cvx Users' Guide

version 0.85 (alpha)

Michael Grant
mcgrant@stanford.edu

Stephen Boyd
boyd@stanford.edu

Yinyu Ye
yyye@stanford.edu

September 1, 2005

1 Introduction

cvx is a modeling system that supports *disciplined convex programming*. Disciplined convex programs (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), (convex) quadratic programs (QPs), second-order cone programs (SOCPs), and semidefinite programs (SDPs). In fact, almost any convex optimization problem that one is likely to encounter in practice, even nondifferentiable problems, can be represented relatively easily as a DCP. For background on convex optimization, see the book *Convex Optimization* [BV04]; for more discussion of disciplined convex programming, see [GBY05, Gra04].

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 converts problems into a canonical form that is compatible with interior-point algorithms. Problems involving nondifferentiable functions can be automatically transformed into equivalent smooth forms, enabling them to be solved without the loss in performance typically associated with nondifferentiability. A significant amount of problem sparsity and block structure is revealed to the underlying solver, which can be exploited to improve performance.

`cvx` comes with a base library of convex and concave functions and convex sets that can be used directly in the construction of DCPs. New functions can be added in three ways:

- Existing functions can be combined according to the disciplined convex programming composition rules.
- A function can be described as the optimal value of an partially specified convex optimization problem, expressed in the `cvx` modeling language.
- A smooth convex or concave function can be described by specifying its convexity and monotonicity properties and supplying code to compute its value, gradient, and Hessian.

This last option is not available in this alpha version (see §1.2 below). The base library also includes several useful convex sets, such as the positive semidefinite matrix cone. In the future, `cvx` will also allow new sets to be added in two ways:

- The set can be described using the `cvx` modeling language in the form of a convex feasibility problem.
- Code to compute the value, gradient, and Hessian of a barrier function for the set may be provided.

`cvx` was designed and implemented by Michael Grant, with input from Stephen Boyd and Yinyu Ye [GBY05]. 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.

1.1 Disciplined convex programming

`cvx` is designed to support *disciplined convex programming*. This name was chosen to communicate the treatment of convex programming as a discipline in which the convexity of the problem being formulated is considered an advance requirement. In other words, `cvx` is designed to support the formulation and construction of optimization problems that the user intends *from the outset* to be convex. To that end, `cvx` imposes a limited set of conventions or rules, which we call the *DCP ruleset*, that enable it to verify convexity and convert problems to solvable form reliably and efficiently. 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 still closely resemble their natural mathematical forms.

1.2 This version

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

Because SeDuMi only solves 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. Some useful functions that are *not* supported in the current version include exponential and log, powers other than 2 and 0.5, entropy, log-sum-exp, and the log-determinant of a matrix.

Several other features planned for `cvx` are *not* supported in this release, including user-defined smooth functions, user-defined sets, and geometric programming. We hope to extend `cvx` to handle these fairly soon.

Any feedback and criticism you might have about this software, particularly at this early stage, would be greatly appreciated. Please contact Michael Grant (`mcgrant@stanford.edu`) or Stephen Boyd (`boyd@stanford.edu`) with your comments. If you discover a bug, please notify us at the above addresses; and if at all possible, please include:

- the code and 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.

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, 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 objective functions 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 it to be solved.

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

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

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

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

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

- `cvx_optval`, which contains the value of the objective function; *i.e.*, $\|Ax - b\|_2$;
- `cvx_status`, which contains a string describing the status of the calculation. In this case, `cvx_status` would contain the string `Solved`.

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
Dispciplined convex programming error:
Objective function in a maximization must be concave.
```

2.2 Bound-constrained least-squares

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

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

where l and u are given data, vectors with the same dimension as the variable x. The vector inequality $u \preceq v$ means componentwise, *i.e.*, $u_i \leq v_i$ for all i. We can no longer use the simple backslash notation to solve this problem, but it can be transformed into 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) )
```

[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.

```
62      subject to
63          x >= l;
64          x <= u;
65  cvx_end
```

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

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

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

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

2.3 Other norms and functions

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

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

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

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

```
108 cvx_begin
109     variable x(n)
110     minimize( norm(A*x-b,Inf) )
111 cvx_end
```

The code based on `linprog`, and the `cvx` specification above, will both solve the Chebyshev norm minimization problem, in the sense that 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_ls = 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 will will omit it here. But this norm is in the base `cvx` library, so to specify and solve the problem using `cvx` is easy; *e.g.*,

```
k = 5;
cvx_begin
    variable x(n)
    minimize( norm(A*x-b,'largest',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(A*x-b,'largest',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

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

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 lines

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

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

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

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

```
cvx_begin
    variable x(n)
    minimize( norm(A*x-b) )
    subject to
        C*x == d;
        norm(x,Inf) <= 1;
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 1s and 0s, 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 upon the numeric value of x.

