`cvx` Users' Guide

version 1.0 (beta)

Michael Grant
mcgrant@stanford.edu

Stephen Boyd
boyd@stanford.edu

Yinyu Ye
yyye@stanford.edu

June 9, 2006

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 adhere to a limited set of construction rules that enable them to be analyzed and solved efficiently. The set of DCPs includes most well-known classes of convex programs, including linear programs (LPs), quadratic programs (QPs), second-order cone programs (SOCPs), and semidefinite programs (SDPs). cvx can solve problems from all of these problem classes and others as well. For background on convex optimization, see the book *Convex Optimization* [BV04].

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 [BKVH05], and transforms such problems into convex form so that they can be solved efficiently.

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 those 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 was designed and implemented by Michael Grant, with input from Stephen Boyd and Yinyu Ye [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 is designed to fully exploit convexity. The specific method for implementing cvx in Matlab draws heavily from YALMIP [Löf05]. We hope to implement a version of cvx in Python, as part of the CVXOPT system [DV05], in the near future.

1.2 What is disciplined convex programming?

The term *disciplined convex programming* refers to a methodology for constructing convex optimization problems proposed by Michael Grant, Stephen Boyd, and Yinyu Ye [GBY06, Gra04]. The name was chosen to communicate the treatment of convex programming as a distinct discipline, one in which convexity is an advance requirement. In other words, it is designed to support the formulation and construction of optimization problems that the user intends *from the outset* to be convex.

Disciplined convex programming imposes a limited 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 convexity of the problem is obvious to the user. That is not to say that such problems cannot be solved; they just need to be rewritten in a conforming manner.

A detailed description of the DCP ruleset is given in §4, and it is important for anyone who wishes to actively use `cvx` to understand it. The ruleset is simple to learn, drawn from basic principles of convex analysis, and resembles the practices of those who study and use convex optimization. In addition, the ruleset actually *enables* considerable benefits, such as the 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

In this version of `cvx`, SeDuMi [Stu99] is the only core solver supported. SeDuMi is an open-source interior-point solver written in Matlab for LPs, SOCPs, SDPs, and combinations thereof. Future versions of `cvx` will support other solvers, such as MOSEK [MOS05], CVXOPT [DV05], and a general purpose solver we are developing.

Because SeDuMi supports only LPs, SOCPs, and SDPs, this version of `cvx` can only handle functions that can be represented using these problem types, and problems that can be reduced to these problem forms. Thus the current version includes such functions as ℓ_1, ℓ_2, and ℓ_∞ norms, maximum, minimum, absolute value, quadratic forms, square root, and maximum and minimum eigenvalue of a symmetric matrix, as well as a few obscure functions such as the sum of the largest k entries of a vector. Geometric programs are handled through a novel approximation method that is described briefly in Appendix E.

Some useful functions that are *not* supported in the current version include exponential and log, powers other than 2 and 0.5, and so forth. Support for such functions will require the use of one of the new solvers mentioned above.

1.4 Feedback

Any feedback and criticism you might have about this software, would be greatly appreciated. Please contact Michael Grant or Stephen Boyd (`mcgrant@stanford.edu`, `boyd@stanford.edu`) with your comments. If you discover a bug, please try to include the following in your communication:

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

Doing so will greatly increase our chances of being able to reproduce and fix the problem.

1.5 What cvx is *not*

Our goal in the design of cvx is to greatly simplify the specification and solution of convex optimization problems. One indication of the progress we have made towards this goal is the ease with which some users have encountered the limits of the software's performance, by creating models that are too large for cvx to solve.

Such limits are unavoidable due to a number of factors, including the finite resources of a typical PC, the overhead imposed by the Matlab environment, and the general-purpose design of the underlying numerical solver. Thus while cvx should perform well for medium scale problems and many structured large-scale problems, it may encounter difficulties when scaling to large problem instances.

Despite this advance warning, we would still greatly appreciate if you would send us feedback if you do encounter limits like these. In some cases, such reports have led to genuine improvements in the implementation of cvx that have increased its capacity or performance. In others, we have been able to suggest ways to find equivalent formulations to problems that cvx is able to process more efficiently.

2 A quick start

Once you have installed cvx (see §A), you can start using cvx 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, but 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 solver (which is SeDuMi, in this first version).

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.*, $(A^TA)^{-1}A^Tb$. 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.

