CVX Tutorial , Lec. 6

Intorduction

- CVX is a modeling system for disciplined convex programming.
- Disciplined convex programs, or DCPs, are convex optimization problems that are described using a limited set of construction rules, which enables them to be analyzed and solved efficiently.
- CVX is implemented in Matlab.
- 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;

Introduction

- If CVX accepts your problem, you can be sure it is convex.
- CVX is not meant for very large problems, so if your problem is very large (for example, a large image processing problem), CVX is unlikely to work well (or at all). For such problems you will likely need to directly call a solver

Disciplined convex programming

- Disciplined convex programming is a methodology for constructing convex optimization problems proposed by Michael Grant, Stephen Boyd, and Yinyu Ye.
- 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.

A Quick start

- Once you have installed CVX, you can start using it by entering a CVX specification into a Matlab script or function, or directly from the command prompt.
- 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.
- As Matlab reads a CVX specification, it builds an internal representation of the optimization problem.

A Quick start

- When Matlab reaches the CVX_end command, it completes the conversion of the CVX specification to another 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.
- 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).

Example 1

- Least-squares

we seek $x \in \mathbf{R}^n$ that minimizes $\|Ax - b\|_2$, where $A \in \mathbf{R}^{m \times n}$

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

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

- CVX requires that all problem variables be declared before they are used in an objective function or constraints.

Example 1

- 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.
- In addition, four additional Matlab variables are created:

1. CVX_optval, contains the value of the objective function
2. CVX_status, which contains a string describing the status of the calculation. In this case, CVX_status would contain the string Solved.
3. CVX_slvtol: the tolerance level achieved by the solver.
4. CVX_slvitr: the number of iterations taken by the solver.

Example 1

- 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.

Example 2

- Bound-constrained least-squares

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

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

- The subject to statement does nothing—CVX provides this statement simply to make specifications more readable. It is entirely optional.

Constraints

```
cvx_begin
    variable x(n);
    minimize( norm(A*x-b) );
    subject to
        C*x == d;
        norm(x,Inf) <= 1;
cvx_end
```

- Expressions using comparison operators (==, >=, etc.) behave differently when they involve CVX optimization variables than when they involve simple numeric values. For example, because x is a declared variable, the expression C*x==d above causes a constraint to be included in the CVX specification, and returns no value at all.

Constraints

- 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 `1-norm is less than or equal to, or more than, 1).
- Because CVX is designed to support convex optimization, it must be able to verify that problems are convex. To that end, CVX adopts certain construction rules that govern how constraint and objective expressions are constructed. For example, CVX requires that the left- and right- hand sides of an equality constraint be affine.
- Inequality constraints of the form f(x) <= g(x) or g(x) >= f(x) are accepted only if f can be verified as convex and g verified as concave.

The Basics

- CVX begin and CVX end
  - All CVX models must be preceded by the command CVX_begin and terminated with the command CVX_end. All variable declarations, objective functions, and constraints should fall in between.
  - The CVX_begin command accepts several modifiers that you may find useful. For instance, CVX_begin quiet prevents the model from producing any screen output while it is being solved.

The Basics

- Data types for variables

- All variables must be declared using the variable command (or the variables command;) before they can be used in constraints or an objective function.
- Variables can be real or complex; and scalar, vector, matrix, or n-dimensional arrays.
- Matrices can have structure such as symmetry.

The Basics

- The structure of a variable is given by supplying a list of descriptive keywords after the name and size of the variable.

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

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

The Basics

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

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

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

The Basics

- Objective functions

- Declaring an objective function requires the use of the minimize or maximize function,as appropriate.

- The objective function in a call to minimize must be convex; the objective function in a call to maximize must be concave. At most one objective function may be declared in a given CVX specification, and the objective function must have a scalar value.

The Basics

- 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.

The Basics

- Constraints

➢ The following constraint types are supported in CVX:

1. Equality == constraints, where both the left- and right-hand sides are affine functions of the optimization variables.
2. Less-than <=, < inequality constraints, where the left-hand expression is convex, and the right-hand expression is concave.
3. Greater-than >=, > constraints, where the left-hand expression is concave, and the right-hand expression is convex.

➢ In CVX, the strict inequalities < and > are accepted, but interpreted as the associated nonstrict inequalities, <= and >=, respectively

The Basics

- These equality and inequality operators work for arrays. When both sides of the constraint are arrays of the same size, the constraint is imposed elementwise.
- CVX also handles cases where one side is a scalar and the other is an array. This is interpreted as a constraint for each element of the array, with the (same) scalar appearing on the other side.
- Note also the important distinction between =, which is an assignment, and ==, which imposes an equality constraint.
- Inequalities cannot be used if either side is complex.

The Basics

- 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.
- DCP

The Basics

- Sets
- CVX supports the definition and use of convex sets. The base library includes the cone of positive semidefinite n×n matrices, the second-order or Lorentz cone, and various norm balls.
- Matlab does not have a set membership operator, such as x in S. To represent a set a function that returns an unnamed variable that is required to be in the set is used.
- Consider, for example the cone of symmetric positive semidefinite n×n matrices. In CVX, we this is represented 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, the syntax X == semidefinite(n) is used.

