The CVX Users' Guide

Release 2.0 (beta)

Michael C. Grant, Stephen P. Boyd
CVX Research, Inc.

March 08, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is CVX?

CVX is a modeling system for constructing and solving *disciplined convex programs* (DCPs). CVX supports a number of standard problem types, including linear and quadratic programs (LPs/QPs), second-order cone programs (SOCPs), and semidefinite programs (SDPs). CVX can also solve much more complex convex optimization problems, including many involving nondifferentiable functions, such as ℓ_1 norms. You can use CVX to conveniently formulate and solve constrained norm minimization, entropy maximization, determinant maximization, and many other convex programs. As of version 2.0, CVX also solves *mixed integer disciplined convex programs* (MIDCPs) as well, with an appropriate integer-capable solver.

To use CVX effectively, you need to know at least a bit about convex optimization. For background on convex optimization, see the book Convex Optimization [BV04] or the Stanford course EE364A.

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

CVX provides special modes to simplify the construction of problems from two specific problem classes. In *semidefinite programming (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 *geometric programming (GP) mode*, CVX accepts all of the special functions and combination rules of geometric programming, including monomials, posynomials, and generalized posynomials, and transforms such problems into convex form so that they can be solved efficiently. For background on geometric programming, see this tutorial paper [BKVH05].

Previous versions of CVX supported two free SQLP solvers, SeDuMi [Stu99] and SDPT3 [TTT03]. These solvers are included with the CVX distribution. This version includes support for two more solvers, Gurobi and MOSEK. For more information, see *Solvers*. These are commercial solvers, and must be acquired separately.

The ability to use CVX with commercial solvers is a new capability that we have decided to include under a new CVX Professional license model. Academic users will be able to utilize these features at no charge, but commercial users will require a paid CVX Professional license. For more details, see *Licensing*.

1.1.1 What's new?

- Support for commercial solvers; see *Solvers*.
- Binary and integer variable support; see *Mixed integer problems*.
- Commercial licensing for advanced features; see *Licensing*.
- New support options; see *Support*.
- The default solver choice and other preferences can be saved across MATLAB using the `cvx_save_prefs` command.

1.2 What is disciplined convex programming?

Disciplined convex programming is a methodology for constructing convex optimization problems proposed by Michael Grant, Stephen Boyd, and Yinyu Ye [GBY06], [Gra04]. It is meant to support the formulation and construction of optimization problems that the user intends *from the outset* to be convex.

Disciplined convex programming imposes a set of conventions or rules, which we call *the DCP ruleset*. Problems which adhere to the ruleset can be rapidly and automatically verified as convex and converted to solvable form. Problems that violate the ruleset are rejected—even when the problem is convex. That is not to say that such problems cannot be solved using DCP; they just need to be rewritten in a way that conforms to the DCP ruleset.

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

1.2.1 Mixed integer problems

With version 2.0, CVX now supports *mixed integer* disciplined convex programs (MIDCPs). A MIDCP is a model that obeys the same convexity rules as standard DCPs, except that one or more of its variables is constrained to take on integral values. In other words, if the integer constraints are removed, the result is a standard DCP.

Unlike a true DCP, a mixed integer problem is *not* convex. Finding the global optimum requires the combination of a traditional convex optimization algorithm with an exhaustive search such as a branch-and-bound algorithm. Some CVX solvers do not include this second piece and therefore do not support MIDCPs; see *Solvers* for more information. What is more, even the best solvers cannot guarantee that every moderately-sized MIDCP can be solved in a reasonable amount of time.

Mixed integer disciplined convex programming represents new territory for the CVX modeling framework—and for the supporting solvers as well. While solvers for mixed integer linear and quadratic programs (MILP/MIQP) are reasonably mature, support for more general convex nonlinearities is a relatively new development. We anticipate that MIDCP support will improve over time.

1.3 What CVX is *not*

CVX is *not* meant to be a tool for checking if your problem is convex. You need to know a bit about convex optimization to effectively use CVX; otherwise you are the proverbial monkey at the typewriter, hoping to (accidentally) type in a valid disciplined convex program. If you are not certain that your problem is convex *before* you enter it into CVX, you are using the tool improperly, and your efforts will likely fail.

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

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

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

1.4 Licensing

CVX is free for use in both academic and commercial settings when paired with a free solver—including the versions of SeDuMi and SDPT3 that are included with the package.

With version 2.0, we have added the ability to connect CVX to *commercial* solvers as well. This new functionality is released under a *CVX Professional* product tier which we intend to license to commercial users for a fee, and offer to academic users at no charge. The licensing structure is as follows:

- *All users* are free to use the standard features of CVX *at no charge*. This includes the ability to construct and solve any of the models supported by the free solvers SeDuMi and SDPT3.
- *Commercial users* who wish to solve CVX models using Gurobi or MOSEK will need to purchase a CVX Professional license. Please send an email to CVX Research for inquiries. for an availability schedule and pricing details.
- *Academic users* may utilize the CVX Professional capability *at no charge*. To obtain an academic license, please visit the Academic licenses page on the CVX Research web site.

The bulk of CVX remains open source under a slightly modified version of the GPL Version 2 license. A small number of files that support the CVX Professional functionality remain closed source. If those files are removed, the modified package remains *fully functional* using the free solvers, SeDuMi and SDPT3. Users may freely modify, augment, and redistribute this free version of CVX, as long as all modifications are themselves released under the same license. This includes adding support for new solvers released under a free software license such as the GPL. For more details, please see the full *Licensing* section.

CHAPTER

TWO

INSTALLATION

2.1 Supported platforms

CVX is supported on 32-bit and 64-bit versions of Linux, Mac OSX, and Windows, running MATLAB versions 7.5 (R2007b) and later. There are some important platform-specific cautions, however:

- Gurobi support requires Matlab 7.7 (R2008b) or later.
- 32-bit Linux: the Gurobi solver is not available for this platform, as Gurobi is phasing out support for 32-bit Linux altogether.
- Older versions of Mac OS X (e.g. 10.5) ship with Java 1.5. The standard version of CVX works properly on this platform, but CVX Professional support requires Java 1.6. To restore this support, upgrade your operating system or Java installation.

As of version 2.0, support for versions 7.4 (R2007a) or older has been discontinued. If you need to use CVX with these older versions of Matlab, please use CVX 1.22 or earlier, which will remain available indefinitely on the CVX Research web site. However, this version is no longer supported, and will not receive bug fixes or improvements. We strongly encourage you to update your Matlab installation to the latest version possible.

2.2 Installation instructions

Note: If you wish to use CVX with Gurobi or MOSEK, they must be installed and accessible from MATLAB *before* running `cvx_setup`. See *below* for more details.

---

1. Retrieve the latest version of CVX from the web site. 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. There are two important exceptions:
   - *Do not* place CVX in Matlab's own `toolbox` directory.

- *Do not* unpack a new version of CVX on top of an old one. We recommend moving the old version out of the way, but do not delete it until you are sure the new version is working as you expect.

3. Start Matlab.
4. Change directories to the top of the CVX distribution, and run the `cvx_setup` command. For example, if you installed CVX into `C:\Matlab\personal\cvx` on Windows, type these commands:

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

   at the Matlab command prompt. If you installed CVX into `~/MATLAB/cvx` on Linux or a Mac, type these commands:

   ```
   cd ~/MATLAB/cvx
   cvx_setup
   ```

   The `cvx_setup` function performs a variety of tasks to verify that your installation is correct, sets your Matlab search path so it can find all of the CVX program files, and runs a simple test problem to verify the installation.
5. In some cases—usually on Linux—the `cvx_setup` command may instruct you to create or modify a `startup.m` file that allows you to use CVX without having to type `cvx_setup` every time you re-start Matlab.

2.3 Installing a CVX Professional license

If you acquire a license key for CVX Professional, the only change required to the above steps is to include the name of the license file as an input to the `cvx_setup` command. For example, if you saved your license file to `~/licenses/cvx_license.mat` on a Mac, this would be the modified command:

```
cd ~/MATLAB/cvx
cvx_setup ~/licenses/cvx_license.mat
```

If you have previously run `cvx_setup` without a license, or you need to replace your current license with a new one, simply run `cvx_setup` again with the filename. Once the license has been accepted and installed, you are free to move your license file anywhere you wish for safekeeping—CVX saves a copy in its preferences.

In order to use Gurobi or MOSEK with CVX, you must first install them according to their respective developer's instructions. In particular, make sure that the MATLAB interface is fully operational in each case. For instance, the commands `mosekopt` and/or `gurobi` should run successfully from the MATLAB command line.

2.4 Using CVX with Gurobi or MOSEK

When `cvx_setup` is run, it configures CVX according to the solvers that it locates at that time. Therefore, if you first install CVX Standard, and later install Gurobi or MOSEK, you must *re-run* `cvx_setup` in

order for CVX to detect the new solver.

When installing these solvers, please make sure that their MATLAB interfaces are operating properly before re-running `cvx_setup`. Please consult the installation instructions provided by the solver developer for details. If MATLAB cannot find the `gurobi` or `mosekopt` commands when `cvx_setup` is run, it will not configure CVX to use those solvers.

2.5 About the included solvers

The CVX distribution includes copies of the solvers SeDuMi and SDPT3 in the directories `cvx/sedumi` and `cvx/sdpt3`, respectively. We have designed CVX to use its own copy of these solvers, because we can better support the specific version that we have chosen. Indeed, CVX has generated quite a few bug reports for these solvers! However, you are free to keep an alternate copy in your MATLAB path. When you are not constructing a CVX model, MATLAB will rely on your copy of the solver instead.

CHAPTER

THREE

A QUICK START

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

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

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

This processing flow will become clearer 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.)

3.1 Least squares

We first consider the most basic convex optimization problem, least-squares (also known as linear regression). 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 the data for a small test problem in Matlab:

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

Then the least-squares solution $x = (A^T A)^{-1} A^T b$ is easily computed using the backslash operator:

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

Using CVX, the same problem can be solved as follows:

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

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

- `cvx_begin` creates a placeholder for the new CVX specification, and prepares Matlab to accept variable declarations, constraints, an objective function, and so forth.
- `variable x(n)` declares `x` to be an optimization variable of dimension n. CVX requires that all problem variables be declared before they are used in the objective function or constraints.
- `minimize( norm(A*x-b) )` specifies the objective function to be minimized.
- `cvx_end` signals the end of the CVX specification, and causes the problem to be solved.