All three of these quantities, `x`, `cvx_optval`, and `cvx_status`, 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.
```

[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.

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:

```
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:

```
f    = [ zeros(n,1); 1          ];
Ane  = [ +A,         -ones(m,1)  ; ...
           -A,       -ones(m,1) ];
bne  = [ +b;         -b         ];
xt   = linprog(f,Ane,bne);
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 following 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 beyond the `norm` command. For example, consider the Huber penalty function minimization problem

$$\text{minimize} \quad \sum_{i=1}^m \phi((Ax-b)_i), \qquad \phi(z) = \begin{cases} |z|^2 & |z| \le 1 \\ 2|z|-1 & |z| \ge 1 \end{cases} \tag{2}$$

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

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

`cvx` transforms this problem automatically into a second-order cone problem (SOCP).

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,

```
227 p = 4;
228 C = randn(p,n);
229 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
```

This problem can be solved using the `quadprog` function, but the transformation is not straightforward, so we will omit it here.

Expressions using comparison operators (`==`, `>=`, *etc.*) behave quite differently when they involve `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—including 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

[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 to within a nonzero numeric tolerance. So the equality constraints will be satisfied closely, but often not exactly.

```
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;
```

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 tradeoff 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 );
```

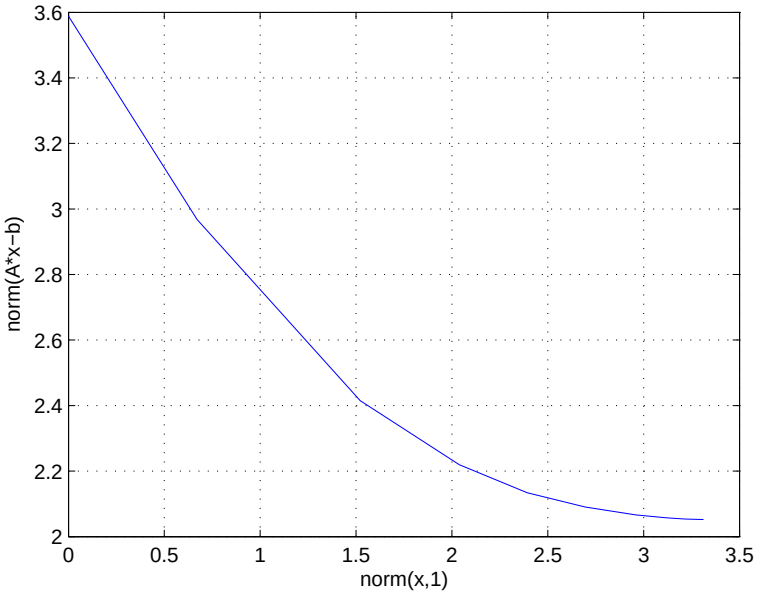

Figure 1: An example tradeoff curve from the `quickstart` demo, lines 268-286.

```
284 xlabel( 'norm(x,1)' );
285 ylabel( 'norm(A*x-b)' );
286 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

```
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 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 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`, `hankel`: An `upper_hankel` matrix is zero below the antidiagonal; a `hankel` matrix is not (necessarily).

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 statement `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.

3.2 Objective functions

Declaring an objective function requires the use of the `minimize` or `maximize` function, as appropriate. 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 §7.2.1).

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.

3.3 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`, `<` is the same as `<=`, but we encourage you to use the longer form `<=`, since it is mathematically correct.

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. If `a` and `b` are $m \times n$ matrices, for example, 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 equalities and inequalities 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); more about this in §G.1. Also note that the non-equality operator `~=` may *not* be used in a constraint, because such a constraint is rarely convex. Inequalities cannot be used if either side is complex.

`cvx` also supports a *set membership* constraint; see §3.5.

3.4 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 §7.2.1.

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. (There is a version of this function that 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.

3.5 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 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 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 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.

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,

$$\mathbf{Q}^m = \{\, (x, y) \in \mathbf{R}^m \times \mathbf{R} \;\mid\; \|x\|_2 \leq y \,\} = \mathbf{epi}\, \|\cdot\|_2, \tag{3}$$

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:

$$\begin{array}{ll} \text{minimize} & y \\ \text{subject to} & (Ax - b, y) \in \mathbf{Q}^m. \end{array} \tag{4}$$

`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 } == 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.

3.6 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

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

The dual variable y is associated with the inequality constraint $Ax \preceq b$. 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`, because they are automatically determined from the constraint to which it is assigned. 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 sucessful, `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*

$$y_i(b - Ax)_i = 0, \quad i = 1, \ldots, m. \tag{5}$$

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

$$c^T x = -1, \quad Ax \leq 0, \tag{6}$$

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

$$b^T y = 1, \quad A^T y = 0, \quad y \geq 0 \tag{7}$$

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.

The `cvx` language 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 §7.1.

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 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.

If a convex (or concave) function is not recognized as convex or concave by `cvx`, it can be added as a new atom; see §7.2.1.

4.1 A taxonomy of curvature

In disciplined convex programming, a numeric 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.

The validity of each constraint is determined *individually*, so it is not sufficient for the intersection of the constraints to be convex. For example, the set

$$\left\{ x \in \mathbf{R} \mid x^2 \geq 1,\ x \geq 0 \right\}$$

is convex (it is just the interval $[1, \infty)$), but its description includes a nonconvex constraint $x^2 \geq 1$.

