^{1/4}]$
- `cvx_precision medium`: $[\epsilon^{1/2}, \epsilon^{3/8}, \epsilon^{1/4}]$
- `cvx_precision default`: $[\epsilon^{1/2}, \epsilon^{1/2}, \epsilon^{1/4}]$
- `cvx_precision high`: $[\epsilon^{3/4}, \epsilon^{3/4}, \epsilon^{3/8}]$
- `cvx_precision best`: $[0, \epsilon^{1/2}, \epsilon^{1/4}]$

In function mode, these calls look like `cvx_precision('low')`, etc. Note that the `best` precision settings sets the solver target to zero, which means that the solver continues as long as it can make progress. It will often be slower than `default`, but it is just as reliable, and sometimes produces more accurate solutions.

Finally, the `cvx_precision` command can be called with a scalar, a length-2 vector, or a length-3 vector. If you pass it a scalar, it will set the solver and standard tolerances to that value, and it will compute a default reduced precision value for you. Roughly speaking, that reduced precision will be the square root of the standard precision, with some bounds imposed to make sure that it stays reasonable. If you supply two values, the smaller will be used for the solver and standard tolerances, and the larger for the reduced tolerance. If you supply three values, their values will be sorted, and each tolerance will be set separately.

The `cvx_precision` command can be used either *within* a `cvx` model or *outside* of it; and its behavior differs in each case. If you call it from within a model, *e.g.*,

```
cvx_begin
    cvx_precision high
    ...
cvx_end
```

then the setting you choose will apply only until `cvx_end` is reached. If you call it outside a model, *e.g.*,

```
cvx_precision high
cvx_begin
    ...
cvx_end
```

then the setting you choose will apply *globally*; that is, to any subsequent models that are created and solved. The local approach should be preferred in an application where multiple models are constructed and solved at different levels of precision.

If you call `cvx_precision` in function mode, either with a string or a numeric value, it will return as its output the *previous* precision vector—the same result you would obtain if you called it with no arguments. This may seem confusing at first, but this is done so that you can save the previous value in a variable, and restore it at the end of your calcuations; e.g.,

```
cvxp = cvx_precision( 'high' );
cvx_begin
...
cvx_end
cvx_precision( cvxp );
```

This is considered good coding etiquette in a larger application where multiple `cvx` models at multiple precision levels may be employed. Of course, a simpler but equally courteous approach is to call `cvx_precision` within the `cvx` model, as described above, so that its effect lasts only for that model.

8.3 Miscellaneous `cvx` commands

- `cvx_problem`: typing this within a `cvx` specification provides a summary of the current problem. Note that this will contain a *lot* of information that you may not recognize—`cvx` performs its conversion to canonical form *as the problem is entered*, generating extra temporary variables and constraints in the process.

- `cvx_clear`: typing this resets the `cvx` system, clearing any and all problems from memory, but without erasing any of your numeric data. This is useful if you make a mistake and need to start over. But note that in current versions of `cvx`, you can simply start another model with `cvx_begin`, and the previous model will be erased (with a warning).

- `cvx_quiet`: typing `cvx_quiet(true)` suppresses screen output from the solver. Typing `cvx_quiet(false)` restores the screen output. In each case, it returns a logical value representing the *previous* state of the quiet flag, so you can restore that state later if you wish—which is good practice if you wish to share your code with others.

- `cvx_pause`: typing `cvx_pause(true)` causes `cvx` to pause and wait for keyboard input before *and* after the solver is called. Useful primarily for demo purposes. Typing `cvx_pause(false)` resets the behavior. In each case, it returns a logical value representing the *previous* state of the pause flag, so you can restore that state later if you wish.

- `cvx_where`: returns the directory where the `cvx` distribution has been installed—assuming that the Matlab path has been set to include that distribution. Useful if you want to find certain helpful subdirectories, such as `doc`, `examples`, *etc.*

- `cvx_version`: returns a description of the `cvx` version number, build number, and hardware platform you are currently using. If you encounter a bug in `cvx`, please run this function and copy its output into your email message along with a description of your problem.

8.4 Assignments versus equality constraints

Anyone who has used the C or Matlab languages for a sufficiently long time understands the differences between *assignments*, which employ a single equals sign `=` operator, and an *equality*, which employs the double equal `==` operator. In `cvx`, this distinction is particularly important; confusing the two operators can have serious consequences. Even when the distinction is well understood, there are important caveats to using assignments in `cvx` that we address here as well.

The consequences of inadvertently using assignments within a `cvx` specification can cause subtle but critical problems. For example, let $A, C \in \mathbf{R}^{n\times n}$ and $b \in \mathbf{R}$ be given, and consider the simple SDP

$$\begin{array}{ll} \text{minimize} & \mathbf{Tr}(CX) \\ \text{subject to} & \mathbf{Tr}(AX) = b \\ & X \succeq 0 \end{array} \tag{14}$$

Suppose that we tried to express this problem in `cvx` as follows:

```
1 n = 5;
2 A = randn(n,n); C = randn(n,n); b = randn;
3 cvx_begin
4     variable X(n,n) symmetric;
5     minimize( trace( C * X ) );
6     subject to
7         trace( A * X ) == b;
8         X = semidefinite(n);
9 cvx_end
```

At first glance, line 8 may look like it constrains `X` to be positive semidefinite; but it is an assignment, not an equality constraint. So `X` is actually *overwritten* with an anonymous, positive semidefinite variable, and the original `X` is not constrained all!

Fortunately, this particular error is easily caught and prevented by `cvx`. When `cvx_end` is reached, `cvx` examines each declared variable to verify that it still points to the variable object it was originally assigned upon declaration. If it does not, it will issue an error like this:

```
??? Error using ==> cvx_end
The following cvx variable(s) have been overwritten:
   X
This is often an indication that an equality constraint was
written with one equals '=' instead of two '=='. The model
must be rewritten before cvx can proceed.
```

We hope that this check will prevent at least some typographical errors from having frustrating consequences in your models.