Clearly 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.

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, several additional Matlab variables are created; for instance,

- `cvx_optval` contains the value of the objective function;
- `cvx_status` contains a string describing the status of the calculation (see *Interpreting the results*).

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

There is not much room for error in specifying a simple least-squares problem, but if you make one, you will get an error or warning message. For example, if you replace the objective function 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):

[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.

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

3.2 Bound-constrained least squares

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

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

where l and u are given data vectors with the same dimension as 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 with a standard QP solver. [2]

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

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

If you have the Matlab Optimization Toolbox, you can use `quadprog` to solve the problem as follows:

```
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

```
cvx_begin
    variable x(n)
    minimize( norm(A*x-b) )
    subject to
        l <= x <= u
cvx_end
```

Two new lines of CVX code have been added to the CVX specification:

- The `subject to` statement does nothing—CVX provides this statement simply to make specifications more readable. As with indentation, it is optional.
- The line `l <= x <= u` represents the $2n$ inequality constraints.

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`

[2] There are also a number of solvers specifically designed to solve bound-constrained least-squares problems, such as BCLS by Michael Friedlander.

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.

3.3 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.*, Section 6.1 of Convex Optimization. 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,:) - xzz(n+1:end,:);
```

The CVX version is, not surprisingly,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4 Other constraints

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

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

Expressions using comparison operators (==, >=, *etc.*) behave quite differently when they involve CVX optimization variables, or expressions constructed from CVX optimization variables, than when they involve simple numeric values. For example, because `x` is a declared variable, the expression `C*x==d` causes a constraint to be included in the CVX specification, and returns no value at all. On the other hand, outside of a CVX specification, if `x` has an appropriate numeric value—for example immediately after the `cvx_end` command—that same expression would return a vector of `1`s and `0`s, corresponding to the truth or falsity of each equality. [3] 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 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.
```

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

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 *The DCP ruleset*. These rules are relatively intuitive if you know the basics of convex analysis and convex optimization.

3.5 An optimal trade-off curve

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

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

The `minimize` statement above illustrates one of the construction rules to be discussed in *The DCP ruleset*. 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)
```

is recognized as convex by CVX, as long as `gamma(k)` is positive or zero. If `gamma(k)` were negative,

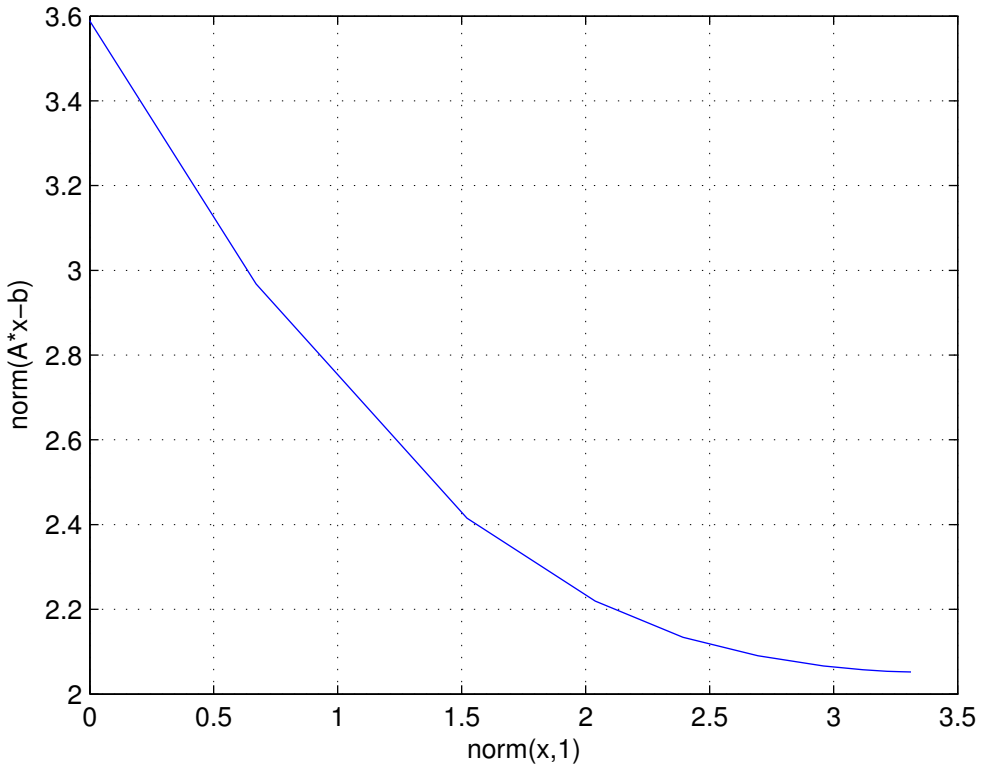

Figure 3.1: An example trade-off curve from the `quickstart.m` demo.

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.
```

CHAPTER

FOUR

THE BASICS

4.1 `cvx_begin` and `cvx_end`

All CVX models must be preceded by the command `cvx_begin` and terminated with the command `cvx_end`. All variable declarations, objective functions, and constraints should fall in between. The cvx_begin command may include one more more modifiers:

`cvx_begin quiet` Prevents the model from producing any screen output while it is being solved.

`cvx_begin sdp` Invokes *semidefinite programming mode*.

`cvx_begin gp` Invokes *geometric programming mode*.

These modifiers may be combined when appropriate; for instance, `cvx_begin sdp quiet` invokes SDP mode and silences the solver output.

4.2 Variables

All variables must be declared using the `variable` command (or `variables` command; see below) before they can be used in constraints or an objective function. A `variable` command includes the name of the variable, an optional dimension list, and one or more keywords that provide additional information about the content or structure of the variable.

Variables can be real or complex scalars, vectors, matrices, or n-dimensional arrays. For instance,

```
variable X
variable Y(20,10)
variable Z(5,5,5)
```

declares a total of 326 (scalar) variables: a scalar `X`, a 20x10 matrix `Y` (containing 200 scalar variables), and a 5x5x5 array `Z` (containing 125 scalar variables).

Variable declarations can also include one or more *keywords* to denote various structures or conditions on the variable. For instance, to declare a complex variable, use the `complex` keyword:

```
variable w(50) complex
```

For MIDCPs, the `integer` and `binary` keywords are used to declare integer and binary variables, respectively:

```
variable p(10) integer
variable q binary
```

A variety of keywords are available to help construct variables with *matrix structure* such as symmetry or bandedness. For example, the code segment

```
variable Y(50,50) symmetric
variable Z(100,100) hermitian toeplitz
```

declares Y to be a real 50×50 symmetric matrix variable, and Z a 100×100 Hermitian Toeplitz matrix variable. (*Note*: the `hermitian` keyword implies that the matrix is also complex.) 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)`	`diagonal`	`hankel`	`hermitian`
`skew_symmetric`	`symmetric`	`toeplitz`	`tridiagonal`
`lower_bidiagonal`	`lower_hessenberg`	`lower_triangular`	
`upper_bidiagonal`	`upper_hankel`	`upper_hessenberg`	`upper_triangular`

The structure keywords are self-explanatory with a couple of exceptions:

`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.

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

When multiple keywords are supplied, the resulting matrix structure is determined by intersection. For example `symmetric lower_triangular` produces a diagonal matrix, because that is the only matrix type that is both symmetric and lower triangular. If the keywords truly conflict, so that there is emph{no} non-zero matrix that satisfies all keywords, an error will result.

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

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

The one limitation of the `variables` command is that it cannot declare complex, integer, or structured variables. These must be declared one at a time, using the singular `variable` command.

4.3 Objective functions

Declaring an objective function requires the use of the `minimize` or `maximize` function, as appropriate. (For the benefit of our users whose English favors it, the synonyms `minimise` and `maximise` are provided as well.) The objective function in a call to minimize must be convex; the objective function in a call to maximize must be concave; for instance:

```
minimize( norm( x, 1 ) )
maximize( geo_mean( x ) )
```

At most one objective function may be declared in a CVX specification, and it must have a scalar value.

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.

4.4 Constraints

The following constraint types are supported in CVX:

- Equality == constraints, where both the left- and right-hand sides are affine expressions.
- 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.

The non-equality operator ~= may *never* be used in a constraint; in any case, such constraints are rarely convex. The latest version of CVX now allows you to chain inequalities together; *e.g.*, `l <= x <= u`. (Previous versions did not allow chained inequalities.)

Note the important distinction between the single equals =, which is an assignment, and the double equals ==, which denotes equality; for more on this distinction, see *Assignment and expression holders* below.

Strict inequalities < and > are accepted as well, but they are interpreted identically to their nonstrict counterparts. We strongly discourage their use, and a future version of CVX may remove them altogether. For the reasoning behind this, please see the fuller discussion in *Strict inequalities*.

Inequality and equality constraints are applied in an elementwise fashion, matching the behavior of MATLAB itself. For instance, if `A` and `B` are $m \times n$ arrays, then `A<=B` is interpreted as mn (scalar) inequalities `A(i,j)<=B(i,j)`. When one side or the other is a scalar, that value is replicated; for instance, `A>0` is interpreted as `A(i,j)>=0`.

The elementwise treatment of inequalities is altered in *semidefinite programming mode*; see that section for more details.

CVX also supports a *set membership* constraint; see *Set membership* below.

4.5 Functions

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

An example of a function in the base library is the quadratic-over-linear function `quad_over_lin`:

$$f : \mathbf{R}^n \times \mathbf{R} \to \mathbf{R}, \quad f(x,y) = \begin{cases} x^T x/y & y > 0 \\ +\infty & y \leq 0 \end{cases}$$

(The function also accepts complex x, but we'll consider real x to keep things simple.) The quadratic-over-linear function is convex in x and y, and so can be used as an objective, in an appropriate constraint, or in a more complicated expression. We can, for example, minimize the quadratic-over-linear function of $(Ax - b, c^T x + d)$ using

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

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

You can also use the function `quad_over_lin` *outside* a CVX specification. In this case, it just computes its (numerical) value, given (numerical) arguments. If `c'*x+d` is positive, then the result is numerically equivalent tp