4.3 Constraints

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

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

Deliberately omitted from this list are *non*-equality constraints; *i.e.*, constraints using the `~=` ($\neq$) operator. This is because, in general, 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 §3.5 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 numeric 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 numeric 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 sums or difference of affine expressions;
  - the product of an affine expression and a constant.
- A valid convex expression is
  - a valid constant, affine, or log-convex expression (§6);
  - a valid call to a function in the atom library with a convex result;
  - 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 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 (or their geometric programming counterparts described in §6.3), then it is rejected by `cvx`. For matrix and array expressions, these rules are applied on an elementwise basis. We note that the set of rules listed above is redundant; there are much smaller, equivalent sets of rules.

Of particular note is that these expression rules 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, so it is rejected by `cvx`. 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.

The function $\sqrt{x^2+1}$, written as the expression `sqrt(x^2+1)`, is another example of a convex function that cannot be verified as convex using the `cvx` ruleset. (It is recognized as convex when it is written as `norm([x,1])`, since `norm` is in the `cvx` base library of convex functions.)

4.5 Functions

Because functions in the `cvx` atom library are created in the Matlab language, they differ in certain ways from functions described in a formal mathematical sense, so let us address these differences here.

Functions are categorized in two "dimensions": *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. As examples, we have:

`sum( x )`	$\sum_i x_i$	affine, nondecreasing
`abs( x )`	$\lvert x \rvert$	convex, nonmonotonic
`log( x )`	$\log x$	concave, nondecreasing

Convexity and monotonicity are always determined in an extended-valued sense. For example, when its argument is a nonconstant `cvx` expression, the square root function `sqrt( x )` is interpreted by `cvx` as follows:

$$\texttt{sqrt(x)} \quad \begin{cases} \sqrt{x} & x \geq 0 \\ -\infty & x < 0 \end{cases} \quad \text{concave, nondecreasing}$$

(This intepretation has the effect of constraining the argument `x` to be nonnegative.) Furthermore, `cvx` does *not* consider a function to be convex or concave if it is so only over a portion of its domain. For example, consider the function

$$\texttt{1/x} \quad 1/x \ \ (x \neq 0) \quad \text{not convex/concave/affine, nonomonotonic}$$

This function is convex if its argument is positive, and concave if its argument is negative. Over its whole domain, though, it is neither convex nor concave; and in `cvx`, of course, it is not recognized as either convex or concave. However, the function

$$\texttt{inv\_pos(x)} \quad \begin{cases} 1/x & x > 0 \\ +\infty & x \leq 0 \end{cases} \quad \text{convex, nonincreasing}$$

which is convex and nonincreasing, has been included in the atom library, so it can be used freely in constraints and objective functions.

For functions that have multiple arguments, additional considerations must be made. First of all, curvature is always considered *jointly*, but monotonicity can be considered on an argument-by-argument basis. For example,

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

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

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

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.*,

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

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 as an argument an affine expression (assuming it is of compatible size).

(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, then

```
square( x )
```

is accepted by `cvx` as a convex expression—and, thanks to the above rule, so is

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

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

DCP also allows nonlinear functions to be combined using more sophisticated composition rules. The general DCP rules that govern these compositions are as follows:

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

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

(In each case, we assume that the argument is of compatible size.) For more on 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 specific manifestations of these four rules.

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

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

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

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

are both valid nonlinear compositions in `cvx`, according to the convex/nondecreasing and concave/nondecreasing rules, respectively.

Let us 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, for example, the first term, `sqrt(f'*x)`. `cvx` recognizes `f'*x` as affine, and it knows that a concave increasing function of a concave function is concave, so it concludes that `sqrt(f'*x)` is concave. (Incidently, when `sqrt` is used inside a `cvx` specification, it adds the constraint that its argument must be nonnegative.) `cvx` recognizes that `-norm(A*x-b)` is concave, since it is the negative of a convex function; and it knows that the minimum of two concave functions is concave, so it concludes that the second term, `min(4,1.3-norm(A*x-b))`, is also concave. The whole expression is then recognized as concave, since it is the sum of two concave functions.

