CVXPY Documentation

Release 0.4.8

Steven Diamond, Eric Chu, Stephen Boyd

Mar 07, 2017

Contents

Join the CVXPY mailing list **and** Gitter chat **for the best CVXPY support!**

CVXPY 1.0 is under development. **There will be some** changes to the user interface.

CVXPY is a Python-embedded modeling language for convex optimization problems. It allows you to express your problem in a natural way that follows the math, rather than in the restrictive standard form required by solvers.

For example, the following code solves a least-squares problem where the variable is constrained by lower and upper bounds:

```
from cvxpy import *
import numpy

# Problem data.
m = 30
n = 20
numpy.random.seed(1)
A = numpy.random.randn(m, n)
b = numpy.random.randn(m)

# Construct the problem.
x = Variable(n)
objective = Minimize(sum_squares(A*x - b))
constraints = [0 <= x, x <= 1]
prob = Problem(objective, constraints)

# The optimal objective is returned by prob.solve().
result = prob.solve()
# The optimal value for x is stored in x.value.
print x.value
# The optimal Lagrange multiplier for a constraint
# is stored in constraint.dual_value.
print constraints[0].dual_value
```

This short script is a basic example of what CVXPY can do. CVXPY also supports simple ways to solve problems in parallel, higher-level abstractions such as object oriented convex optimization, and extensions for non-convex optimization.

CVXPY was designed and implemented by Steven Diamond, with input from Stephen Boyd and Eric Chu.

CVXPY was inspired by the MATLAB package CVX. See the book Convex Optimization by Boyd and Vandenberghe for general background on convex optimization.

CVXPY relies on the open source solvers ECOS, CVXOPT, and SCS. Additional solvers are supported, but must be installed separately.

CHAPTER 1

Install Guide

Mac OS X and Linux

CVXPY supports both Python 2 and Python 3 on OS X and Linux.