```
( ( A * x - b )' * ( A * x - b ) ) / ( c' * x + d )
```

However, the `quad_over_lin` function also performs a domain check, so it returns `Inf` if `c'*x+d` is zero or negative.

4.6 Set membership

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 *Sets*.

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

```
X == semidefinite(n)
```

The literal meaning of this is that `X` is constrained to be equal to some unnamed variable, which is required to be an $n \times n$ symmetric positive semidefinite matrix. This is, of course, equivalent to saying that X must itself be symmetric positive semidefinite.

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

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

The second line here imposes the constraint that X be positive semidefinite. (You can read "==" here as "is" or "is in", 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.

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

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

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

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

```
A*X*A'-X <In> 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,

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

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 the section Least squares (in a fairly complicated way) as follows:

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

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

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

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

4.7 Dual variables

When a disciplined convex program is solved, the associated *dual problem* is also solved. (In this context, the original problem is called the *primal problem*.) The optimal dual variables, each of which is associated with a constraint in the original problem, give valuable information about the original problem, such as the

sensitivities with respect to perturbing the constraints (*c.f.* Convex Optimization, chapter 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. To associate the dual variable y with the inequality constraint $Ax \preceq b$ in this LP, we use the following syntax:

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

The line

```
dual variable y
```

tells CVX that y will represent the dual variable, and the line

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

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

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

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

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

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

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

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

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

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

You can check this in Matlab with the line

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

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

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

$$c^T x = -1, \quad Ax \preceq 0,$$

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

$$b^T y = -1, \quad A^T y = 0, \quad y \succeq 0$$

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

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

4.8 Assignment and expression holders

Anyone with experience with C or Matlab understands the difference between the single-equal *assignment* operator = and the double-equal *equality* operator ==. This distinction is vitally important in CVX as well, and CVX takes steps to ensure that assignments are not used improperly. For instance, consider the following code snippet:

```
variable X(n,n) symmetric;
X = semidefinite(n);
```

At first glance, the statement `X = semidefinite(n);` may look like it constrains `X` to be positive semidefinite. But since the assignment operator is used, `X` is actually *overwritten* by the anonymous semidefinite variable instead. Fortunately, CVX forbids declared variables from being overwritten in this way; when `cvx_end` is reached, this model would issue the following error:

```
??? 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.

Despite this warning, assignments can be genuinely useful, so we encourage their use with appropriate care. For instance, consider the following excerpt:

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

The construction `z = 2 * x - y` is *not* an equality constraint; it is an assignment. It is storing an intermediate calculation `2 * x - y`, which is an affine expression, which is then used later in two different constraints. We call `z` an *expression holder* to differentiate it from a formally declared CVX variable.

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

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

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

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

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

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

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

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

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

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

CHAPTER

FIVE

THE DCP RULESET

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

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

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

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

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

which is in the base CVX library, and thus recognized as concave by CVX. If a convex (or concave) function is not recognized as convex or concave by CVX, it can be added as a new atom; see *Adding new functions to the atom library*.

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

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

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

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

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

5.1 A taxonomy of curvature

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

the categories have the following meanings:

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

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

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

5.2 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 and no objective.

5.3 Constraints

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

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

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

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

As discussed in *Set membership*, 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.

5.3.1 Strict inequalities

As mentioned in *Constraints*, strict inequalities <, > are interpreted in an identical fashion to nonstrict inequalities >=, <=. It is important to note that CVX cannot guarantee that an inequality will be strictly satisfied at the solution it computes. This is not simply a choice we have made in CVX; it is a natural consequence of both the underlying mathematics and the design of convex optimization solvers. For that reason, we *strongly* discourage the use of strict inequalities in CVX, and a future version may remove them altogether.

When a strict inequality is essential to your model, you may need to take additional steps to ensure compliance. In some cases, this can be accomplished through *normalization*. For instance, consider a set of homogeneous equations and inequalities:

$$Ax = 0, \quad Cx \preceq 0, \quad x \succ 0$$

Except for the strict inequality, $x = 0$ would be an acceptable solution; indeed the need to avoid the origin is the very reason for the strict inequality. However, note that if a given x satisfies these constraints, then so does αx for all $\alpha > 0$. By eliminating this degree of freedom with normalization, we can eliminate the strict inequality; for instance:

$$Ax = 0, \quad Cx \preceq 0, \quad x \succ 0, \quad \mathbf{1}^T x = 1$$

If normalization is not a valid approach for your model, you may simply need to convert the strict inequality into a non-strict one by adding a small offset; *e.g.*, convert `x > 0` to, say, `x >= 1e-4`. Note that the bound needs to be large enough so that the underlying solver considers it numerically significant.

Finally, note that for some functions like `log(x)` and `inv_pos(x)`, which have domains defined by strict inequalities, the domain restriction is handled *by the function itself*. You do not need to add an explicit constraint `x > 0` to your model to guarantee that the solution is positive.

5.4 Expression rules

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

Disciplined convex programming determines the curvature of scalar expressions by recursively applying the following rules. While this list may seem long, it is for the most part an enumeration of basic rules of convex analysis for combining convex, concave, and affine forms: sums, multiplication by scalars, and so forth.

- A valid constant expression is
  - any well-formed Matlab expression that evaluates to a finite value.
- A valid affine expression is
  - a valid constant expression;
  - a declared variable;
  - a valid call to a function in the atom library with an affine result;

  - the sum or difference of affine expressions;
  - the product of an affine expression and a constant.
- A valid convex expression is
  - a valid constant or affine expression;
  - a valid call to a function in the atom library with a convex result;
  - an affine scalar raised to a constant power $p \geq 1$, $p \neq 3, 5, 7, 9, ...$;
  - a convex scalar quadratic form—see *Scalar quadratic forms*;
  - the sum of two or more convex expressions;
  - the difference between a convex expression and a concave expression;
  - the product of a convex expression and a nonnegative constant;
  - the product of a concave expression and a nonpositive constant;
  - the negation of a concave expression.
- A valid concave expression is
  - a valid constant or affine expression;
  - a valid call to a function in the atom library with a concave result;
  - a concave scalar raised to a power $p \in (0, 1)$;
  - a concave scalar quadratic form—see *Scalar quadratic forms*;
  - the sum of two or more concave expressions;
  - the difference between a concave expression and a convex expression;
  - the product of a concave expression and a nonnegative constant;
  - the product of a convex expression and a nonpositive constant;
  - the negation of a convex expression.

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

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

5.5 Functions

In CVX, functions are categorized in two attributes: *curvature* (*constant*, *affine*, *convex*, or *concave*) and *monotonicity* (*nondecreasing*, *nonincreasing*, or *nonmonotonic*). Curvature determines the conditions under

which they can appear in expressions according to the expression rules given above. Monotonicity determines how they can be used in function compositions, as we shall see in the next section.

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

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

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

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

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

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

For functions with multiple arguments, curvature is always considered *jointly*, but monotonicity can be considered on an *argument-by-argument* basis. For example, the function `quad_over_lin(x,y)`

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

is jointly convex in both x and y, but it is monotonic (nonincreasing) only in y.

Some functions are convex, concave, or affine only for a *subset* of its arguments. For example, the function `norm(x,p)` where `p \geq 1` 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.*,

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

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

5.6 Compositions

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

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

For example, consider the function `square(x)`, which is provided in the CVX atom library. This function squares its argument; *i.e.*, it computes `x.*x`. (For array arguments, it squares each element independently.) It is in the CVX atom library, and known to be convex, provided its argument is real. So if `x` is a real variable of dimension n, `a` is a constant n-vector, and `b` is a constant, the expression

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

is accepted by CVX, which knows that it is convex.

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

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

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

The corresponding rules for a concave function are as follows:

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

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

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

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

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

obeys the first of the six composition rules and is therefore accepted by CVX, and classified as convex. As another example, consider the sum function, which is both convex and concave (since it is affine), and nondecreasing in each argument. Therefore the expressions

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

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

Most people who know basic convex analysis like to think of these examples in terms of the more specific rules: a maximum of convex functions is convex, and a sum of convex (concave) functions is convex (concave). But these rules are just special cases of the general composition rules above. Some other well known basic rules that follow from the general composition rules are:

- a nonnegative multiple of a convex (concave) function is convex (concave);
- a nonpositive multiple of a convex (concave) function is concave (convex).

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

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

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

The composition rules are sufficient but not necessary for the classification to be correct, so some expressions which are in fact convex or concave will fail to satisfy them, and so will be rejected by CVX. For example, if `x` is a vector variable, the expression

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

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

5.7 Monotonicity in nonlinear compositions

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

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

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

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

Another approach is to use the alternate outer function `square_pos`, included in the CVX library, which represents the function $(x_+)^2$, where $x_+ = \max\{0, x\}$. Obviously, `square` and `square_pos` coincide when their arguments are nonnegative. But `square_pos` is nondecreasing, so it can accept a convex argument. Thus, the expression

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

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

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

5.8 Scalar quadratic forms

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

`x .* x`	`square( x )` (real `x`)
`conj( x ) .* x`	`square_abs( x )`
`y' * y`	`sum_square_abs( y )`
`(A*x-b)'*Q*(Ax-b)`	`quad_form( A*x - b, Q )`

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

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

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

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

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

will be accepted.

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

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

will be recognized as convex. Once a quadratic form has been verified by CVX, it can be freely used in any way that a normal convex or concave expression can be, as described in *Expression rules*.

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 is more naturally represented using a second-order cone constraint—so the second form may actually be more efficient. In fact, in our experience, the non-squared form will often be handled more accurately. So we strongly encourage you to re-evaluate the use of quadratic forms in your models, in light of the new capabilities afforded by disciplined convex programming.

CHAPTER

SIX

SEMIDEFINITE PROGRAMMING MODE

Those who are familiar with *semidefinite programming* (SDP) know that the constraints that utilize the set `semidefinite(n)` in the discussion on *Set membership* 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* that allows this LMI notation to be employed inside CVX models using Matlab's standard inequality operators `>=`, `<=`. In order to use it, one simply begins 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`	becomes	`X - Y == semidefinite(n)`
`X <= Y`	becomes	`Y - X == semidefinite(n)`

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

`X >= Y`	becomes	`X - Y == hermitian_semidefinite(n)`
`X <= Y`	becomes	`Y - X == hermitian_semidefinite(n)`

- 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, 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 to within a reasonable numeric tolerance, a warning will be issued. We have provided a function `sym(X)` that extracts the symmetric part of a square matrix; that is, `sym(X) = 0.5*(X+X')`.
- 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 *Indexed dual variables*.

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. Using 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`.

CHAPTER

SEVEN

GEOMETRIC PROGRAMMING MODE

Geometric programs (GPs) are special mathematical programs that can be converted to convex form using a change of variables. The convex form of GPs can be expressed as DCPs, but CVX also provides a special mode that allows a GP to be specified in its native form. CVX will automatically perform the necessary conversion, compute a numerical solution, and translate the results back to the original problem.

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

```
cvx_begin gp
    variables w h d
    maximize( w * h * d )
    subject to
        2*(h*w+h*d) <= Awall;
        w*d <= Afloor;
        alpha <= h/w >= beta;
        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. More information about generalized geometric programs is provided in this tutorial.