As these composition rules are sufficient but not necessary, some expressions which are obviously convex or concave will fail to satisfy them. 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( A * x + b ) )
```

where `square( A * x + b )` was defined in the previous section. This expression effectively raises each element of `A*x+b` to the fourth power, and it is clearly elementwise convex. But `cvx` will not recognize it as such, because a convex function of a convex expression is *not* necessarily convex: for example,

```
square( square( A * x + b ) - 1 )
```

is not convex. The problem is that `square` is not monotonic, so the composition rules cannot be applied.

Examination of these examples reveals a key difference: in the first, the argument to the outer `square` is nonnegative; in the second, it is not. If `cvx` had access to this information, it could indeed conclude that the first composition is convex by applying a modified form of the first nonlinear composition rule. Alas, for now, `cvx` does not have such access; so it cannot make such conclusions. It must instead require that outer functions be *globally* monotonic.

There are many ways to modify this example that it complies with the ruleset. One is to recognize that a constraint such as

```
square( square( a * x + b ) ) <= 1
```

is equivalent to

```
square( a * x + b ) <= z
square( z ) <= 1
```

where `z` is a new variable. The equivalence of this decomposition is a direct consequence of the monotonicity of `square` when `z` is nonnegative. In fact, `cvx` performs this decomposition itself for those nonlinear compositions that it accepts.

Another approach is to use an alternate outer function `square_pos`, which we have included in the library to implement the following function f:

$$f(x) \triangleq (\max\{x, 0\})^2 = \begin{cases} x^2 & x \geq 0 \\ 0 & x \leq 0 \end{cases}$$

Obviously, `square` and `square_pos` coincide when their inputs are nonnegative. But `square_pos` is globally nondecreasing, so it can be used as the outer function in a nonlinear composition. Thus, the expression

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

is mathematically equivalent to the rejected version, but satisfies the DCP ruleset and is therefore acceptable to `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 be a useful addition as well.

Yet another approach is to use one of the above techniques within a *new* function, say `power4`, that enables the original expression to be rewritten