1. Install Anaconda.
2. Install `cvxpy` with `conda`.

```
conda install -c cvxgrp cvxpy
```

3. Test the installation with `nose`.

```
conda install nose
nosetests cvxpy
```

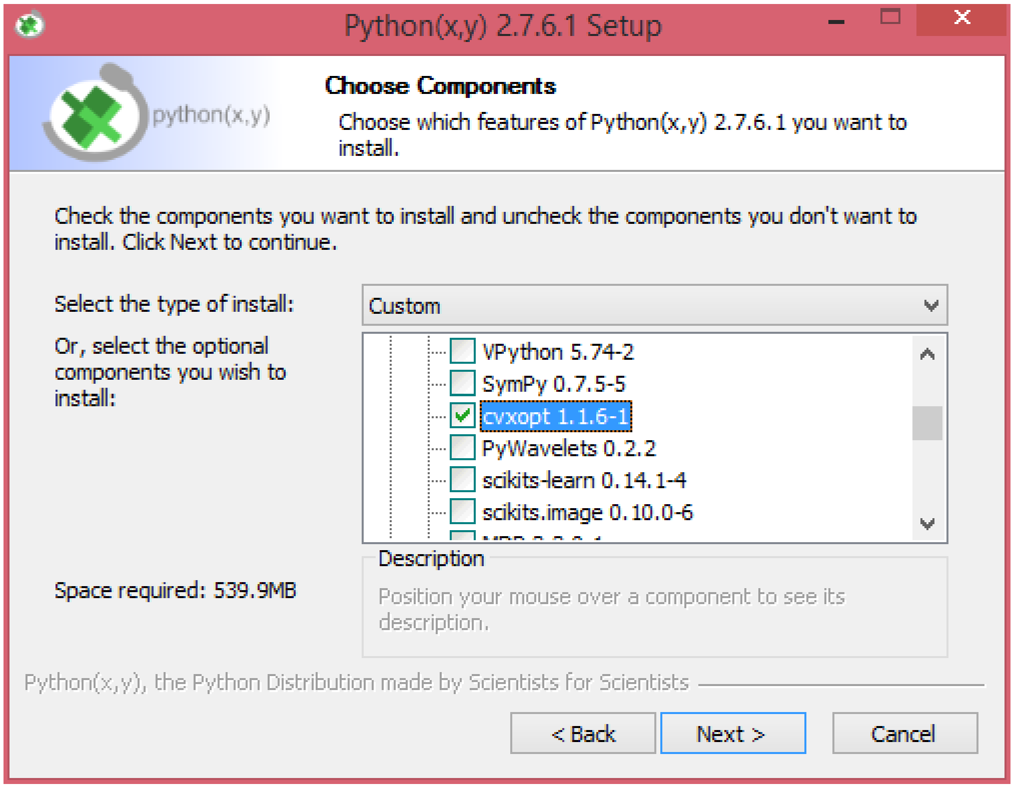
Windows

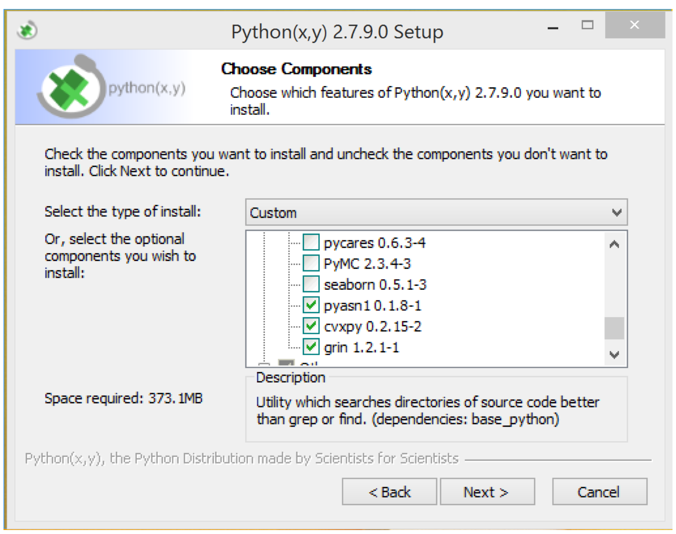
There are two ways to install CVXPY on Windows. One method uses Python(x,y), while the other uses Anaconda. Installation with Python(x,y) is less likely to have problems. Both installation methods use Python 2.

Windows with Python(x,y)

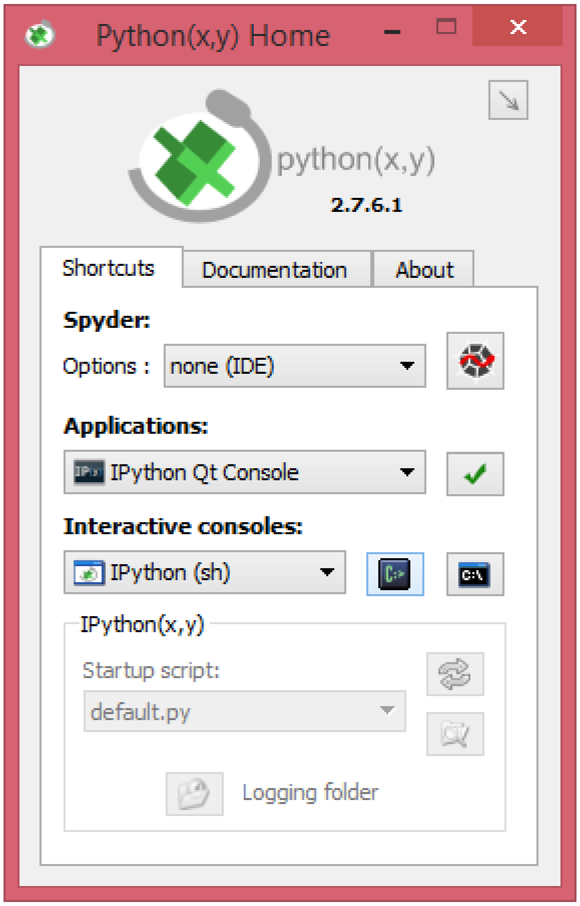
1. If you have Python installed already, it's probably a good idea to remove it first. If you uninstall Anaconda, you may need to take extra steps to remove all traces of the Anaconda install.

2. Download the latest version of Python(x,y).
3. Install Python(x,y). When prompted to select optional components, make sure to check CVXOPT and CVXPY, as shown below.





4. To test the CVXPY installation, open Python(x,y) and launch the interactive console (highlighted button in the picture). This will bring up a console.



5. From the console, run `nosetests cvxpy`. If all but one of the tests pass, your installation was successful.

Windows with Anaconda

1. Download and install the latest version of Anaconda. You must use the Python 2 version.
2. Download the Visual Studio C++ compiler for Python.
3. Install SCS from the Anaconda prompt by running the following command:

```
conda install -c https://conda.anaconda.org/omnia scs
```

4. Install CVXPY from the Anaconda prompt by running the following command:

```
pip install cvxpy
```

5. From the console, run `nosetests cvxpy`. If all the tests pass, your installation was successful.

Other Platforms

The CVXPY installation process on other platforms is less automated and less well tested. Check this page for instructions for your platform.