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

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

7.1 Top-level rules

CVX supports three types of geometric programs:

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

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

7.2 Constraints

Three types of constraints may be specified in geometric programs:

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

As with DCPs, non-equality constraints are not permitted; and while strict inequalities <, > are supported, they are treated as non-strict inequalities and should therefore be avoided.

7.3 Expressions

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

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

A valid posynomial expression is

- a valid monomial;
- the sum of two or more posynomials;
- the product of two or more posynomials;
- the ratio of a posynomial and a monomial;
- a posynomial raised to a positive integral power; or

- a call to one of the following functions with posynomial arguments: `sum`, `cumsum`, `mean`, `prod`, `cumprod`.

A valid generalized posynomial expression is

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

It is entirely possible to create and manipulate arrays of monomials, posynomials, and/or generalized posynomials in CVX, in which case these rules extend in an obvious manner. For example, the product of two monomial matrices produces a matrix whose entries are polynomials (or monomials in special cases).

CHAPTER

EIGHT

SOLVERS

8.1 Supported solvers

This version of CVX supports four solvers, each with different capabilities. Support for a fifth solver is currently under development:

Solver name	LP	QP	SOCP	SDP	GP	Integer	License?
SeDuMi	Y	Y	Y	Y	E	N	N
SDPT3	Y	Y	Y	Y	E	N	N
Gurobi	Y	Y	Y	N	N	Y	Y
MOSEK	Y	Y	Y	Y*	E	Y	Y
GLPK (in development)	Y	N	N	Y	N	Y	N

(key: Y = Yes, N = No, E = Experimental, * = Mosek 7 or later is required.)

Each solver has different capabilities and different levels of performance. For instance, SeDuMi [Stu99] and SDPT3 [TTT03] support all of the continuous (non-integer) models that CVX itself supports, while Gurobi, MOSEK, and GLPK are more limited. On the other hand, these latter solvers all support integer constraints, while SeDuMi and SDPT3 do not.

SeDuMi and SDPT3 are included with the CVX distribution, so you do not need to download an additional solver to start using CVX. We have reached agreements with both Gurobi Optimization and MOSEK ApS to bundle their solvers with CVX. We will make an announcement and update this documentation once the bundling implementation has been completed.

If you are having difficulty with one solver, please try another. No one solver performs better than the others on *every* model CVX can generate—including commercial solvers. That said, if you encounter a problem that one solver can handle well and another cannot, please send us a bug report (see *Support*) and we will forward the results to the solver's authors.

8.2 Selecting a solver

The default solver is currently SDPT3. We have found that SeDuMi is faster for most problems, but unfortunately not as reliable. None of the solvers are perfect, however, and you may find for your application that another solver is preferred.

To see which solver is currently selected, simply type

```
cvx_solver
```

To change the current solver, simply follow the `cvx_solver` with the name of your chosen solver. For example, to select SeDuMi, type

```
cvx_solver sedumi
```

The `cvx_solver` command is case insensitive, so `cvx_solver SeDuMi` will work just fine as well.

If you issue this command inside a model—that is, between `cvx_begin` and `cvx_end` it will change the solver *only* for that model; the next model will use the previous choice. If, only the other hand, you issue a `cvx_solver` command *outside* of a model, it will change the solver used for the remainder of your Matlab session (or until you change it again).

If you would like to change the default solver *permanently*—that is, so that it remains the default even if you quit and re-start Matlab—then make sure it is set properly, and then issue the command

```
cvx_save_prefs
```

This command saves not only your solver choice, but also your settings for `cvx_expert`, `cvx_power_warning`, and `cvx_precision` as well.

8.3 Controlling screen output

Once you gain confidence in using CVX and start incorporating it into your larger algorithms and programs, you are likely going to want to silence the messages it delivers to the screen. To do so, simply add the `quiet` keyword to the `cvx_begin` command; that is,

```
cvx_begin quiet
    ...
cvx_end
```

Previous versions of CVX utilized a separate `cvx_quiet` command and that command is still available in this version as well, if you prefer it. Entering `cvx_quiet true` suppresses screen output from the solver, while entering `cvx_quiet false` restores the screen output. If you enter these commands within a model—that is, between `cvx_begin` and `cvx_end`—it will affect only that model. If you enter it *outside* of a model, it will affect all subsequent models. Entering cvx_quiet with no arguments returns the current setting.

8.4 Interpreting the results

After a complete CVX specification has been entered and the cvx_end command issued, the solver is called to generate a numerical result. It proceeds to replace the variables in your model with the computed numerical values, and creates the variable cvx_optval containing the value of the objective function. It also summarizes the result of its efforts in the form of a string named `cvx_status`. The possible values of `cvx_status` are as follows:

Solved A complementary (primal and dual) solution has been found. The primal and dual variables are replaced with their computed values, and the the optimal value of the problem is placed in cvx_optval (which, by convention, is 0 for feasibility problems).

Unbounded The solver has determined that the problem is unbounded. The value of `cvx_optval` is set to `-Inf` for minimizations, and `+Inf` for maximizations. (Feasibility problems, by construction, never produce an `Unbounded` status.) The values of any dual variables are replaced with `NaN`, as the dual problem is in fact infeasible.

For unbounded problems, CVX stores an *unbounded direction* into the problem variables. This is is a *direction* along which the feasible set is unbounded, and the optimal value approaches $\pm\infty$. It is important to understand that this value is very likely *not* a feasible point. If a feasible point is required, the problem should be re-solved as a feasibility problem by omitting the objective. Mathematically speaking, given an unbounded direction v and a feasible point x, $x + tv$ is feasible for all $t \geq 0$, and the objective tends to $-\infty$ (for minimizations; $+\infty$ for maximizations) as $t \rightarrow +\infty$ itself.

Infeasible The problem has been proven to be infeasible through the discovery of an unbounded direction. The values of the variables are filled with `NaN`, and the value of `cvx_optval` is set to `+Inf` for minimizations and feasibility problems, and `-Inf` for maximizations.

Associated with a provably infeasible problem is an *unbounded dual direction*. Appropriate components of this direction are stored in the dual variables. Similarly to the `Unbounded` case, it is important to understand that the unbounded dual direction is very likely not a feasible dual point.

Inaccurate/Solved, Inaccurate/Unbounded, Inaccurate/Infeasible These three status values indicate that the solver was unable to make a determination to within the default numerical tolerance. However, it determined that the results obtained satisfied a "relaxed" tolerance leve and therefore may still be suitable for further use. If this occurs, you should test the validity of the computed solution before using it in further calculations. See *Controlling precision* for a more advanced discussion of solver tolerances and how to make adjustments.

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

Overdetermined The presolver has determined that the problem has more equality constraints than variables, which means that the coefficient matrix of the equality constraints is singular. In practice, such problems are often, but not always, infeasible. Unfortunately, solvers typically cannot handle such problems, so a precise conclusion cannot be reached. The situations that most commonly produce an Overdetermined result are discussed in *Overdetermined problems*.

8.5 Controlling precision

Note: We consider the modification of solver precision to be an advanced feature, to be used sparingly, if at all—and only after you have become comfortable building models in CVX.

Numerical methods for convex optimization are not exact; they compute their results to within a predefined numerical precision or tolerance. Upon solution of your model, the tolerance level the solver has achieved

is returned in the `cvx_slvtol` variable. Attempts to interpret this tolerance level in any absolute sense are not recommended. For one thing, each solver computes it differently. For another, it depends heavily on the considerable transformations that CVX applies to your model before delivering it to the solver. So while you may find its value interesting we strongly discourage dependence upon it within your applications.

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

- The *solver tolerance* ϵ_{solver} is the level requested of the solver. The solver will stop as soon as it achieves this level, or until no further progress is possible.
- The *standard tolerance* $\epsilon_{\text{standard}}$ is the level at which CVX considers the model solved to full precision.
- The *reduced tolerance* $\epsilon_{\text{reduced}}$ is the level at which CVX considers the model "inaccurately" sovled, returning a status with the `Inaccurate/` prefix. If this tolerance cannot be achieved, CVX returns a status of `Failed`, and the values of the variables should not be considered reliable.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.6 Advanced solver settings

Warning: This is an **advanced topic** for users who have a deep understanding of the underlying solver they are using, or who have received specific advice from the solver's developer for improving performance. Improper use of the `cvx_solver_settings` command can cause unpredictable results.

Solvers can be tuned and adjusted in a variety of ways. Solver vendors attempt to select default settings that will provide good performance across a broad range of problems. But no solver, and no choice of settings, will perform well for every possible model. On occasion, it may be worthwhile to give a particular special instructions to improve its performance for a specific application. Unfortunately, such settings differ from solver to solver, so there is no way for CVX to provide this ability in a verifiable, reliable, global fashion.

Nevertheless, using the new `cvx_solver_settings` command, you can customize a solver's settings when a specific model demands it. We cannot emphasize enough that this is an *expert* feature to be employed by experienced modelers only. Indeed, if you are an expert, you understand that these warnings are essential:

- CVX does not check the correctness of the settings you supply. If the solver rejects the settings, CVX will fail until you change or remove those settings.

- There is no guarantee that altering the settings will improve performance in any way; indeed, it can make the performance worse.
- CVX Research provides *no* documentation on the specific settings available for each solver; you will have to consult the solver's own documentation for this.
- The settings set here *override* any default values CVX may have chosen for each solver. Thus in certain cases, using this feature this may actually confuse CVX and cause it to misinterpret the results. For this reason, we cannot support all possible combinations of custom settings.
- Unless you have turned off solver output completely, CVX will warn you if any custom settings are in effect every time you solve model.