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

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

results in the following error:

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

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

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

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. For now, let us just say that the rules are relatively intuitive and are unlikely to significantly hinder the construction of problems in practice.

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 solves the problem of minimizing $\|Ax-b\|_2+\gamma\|x\|_1$,

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

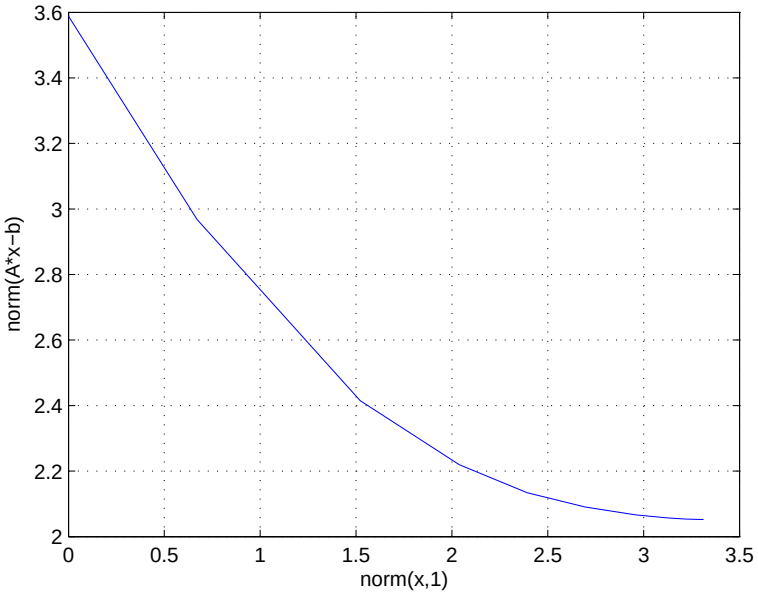

Figure 1: An example tradeoff curve from the `quickstart` demo, lines 268-286.

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.

```
268 gamma = logspace( -2, 2, 20 );
269 l2norm = zeros(size(gamma));
270 l1norm = zeros(size(gamma));
271 fprintf( 1, '   gamma       norm(x,1)    norm(A*x-b)\n' );
272 fprintf( 1, '-----------------------------------------\n' );
273 for k = 1:length(gamma),
274     fprintf( 1, '%8.4e', gamma(k) );
275     cvx_begin
276         variable x(n)
277         minimize( norm(A*x-b)+gamma(k)*norm(x,1) )
278     cvx_end
279     l1norm(k) = norm(x,1);
280     l2norm(k) = norm(A*x-b);
281     fprintf( 1, '   %8.4e   %8.4e\n', l1norm(k), l2norm(k) );
282 end
283 plot( l1norm, l2norm );
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 function known to be convex can be multiplied by a nonnegative scalar, or added to another function

known to be convex, and the result is then known to be 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 objective functions.

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, `X` is a real 20×10 matrix optimization variable, `Y` is a real 50×50 symmetric matrix optimization variable, and `Z` is a complex 100×100 Hermitian matrix variable. It is actually possible for the structure keywords to be applied to n-dimensional arrays as well: each 2-dimensional "slice" of the array is given the stated structure. The list of possible 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; that is, it is 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.*). Those 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 §5.1).

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). Also note that the non-equality operator `~=` may *not* be used in a constraint, because it 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 §5.1.

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, Matlab allows us to simplify the third line to

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

because Matlab accepts comparisons between arrays and scalars 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,

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

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`, together, lie in the appropriately-sized Lorentz cone.

`cvx` also provides a function `in(x,S)` as an alternative to the set membership constraint syntax `x==S()`. For example, the lines

```
X == semidefinte(n);
{ x, y } == lorentz( n );
```

can be rewritten

```
in( X, semidefinite(n) );
in( { x, y }, lorentz( n ) );
```

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, and m linear inequality constraints. The dual of this problem is

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

The dual variable λ is associated with the inequality constraint $Ax \preceq b$. To represent this association 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;s
cvx_end
```

The line

```
dual variable y
```

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

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

assigns `y` to the given inequality constraint. Notice how the colon `:` operator is being used in a different manner than Matlab usually intends, which is 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. The dimensions of dual variables are not specified when they are declared; they are automatically determined from the constraints to which they are assigned. For example, if $m = 20$, `y` at the Matlab command prompt immediately before `cvx_end` yields

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

After the `cvx_end` statement is processed, and assuming the optimization was sucessful, `cvx` assigns numerical values to `x` and `y` (optimal primal and dual variable values, respectively).