The Basics

- Example

Consider the second-order or Lorentz cone

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

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

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

The Basics

- Dual variables
- When a disciplined convex program is solved, the associated dual problem is also solved.
- The optimal dual variables, each of which is associated with a constraint in the original problem, give valuable information about the original problem.
- To get access to the optimal dual variables in CVX, you simply declare them, and associate them with the constraints.

The Basics

- Example of LP

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

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

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

The Basics

- No dimensions are given for y; they are automatically determined from the constraint with which it is associated.
- 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;
```

The Basics

- Expression holders
- Sometimes it is useful to store a CVX expression into a Matlab variable for future use.

```
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.

The Basics

- The keywords expression and expressions have been provided 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.

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

The Basics

- 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. 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-ssigned, and overwritten in a CVX specification.

DCP rule set

- 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 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.

DCP rule set

- As an example consider the entropy If it is expressed as

$$-\sum_{i=1}^{n} x_i \log x_i,$$

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

- CVX will reject it, because its concavity does not follow from any of the composition 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.

DCP rule set

- If a convex (or concave) function is not recognized as convex or concave by CVX, it can be added as a new atom.
- As another example consider the function

$$\sqrt{x^2+1} = \|[x\ 1]\|_2$$

- which is convex. If it is written as `norm([x 1])`
- it will be recognized by CVX as a convex expression, and therefore can be used in (appropriate) constraints and objectives.

DCP rule set

- 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.

DCP rule set

- Curvature
- In disciplined convex programming, a scalar expression is classified by its curvature. There are four categories of curvature: constant, affine, convex, and concave.

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

DCP rule set

- Top-level rules

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

DCP rule set

- Constraints

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

- One or both sides of an equality constraint may be complex; inequality constraints, on the other hand, must be real.

DCP rule set

- Expression rules
- 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.

DCP rule set

- 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.

DCP rule set

- A valid convex expression is
  - a valid constant or affine expression;
  - a valid call to a function in the atom library with a convex result;
  - an affine scalar raised to a constant power $p \geq 1$, $p \neq 3, 5, 7, 9, ...$;
  - a convex scalar quadratic form (§4.8);
  - the sum of two or more convex expressions;
  - the difference between a convex expression and a concave expression;
  - the product of a convex expression and a nonnegative constant;
  - the product of a concave expression and a nonpositive constant;
  - the negation of a concave expression.

DCP rule set

- A valid concave expression is
  - a valid constant or affine expression;
  - a valid call to a function in the atom library with a concave result;
  - a concave scalar raised to a power $p \in (0, 1)$;
  - a concave scalar quadratic form (§4.8);
  - the sum of two or more concave expressions;
  - the difference between a concave expression and a convex expression;
  - the product of a concave expression and a nonnegative constant;
  - the product of a convex expression and a nonpositive constant;
  - the negation of a convex expression.

DCP rule set

- 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.
- Of particular note is that these expression rules generally forbid products between nonconstant expressions.
- 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.

DCP rule set

- 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. Monotonicity determines how they can be used in function compositions, as we shall see.
- Following standard practice in convex analysis, convex functions are interpreted as +inf when the argument is outside the domain of the function, and concave functions are interpreted as −inf when the argument is outside its domain. In other words, convex and concave functions in CVX are interpreted as their extended-valued extensions.

DCP rule set

- 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 (−inf) for negative values of its argument; then jumps up to 0 for argument zero, and increases for positive values of its argument.

DCP rule set

- 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 (inf) therwise, for the convex portion of 1/x; CVX recognizes this function as convex and nonincreasing.

DCP rule set

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

`quad_over_lin( x, y )` $\begin{cases} |x|^2/y & y > 0 \\ +\infty & y \leq 0 \end{cases}$ convex, nonincreasing in y

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

DCP rule set

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

`norm( x, p )`	$\|x\|_p \quad (1 \leq p)$	convex in x, nonmonotonic

is convex only in its first argument. Whenever this function is used in a CVX specification, then, the remaining arguments must be constant, or CVX will issue an error message.

DCP rule set

- Compositions

1. 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.

- Example :Consider the function square(x), which is provided in the CVX atom library. This function squares is 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.

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

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

DCP rule set

- We consider a function, of known curvature and monotonicity, that accepts multiple arguments. For convex functions, the rules are:

1. If the function is nondecreasing in an argument, that argument must be convex.
2. If the function is nonincreasing in an argument, that argument must be concave.
3. 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

DCP rule set

- We consider a function, of known curvature and monotonicity, that accepts multiple arguments. For concave functions, the rules are:

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

- the expression is accepted by CVX, and classified as concave.

DCP rule set

- 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.

Semidefinite programming

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

Semidefinite programming

1. Equality constraints are interpreted the same (i.e., elementwise).
2. Inequality constraints involving vectors and scalars are interpreted the same; i.e., elementwise.
3. Inequality constraints involving non-square matrices are disallowed; attemptingto use them causes an error. If you wish to do true elementwise comparison of matrices X and Y, use a vectorization operation.

Semidefinite programming

- Inequality constraints involving real, square matrices are interpreted as follows:

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

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

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

Semidefinite programming

- 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*

Semidefinite programming

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

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

Semidefinite programming

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

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