Of course, this single check does not prevent you from using *all* assignments inside your `cvx` specifications, only those that overwrite formally declared variables. As discussed in §3.8, other kinds of assignments are permitted, and may be genuinely useful. But in our view they should be used sparingly. For instance, consider the first example from §3.8:

```
variables x y
z = 2 * x - y;
square( z ) <= 3;
quad_over_lin( x, z ) <= 1;
```

The following alternative formulation, which declares `z` as a formal variable, is numerically equivalent:

```
variables x y z
z == 2 * x - y;
square( z ) <= 3;
quad_over_lin( x, z ) <= 1;
```

We recommend taking this approach whenever possible. Declaring intermediate calculations as variables provides an extra measure of clarity in your models, and it exposes more of the model's structure to the solver, possibly improving performance.

8.5 Indexed dual variables

In some models, the *number* of constraints depends on the model parameters—not just their sizes. It is straightforward to build such models in `cvx` using, say, a Matlab `for` loop. In order to assign each of these constraints a separate dual variable, we must find a way to adjust the number of dual variables as well. For this reason, `cvx` supports *indexed dual variables*. In reality, they are simply standard Matlab cell arrays whose entries are `cvx` dual variable objects.

Let us illustrate by example how to declare and use indexed dual variables. Consider the following semidefinite program:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n}(n-i)X_{ii} \\ \text{subject to} & \sum_{i=1}^{n} X_{i,i+k} = b_k, \;\; k = 1, 2, \ldots, n \\ & X \succeq 0 \end{array} \tag{15}$$

([Stu99]). This problem minimizes a weighted sum of the main diagonal of a positive semidefinite matrix, while holding the sums along each diagonal constant. The parameters of the problem are the elements of the vector $b \in \mathbf{R}^n$, and the optimization variable is a symmetric matrix $X \in \mathbf{R}^{n \times n}$. The `cvx` version of this model is

```
cvx_begin
    variable X( n, n ) symmetric
    minimize( ( n - 1 : -1 : 0 ) * diag( X ) );
    for k = 0 : n-1,
        sum( diag( X, k ) ) == b( k+1 );
    end
    X == semidefinite(n);
cvx_end
```

If we wish to obtain dual information for the n simple equality constraints, we need a way to assign each constraint in the `for` loop a separate dual variable. This is accomplished as follows:

```
cvx_begin
    variable X( n, n ) symmetric
    dual variables y{n}
    minimize( ( n - 1 : -1 : 0 ) * diag( X ) );
    for k = 0 : n-1,
        sum( diag( X, k ) ) == b( k+1 ) : y{k+1};
    end
    X == semidefinite(n);
cvx_end
```

The statement

```
dual variables y{n}
```

allocates a cell array of n dual variables, and stores the result in the Matlab variable `Z`. The equality constraint in the `for` loop has been augmented with a reference to `y{k+1}`, so that each constraint is assigned a separate dual variable. When the `cvx_end` command is issued, `cvx` will compute the optimal values of these dual variables, and deposit them into an n-element cell array `y`.

This example admittedly is a bit simplistic. With a bit of careful arrangement, it is possible to rewrite this model so that the n equality constraints can be combined into a single vector constraint, which in turn would require only a single vector dual

variable.[3] For a more complex example that is not amenable to such a simplification, see the file

```
examples/cvxbook/Ch07_statistical_estim/cheb.m
```

in the `cvx` distribution. In that problem, each constraint in the `for` loop is a linear matrix inequality, not a scalar linear equation; so the indexed dual variables are symmetric matrices, not scalars.

[3] Indeed, a future version of `cvx` will support the use of the Matlab function `spdiags`, which will reduce the entire for loop to the single constraint `spdiags(X,0:n-1)==b`.

A Installation and compatability

`cvx` requires Matlab 6.5.2 or later. (Previous versions of `cvx` supported Matlab 6.1, but this has been discontinued.) Primary `cvx` development is performed using an Intel-based Mac running Matlab 7.5, with additional development performed on the following platforms and Matlab versions:

- Mac (32-bit): Matlab 7.3, 7.5, 7.9
- Mac (64-bit): Matlab 7.9
- Windows (32-bit): Matlab 6.5.2, 7.5, 7.9
- Linux (32-bit): Matlab 6.5.2, 7.5, 7.9
- Linux (64-bit): Matlab 7.3, 7.5, 7.9

For purposes of testing and support, we have access to most other versions of Matlab for Windows, Linux, and Mac (both Intel and PowerPC).

We strongly recommend against using version 7.0.x of Matlab on any platform. Version 7.0.0 has exhibited numerical problems for at least one user; the problems were eliminated by upgrading. We have also identified another issue using `cvx` with Matlab 7.0.1 and 7.0.4; workarounds are provided in §A.3 below.

Precompiled MEX files are included for all of the above platorms. For other platforms (*e.g.*, Solaris), `cvx` will attempt to compile the MEX files will be compiled during the setup process. While `cvx` is quite likely to work for these platforms, we are unable to provide support. In particular you are responsible for making sure that your system is set up to compile MEX files.

A.1 Basic instructions

1. Retrieve the latest version of cvx from `http://www.stanford.edu/~boyd/cvx`. You can download the package as either a `.zip` file or a `.tar.gz` file.
2. If you have been running a previous version of `cvx`, remove it or rename it to, say, `cvx_old`—before proceeding. *DO NOT* allow the new version of `cvx` to be unpacked on top of the old.
3. Unpack the file anywhere you like; a directory called `cvx` will be created. There is one important exception: *do not* place `cvx` in Matlab's own `toolbox` directory.
4. Start Matlab.
5. Change the current directory to the location of `cvx`. For example, if you unpacked `cvx.zip` into the directory `C:\Matlab\personal`, then you would type

```
cd C:\Matlab\personal\cvx
```

at the Matlab command prompt. Alternatively, if you unpacked `cvx.tar.gz` into the directory `~/matlab` on a Linux machine, then you would type

```
cd ~/matlab/cvx
```

at the Matlab command prompt.