With this warning out of the way, let us introduce `cvx_solver_settings`. Typing

```
cvx_solver_settings
```

at the command prompt provides a listing of the custom settings that have been provided for the active solver. Custom settings are *specific to each solver*. Typing

```
cvx_solver_settings -all
```

will provide a full list of the custom settings provided for *all* solvers.

To create a new custom setting for the current solver, use this syntax:

```
cvx_solver_settings( '{name}', {value} )
```

`{name}` must be a valid MATLAB variable/field name. `{value}` can be *any* valid Matlab object; CVX does not check its value in any way.

To clear all custom settings for the active solver, type

```
cvx_solver_settings -clear
```

To clear just a single setting, type

```
cvx_solver_settings -clear {<name>}
```

To clear all settings for all solvers, type

```
cvx_solver_settings -clearall
```

The settings created by the `cvx_solver_settings` command enjoy the same scope as `cvx_solver`, `cvx_precision`, and so forth. For instance, if you use this command *within* a model—between `cvx_begin` and `cvx_end`—the changes will apply only to that particular model. If you issue the command *outside* of a particular model, the change will persist through the end of the MATLAB session (or until you change it again). Finally, if you use the `cvx_save_prefs` command, any custom settings you have added will be saved and restored the next time you start Matlab.

CHAPTER

NINE

REFERENCE GUIDE

In this section we describe each operator, function, set, and command that you are likely to encounter in CVX. In some cases, limitations of the underlying solver place certain restrictions or caveats on their use:

- Functions marked with a dagger (†) are not supported natively by the solvers that CVX uses. They are handled using a successive approximation method which makes multiple calls to the underlying solver, achieving the same final precision. If you use one of these functions, you will be warned that successive approximation will be used. This technique is discussed further in *The successive approximation method*. As this section discusses, this is an experimental approach that works well in many cases, but cannot be guaranteed.
- Functions involving powers (*e.g.*, `x^p`) and p-norms (*e.g.*, `norm(x,p)`) are marked with a double dagger (‡). CVX represents these functions exactly when p is a rational number. For irrational values of `p`, a nearby rational is selected instead. See *Power functions and p-norms* for details on how both cases are handled.

9.1 Arithmetic operators

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.

Numerous other combinations are possible, of course. The use of the exponentiation operators `^ .^` are somewhat limited; see the definitions of `power` in *Nonlinear* below.

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

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

9.2 Built-in functions

9.2.1 Linear

A number of Matlab's basic linear functions have been extended to work with `cvx` expressions as well: `conj`, `conv`, `cumsum`, `diag`, `dot`, `find`, `fliplr`, `flipud`, `flipdim`, `horzcat`, `hankel`, `ipermute`, `kron`, `permute`, `repmat`, `reshape`, `rot90`, `sparse`, `sum`, `trace`, `tril`, `triu`, `toeplitz`, `vertcat`.

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

9.2.2 Nonlinear

`abs` absolute value for real and complex arrays. Convex.

† **`exp`** exponential. Convex and nondecreasing.

† **`log`** logarithm. Concave and nondecreasing.

`max` maximum. Convex and nondecreasing.

`min` minimum. Concave and nondecreasing.

`norm` norms for real and complex vectors and matrices. Convex. Thus function follows the Matlab conventions closely. Thus the one-argument version `norm(x)` computes the 2-norm for vectors, and the 2-norm (maximum singular value) for matrices. The two-argument version `norm(x,p)` is supported as follows:

- ‡ For vectors, all values $p \geq 1$ are accepted.
- For matrices, `p` must be `1`, `2`, `Inf`, or `'Fro'`.

`polyval` polynomial evaluation. `polyval(p,x)`, where `p` is a vector of length `n`, computes

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

This function can be used in CVX in two ways:

- If `p` is a variable and `x` is a constant, then `polyval(x,p)` computes a linear combination of the elements of `p`. The combination must satisfy the DCP rules for addition and scaling.

- If `p` is a constant and `x` is a variable, then `polyval(x,p)` constructs a polynomial function of the variable `x`. The polynomial must be affine, convex, or concave, and `x` must be real and affine.

‡ **`power(x,p)`** `x^p` and `x.^p`, where `x` is a real variable and and `p` is a real constant. For `x^p`, both `x` and `p` must be scalars. Only those values of `p` which can reasonably and unambiguously interpreted as convex or concave are accepted:

- $p = 0$. Constant. `x.^p` is treated as identically 1.
- $0 < p < 1$. Concave. The argument x must be concave (or affine), and is implicitly constrained to be nonnegative.
- $p = 1$. Affine. `x.^p` is simply `x`.
- $p \in \{2, 4, 6, 8, ...\}$. Convex. Argument x must be affine.
- $p > 1, p \notin \{2, 3, 4, 5, ...\}$. Convex. Argument x must be affine, and is implicitly constrained to be nonnegative.

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

† **`power(p,x)`** `p.^x` and `p^x`, where `p` is a real constant and `x` is a real variable. For `p^x`, both `p` and `x` must be scalars. Valid values of `p` include:

- $p \in \{0, 1\}$. Constant.
- $0 < p < 1$. Convex and nonincreasing; `x` must be concave.
- $p > 1$. Convex and nondecreasing; `x` must be convex.

Negative values of `p` are not permitted.

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

9.3 New functions

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

`berhu(x,M)`

The reversed Huber function (hence, Berhu), defined as

$$f_{\text{berhu}}(x, M) \triangleq \begin{cases} |x| & |x| \leq M \\ (|x|^2 + M^2)/2M & |x| \geq M \end{cases}$$

Convex. If M is omitted, $M = 1$ is assumed; but if supplied, it must be a positive constant. Also callable with three arguments as `berhu(x,M,t)`, which computes `t+t*berhu(x/t,M)`, useful for concomitant scale estimation (see [Owen06]).

`det_inv` determinant of inverse of a symmetric (or Hermitian) positive definite matrix, $\det X^{-1}$, which is the same as the product of the inverses of the eigenvalues. When used inside a CVX specification,

`det_inv` constrains the matrix to be symmetric (if real) or Hermitian (if complex) and positive semidefinite. When used with numerical arguments, `det_inv` returns `+Inf` if these constraints are not met. Convex.

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

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

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

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

`huber(x,M)`

The Huber function, defined as

$$f_{\text{huber}}(x,M) \triangleq \begin{cases} |x|^2 & |x| \leq M \\ 2M|x| - M^2 & |x| \geq M \end{cases}$$

Convex. If x is a vector or array, the function is applied on an elementwise basis. If M is omitted, then $M=1$ is assumed; but if it supplied, it must be a positive constant. Also callable as `huber(x,M,t)`, which computes `t+t*huber(x/t,M)`, useful for concomitant scale estimation (see [Owen06]).

`huber_circ(x,M)`

The circularly symmetric Huber function, defined as

$$f_{\text{huber\_circ}}(x,M) \triangleq \begin{cases} \|x\|_2^2 & \|x\|_2 \leq M \\ 2M\|x\|_2 - M^2 & \|x\|_2 \geq M \end{cases}$$

Convex. Same (and implemented) as `huber_pos(norm(x),M)`.

`huber_pos(x,M)` The same as the Huber function for nonnegative `x`; zero for negative `x`. Convex and nondecreasing.

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

† **`kl_div`** Kullback-Leibler distance:

$$f_{\text{kl}}(x,y) \triangleq \begin{cases} x\log(x/y) - x + y & x,y > 0 \\ 0 & x = y = 0 \\ +\infty & \text{otherwise} \end{cases}$$

Convex. Outside CVX specification, returns $+\infty$ if arguments aren't in the domain.

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

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

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

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

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

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

† **log_prod(x)** $\log\prod_i x_i$ if when x is positive; $-\infty$ otherwise. Concave and nonincreasing. Equivalent to `sum_log(x)`.

† **log_sum_exp(x)** the logarithm of the sum of the elementwise exponentials of `x`. Convex and nondecreasing.

logsumexp_sdp a polynomial approximation to the log-sum-exp function with global absolute accuracy. This can be used to estimate the log-sum-exp function without using the successive approximation method.

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

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

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

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

poly_env(p,x) Computes the value of the *convex or concave envelope* of the polynomial described by `p` (in the `polyval` sense). `p` must be a real constant vector whose length `n` is 0, 1, 2, 3, or some other *odd* length; and `x` must be real and affine. The sign of the first nonzero element of `p` determines whether a convex (positive) or concave (negative) envelope is constructed. For example, consider the function $p(x) \triangleq (x^2-1)^2 = x^4 - 2x^2 + 1$, depicted along with its convex envelope in the figure below.

The two coincide when $|x| \geq 1$, but deviate when $|x| < 1$. Attempting to call `polyval([1,0,2,0,1],x)` in a CVX model would yield an error, but a call to `poly_env([1,0,2,0,1],x)` yields a valid representation of the envelope. For convex or concave polynomials, this function produces the same result as `polyval`.