If x and y are primal and dual optimal variables for this LP, they must satisfy $y_i(b - Ax)_i = 0$, the so-called *complementary slackness conditions*. 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.

4 The DCP ruleset

As mentioned previously, `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 one of the rules; hence it is important to

understand them before beginning to build models. Thankfully, the rules are drawn from basic principles of convex analysis, and are easy to learn—especially if you already construct convex programs regularly.

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. Indeed, it is not difficult to do so; consider the well-known (and useful) entropy function,

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

This expression is easily verified to be concave over the values of `x` for which it is defined. But `cvx` rejects it as written, because it violates the no-product rule described in §4.3. Problems involving entropy, however, can be solved, by explicitly using the entropy function,

```
entropy( x )
```

which is in the base `cvx` library, and thus recognized as concave by `cvx`. (The first version of `cvx`, however, does not yet support the entropy function.) If a convex (or concave) function is not recognized as convex or concave by `cvx`, it can be added as a new atom; see §5.1.

At first, these restrictions may seem inconvenient, especially to those accustomed to the more permissive nature of more traditional modeling frameworks. We hope you will find that the rules are, in fact, not particularly restrictive in practice; and certainly that they are a small price to pay for the considerable benefits they allow: automatic convexity verification, automatic conversion to solvable form, and full support for nonsmooth functions.

4.1 Top-level rules

`cvx` currently allows three different types of disciplined convex programs:

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

Convexity is enforced for each constraint *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 (indeed, it is just the interval $[1, \infty)$), but its description includes a non-convex constraint $x^2 \geq 1$.

4.2 Constraints

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

- An *equality constraint* `==` where both sides are affine expressions.
- 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.

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

4.3 Expression rules

So far, the rules as stated are not particularly restrictive, in that all convex programs (disciplined or otherwise) must 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.

In disciplined convex programming, an expression is categorized by its *curvature*, which is either *constant*, *affine*, *convex*, or *concave*. Disciplined convex programming determines curvature by recursively applying the following rules:

- A valid constant is a MATLAB expression that evaluates to a finite numeric value.
- A valid affine expression is
  - a constant expression;
  - a declared variable;
  - the sum of two or more affine expressions;
  - the difference between two affine expressions; or
  - the product of an affine expression and a constant.
- A valid convex expression is

  - a constant or affine expression;
  - a valid call to a convex function in the atom library;
  - a convex scalar quadratic form (§4.7);
  - 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; or
  - the negation of a concave expression.

- A valid concave expression is
  - a constant or affine expression;
  - a valid call to a concave function in the atom library;
  - a concave scalar quadratic form (§4.7);
  - 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; or
  - the negation of a convex expression.

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

These expression rules forbid *products* between nonconstant expressions, with the exception of scalar quadratic forms (see §4.7 below). We call this the *no-product rule*. These expression rules also forbid the use of nonconstant expressions as either the argument or exponent of a power expression (`x ^ y`).

4.4 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.

In disciplined convex programming, functions are categorized by their curvature (*constant*, *affine*, *convex*, or *concave*) as well as their *monotonicity* (*nondecreasing*, *nonincreasing*, or *nonmonotonic*). Curvature determines the conditions under which they can appear as in expressions according to the expression rules given in §4.3 above. Monotonicity determines how they can be used in function compositions, as we shall see in §4.5 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, Matlab's square root function `sqrt( x )` is interpreted 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 montonicity can be considered on an argument-by-argument basis. For example,

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

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

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

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 a 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.

4.5 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 any 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. (The first is recognized as convex and nondecreasing, and the second is recognized as concave and nondecreasing.)

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 )` function instead, since `norm` is in the atom library and known by `cvx` to be convex.

4.6 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 equivalency 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 §5.1, we show how to accomplish this.

4.7 Scalar quadratic forms

In its original form described in [Gra04, GBY05], 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 )          ⟹     conj(x) .* x
       sum_square( y )         ⟹        y' * y
quad_form( A * x - b, Q )      ⟹  (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 §4.3.

The use of quadratic forms should actually be of *less* use in disciplined convex programming than in a more traditional mathematical programming framework, where a quadratic form is often a smooth subsitute for a non-smooth 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 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.