6. Type the command

```
cvx_setup
```

at the Matlab prompt. This does two things: it sets the Matlab search path so it can find all of the `cvx` program files, and it runs a simple test problem to verify the installation. If all goes well, the command will output the line

```
No errors! cvx has been successfully installed.
```

(among others). If this message is not displayed, or any warnings or errors are generated, then there is a problem with the `cvx` installation. Try installing the package again; and if that fails, send us a bug report and tell us about it.

7. If you plan to use `cvx` regularly, you will need to save the current Matlab path for subsequent Matlab sessions. Follow the instructions provided by `cvx_setup` to accomplish that.

A.2 About SeDuMi and SDPT3

The `cvx` distribution includes copies of the solvers SeDuMi and SDPT3 in the directories `cvx/sedumi` and `cvx/sdpt3`, respectively. We strongly recommend that you use our versions of these solvers, and remove any other versions that you have in your Matlab path.

A.3 A Matlab 7.0 issue

The techniques that `cvx` use to embed a new modeling language inside Matlab seem to cause some confusion for Matlab 7.0, specifically version 7.0.4 (R14SP2). We are reasonably confident that the issue is due to a bug in Matlab itself. It does not occur in version 6.5.2 or 7.1 and later. It may occur in earlier versions of Matlab 7.0 as well (R14,R14SP1).

The bug is this: in some circumstances, a `.m`-file containing a `cvx` model will cause an error that looks like this:

```
??? Error: File: thrusters.m Line: 43 Column: 5
"p" was previously used as a variable,
conflicting with its use here as the name of a function.
```

The file name, line/column numbers, and variable name may differ, but the error message remains the same. The example that produced this particular error is a simple inequality constraint

```
p >= -5;
```

where `p` is a `cvx` variable previously declared with a `variable` statement. Interestingly, a different inequality constraint

```
u >= 0
```

in the same model causes no complaint. So we cannot offer you a precise set of circumstances that cause the error.

Fortunately, the workaround is very simple: simply remove all of the extra space in the constraint. In this example, changing the constraint to

```
p>=-5;
```

eliminates the error. You may still indent the constraint as you wish—just remove the intermediate spaces.

We have no idea why this workaround works, but it does—reliably.

B Operators, functions, and sets

B.1 Basic operators and linear functions

Matlab's standard arithmetic operations for addition `+`, subtraction `-`, multiplication `* .*`, division `/ \ ./ .\`, and exponentiation `^ .^` have been overloaded to work in `cvx` whenever appropriate—that is, whenever their use is consistent with both standard mathematical and Matlab conventions *and* the DCP ruleset. For example:

- Two `cvx` expressions can be added together if they are of the same dimension (or one is scalar) and have the same curvature (*i.e.*, both are convex, concave, or affine).
- A `cvx` expression can be multiplied or divided by a scalar constant. If the constant is positive, the curvature is preserved; if it is negative, curvature is reversed.
- An affine column vector `cvx` expression can be multiplied by a constant matrix of appropriate dimensions; or it can be left-divided by a non-singular constant matrix of appropriate dimension.

Numerious other combinations are possible, of course. For example, the use of the exponentiation operators `^ .^` are somewhat limited; see §B.2 below.

Matlab's basic matrix manipulation and arithmetic operations have been extended to work with `cvx` expressions as well, including:

- Concatenation: `[ A, B ; C, D ]`
- Indexing: `x(n+1:end)`, `X([3,4],:)`, *etc.*
- Indexed assignment, including deletion: `y(2:4) = 1`, `Z(1:4,:) = []`, *etc.*
- Transpose and conjugate transpose: `Z.'`, `y'`

A number of Matlab's basic functions have been extended to work with `cvx` expressions as well:

```
conj  conv  cumsum  diag  dot  find  fliplr  flipud  flipdim  horzcat
hankel  ipermute  kron  permute  repmat  reshape  rot90  sparse  sum
                trace  tril  triu  toeplitz  vertcat
```

Most should behave identically with `cvx` expressions as they do with numeric expressions. Those that perform some sort of summation, such as `cumsum`, `sum`, or multiplication, such as `conv`, `dot` or `kron`, can only be used in accordance with the disciplined convex programming rules. For example, `kron(X,Y)` is valid only if either `X` or `Y` is constant; and `trace(Z)` is valid only if the elements along the diagonal have the same curvature.

B.2 Nonlinear functions

What follows are two lists of nonlinear functions supported by `cvx`: first, a list of standard Matlab functions extended to work with `cvx`; and second, a list of new functions created specifically for use in `cvx`.

In some cases, limitations of the underlying solvers place certain restrictions or caveats on their use:

- Functions marked with a dagger (†) are not supported natively by the solvers that `cvx` uses. They are handled using a successive approximation method, which makes multiple calls to the underlying solver, achieving the same final precision. (See §D.1 for details.) If you use one of these functions, you will be warned that successive approximation will be used.
- Functions involving powers (*e.g.*, `x^p`) and p-norms (*e.g.*, `norm(x,p)`) are marked with a star ($\star$). `cvx` represents these functions exactly when p is a rational number. For irrational values of `p`, a nearby rational is selected using Matlab's `rat` function. See §D.2 for details.