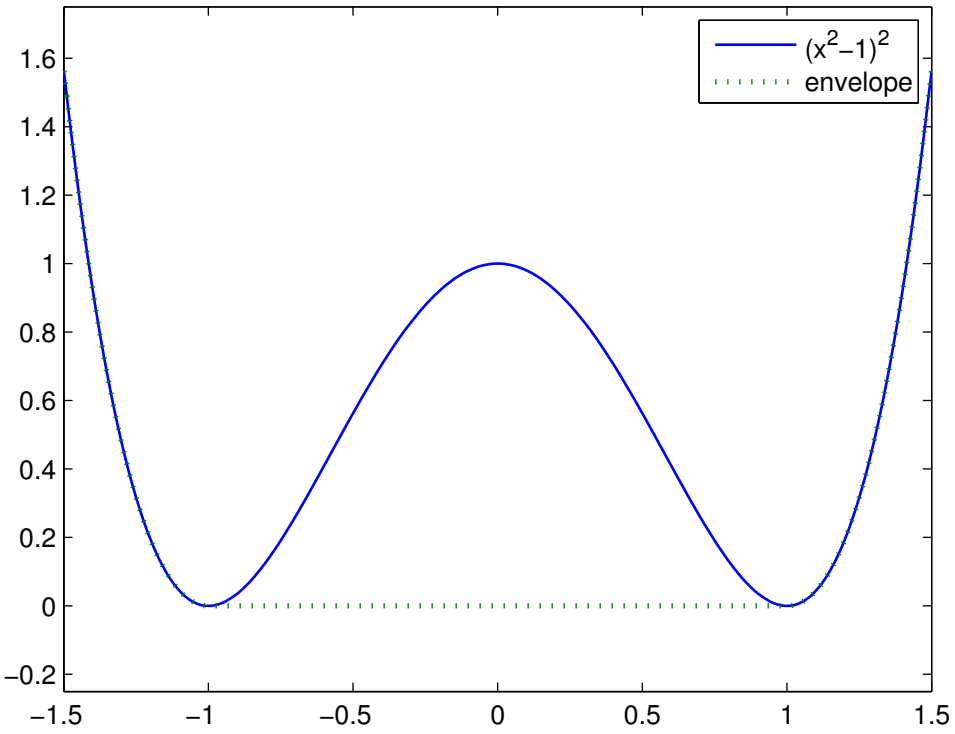

Figure 9.1: The polynomial function $p(x) = x^4 - 2x^2 + 1$ and its convex envelope.

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

‡ **`pow_abs(x,p)`** $|x|^p$ for $x \in \mathbf{R}$ or $x \in \mathbf{C}$ and $p \geq 1$. Convex.

‡ **`pow_pos(x,p)`** $\max\{x, 0\}^p$ for $x \in \mathbf{R}$ and $p \geq 1$. Convex and nondecreasing.

‡ **`pow_p(x,p)`** for $x \in \mathbf{R}$ and real constant p, computes nonnegative convex and concave branches of the power function:

$$
\begin{array}{lll}
p \leq 0 & f_p(x) \triangleq \begin{cases} x^p & x > 0 \\ +\infty & x \leq 0 \end{cases} & \text{convex, nonincreasing} \\
0 < p \leq 1 & f_p(x) \triangleq \begin{cases} x^p & x \geq 0 \\ -\infty & x < 0 \end{cases} & \text{concave, nondecreasing} \\
p \geq 1 & f_p(x) \triangleq \begin{cases} x^p & x \geq 0 \\ +\infty & x < 0 \end{cases} & \text{convex, nonmonotonic}
\end{array}
$$

prod_inv(x) $\prod_i x_i^{-1}$ when x is positive; $+\infty$ otherwise. Convex and nonincreasing.

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.

Note: Quadratic functions such as `quad_form`, `sum_square` can often be replaced by the `norm` function without sacrificing equivalence. For numerical reasons, this alternate formulation is *preferred*. Please see *Eliminating quadratic forms* for more information.

---

quad_over_lin(x,y) $x^T x/y$ 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(x,y) `sum_square_pos( x )/y` for $x \in \mathbf{R}^n$, $y > 0$. Convex, increasing in x, and decreasing in y.

† **rel_entr(x)** Scalar relative entropy; `rel_entr(x,y)=x.*log(x/y)`. Convex.

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

square x^2 for $x \in \mathbf{R}$. Convex.

square_abs $|x|^2$ for $x \in \mathbf{R}$ or $x \in \mathbf{C}$.

square_pos $\max\{x, 0\}^2$ for $x \in \mathbf{R}$. Convex and increasing.

sum_largest(x,k) sum of the largest k values, for real vector x. Convex and increasing.

† **sum_log(x)** $\sum_i \log(x_i)$ when x is positive; $-\infty$ otherwise. Concave and nondecreasing.

sum_smallest(x,k) sum of the smallest k values, *i.e.*, equivalent to `-sum_largest(-x,k)`. Concave and decreasing.

sum_square Equivalent to `sum(square(x))`, but more efficient. Convex. Works only for real values.

sum_square_abs Equivalent to `sum(square_abs(x))`, but more efficient. Convex.

sum_square_pos Equivalent to `sum(square_pos(x))`, but more efficient. Works only for real values. Convex and increasing.

trace_inv(X) trace of the inverse of an SPD matrix `X`, which is the same as the sum of the inverses of the eigenvalues. Convex. Outside of CVX, returns `+Inf` if argument is not positive definite.

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

9.4 Sets

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

nonnegative(n)

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

simplex(n)

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

lorentz(n)

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

rotated_lorentz(n)

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

complex_lorentz(n)

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

rotated_complex_lorentz(n)

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

semidefinite(n)

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

hermitian_semidefinite(n)

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

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

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

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

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

exp_cone

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

geo_mean_cone(n)

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

9.5 Commands

cvx_begin Begins a new CVX model. If a model is already in progress, it will issue a warning and clear it. See *cvx_begin and cvx_end* for a full description, including the modifying keywords that control solver output, SDP mode, GDP mode, etc.

cvx_clear Clears any model being constructed. Useful when an error has been made and it is necessary to start from the beginning. Whereas `cvx_begin` issues a warning if called with a model in progress, `cvx_clear` is silent.

cvx_end Signals the end of a CVX model. In typical use, this instructs CVX to begin the solution process. See *cvx_begin and cvx_end*.

cvx_expert Controls the issuance of warnings when models requiring the use of successive approximation are employed; see *The successive approximation method* more details.

cvx_power_warning Controls if and when CVX issues warnings during the construction of models involving rational power functions (i.e., `x^p`, where `x` is a variable and `p` is a constant); see *Power functions and p-norms*.

cvx_precision Controls solver precision; see *Controlling precision*.

cvx_quiet Enables or disables screen output during the solution process; see *Controlling screen output*. Also see *cvx_begin and cvx_end* for the newer, preferred syntax `cvx_begin quiet`.

cvx_save_prefs Saves the current states for `cvx_expert`, `cvx_power_warning`, `cvx_precision`, and `cvx_solver` to disk, so that their values are retained when quitting and re-starting MATLAB. The file is saved in MATLAB's preference directory, which can be located by typing the `prefdir` command.

cvx_setup The setup script used to install and configure CVX; see *Installation*.

cvx_solver Selects the solver to be employed when solving CVX models; see *Selecting a solver*.

cvx_solver_settings Allows the user to deliver advanced, solver-specific settings to the solver that CVX does not otherwise support; see *Advanced solver settings*.

cvx_version Prints information about the current versions of CVX, Matlab, and the operating system. When submitting bug reports, please include the output of this command.

cvx_where Returns the directory where CVX is installed.

dual variable, dual variables Creates one or more dual variables to be connected to constraints in the current model; see *Dual variables*.

expression, expressions Creates one or more expression holders; see *Assignment and expression holders*.

maximise, maximize Specifies a maximization objective; see *Objective functions*.

minimise, minimize Specifies a minimization objective; see *Objective functions*.

variable, variables Creates one or more variables for use in the current CVX model; see *Variables*.

CHAPTER

TEN

SUPPORT

The user base for CVX has grown to such an extent that full email-based support is no longer feasible for our free version of CVX. Therefore, we have created several avenues for obtaining support.

For help on how to *use* CVX, this users' guide is your first line of support. Please make sure that you have attempted to find an answer to your question here before you pursue other avenues. We have placed this document online and made it searchable in order to help you find the answers to the questions you may have.

With a package like CVX that encapsulates such mathematical complexity, it can sometimes be unclear if a problem with a model is due to model formulation or a bug in CVX. See *What is a bug?* below to help you discern the difference, and to determine the most appropriate channel for support.

10.1 The CVX Forum

If your answers cannot be found here, consider posting your question to the CVX Forum. This is a community forum that allows our users to submit questions and to answer other people's questions. The forum uses the open-source Askbot system, and its format should be familiar to anyone who participates in OR-Exchange, Stack Overflow, or any one of the Stack Exchange family of sites.

We highly encourage our expert users who enjoy helping others to participate in this forum. We hope that it will not only serve as a resource for diagnosing problems and issues, but a clearinghouse for advanced usage tips and tricks.

10.2 Bug reports

If you believe you have found a *bug* in CVX or in one of the underlying solvers, then we encourage you to submit a bug report—either by email to to `cvx@cvxr.com` or through our web-based support portal. Please include the following in your bug report so we can fully reproduce the problem:

1. the CVX model and supporting data that caused the error.
2. a copy of any error messages that it produced
3. the CVX version number and build number
4. the version number of Matlab that you are running

5. the name and version of the operating system you are using

The easiest way to supply items 3-5 is to type `cvx_version` at the command prompt and copy its output into your email message.

Please note that we do not own all of Matlab's toolboxes. We cannot debug a model that employs functions from a toolbox we do not own.

10.3 What *is* a bug?

Certain issues are unambiguously bugs, and you should feel free to report them immediately. In particular, CVX often attempts to catch unexpected errors in key places—including `cvx_setup`, `cvx_end`, etc. It will report those errors and instruct you to report them to us. If your model produces a MATLAB error that CVX did not itself generate, and you cannot readily tie it to a syntax error in your model, please report that as well.

That said, because disciplined convex programming is a new concept for many, we often receive support requests for problems with their models that are not, in fact, bugs. A particularly common class of support requests are reports of models being rejected due to `Disciplined convex programming error` messages. For instance, the following code

```
variable x(10)
norm(x) == 1
```

will produce this error in CVX:

```
Error using cvxprob/newcnstr (line 181)
Disciplined convex programming error:
  Invalid constraint: {convex} == {real constant}
```

Disciplined convex programming errors indicate that the model fails to adhere to the rules in the *DCP ruleset*. In nearly all cases, this is *not* a bug, and should not be reported as such. Rather, the underlying issue falls into one of two categories:

1. The model is *not convex* (mixed-integer or otherwise). A model with the nonlinear equation above would fall squarely in this category. CVX simply cannot solve such problems. In some cases, it is possible to transform a problem that is non-convex into one that is convex (for example, *geometric programs*). This has not been the case for any problem submitted by our users—so far.

2. The model *is* convex, but it is still written in a manner that violates the rules. For instance, given the same vector `x` above, the constraint

   ```
   sqrt( sum( square( x ) ) ) <= 1
   ```

   is convex, but it violates the ruleset—so it is rejected. However, the mathematically equivalent form

   ```
   norm( x ) <= 1
   ```

   is acceptable. If your error is of this type, you will need to find a way to express your problem in a DCP-compliant manner. We have attempted to supply all of the commonly used functions that the underlying solvers can support; so if you cannot easily rewrite your problem using the functions

supplied, it may not be possible. If you think this is a possibility, you may wish to see if the wizards on the CVX Forum have suggestions for you.