```
power4( A * x + b )
```

and new expressions can now take advantage of the effort. In §7.2.1, we show how to accomplish this.

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 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:

`square( x )`	$\Longleftarrow$	`conj(x) .* x`
`sum_square( y )`	$\Longleftarrow$	`y' * y`
`quad_form( A * x - b, Q )`	$\Longleftarrow$	`(A*x-b)'*Q*(Ax-b)`

In other words, `cvx` detects quadratic expressions such as those on the right 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 left above.

When we say "single-term" above, we mean either a single product of affine expressions, or a single squaring of an affine expression. That is, `cvx` verifies each quadratic term *independently*, not collectively. So, 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 not convex. But 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 Semidefinite programming using `cvx`

Those who are familiar with *semidefinite programming* (SDP) know that the constraints that utilize the set `semidefinite(n)` in §3.5 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 §7.1.

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`.

6 Geometric programming using `cvx`

Geometric programs (GPs) are a special class 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. A tutorial on geometric programming is beyond the scope of this document, so we refer the reader to [BKVH05].

To use utilize this 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 solver used in this version of `cvx`, SeDuMi, does not support geometric programming natively. Therefore, we have developed an approximation method which allows `cvx` to use SeDuMi to solve GPs and related problems. The method converts GPs to SDPs, and the approximation error can be bounded in advance. Details of the

method are outside of the scope of this document, but Appendix §E provides a brief introduction and describes how to control its accuracy. While this approach is ultimately *not* suited for very large GPs, it has proven in practice to perform respectably on small- and medium-scale problems.

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

6.1 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.

6.2 Constraints

Three types of constraints may be specified in geometric programs:

- An *equality constraint* `==` 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.

6.3 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`, `geomean`, `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`, `geomean`, `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.

7 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.

7.1 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:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n}(n-i)X_{ii} \\ \text{subject to} & \sum_{i=1}^{n} X_{i,i+k} = b_k, \;\; k = 1, 2, \ldots, n \\ & X \succeq 0 \end{array} \tag{8}$$

([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 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.

7.2 Expanding the `cvx` atom library

The restrictions imposed by the DCP ruleset imply that, for a fixed atom library, the variety of constraints and objective functions that can be constructed is limited. Thus in order to preserve generality in disciplined convex programming, the atom library must be extensible. `cvx` allows you to define new convex and concave functions, and new convex sets, that can be used in `cvx` specifications. This can be accomplished in a number of ways, which we describe in this section.

7.2.1 Defining new functions via overloading

The simplest method of constructing a new function is to use the standard function definition mechanism in Matlab, and to rely on the overloaded operators to do the right thing when the function is invoked inside a `cvx` specification.

To illustrate this, we consider the *deadzone* function, defined as

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

The deadzone function is convex in x.

It's very straightforward to add this function to `cvx`. In a file `deadzone.m` we put the following code:

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

[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`.

This is nothing but a standard Matlab function, and it works outside of `cvx` (*i.e.*, when it arguments are numerical), and also inside `cvx` (when its arguments are expressions). The reason it works inside a `cvx` specification is that the operations carried out all 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. Inside `cvx`, we can use `deadzone` anywhere we can use `norm`, for example; `cvx` knows that it is a convex function.

Now consider what happens if we replace `max` with `min` in our definition of the `deadzone` function. (The function $\min\{|x|-1,0\}$ is neither convex nor concave.) The modified function will work *outside* a `cvx` specification, happily computing and returning the *number* $\min\{|x|-1,0\}$, given a *numerical* argument x. But inside a `cvx` specification, and invoked with a nonconstant argument, the modified function won't work, since we haven't followed the DCP composition rules.

This method of defining an ordinary function, and relying on overloading of operators inside a `cvx` specification, works provided you make sure that all the calculations carried out inside the function satisfy the DCP composition rules.

7.2.2 Defining new functions via incomplete specifications

Suppose that $S \subset \mathbf{R}^n \times \mathbf{R}^m$ is a convex set and $g : (\mathbf{R}^n \times \mathbf{R}^m) \rightarrow (\mathbf{R} \cup +\infty)$ is a convex function. Then convex analysis tells us that

$$f : \mathbf{R}^n \rightarrow (\mathbf{R} \cup +\infty), \quad f(x) \triangleq \inf \{\, g(x,y) \mid (x,y) \in S \,\} \tag{9}$$

is also a convex function. In effect, f has been expressed in terms of a parameterized family of convex optimization problems, where x is the family parameter. One special case should be very familar: if $n = 1$ and $g(x,y) \triangleq y$, then

$$f(x) \triangleq \inf \{\, y \mid (x,y) \in S \,\} \tag{10}$$

gives the classic *epigraph* representation of f; that is, $S = \mathbf{epi}\, f$.

A key feature of `cvx` is the ability to define a convex function in this very manner: that is, in terms of a parameterized family of disciplined convex programs. We call the underlying convex program in such cases an *incomplete specification*—so names 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 quite powerful, allowing `cvx` to support a variety of functions that cannot be employed in a traditional, derivative-based optimization framework.

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

$$h : \mathbf{R} \rightarrow \mathbf{R}, \quad h(x) \triangleq \begin{cases} x^2 & |x| \leq 1 \\ 2|x| - 1 & |x| \geq 1 \end{cases} \tag{11}$$

The Huber penalty function cannot be used in an optimization algorithm utilizing Newton's method, because its Hessian is zero for $|x| \geq 1$. However, it can be expressed

in terms of the following family of convex QPs, parameterized by x:

$$\begin{array}{ll} \text{minimize} & 2v + w^2 \\ \text{subject to} & |x| \leq v + w \\ & w \leq 1 \end{array} \tag{12}$$

Specifically, the optimal value of the this QP is equal to the Huber penalty function. 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;
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.

What is most important, however, is that if `huber` is used within a `cvx` specification, with an affine `cvx` expression for it argument, then `cvx` will do the right thing. In this case, the function `huber` will contain a special Matlab object that represents the function call in constraints and objectives. Thus `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 the set S above and a concave function $\bar{g} : (\mathbf{R}^n \times \mathbf{R}^m) \rightarrow (\mathbf{R} \cup +\infty)$ is concave, the function

$$\bar{f} : \mathbf{R} \rightarrow (\mathbf{R} \cup +\infty), \quad \bar{f}(x) \triangleq \sup \{ \, g(x,y) \mid (x,y) \in S \, \} \tag{13}$$

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

$$\bar{f}(x) \triangleq \sup \{ \, y \mid (x,y) \in S \, \} \tag{14}$$

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

$$f : \mathbf{R}^{n \times n} \rightarrow \mathbf{R}, \quad f(X) = \lambda_{\min}(X + X^T) \tag{15}$$

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

$$\mathbf{hypo}\, f = \{ \, (X,t) \mid f(X) \geq t \, \} = \left\{ \, (X,t) \mid X + X^T - tI \succeq 0 \, \right\} \tag{16}$$

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 are illustrated well by our `huber` example above. First of all, as written the function works only with scalar values. To apply it 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;
cvx_end
```

This version of `huber` will in effect create `prod( sx )` "instances" of the problem in parallel; and when used in a `cvx` specification, `cvx` will do the right thing.

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 convex program. (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;
        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 numeric version in a subdirectory called `@double`. Matlab will call the `@double` version only when its arguments are numeric, and it will call your `cvx` version in other cases. 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.

A Installing cvx

cvx requires Matlab 6.1 or later.[4] We are able to precompile and test the SeDuMi and cvx MEX files on the following platform combinations:

- Windows (32-bit): Matlab 6.1, 6.5, 7.1, and 7.2
- Linux (32-bit): Matlab 6.5, 7.0, 7.1
- Solaris 8: Matlab 6.5, 7.1
- Mac OS X (PowerPC only): Matlab 7.1

If your platform combination differs only slightly from these—say, for example, you are running Matlab 7.0 on Windows—then the instructions in §A.1 below should still work. Nevertheless, we encourage you to upgrade to at least upgrade to the latest minor version of Matlab if at all possible. For other platforms besides those listed above, see see §A.2 before proceeding.