B.2.1 Built-in functions

- `abs`: absolute value for real and complex arrays. Convex.
- † `exp`: exponential. Convex and nondecreasing.
- † `log`: logarithm. Concave and nondecreasing.
- `max`: maximum. Convex and nondecreasing.
- `min`: minimum. Concave and nondecreasing.
- $\star$ `norm`: norms for real and complex vectors and matrices. Convex. The one-argument version `norm(x)` computes the 2-norm for vectors and induced 2-norm (maximum singular value) for matrices. The two-argument version `norm(x,p)` is supported as follows:
  - For vectors, all `p` ≥ 1 are accepted, but see Appendix D.2 for more details about how `cvx` handles values other than `1`, `2`, and `Inf`.
  - For matrices, `p` must be `1`, `2`, `Inf`, or `'Fro'`.
- `polyval`: polynomial evaluation. `polyval(p,x)`, where `p` is a vector of length `n`, computes

  ```
  p(1) * x.^(n-1) + p(2) * x.^(n-2) + ... + p(n-1) * x + p(n)
  ```

  This function can be used in `cvx` in two ways:

- If `p` is a variable and `x` is a constant, then `polyval(x,p)` computes a linear combination of the elements of `p`. The combination must satisfy the DCP rules for addition and scaling.
- If `p` is a constant and `x` is a variable, then `polyval(x,p)` constructs a polynomial function of the variable `x`. The polynomial must be affine, convex, or concave, and `x` must be real and affine.

- $\star$ `power`: `x^p` and `x.^p`, where `x` is a real variable and and `p` is a real constant. For `x^p`, `x` and `p` must be scalars. Only those values of `p` which can reasonably and unambiguously interpreted as convex or concave are accepted:
  - $p = 0$. Constant. `x.^p` is identically 1.
  - $0 < p < 1$. Concave. The argument x must be concave (or affine), and is implicitly constrained to be nonnegative.
  - $p = 1$. Affine. `x.^p` is then `x`.
  - $p \in \{2, 4, 6, 8, ...\}$. Convex. Argument x must be affine.
  - $p > 1$, $p \notin \{2, 3, 4, 5, ...\}$. Convex. Argument x must be affine, and is implicitly constrained to be nonnegative.

  Negative and odd integral values of p are not permitted, but see the functions `pow_p`, `pow_pos`, and `pow_abs` in the next section for useful alternatives.

- † `power`: `p.^x` and `p^x`, where `p` is a real constant and `x` is a real variable. For `p^x`, `p` and `x` must be scalars. Valid values of `p` include:
  - `p` $\in \{0, 1\}$. Constant.
  - $0 <$ `p` < 1. Convex and nonincreasing; `x` must be concave.
  - `p` > 1. Convex and nondecreasing; `x` must be convex.

  Negative values of `p` are not permitted.

- `sqrt`: square root. Implicitly constrains its argument to be nonnegative. Concave and nondecreasing.

B.2.2 New nonlinear functions

Even though these functions were developed specifically for `cvx`, they work outside of a `cvx` specification as well, when supplied with numeric arguments.

- `berhu(x,M)`: The reversed Huber function (hence, Berhu), defined as $|x|$ for $|x| \leq M$, and $(|x|^2 + M^2)/2M$ for $|x| \geq M$. Convex. If M is omitted, $M = 1$ is assumed; but if supplied, it must be a positive constant. Also callable with three arguments as `berhu(x,M,t)`, which computes `t+t*berhu(x/t,M)`, useful for concomitant scale estimation (see [Owe06]).

- `det_inv`: determinant of inverse of a symmetric (or Hermitian) positive definite matrix, $\det X^{-1}$, which is the same as the product of the inverses of the eigenvalues. When used inside a `cvx` specification, `det_inv` constrains the matrix to be symmetric (if real) or Hermitian (if complex) and positive semidefinite. When used with numerical arguments, `det_inv` returns `+Inf` if these constraints are not met. Convex.

- `det_rootn`: n-th root of the determinant of a semidefinite matrix, $(\det X)^{1/n}$. When used inside a `cvx` specification, `det_rootn` constrains the matrix to be symmetric (if real) or Hermitian (if complex) and positive semidefinite. When used with numerical arguments, `det_rootn` returns `-Inf` if these constraints are not met. Concave.

- `det_root2n`: the $2n$-th root of the determinant of a semidefinite matrix; *i.e.*, `det_root2n(X)=sqrt(det_rootn(X))`. Concave. Maintained solely for backcompatibility purposes.

- † `entr`, the elementwise entropy function: `entr(x)=-x.*log(x)`. Concave. Returns `-Inf` when called with a constant argument that has a negative entry.

- `geo_mean`: the geometric mean of a vector, $\left(\prod_{k=1}^{n} x_k\right)^{1/n}$. When used inside a `cvx` specification, `geo_mean` constrains the elements of the vector to be nonnegative. When used with numerical arguments, `geo_mean` returns `-Inf` if any element is negative. Concave and increasing.

- `huber(x,M)`, defined as $2M|x| - M^2$ for $|x| \geq M$, and $|x|^2$ for $|x| \leq M$. Convex. If M is omitted, then $M = 1$ is assumed; but if it supplied, it must be a positive constant. Also callable as `huber(x,M,t)`, which computes `t+t*huber(x/t,M)`, useful for concomitant scale estimation (see [Owe06]).

- `huber_circ(x,M)`, the circularly symmetric Huber function, defined as $\|x\|_2$ for $\|x\|_2 \leq M$, and $2M\|x\|_2 - M^2$ for $\|x\|_2 \geq M$. Same (and implemented) as `huber_pos(norm(x),M)`. Convex.

- `huber_pos(x,M)`. Same as Huber function for nonnegative `x`; zero for negative `x`. Convex and nondecreasing.

- `inv_pos`, inverse of the positive portion, $1/\max\{x, 0\}$. Inside `cvx` specification, imposes constraint that its argument is positive. Outside `cvx` specification, returns $+\infty$ if $x \leq 0$. Convex and decreasing.