In rare cases, users have discovered that certain models were rejected with `Disciplined convex programming error` messages even though they satisfied the *DCP ruleset*. We have not received a bug report of this type in quite a long time, however, so we suspect our users have helped us iron out these issues.

10.4 Handling numerical issues

No developer likes to tell their customers that their software may not work for them. Alas, we have no choice. The fact is that we cannot guarantee that CVX will be able to solve your problem, *even if* it is formulated properly, even if it avoids the use of integer or binary variables, even if it is of reasonable size, even if it avoids the use of our experimental *exponential cone support*.

We blame the solvers—but we must come to their defense, too. Even the best and most mature solvers will struggle with a particular problem that seems straightforward. Another solver may have no difficulty with that one, but fail to find an accurate solution on another. While sometimes these challenges are due to bugs in the solver's implementation, quite often it is simply due to limits imposed by the nature of finite numerical precision computation. And different problems push those limits to different degrees. So the fact is that no solver is perfect, but no solver *can* be.

When we consider *mixed-integer problems*, the situation is even worse. Solvers must perform what is effectively an exhaustive search among the integer variables to determine the correct solution. Yes, there are some intelligent and innovative ways to speed up that search, and the performance of mixed-integer solvers has improved dramatically over the years. But there will always be models for which the exhaustive search will simply take too long.

None of this is much comfort if it is *your* model the solver is struggling with. Here are some practical tips if you encounter this problem:

Try a different solver. Use the `cvx_solver` command for this. If you are using Gurobi or MOSEK, don't hesitate to try one of the free solvers if they are compatible with your problem.

Reduce the precision. Consider inserting `cvx_precision medium` or even `cvx_precision low` into your problem to see if that allows the solver to exit successfully. Of course, if it does succeed, make sure to check the results to see if they are acceptable for your application. If they are not, consider some of the other advice here to see if the solvability of your model may be improved.

Remove constraints. If you think that one or more of the constraints might not be active at the solution, try removing them. If the solver terminates, you can confirm that your guess was correct by examining the solution to the modified problem.

Add constraints. Consider adding simple bounds to the constraints to reduce the size of the feasible set. This will sometimes improve the numerical conditioning of the problem. Make them as tight as you can without impinging on the optimal set. If this modified problem is successfully solved, check the solution to see if any of the added bounds are active. If they are, relax them, and try again.

Watch for scaling issues. Scaling issues are the most vexing problems for numerical solvers to deal with. Solvers will often re-scale the problem to reduce the dynamic range of the numerical coefficients, but doing so sometimes leads to undesirable effects on the solution. It is better to avoid scaling issues

during the modeling process. Don't mix values of wildly different magnitudes, such as `1e-3` and `1e20`. Even better, try to avoid any numerical values (both in fixed parameters and likely values of the variables) that exceed `1e8` in absolute value.

Try equivalent reformulations. It is quite likely that your model can be expressed in a variety of different ways. Certainly, you should begin with the most obvious and natural formulation; but if you encounter numerical issues, a reformulation may often solve them. One reformulation we highly recommend is to eliminate quadratic forms; see *this section* for more details.

Reach out to the CVX Forum. Share your struggles with the larger CVX community! Perhaps they will have concrete suggestions for improving the solvability of your model.

10.5 CVX Professional support

Paid CVX Professional users will receive support through a trouble-ticket support system, so that they can have confidence that their issues are being addressed promptly. This infrastructure is still under development; we will update this section and the Web site with more information once it has been completed.

CHAPTER

ELEVEN

ADVANCED TOPICS

Note: 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.

11.1 Eliminating quadratic forms

One particular reformulation that we *strongly* encourage is to eliminate quadratic forms—that is, functions like `sum_square`, `sum(square(.))` or `quad_form`—whenever it is possible to construct equivalent models using `norm` instead. Our experience tells us that quadratic forms often pose a numerical challenge for the underlying solvers that CVX uses.

We acknowledge that this advice goes against conventional wisdom: quadratic forms are the prototypical smooth convex function, while norms are nonsmooth and therefore unwieldy. But with the *conic* solvers that CVX uses, this wisdom is *exactly backwards*. It is the *norm* that is best suited for conic formulation and solution. Quadratic forms are handled by *converting* them to a conic form—using norms, in fact! This conversion process poses some interesting scaling challenges. It is better if the modeler can eliminate the need to perform this conversion.

For a simple example of such a change, consider the objective

```
minimize( sum_square( A * x - b ) )
```

In exact arithmetic, this is precisely equivalent to

```
minimize( square_pos( norm( A * x - b ) ) )
```

But equivalence is also preserved if we eliminate the square altogether:

```
minimize( norm( A * x - b ) )
```

The optimal value of `x` is identical in all three cases, but this last version is likely to produce more accurate results. Of course, if you *need* the value of the squared norm, you can always recover it by squaring the norm after the fact.

Conversions using `quad_form` can sometimes be a bit more difficult. For instance, consider

```
quad_form( A * x - b, Q ) <= 1
```

where `Q` is a positive definite matrix. The equivalent `norm` version is

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

where `Qsqrt` is an appropriate matrix square root of `Q`. One option is to compute the symmetric square root `Qsqrt = sqrtm(Q)`, but this computation destroys sparsity. If `Q` is sparse, it is likely worth the effort to compute a sparse Cholesky-based square root:

```
[ Qsqrt, p, S  ] = chol( Q );
Qsqrt = Qsqrt * S;
```

Sometimes an effective reformulation requires a practical understanding of what it means for problems to be equivalent. For instance, suppose we wanted to add an ℓ_1 regularization term to the objective above, weighted by some fixed, positive `lambda`:

```
minimize( sum_square( A * x - b ) + lambda * norm( x, 1 ) )
```

In this case, we typically do not care about the *specific* values of `lambda`; rather we are varying it over a range to study the tradeoff between the residual of `A*x-b` and the 1-norm of `x`. The same tradeoff can be studied by examining this modified model:

```
minimize( norm( A * x - b ) + lambda2 * norm( x, 1 ) )
```

This is not precisely the same model; setting `lambda` and `lambda2` to the same value will not yield identical values of `x`. But both models *do* trace the same tradeoff curve—only the second form is likely to produce more accurate results.

11.2 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 from the SeDuMi examples:

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

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

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

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

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

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

This example admittedly is a bit simplistic. With a bit of careful arrangement, it is possible to rewrite this model so that the n equality constraints can be combined into a single vector constraint, which in turn would require only a single vector dual variable. [1] 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.

11.3 The successive approximation method

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

For this reason, we have constructed a *successive approximation* method that allows symmetric primal/dual solvers to support the exponential family of functions. A precise description of the approach is beyond the scope of this text, but we can provide a highly simplified description here. First, we construct a global

[1] 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`.

approximation for each exponential family function which is accurate within a neighborhood of some center point x_0. Solving this approximate model yields an approximate optimal point $\bar{x}$. We shift the center point x_0 towards $\bar{x}$, construct a new approximation, and solve again. This process is repeated until $|\bar{x} - x_0|$ is small enough to conclude that our approximation is accurate enough to represent the original model. Again, this is a highly simplified description of the approach; for instance, we actually employ both the primal and dual solutions to guide our judgements for shifting x_0 and terminating.

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

```
cvx_expert true
```

into your model before the first use of such features.

11.4 Power functions and p-norms

In order to implement the convex or concave branches of the power function x^p and p-norms $\|x\|_p$ for general values of p, CVX uses an enhanced version of the SDP/SOCP conversion method described by [AG00]. This approach is exact—as long as the exponent p is rational. To determine integral values p_n, p_d such that $p_n/p_d = p$, CVX uses Matlab's `rat` function with its default tolerance of 10^{-6}. There is currently no way to change this tolerance. See the MATLAB documentation for the `rat` function for more details.

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

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

For other values of $p = p_n/p_d$, CVX generates a number of 2×2 LMIs that depends on both p_n and p_d; we denote this number by $k(p_n, p_d)$. (In some cases additional linear constraints are also generated, but we ignore them for this analysis.) For instance, for $p = 3/1$, we have

$$x^3 \leq y, x \geq 0 \quad \Longleftrightarrow \quad \exists z \; \begin{bmatrix} z & x \\ x & 1 \end{bmatrix} \succeq 0. \; \begin{bmatrix} y & z \\ z & x \end{bmatrix} \succeq 0.$$

So $k(3,1) = 2$. An empirical study has shown that for $p = p_n/p_d > 1$, we have

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

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

The cost of this SDP representation is relatively small for nearly all useful values of p. Nevertheless, CVX issues a warning whenever $k(p_n, p_d) > 10$ to insure that the user is not surprised by any unexpected slowdown. In the event that this threshold does not suit you, you may change it using the command `cvx_power_warning(`*`thresh`*`)`, where *`thresh`* is the desired cutoff value. Setting the threshold to `Inf` disables it completely. As with the command `cvx_precision`, you can place a call to

`cvx_power_warning` within a model to change the threshold for a single model; or outside of a model to make a global change. The command always returns the *previous* value of the threshold, so you can save it and restore it upon completion of your model, if you wish. You can query the current value by calling `cvx_power_warning` with no arguments.

11.5 Overdetermined problems

The 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:

$$\begin{array}{ll} \text{minimize} & \text{Tr}(D) \\ \text{subject to} & W + D \succeq 0 \\ & D \text{ diagonal} \end{array}$$

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. We can correct this problem by replacing the equality constraint with

```
sym( W ) + D == semidefinite(n);
```

`sym` is a function we have provided that extracts the symmetric part of its argument; that is, `sym(W)` equals `0.5 * ( W + W' )`.

11.6 Adding new functions to the atom library

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

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

11.6.1 New functions via the DCP ruleset

The simplest way to construct a new function that works within CVX is to construct it using expressions that fully conform to the DCP ruleset. Consider, for instance, the deadzone function

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

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

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

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

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

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

then the modified function fails to satisfy the DCP ruleset. The function will work *outside* of a CVX specification, happily computing the value $\min\{|x| - 1, 0\}$ for a *numerical* argument x. But inside a CVX specification, invoked with a nonconstant argument, it will generate an error.