5.1 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 )
```

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 that all the calculations carried out inside the function satisfy the DCP composition rules.

5.2 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

$$f : \mathbf{R}^n \rightarrow (\mathbf{R} \cup +\infty), \quad f(x) \triangleq \inf \{ \, g(x,y) \;\mid\; (x,y) \in S \, \} \tag{5}$$

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

$$f(x) \triangleq \inf \{ \, y \;\mid\; (x,y) \in S \, \} \tag{6}$$

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

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 $h(x)$:

$$h : \mathbf{R} \rightarrow \mathbf{R}, \quad h(x) \triangleq \begin{cases} x^2 & |x| \leq 1 \\ 2|x| - 1 & |x| \geq 1 \end{cases} \tag{7}$$

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:

$$\begin{array}{ll} \text{minimize} & 2v + w^2 \\ \text{subject to} & |x| \leq v + w \\ & w \leq 1 \end{array} \tag{8}$$

Specifically, the optimal value of the this QP is equal to the Huber penlaty 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

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

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

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

gives the *hypograph* representation of $\bar{f}$; that is, $S = \operatorname{hypo} f$. 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

$$f : \mathbf{R}^{n \times n} \rightarrow \mathbf{R}, \quad f(X) = \lambda_{\min}(X + X^T) \tag{11}$$

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

$$\operatorname{hypo} f = \{ \, (X,t) \mid f(X) \geq t \, \} = \left\{ \, (X,t) \mid X + X^T - tI \succeq 0 \, \right\} \tag{12}$$

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 isa( x, 'double' ),
    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.

5.3 Defining new sets

This version of `cvx` does not support user-defined sets, since we haven't yet settled on the best syntax. This support will be added soon.

A Installing cvx

The basic system requirements for running cvx are Matlab 6.0 or later, on any platform that supports SeDuMi 1.1R2 [RP05]. A copy of SeDuMi has been included with cvx. The SeDuMi MEX files, and the lone cvx MEX file, have been precompiled for Windows, Linux, and Solaris 8, and we are able to test them on the following specific platform combinations:

- Windows: MATLAB 6.1, 6.5.2, and 7.0.4.
- Linux: MATLAB 6.5.1, 7.0.0, and 7.0.4.
- Solaris 8: MATLAB 6.5.1 and 7.0.0.

If any of these combinations matches your own, then simply follow the directions in §A.1 below to install cvx, and you should be able to proceed without difficulty. For other platforms, please see the instructions in §A.2.

If your platform combination differs only slightly from these—say, for example, you are running Matlab 7.0.0 on Windows—then the instructions in §A.1 *should* work, but we cannot be sure. We encourage you to upgrade to at least upgrade to the latest minor version of Matlab if at all possible; your license likely allows it.

If you have been running a previous version of cvx, and you are installing a new version, please do not simply unpack it on top of the old version. Instead, remove the old version entirely, or move it out of the way—say, by renaming the directory cvx to cvx.old. Then follow the directions as if you were installing cvx for the first time.

A.1 Precompiled platforms

Installing cvx is simplest on Windows and Linux systems. If you are running Matlab R13 (6.5), please note the additional instructions below:

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.
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
   ```

   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 Other platforms

According to the SeDuMi web site, SeDuMi 1.1R2 will also run on MacOS X, AIX, and Solaris. If you wish to `cvx` on these platforms, you will have to compile SeDuMi's MEX files yourself. The installation thus proceeds as follows:

1. Install `cvx` according to Steps 1-3 in §A.1.

2. At the Matlab command prompt, `cd` to the directory `sedumi` within the `cvx` installation, and execute the command

   ```
   install_sedumi
   ```

   which will compile the MEX files.

3. Now complete steps 4-6 in §A.1 above.

4. It is likely that `cvx_setup` will warn you that it was unable to find the MEX file `cvx_bcompress_mex`. To create it, `cd` to the directory `lib` within the `cvx` installation, and type the command

   ```
   mex cvx_bcompress_mex.c
   ```

   This assumes that you are properly set up to compile MEX files already; see your MATLAB documentation for more details.

If you succced in compiling the MEX files for a new platform, please consider sending them to us so that we may consider adding them to the master `cvx` distribution.

Important note: If you are using Linux, do *not* run `install_sedumi`. Doing so can result in errors that prevent SeDuMi and `cvx` from running.

A.3 About SeDuMi

As stated above, the `cvx` distribution includes a full copy of the SeDuMi package in the directory `cvx/sedumi`. If you wish to use SeDuMi independently of `cvx`, add that subdirectory to your Matlab path as well. If you already have SeDuMi installed on your system, you are free to remove `cvx/sedumi` to save space; `cvx` will detect the presence of your other copy and use it instead. If you do this, it is important that your version of SeDuMi coincides with the version used by `cvx` (1.1R2).