- † `kl_div`, elementwise Kullback-Leibler distance, `kl_div(x,y)=x.*log(x./y)-x+y`, for `x`, `y` nonnegative, with `x(i)` zero whenever `y(i)` is zero. Convex. Outside `cvx` specification, returns $+\infty$ if arguments aren't in the domain.

- `lambda_max`: maximum eigenvalue of a real symmetric or complex Hermitian matrix. Inside `cvx`, imposes constraint that its argument is symmetric (if real) or Hermitian (if complex). Convex.

- `lambda_min`: minimum eigenvalue of a real symmetric or complex Hermitian matrix. Inside `cvx`, imposes constraint that its argument is symmetric (if real) or Hermitian (if complex). Concave.

- `lambda_sum_largest(X,k)`: sum of the largest k values of a real symmetric or complex Hermitian matrix. Inside `cvx`, imposes constraint that its argument is symmetric (if real) or Hermitian (if complex). Convex.

- `lambda_sum_smallest(X,k)`: sum of the smallest k values of a real symmetric or complex Hermitian matrix. Inside `cvx`, imposes constraint that its argument is symmetric (if real) or Hermitian (if complex). Concave.

- `log_det`: log of determinant of a positive definite matrix, $\log\det(X)$. When used inside a `cvx` specification, `log_det` constrains its argument to be symmetric (if real) or Hermitian (if complex) and positive definite. With numerical argument, `log_det` returns `-Inf` if these constraints are not met. Concave.

- $\star$ `log_normcdf(x)`: logarithm of cumulative distribution function of standard normal random variable. Concave and increasing. The current implementation is a fairly crude SDP-representable approximation, with modest accuracy over the interval $[-4, 4]$; we intend to replace it with a much better approximation at some point.

- † `log_sum_exp(x)`: the logarithm of the sum of the elementwise exponentials of `x`. Convex and nondecreasing. This is used internally in expert GP mode, but can also be used in standard DCPs.

- `logsumexp_sdp`: a polynomial approximation to the log-sum-exp function with global absolute accuracy. This approximation is used in default GP mode, but can also be used in standard DCPs.

- `matrix_frac(x,Y)`: matrix fractional function, $x^T Y^{-1} x$. In `cvx`, imposes constraint that Y is symmetric (or Hermitian) and positive definite; outside `cvx`, returns $+\infty$ unless $Y = Y^T \succ 0$. Convex.

- `norm_largest( x, k )`, for real and complex vectors, returns the sum of the largest `k` *magnitudes* in the vector `x`. Convex.

- `norm_nuc(X)`, is the sum of the singular values of a real or complex matrix `X`. (This is the dual of the usual spectral matrix norm, *i.e.*, the largest singular value.) Convex.

- $\star$ `norms( x, p, dim )` and `norms_largest( x, k, dim )`. Computes *vector* norms along a specified dimension of a matrix or N-d array. Useful for sum-of-norms and max-of-norms problems. Convex.

- `poly_env( p, x )`. Computes the value of the *convex or concave envelope* of the polynomial described by `p` (in the `polyval` sense). `p` must be a real

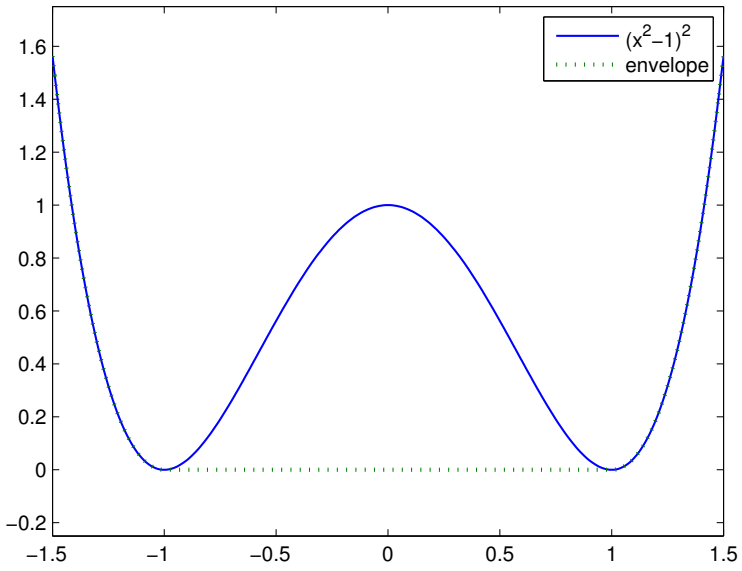

Figure 2: The polynomial function $p(x) = x^4 - 2x^2 + 1$ and its convex envelope.

constant vector whose length `n` is 0, 1, 2, 3, or some other *odd* length; and `x` must be real and affine. The sign of the first nonzero element of `p` determines whether a convex (positive) or concave (negative) envelope is constructed. For example, consider the function $p(x) \triangleq (x^2-1)^2 = x^4 - 2x^2 + 1$, depicted along with its convex envelope in Figure 2. The two coincide when $|x| \geq 1$, but deviate when $|x| < 1$. Attempting to call `polyval([1,0,2,0,1],x)` in a `cvx` model would yield an error, but a call to `poly_env([1,0,2,0,1],x)` yields a valid representation of the envelope. For convex or concave polynomials, this function produces the same result as `polyval`.