11.6.2 New functions via partially specified problems

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

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

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

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

with optimization variable y.

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

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

gives the classic *epigraph* representation of f:

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

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

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

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

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

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

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

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

We can implement the Huber penalty function in CVX as follows:

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

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

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

function `huber` can be used anywhere a traditional convex function can be used, in constraints or objective functions, in accordance with the DCP ruleset.

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

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

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

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

gives the *hypograph* representation of f:

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

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

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

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

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

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

So we can implement this function in CVX as follows:

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

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

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

```
function cvx_optval = huber( x )
sx = size( x );
cvx_begin
    variables w( sx ) v( sx );
```

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

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

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

```
function cvx_optval = huber( x )
if isnumeric( x ),
    xa   = abs( x );
    flag = xa < 1;
    cvx_optval = flag .* xa.^2 + (~flag) * (2*xa-1);
else,
    sx = size( x );
    cvx_begin
        variables w( sx ) v( sx );
        minimize( w .^ 2 + 2 * v );
        subject to
            abs( x ) <= w + v;
            w <= 1; v >= 0;
    cvx_end
end
```

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

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

CHAPTER

TWELVE

LICENSE

CVX: A system for disciplined convex programming
© 2012 CVX Research, Inc., Austin, TX.
http://cvxr.com
info@cvxr.com

Thank you for using CVX!

The files contained in the CVX distribution come from several different sources and are covered under a variety of licenses. The files owned by CVX Research, Inc. are covered under one of two licenses: the *CVX Standard License* and the *CVX Professional License*. The CVX Standard License is effectively the GNU General Public License, Version 2 (GPLv2), but with additional terms that govern modifications that connect CVX to additional solvers. The added terms are discussed in "Solver Interfaces" below.

12.1 CVX Professional License

The standard CVX distribution includes several files in Matlab *p-code* format, which contain encrypted binary versions of Matlab bytecode. As their name implies, they are recognized by their `.p` suffix. Currently the following files are distributed this way:

```
shims/cvx_mosek.p
shims/cvx_gurobi.p
cvx_license.p
```

You may redistribute these files only as part of the complete, unmodified CVX package as distributed by CVX Research, Inc. itself.

12.2 CVX Standard License

If you wish, you may distribute a version of CVX with all of the p-code files *removed*. The resulting package retains full functionality with the exception of its ability to connect to commercial solvers. This modified package is covered by the CVX Standard License. Under this license, you are free to redistribute and/or

modify the files under the terms of the GPLv2, plus the additional terms discussed in "Solver Interfaces" below.

You must include the files `LICENSE.txt` and `GPL.txt` in unmodified form when redistributing this software. If you did not receive a copy of either of these files with your distribution, please contact us.

12.3 Solver Interfaces

CVX relies upon other software packages, called *solvers*, to perform many of its underlying calculations. Currently CVX supports free solvers SeDuMi and SDPT3, and commercial solvers Gurobi and MOSEK. The resulting nexus of free and commercial software presents a licensing challenge. Our vision is guided by three goals:

- to ensure that CVX remains free to use by *all* users with any compatible *free* solver.
- to generate revenue by selling interfaces to *commercial* solvers to *commercial* customers.
- to provide the academic community with the full commercial capability at no charge.

The terms we lay out here are intended to support these trifold goals.

We invite our users to create new interfaces between CVX and other *free* solvers. By "free", we mean that the solver must be made available at no charge for to *all* users, including commercial users, without restriction. Please contact us if you are interested in creating such an interface; we can offer assistance. If you do create one, please consider submitting it to us for inclusion in the standard CVX distribution. But you are under no obligation to do this. Instead, you can ship the interface code with the solver itself; or you can construct a modified version of CVX with your interface included.

We do not permit the creation and distribution of new interfaces between CVX and *non-free* solvers—even if those solvers are made available to academic users at no charge. If you are a vendor or developer of a commercial solver, and would like to develop or offer a CVX interface to your users, please contact us at info@cvxr.com. We welcome the opportunity to support a wider variety of commercial solvers with CVX, and are willing to devote engineering resources to make those connections.

If you are a user of a particular commercial solver and would like to see it supported by CVX, please contact your solver vendor—but please contact us at info@cvxr.com as well. If there is sufficient demand, and it proves technically and financially feasible, we will reach out to the solver vendor to work on an implementation.

12.4 Bundled solvers

The solvers SDPT3 and SeDuMi are distributed with CVX in the `sdpt3/` and `sedumi/` subdirectories, respectively. Neither of these packages is owned by CVX Research, Inc. Both are included with permission of the authors, and licensed under the terms of the GPLv2. Please consult the plain-text documentation contained in each of these directories for more information about copying, citation, and so forth.

12.5 Example library

The contents of the example library, which is distributed with CVX in the `examples/` subdirectory, is *public domain*. You are free to use them in any way you wish; but when you do, we request that you give appropriate credit to the authors. A number of people have contributed to the examples in this library, including Lieven Vandenberghe, Joëlle Skaf, Argyris Zymnis, Almir Mutapcic, Michael Grant, and Stephen Boyd.

12.6 No Warranty

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

CHAPTER

THIRTEEN

CITING CVX

If you are actively using CVX in teaching, research, or applications, and haven't yet told us about it, please do so! Drop us an email at CVX Research Support. It is truly encouraging to hear about new uses for CVX, and we like to keep track of geographic and technical diversity of our user base. And of course, it is always a pleasure to receive an email at CVX Research Support that is not a bug report!

Are you using CVX in research work to be published? If so, please include explicit mention of our work in your publication. We suggest language such as this:

> To solve problem (17) we used `CVX`, a package for specifying and solving convex programs [1],[2].

with the following corresponding entries in your bibliography:

> CVX Research, Inc. CVX: Matlab software for disciplined convex programming, version 2.0. `http://cvxr.com/cvx`, April 2011.
>
> M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs, Recent Advances in Learning and Control (a tribute to M. Vidyasagar), V. Blondel, S. Boyd, and H. Kimura, editors, pages 95-110, Lecture Notes in Control and Information Sciences, Springer, 2008. http://stanford.edu/~boyd/graph_dcp.html.

The corresponding BiBTeX citations are given below:

```
@misc{cvx,
  author       = {CVX Research, Inc.},
  title        = {{CVX}: Matlab Software for Disciplined Convex Programming, version 2.0},
  howpublished = {\url{http://cvxr.com/cvx}},
  month        = aug,
  year         = 2012
}
@incollection{gb08,
  author    = {M. Grant and S. Boyd},
  title     = {Graph implementations for nonsmooth convex programs},
  booktitle = {Recent Advances in Learning and Control},
  series    = {Lecture Notes in Control and Information Sciences},
  editor    = {V. Blondel and S. Boyd and H. Kimura},
  publisher = {Springer-Verlag Limited},
  pages     = {95--110},
  year      = 2008,
```

```
  note       = {\url{http://stanford.edu/~boyd/graph_dcp.html}}
}
```

CHAPTER

FOURTEEN

CREDITS AND ACKNOWLEDGEMENTS

CVX was designed by Michael Grant and Stephen Boyd, with input from Yinyu Ye; and was implemented by Michael Grant [GBY06]. It incorporates ideas from earlier works by Löfberg [Löf04], Dahl and [DV04], Wu and Boyd [WB00], and many others. The modeling language follows the spirit of AMPL or GAMS; unlike these packages, however, CVX was designed from the beginning to fully exploit convexity. The specific method for implementing CVX in Matlab draws heavily from YALMIP.

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

We would like to thank Gurobi Optimization and MOSEK ApS for their generous assistance as we developed the interfaces to their commercial products.

BIBLIOGRAPHY

[AG00] F. Alizadeh and D. Goldfarb. Second-order cone programming. *Mathematical Programming, Series B*, 95:3-51, 2001. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.5133

[BKVH05] S. Boyd, S. J. Kim, L. Vandenberghe, and A. Hassibi,. A tutorial on geometric programming. *Optimization and Engineering*, 8(1):67-127, 2007. http://stanford.edu/~boyd/papers/gp_tutorial.html

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

[Cru02] C. Crusius. *A Parser/Solver for Convex Optimization Problems*. Ph.D. thesis, Information Systems Laboratory, Department of Electrical Engineering, Stanford University, 2002.

[DV04] J. Dahl and L. Vandenberghe, CVXOPT: A Python package for convex optimization (version 1.1.5). http://abel.ee.ucla.edu/cvxopt/

[GBY06] M. Grant and S. Boyd and Y. Ye. Disciplined convex programming. In *Global Optimization: from Theory to Implementation*, Nonconvex Optimization and Its Applications, L. Liberti and N. Maculan, *eds.*, Springer, 2006. http://stanford.edu/~boyd/disc_cvx_prog.html

[Gra04] M. Grant. *Disciplined Convex Programming*. Ph.D. thesis, Information Systems Laboratory, Department of Electrical Engineering, Stanford University, 2004. http://stanford.edu/~boyd/disc_cvx_prog.html

[Löf04] J. Löfberg. YALMIP: a toolbox for modeling and optimization in MATLAB. *Proceedings of the 2004 International Symposium on Computer Aided Control Systems Design*, IEEE Press, September 2004, pp. 284-289. http://users.isy.liu.se/johanl/yalmip/

[Owen06] A. Owen. A robust hybrid of lasso and ridge regression. Technical report, Department of Statistics, Stanford University, October 2006. http://www-stat.stanford.edu/~owen/reports/hhu.pdf

[Stu99] J.F. Sturm, Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11-12:625-633, 1999. Special issue on Interior Point Methods (CD supplement with software). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6954

[TTT03] R.H. Tütüncü, K.C. Toh, and M.J. Todd. Solving semidefinite-quadratic-linear programs using SDPT3. *Mathematical Programming, Series B*, 95:189-217, 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.4807

[WB00] S.P. Wu and S. Boyd. SDPSOL: A parser/solver for semidefinite programs with matrix structure. In *Recent Advances in LMI Methods for Control*, L. El Ghaoui and S.I. Niculescu, *eds.*, SIAM, pp. 79-91, 2000. http://www.stanford.edu/~boyd/sdpsol.html

INDEX

W