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. Unpack the file anywhere you like; a directory called `cvx` will be created. However, if you have been running a previous version of `cvx`, *remove the old version*, or move it out of the way, before proceeding.
3. Start Matlab.
4. Change the current directory to the location of `cvx`. For example, if you unpacked `cvx.zip` on your Windows machine 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.
5. Type the command

   ```
   cvx_setup
   ```

[4] Previous versions of this manual stated that cvx required Matlab 6.0 or later; we have since discovered that version 6.0 is not sufficient.

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.

6. 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 Unsupported platforms

It may be possible to run `cvx` on other platforms that run MATLAB 6.0, but the MEX files must be successfully compiled to do so. Therefore, before following the instructions in §A.1 above, you must confirm that your MEX system has been set up properly. If you have compiled MEX files before, this is likely the case; otherwise, run the command

```
mex -setup
```

and answer the prompted questions. If necessary, consult the Matlab documentation for details.

With the MEX system properly configured, following the instructions in §A.1 should work. The `cvx_setup` script will automatically compile the MEX files for both SeDuMi and `cvx`. Unfortunately, the authors do not have ready access to these platforms, so if you are unable to compile the MEX files for your platform, we may not be able to help. Nevertheless, we have tried to make the installation process simple, and want to hear from you if you run into problems, so please send us a note. If you *are* successful, please consider sending us the MEX files produced, so that we make consider including them in a future `cvx` distribution.

Important note: If you are using Linux, do *not* delete the MEX files and recompile them again. Doing so can result in errors that prevent SeDuMi and `cvx` from running. If you remove them inadvertently, just reinstall them from the distribution.

A.3 About SeDuMi

The `cvx` distribution includes a full copy of the SeDuMi package in the directory `cvx/sedumi`. This version incorporates some bug fixes that are *not* currently found in the versions fo SeDuMi available from the web. Therefore, we *strongly* recommend that you use our version of SeDuMi, and remove any other versions that you may have from your Matlab path.

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. The use of the exponentiation operators `^ .^` are limited to instances that produce quadratic forms; so, for example, the exponent must be 2.

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  cumsum  diag  dot  fliplr  flipud  flipdim  horzcat  hankel
ipermute  kron  permute  repmat  reshape  rot90  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 `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 Standard Matlab nonlinear functions

The following standard Matlab nonlinear functions are overloaded to work in `cvx`:

- `abs`, elementwise absolute value for real and complex arrays. Convex.
- `max`, `min`, elementwise maximum and minimum, for real arrays only. The 1, 2, and 3-argument syntaxes are all supported. `max` is convex and increasing; `min` is concave and decreasing.
- `norm( x, 1 )` and `norm( x, Inf )`, for real and complex vectors and matrices. Convex.
- `norm( x )` or `norm( x, 2 )`, for real and complex vectors and matrices. Convex.
- `norm( X, 'fro' )`, for real and complex matrices. Convex.
- `sqrt`, elementwise squareroot, for real arrays only. Adds constraint that entries are nonnegative. Concave and increasing.

We note that in this version of `cvx`, most powers such as `x^4` or `x^(1/3)`, are *not* supported. The only exception is the power of 2 which is translated to an appropriate call to `square`. Furthermore, an expression such as `x^(1/2)` can usually be adequately represented by `sqrt(x)`, as long as `x` is intended to be real and nonnegative.

B.3 New nonlinear functions

The following functions are part of the base `cvx` library. These functions work both inside a `cvx` specification, and outside a `cvx` specification (with numerical arguments).

- `geomean`: the geometric mean of a vector, $(\prod_{k=1}^{n} x_k)^{1/n}$. When used inside a `cvx` specification, `geomean` constrains the elements of the vector to be nonnegative. When used with numerical arguments, `geomean` returns `-Inf` if any element is negative. Concave and increasing.
- `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.
- `huber(x)`, defined as $2|x| - 1$ for $|x| \geq 1$, and x^2 for $|x| < 1$. Convex.

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

- `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.

- `log(sum(exp(x)))`: the logarithm of the sum of the elementwise exponentials of `x`. Convex and nondecreasing. This is used internally in GP mode, but can also be used in standard DCPs. This function is actually *approximated* in the current version of `cvx`; see §E for details.

- `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. Convex.

- `norms( x, p, dim )` and `norms( x, 'largest', 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.

- `pos`, $\max\{x, 0\}$, for real x. Convex and increasing.

- `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.

- `sigma_max`: maximum singular value of real or complex matrix. Same as `norm`. Convex.

- `square`: x^2 for $x \in \mathbf{R}$; $|x|^2$ for $x \in \mathbf{C}$. Convex.

- `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_smallest(-x,k)`. Concave and decreasing.
- `sum_square`: `sum( square( x ) )`. Convex.
- `sum_square_pos`: `sum( square_pos( x ) )`; works only for real values. Convex and increasing.