- `pos`: $\max\{x, 0\}$, for real x. Convex and increasing.
- $\star$ `pow_abs(x,p)`: $|x|^p$ for $x \in \mathbf{R}$ or $x \in \mathbf{C}$ and $p \geq 1$. Convex. If `p` is irrational, a nearby rational value is chosen; see Appendix D.2 for details.
- $\star$ `pow_pos(x,p)`: $\max\{x, 0\}^p$ for $x \in \mathbf{R}$ and $p \geq 1$. Convex and nondecreasing. If `p` is irrational, a nearby rational value is chosen; see Appendix D.2 for details.
- $\star$ `pow_p(x,p)`, for $x \in \mathbf{R}$ and real constant p computes nonnegative convex and

concave branches of the power function:

$$p \leq 0: \quad f_p(x) \triangleq \begin{cases} x^p & x > 0 \\ +\infty & x \leq 0 \end{cases} \quad \text{convex, nonincreasing}$$

$$0 < p \leq 1: \quad f_p(x) \triangleq \begin{cases} x^p & x \geq 0 \\ -\infty & x < 0 \end{cases} \quad \text{concave, nondecreasing}$$

$$p \geq 1: \quad f_p(x) \triangleq \begin{cases} x^p & x \geq 0 \\ +\infty & x < 0 \end{cases} \quad \text{convex, nonmonotonic}$$

- `quad_form(x,P)`, $x^T P x$ for real x and symmetric P, and $x^H P x$ for complex x and Hermitian P. Convex in x for P constant and positive semidefinite; concave in x for P constant and negative semidefinite. This function is provided since `cvx` will *not* recognize `x'*P*x` as convex (even when `P` is positive semidefinite).
- `quad_over_lin`, $x^T x/y$ for $x \in \mathbf{R}^n$, $y > 0$; for $x \in \mathbf{C}^n$, $y > 0$: $x^* x/y$. In `cvx` specification, adds constraint that $y > 0$. Outside `cvx` specification, returns $+\infty$ if $y \leq 0$. Convex, and decreasing in y.
- `quad_pos_over_lin`: `sum_square_pos( x ) / y` for $x \in \mathbf{R}^n$, $y > 0$. Convex, increasing in x, and decreasing in y.
- † `rel_entr`: Scalar relative entropy: `rel_entr(x,y)=x.*log(x/y)`. Convex.
- `sigma_max`: maximum singular value of real or complex matrix. Same as `norm`. Convex.
- `square`: x^2 for $x \in \mathbf{R}$. Convex.
- `square_abs`: $|x|^2$ for $x \in \mathbf{R}$ or $x \in \mathbf{C}$.
- `square_pos`: $\max\{x, 0\}^2$ for $x \in \mathbf{R}$. Convex and increasing.
- `sum_largest(x,k)` sum of the largest k values, for real vector x. Convex and increasing.
- `sum_smallest(x,k)`, sum of the smallest k values, *i.e.*, `-sum_largest(-x,k)`. Concave and decreasing.
- `sum_square`: `sum( square( x ) )`. Convex.
- `sum_square_abs`: `sum( square_abs( x ) )`. Convex.
- `sum_square_pos`: `sum( square_pos( x ) )`; works only for real values. Convex and increasing.
- `trace_inv(X)`, trace of the inverse of an SPD matrix `X`, which is the same as the sum of the inverses of the eigenvalues. Convex. Outside of `cvx`, returns `+Inf` if argument is not positive definite.

- `trace_sqrtm(X)`, trace of the matrix squareroot of a positive semidefinite matrix `X`. which is the same as the sum of the squareroots of the eigenvalues. Concave. Outside of `cvx`, returns `+Inf` if argument is not positive semidefinite.

B.3 Sets

`cvx` currently supports the following sets; in each case, `n` is a positive integer constant.

- `nonnegative(n)`:

$$\mathbf{R}^n_+ \triangleq \{\, x \in \mathbf{R}^n \mid x_i \geq 0,\ i = 1, 2, \ldots, n \,\}$$

- `simplex(n)`:

$$\mathbf{R}^n_{1+} \triangleq \{\, x \in \mathbf{R}^n \mid x_i \geq 0,\ i = 1, 2, \ldots, n,\ \textstyle\sum_i x_i = 1 \,\}$$

- `lorentz(n)`:

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

- `rotated_lorentz(n)`:

$$\mathbf{Q}^n_r \triangleq \{\, (x, y, z) \in \mathbf{R}^n \times \mathbf{R} \times \mathbf{R} \mid \|x\|_2 \leq \sqrt{yz},\ y, z \geq 0 \,\}$$

- `complex_lorentz(n)`:

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

- `rotated_complex_lorentz(n)`:

$$\mathbf{Q}^n_{rc} \triangleq \{\, (x, y, z) \in \mathbf{C}^n \times \mathbf{R} \times \mathbf{R} \mid \|x\|_2 \leq \sqrt{yz},\ y, z \geq 0 \,\}$$

- `semidefinite(n)`:

$$\mathbf{S}^n_+ \triangleq \left\{\, X \in \mathbf{R}^{n \times n} \mid X = X^T,\ X \succeq 0 \,\right\}$$

- `hermitian_semidefinite(n)`:

$$\mathbf{H}^n_+ \triangleq \left\{\, Z \in \mathbf{C}^{n \times n} \mid Z = Z^H,\ X \succeq 0 \,\right\}$$

- `nonneg_poly_coeffs(n)`: The cone of all coefficients of nonnegative polynomials of degree n; n must be even:

$$\mathbf{P}_{+,n} \triangleq \left\{ p \in \mathbf{R}^{n+1} \mid \sum_{i=0}^{n} p_{i+1} x^{n-i} \geq 0 \ \forall x \in \mathbf{R} \right\}$$

- `convex_poly_coeffs(n)`: The cone of all coefficients of convex polynomials of degree n; n must be even:

$$\mathbf{P}_{+,n} \triangleq \left\{ p \in \mathbf{R}^{n+1} \;\mid\; \sum_{i=0}^{n-2} (n-i)(n-i-1) p_{i+1} x^{n-i-2} \geq 0 \;\forall x \in \mathbf{R} \right\}$$

- `exponential_cone`:

$$\mathbf{E} \triangleq \mathrm{cl}\left\{ (x,y,z) \in \mathbf{R} \times \mathbf{R} \times \mathbf{R} \;\mid\; y > 0,\; ye^{x/y} \leq z \right\}$$