B `cvx` operators, functions, and sets

B.1 Basic operators and linear functions

The standard addition, subtraction, multiplication and division operators are overloaded to work with `cvx`, when appropriate. For example, two expressions known to be concave can be added together, and an expression known to be convex can be divided by a negative scalar (and the result is concave). As another example, a vector affine expression can be multiplied by a constant matrix (of appropriate dimensions), or added to another affine expression. General arrays of expressions can be added or subtracted, provided the rules of disciplined convex programming are respected. For example, you can subtract one array of expressions from another provided the corresponding entries have opposite curvature, so each element of the result is either the difference of a convex and concave function, or the difference of a concave and a convex function.

The usual constructors for arrays work, as in `[A B; C D]`, or `X([3,4],:)`. Matlab functions and operators that are linear have been overloaded to handle `cvx` variables and expressions: `cumsum`, `diag`, `dot` (if one argument is constant), `fliplr`, `flipud`, `flipdim`, `horzcat`, `ipermute`, `kron`, `permute`, `repmat`, `reshape`, `rot90`, `sum`, `tril`, `triu`, `vertcat`, `hankel`, `toeplitz`, `trace`. When these functions involve summation, the rules of disciplined convex programming must be followed. For example, `trace(Z)` will only work if all entries of the diagonal of the (matrix) expression `Z` are known to be convex, or known to be concave.

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`, powers such as `x^4` or `x^(1/2)`, are *not* supported.

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).

- `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.
- `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( x, 'largest', 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

- `lorentz(n)`: $\mathbf{Q}^n = \{\, (x, y) \in \mathbf{R}^n \times \mathbf{R} \mid \|x\|_2 \leq y \,\}$
- `rotated_lorentz(n)`: $\{\, (x, y, z) \in \mathbf{R}^n \times \mathbf{R} \times \mathbf{R} \mid \|x\|_2 \leq yz,\ y, z \geq 0 \,\}$
- `complex_lorentz(n)`: $\{\, (x, y) \in \mathbf{C}^n \times \mathbf{R} \mid \|x\|_2 \leq y \,\}$
- `complex_rotated_lorentz(n)`: $\{\, (x, y, z) \in \mathbf{C}^n \times \mathbf{R} \times \mathbf{R} \mid \|x\|_2 \leq yz,\ y, z \geq 0 \,\}$
- `semidefinite(n)`: $\mathbf{S}^n_+ = \left\{\, X \in \mathbf{R}^{n \times n} \mid X = X^T,\ X \succeq 0 \,\right\}$
- `hermitian_semidefinite(n)`: $\left\{\, X \in \mathbf{C}^{n \times n} \mid X = X^H,\ X \succeq 0 \,\right\}$

C `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.
- `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.

- `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.*

D Caveats

When `cvx_end` is processed, the problem is solved, and primal and dual variables, which were `cvx` objects inside the `cvx` specification, become ordinary numerical arrays, containing optimal values. Numerical (*i.e.*, constant) data defined inside the `cvx` specification persists unchanged after `cvx_end` is processed. For example, if we have the assignment `a=2` inside a `cvx` specification, then `a` will persist after the `cvx_end` command, and will still contain the value `2`.

Our caveat has to do with assignments inside a `cvx` specification in which the righthand side is not a constant, *i.e.*, it is a function of the variables. For example, we might have the assignment `y=2*x-1` inside a `cvx` specification, where `x` is a variable. There's nothing wrong with this assignment; it just says that `y` will refer to the function or expression `2*x-1`. Note that this is very different from the equality constraint operator. If we type `y==2*x-1` we get an error, unless `y` has been previously defined as a variable or expression.

When `cvx_end` is processed, we'd like to have all these nonconstant expressions just go away, or perhaps convert to the numerical values you'd obtain if you evaluated them at the optimal values of the variables. Unfortunately in this implementation, however, these nonconstant expressions persist, but have no meaning. You really have no right referring to them after the `cvx_end` command, but if you do, you'll get an error telling you it no longer has any meaning.

References

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

[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 from `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.

[GBY05] M. Grant, S. Boyd, and Y. Ye. Disciplined convex programming. In L. Liberti and N. Maculan, editors, *Globval Optimization: from Theory to Implementation*, Nonconvex Optimization and Its Applications, pages 155–210. Kluwer, 2005. See `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.

[RP05] O. Romanko and I. Pólik. SeDuMI 1.1R2 (software package). `http://sedumi.mcmaster.ca`, July 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. See `http://www.stanford.edu/~boyd/sdpsol.html`.