B.4 Sets

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

- `nonnegative(n)`:

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

- `lorentz(n)`:

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

- `rotated_lorentz(n)`:

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

- `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)`:

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

- `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)`:

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

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 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 `NaN`s. The value of `cvx_optval` is set to `+Inf` for minimizations and feasibility problems, and `-Inf` for maximizations.

In some cases, SeDuMi 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. In such cases, one of the following results is returned:

- `Solved/Inaccurate`: The problem is likely to have a complementary solution.

- `Unbounded/Inaccurate`: The problem is likely to be unbounded.

- `Infeasible/Inaccurate`: 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 `NaN`s. This result can occur because of significant numerical problems within SeDuMi, or 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 necessarily, infeasible. Unfortunately, SeDuMi cannot handle such problems, so a precise conclusion cannot be reached.

The situations that most commonly produce an `Overdetermined` result are discussed in §G.2 below.

D Controlling solver precision

Numerical methods for convex optimization are not exact; they compute their results to within a predefined numerical precision or tolerance. The precision chosen by default in `cvx`, which in turn is inherited from the defaults chosen by its solver SeDuMi, should be entirely acceptable for most applications. Nevertheless, you may wish to tighten or relax that precision in some applications.

There are several ways to call the `cvx_precision` command. If you call it with no arguments, it simply returns a two-element vector of the current precision settings. The first element in that vector is the standard precision; the precision the solver must obtain to return `Solved`, `Unbounded`, or `Infeasible`. The second element in the vector is the "reduced" precision, the precision that the solver must achieve in order to return `Solved/Inaccurate`, `Unbounded/Inaccurate`, `Infeasible/Inaccurate`.

Calling `cvx_precision` with an argument allows you to actually *change* the precision level. One way is to supply a string as an argument, either in command mode or function mode, chosen from one of five values:

- `cvx_precision low`: standard = reduced = $\epsilon^{1/4} \approx 1.2 \times 10^{-4}$.
- `cvx_precision medium`: standard = $\epsilon^{3/8} \approx 1.3 \times 10^{-6}$, reduced = $\epsilon^{1/4}$.
- `cvx_precision default`: standard = $\epsilon^{1/2} \approx 1.5 \times 10^{-8}$, reduced = $\epsilon^{1/4}$.
- `cvx_precision high`: standard = $\epsilon^{3/4} \approx 1.1 \times 10^{-11}$, reduced = $\epsilon^{3/8}$.
- `cvx_precision best`: standard = 0, "reduced" = $\epsilon^{1/2}$ (see below).

In function mode, these calls look like `cvx_precision('low')`, etc. The `best` precision setting is special: it instructs the solver to continue until it is completely unable to make progress. Then, as long as it reaches at least the "reduced" precision of $\epsilon^{1/2}$, it may claim a successful solution; otherwise, it returns a `cvx_status` value of `Failed`. An `Inaccurate` status value is not possible in `best` mode (nor, for that matter, in `low` mode, which sets the standard and reduced precisions to be identical).

The `cvx_precision` command can also be called with either a scalar or a length-two vector. If you pass it a scalar, it will assume that as the standard precision, 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 a vector of values, then the smallest value will be chosen as the standard precision, and the larger value as the reduced precision.

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, for example,

```
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, for example,

```
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. with multiple 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.

E The GP/SDP approximation

The solver currently employed by `cvx` does not support geometric programs natively. Instead, `cvx` *approximates* the convex form of a DGP in an SDP-representable manner. Details of the approximation method will be published separately; but in short, it replaces each posynomial constraint with a series of polynomial approximations. (Equalities and monomial inequalities do not require approximation.)

A key feature of this approximation method is that the error introduced at each posynomial can be bounded in advance. Specifically, given a constraint of the form

$$f_i(x) \leq h_i(x) \quad f_i(x) \text{ posynomial, } h_i(x) \text{ monomial} \tag{17}$$

the approximation produced by `cvx` satisfies

$$f_i(x) \cdot (1 + \delta_i) \leq g_i(x), \quad \delta \in [0, \epsilon_{GP}] \tag{18}$$

where δ_i is unknown but bounded as indicated. The approximation is applied to posynomial objective functions as well. Thus if the objective function is a posynomial $f_0(x)$, it will be perturbed to $f_0(x)(1 + \delta_0)$, where $\delta_0 \in [0, \epsilon_{GP}]$.

The approximations are *conservative*, in that any value of x that satisfies an approximated constraint (18) will, in fact, satisfy the original (17) as well. Applied to an entire DGP, this means that if `cvx` locates a feasible point for the approximated problem, that point is guaranteed to be feasible as well. Furthermore, if `cvx` successfully obtains an optimal value for the approximate problem, that value will serve as an upper bound on the optimal value of the original problem (for a minimization; for a maximization, it will serve as a lower bound).