- `geo_mean_cone(n)`:

$$\mathbf{G}_n \triangleq \mathrm{cl}\left\{ (x,y) \in \mathbf{R}^n \times \mathbf{R}^n \times \mathbf{R}^n \;\mid\; x \geq 0,\; (\prod_{i=1}^{n} x_i)^{1/n} \geq y \right\}$$

C cvx status messages

After a complete cvx specification has been entered and the cvx_end command issued, the solver is called to generate a numerical result. The solver can produce one of five exit conditions, which are indicated by the value of the string variable cvx_status. The nominal values of cvx_status, and the resulting values of the other variables, are as follows:

- Solved: A complementary (primal and dual) solution has been found. The primal and dual variables are replaced with their computed values, and the the optimal value of the problem is placed in cvx_optval (which, by convention, is 0 for feasibility problems).
- Unbounded: The problem has been proven to be unbounded below through the discovery of an unbounded primal direction. This direction is stored in the primal variables. The value of cvx_optval is set to -Inf for minimizations, and +Inf for maximizations. Feasibility problems by construction cannot be unbounded below.

  It is important to understand that the unbounded primal direction is very likely *not* a feasible point. If a feasible point is required, the problem should be resolved as a feasibility problem by omitting the objective.
- Infeasible: The problem has been proven to be infeasible through the discovery of an unbounded dual direction. Appropriate components of this direction are stored in the dual variables. The values of the primal variables are filled with NaNs. The value of cvx_optval is set to +Inf for minimizations and feasibility problems, and -Inf for maximizations.

In some cases, the solver is unable to achieve the numerical certainty it requires to make one of the above determinations—but is able to draw a weaker conclusion by relaxing those tolerances somewhat; see §8.2. In such cases, one of the following results is returned:

- Inaccurate/Solved: The problem is likely to have a complementary solution.
- Inaccurate/Unbounded: The problem is likely to be unbounded.
- Inaccurate/Infeasible: The problem is likely to be infeasible.

The values of the primal and dual variables, and of cvx_optval, are updated identically to the "accurate" cases. Two final results are also possible:

- Failed: The solver failed to make sufficient progress towards a solution. The values of cvx_optval and primal and dual variables are filled with NaNs. This result can occur because of numerical problems within SeDuMi, often because the problem is particularly "nasty" in some way (*e.g.*, a non-zero duality gap).

- `Overdetermined`: The presolver has determined that the problem has more equality constraints than variables, which means that the coefficient matrix of the equality constraints is singular. In practice, such problems are often, but not always, infeasible. Unfortunately, solvers typically cannot handle such problems, so a precise conclusion cannot be reached.

The situations that most commonly produce an `Overdetermined` result are discussed in §D.3 below.

D Advanced solver topics

D.1 The successive approximation method

Prior to version 1.2, the functions requested most often to be added to the `cvx` function library were those from the exponential family, including `exp`, `log`, and various entropy functions. Unfortunately, `cvx` utilizes symmetric primal/dual solvers that simply cannot support those functions natively; and a variety of practical factors has delayed the use of other types of solvers with `cvx`.

For this reason, we have constructed a *successive approximation* method that allows symmetric primal/dual solvers to support the exponential family of functions. The precise nature of the method will be published elsewhere, but we can provide a highly simplified description here. First, we construct a global approximation for `exp` (or `log`, *etc.*.) which is accurate within a neighborhood of some center point x_0. Solving this approximation yields an approximate optimal point $\bar{x}$. We shift the center point x_0 towards $\bar{x}$, construct a new approximation, and solve again. This process is repeated until $|\bar{x} - x_0|$ is small enough to conclude that our approximation is accurate enough to represent the original model. Again, this is a highly simplified description of the approach; for instance, we actually employ both the primal and dual solutions to guide our judgements for shifting x_0 and terminating.

So far, we have been pleased with the effectiveness of the successive approximation method. Nevertheless, we believe that it is necessary to issue a warning when it is used so that users understand its experimental nature. Therefore, the first time that you solve a problem that will require successive approximation, `cvx` will issue a warning saying so. If you wish to suppress this warning, insert the command

```
cvx_expert true
```

into your model before the first use of such features.

D.2 Irrational powers

In order to implement power expressions like x^p and p-norms $\|x\|_p$ for $1 < p < \infty$, `cvx` uses an SDP-compatible method described in [AG01], and enhanced by the authors of `cvx`. This approach is exact—as long as the exponent p is rational. To determine integral values p_n, p_d such that $p_n/p_d = p$, `cvx` uses Matlab's `rat` function with its default tolerance of 10^{-6}. There is currently no way to change this tolerance. See the documentation for `rat` for more details.

The complexity of the SDP implementation depends on roughly on the size of the values p_n and p_d. Let us introduce a more precise measure of this complexity. For $p = 2$, a constraint $x^p \leq y$ can be represented with exactly one 2×2 LMI:

$$x^2 \leq y \quad \Longrightarrow \quad \begin{bmatrix} y & x \\ x & 1 \end{bmatrix} \succeq 0.$$

For other values of $p = p_n/p_d$, `cvx` generates a number of 2×2 LMIs that depends on both p_n and p_d; we denote this number by $k(p_n, p_d)$. (A number of internal variables

and equality constraints are also generated, but we ignore them for this analysis.) An empirical study has shown that for $p = p_n/p_d > 1$, `cvx` achieves

$$k(p_n, p_d) \leq \log_2 p_n + \alpha(p_n),$$

where the $\alpha(p_n)$ term grows very slowly compared to the $\log_2$ term. Indeed, for $p_n \leq 4096$, we have verified that $\alpha(p_n)$ is usually 1 or 2, but occasionally 0 or 3. Similar results are obtained for $0 < p < 1$ and $p < 0$.