In its default setting, `cvx` uses a tolerance value of $\epsilon_{GP} = 0.001$. For most engineering applications, this is likely to be quite sufficient. However, if you wish to change it, you may do so using the command `cvx_gp_precision(`*tol*`)`, where *tol* is the new value of ϵ_{GP}. As with the command `cvx_precision`, you can place the call either within a model,

```
cvx_begin gp
    cvx_gp_precision(tol);
    ...
cvx_end
```

to change the approximation quality for a single model; or outside of a model,

```
cvx_gp_precision(tol);
cvx_begin gp
    ...
cvx_end
```

to make a global change. Consistent with `cvx`'s other control commands, the command returns the *previous* value of the ϵ_{GP}, so you can save it and restore it upon completion of your model, if you wish. You can query the value of ϵ_{GP} at any time by calling `cvx_gp_precision` with no arguments.

It is important to understand that the complexity of the approximation—the order of the polynomials used, as well as the number of them—increases as the number of terms in the posynomial increases, and as the value of ϵ_{GP} decreases. Therefore, seeking high accuracies or solving problems involving posynomials with a vary large number terms can easily produce very large, slow approximations.

This approximation can actually be used in DCPs as well. Specifically, the function `logsumexp_sdp` uses this approximation method to implement the function

$$f : \mathbf{R}^n \to \mathbf{R}, \quad f(x) \triangleq \log(\textstyle\sum_{i=1}^n e^{x_i}) + [0, \delta(x)], \quad \delta(x) \in [0, \epsilon] \tag{19}$$

If you call this function directly, it uses a default tolerance of $\epsilon = 0.001$, regardless of the value set in `cvx_gp_precision`. You can supply a different tolerance as a second argument; *i.e.*, `logsumexp_sdp( x, tol )`. However, it turns out that `cvx` admits a particular exception to the DCP nonlinear composition rules: it allows the construction of expressions of the form `log(sum(exp(x)))`. (This is not an exception so much as it is a byproduct of the unified approach to verifying DCPs and GPs.) Such constructions are automatically recognized by `cvx`, which utilizes the precision value ϵ_{GP} in determining the accuracy of their approximation.

F 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.*

G Caveats

G.1 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, so let us address it for a moment.

The consequences of using assignments within a `cvx` specification, whether accidentally or inadvertently, can often cause subtle problems, so in certain instances, `cvx` takes steps to prevent you from doing anything dangerous. For example, suppose $A, C \in \mathbf{R}^{n\times n}$ and $b \in \mathbf{R}$ are given, and consider the simple SDP

$$\begin{array}{ll} \text{minimize} & \mathbf{Tr}\, CX \\ \text{subject to} & \mathbf{Tr}\, AX = b \\ & X \succeq 0 \end{array} \tag{20}$$

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 looks like it constrains `X` to be positive semidefinite; but in fact, it is an assignment, not an equality constraint. So it actually *overwrites* the variable `X` with an anonymous, positive semidefinite variable; and `X` is *not* constrained to be positive semidefinite. The numerical consequences are, of course, considerable—but Matlab (and previous versions of `cvx`) would happily accept and process this problem without complaint, leaving you (perhaps) mystified why the results are not what you expect.

No longer. From now on, when `cvx_end` is reached, `cvx` will examine each declared variable, primal and dual, and verify that it still points to the 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. Other kinds of assignments are still permitted; and in some cases, they may be genuinely useful. For example, consider the following `cvx` excerpt:

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

This excerpt uses a Matlab variable `z` to hold an intermediate calculation `2 * x - y` so that it can be used in two different constraints. It is technically correct, and `cvx` will do the right thing with it. In fact, if the value of `z` is not altered for the remainder of the problem, `cvx` will actually replace it with its proper *numerical* value (*i.e.*, `2 * x - y`) after solving the problem.

Having said that, 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;
```

This is the approach we would recommend that you adopt instead. Declaring intermediate calculations as variables provides an extra measure of clarity in your models, in our opinion. You need not worry that these "extra" variables will make your models slower. In fact, in many cases, this practice may *improve* performance by exposing more of your problem's structure. Even when this is not the case, `cvx`'s presolve routine insures that it will perform no *worse* than the first form.

What about assignments involving purely constant expressions, such as `a=2`? We would simply recommend that for style reasons they be placed *before* the `cvx_begin` command, as we do in the examples presented in this document.

G.2 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:

$$\begin{array}{ll} \text{minimize} & \text{Trace}\, D \\ \text{subject to} & W + D \succeq 0 \\ & D \text{ diagonal} \end{array} \tag{21}$$

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
```

H Acknowledgements

We wish to thank the following people for their contributions to the development of `cvx`: Laurent El Ghaoui, Arpita Ghosh, Siddharth Joshi, Johan Löfberg, Almir Mutapcic, Michael Overton and his students, Rahul Panicker, 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.

References

[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 Science+Business Media, Inc., 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.

[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/`.

[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