The cost of this SDP representation is relatively small for nearly all useful values of p. Nevertheless, `cvx` issues a warning whenever $k(p_n, p_d) > 10$ to insure that the user is not surprised by any unexpected slowdown. In the event that this threshold does not suit you, you may change it using the command `cvx_power_warning(`*thresh*`)`, where *thresh* is the desired cutoff value. Setting the threshold to `Inf` disables it completely. As with the command `cvx_precision`, you can place a call to `cvx_power_warning` within a model to change the threshold for a single model; or outside of a model to make a global change. The command always returns the *previous* value of the threshold, so you can save it and restore it upon completion of your model, if you wish. You can query the current value by calling `cvx_power_warning` with no arguments.

D.3 Overdetermined problems

This status message `Overdetermined` commonly occurs when structure in a variable or set is not properly recognized. For example, consider the problem of finding the smallest diagonal addition to a matrix $W \in \mathbf{R}^{n\times n}$ to make it positive semidefinite:

$$\begin{array}{ll} \text{minimize} & \text{Trace}\, D \\ \text{subject to} & W + D \succeq 0 \\ & D \text{ diagonal} \end{array} \tag{16}$$

In `cvx`, this problem might be expressed as follows:

```
n = size(W,1);
cvx_begin
    variable D(n,n) diagonal;
    minimize( trace( D ) );
    subject to
        W + D == semidefinite(n);
cvx_end
```

If we apply this specification to the matrix `W=randn(5,5)`, a warning is issued,

```
Warning: Overdetermined equality constraints;
    problem is likely infeasible.
```

and the variable `cvx_status` is set to `Overdetermined`.

What has happened here is that the unnamed variable returned by statement `semidefinite(n)` is *symmetric*, but W is fixed and *unsymmetric*. Thus the problem,

as stated, is infeasible. But there are also n^2 equality constraints here, and only $n + n * (n + 1)/2$ unique degrees of freedom—thus the problem is overdetermined. The following modified version of the specification corrects this problem by extracting the symmetric part of W:

```
n = size(W,1);
cvx_begin
    variable D(n,n) diagonal;
    minimize( trace( D ) );
    subject to
        0.5 * ( W + W' ) + D == semidefinite(n);
cvx_end
```

E Acknowledgements

We wish to thank the following people for their contributions to the development of `cvx`: Toh Kim Chuan, Laurent El Ghaoui, Arpita Ghosh, Siddharth Joshi, Johan Löfberg, Almir Mutapcic, Michael Overton and his students, Art Owen, Rahul Paniker, Imre Polik, Joëlle Skaf, Lieven Vandenberghe, Argyris Zymnis. We are also grateful to the many students in several universities who have (perhaps unwittingly) served as beta testers by using `cvx` in their classwork. We thank Igal Sason for catching many typos in an earlier version of this document, and generally helping us to improve its clarity.

References

[AG01] F. Alizadeh and D. Goldfarb. Second-order cone programming. Technical Report RRR 51-2001, RUTCOR, Rutgers University, November 2001. Available at `http://rutcor.rutgers.edu/pub/rrr/reports2001/51.ps`.

[BKMR98] A. Brooke, D. Kendrick, A. Meeraus, and R. Raman. *GAMS: A User's Guide*. The Scientific Press, South San Francisco, 1998. Available at `http://www.gams.com/docs/gams/GAMSUsersGuide.pdf`.

[BKVH05] S. Boyd, S. J. Kim, L. Vandenberghe, and A. Hassibi. A tutorial on geometric programming. *Optimization and Engineering*, 2005. Available at `http://www.stanford.edu/~boyd/gp_tutorial.html`.

[BV04] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. Available at `http://www.stanford.edu/~boyd/cvxbook.html`.

[Cru02] C. Crusius. *A Parser/Solver for Convex Optimization Problems*. PhD thesis, Stanford University, 2002.

[DV05] J. Dahl and L. Vandenberghe. *CVXOPT: A Python Package for Convex Optimization*. Available at `http://www.ee.ucla.edu/~vandenbe/cvxopt`, 2005.

[FGK99] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, December 1999.

[GBY06] M. Grant, S. Boyd, and Y. Ye. Disciplined convex programming. In L. Liberti and N. Maculan, editors, *Global Optimization: from Theory to Implementation*, Nonconvex Optimization and Its Applications, pages 155–210. Springer, New York, 2006. Available at `http://www.stanford.edu/~boyd/disc_cvx_prog.html`.

[Gra04] M. Grant. *Disciplined Convex Programming*. PhD thesis, Department of Electrical Engineering, Stanford University, December 2004. See `http://www.stanford.edu/~boyd/disc_cvx_prog.html`.

[Löf05] J. Löfberg. YALMIP version 3 (software package). `http://control.ee.ethz.ch/~joloef/yalmip.php`, September 2005.

[Mat04] The MathWorks, Inc. MATLAB (software package). `http://www.mathworks.com`, 2004.

[Mat05] The MathWorks, Inc. MATLAB optimization toolbox (software package). `http://www.mathworks.com/products/optimization/`, 2005.

[MOS05] MOSEK ApS. Mosek (software package). `http://www.mosek.com`, February 2005.

[Owe06] A. Owen. A robust hybrid of lasso and ridge regression. Technical report, Department of Statistics, Stanford University, October 2006. Author's internal draft.

[Stu99] J. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11:625–653, 1999. Software available at `http://sedumi.mcmaster.ca/`.

[TTT06] K. Toh, R. Tütüncü, and M. Todd. SDPT3 4.0 (beta) (software package). `http://www.math.nus.edu.sg/~mattohkc/sdpt3.html`, July 2006.

[WB00] 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. Available at `http://www.stanford.edu/~boyd/sdpsol.html`.

Index