Install from source

CVXPY has the following dependencies:

- Python 2.7 or Python 3.4
- setuptools >= 1.4
- toolz
- six
- fastcache
- multiprocess
- ECOS >= 2
- SCS >= 1.1.3
- NumPy >= 1.8
- SciPy >= 0.15
- CVXcanon >= 0.0.22

To test the CVXPY installation, you additionally need Nose.

CVXPY automatically installs ECOS, SCS, toolz, six, fastcache, and multiprocess. NumPy and SciPy will need to be installed manually. You may also wish to install Swig to build CVXcanon from source. Once you've installed NumPy and SciPy, installing CVXPY from source is simple:

1. Clone the CVXPY git repository.
2. Navigate to the top-level of the cloned directory and run

```
python setup.py install
```

Install with CVXOPT support

CVXPY supports the CVXOPT solver. Simply install CVXOPT by running `pip install cvxopt`. If you use Anaconda you will need to run `conda install nomkl` first.

Install with Elemental support

CVXPY supports the Elemental solver. Simply install Elemental such that you can `import El` in Python. See the Elemental website for installation instructions.

Install with GUROBI support

CVXPY supports the GUROBI solver. Simply install GUROBI such that you can `import gurobipy` in Python. See the GUROBI website for installation instructions.

Install with MOSEK support

CVXPY supports the MOSEK solver. Simply install MOSEK such that you can `import mosek` in Python. See the MOSEK website for installation instructions.

Install with GLPK support

CVXPY supports the GLPK solver, but only if CVXOPT is installed with GLPK bindings. To install CVXPY and its dependencies with GLPK support, follow these instructions:

1. Install GLPK. We recommend either installing the latest GLPK from source or using a package manager such as apt-get on Ubuntu and homebrew on OS X.
2. Install CVXOPT with GLPK bindings.

```
CVXOPT_BUILD_GLPK=1
CVXOPT_GLPK_LIB_DIR=/path/to/glpk-X.X/lib
CVXOPT_GLPK_INC_DIR=/path/to/glpk-X.X/include
pip install cvxopt
```

3. Follow the standard installation procedure to install CVXPY and its remaining dependencies.

Install with Cbc (Clp, Cgl) support

CVXPY supports the Cbc solver (which includes Clp and Cgl) with the help of cylp. Simply install cylp (you will need the Cbc sources which includes Cgl) such you can import this library in Python. See the cylp documentation for installation instructions.

CHAPTER 2

Tutorial

What is CVXPY?

CVXPY is a Python-embedded modeling language for convex optimization problems. It automatically transforms the problem into standard form, calls a solver, and unpacks the results.

The code below solves a simple optimization problem in CVXPY:

```
from cvxpy import *

# Create two scalar optimization variables.
x = Variable()
y = Variable()

# Create two constraints.
constraints = [x + y == 1,
               x - y >= 1]

# Form objective.
obj = Minimize(square(x - y))

# Form and solve problem.
prob = Problem(obj, constraints)
prob.solve()  # Returns the optimal value.
print "status:", prob.status
print "optimal value", prob.value
print "optimal var", x.value, y.value
```

```
status: optimal
optimal value 0.999999999761
optimal var 1.00000000001 -1.19961841702e-11
```

The status, which was assigned a value "optimal" by the solve method, tells us the problem was solved successfully. The optimal value (basically 1 here) is the minimum value of the objective over all choices of variables that satisfy the constraints. The last thing printed gives values of x and y (basically 1 and 0 respectively) that achieve the optimal objective.

`prob.solve()` returns the optimal value and updates `prob.status`, `prob.value`, and the `value` field of all the variables in the problem.

Namespace

The Python examples in this tutorial import CVXPY using the syntax `from cvxpy import *`. This is done to make the examples simpler and more concise. But for production code you should always import CVXPY as a namespace. For example, `import cvxpy as cvx`. Here's the code from the previous section with CVXPY imported as a namespace.

```
import cvxpy as cvx

# Create two scalar optimization variables.
x = cvx.Variable()
y = cvx.Variable()

# Create two constraints.
constraints = [x + y == 1,
               x - y >= 1]

# Form objective.
obj = cvx.Minimize(cvx.square(x - y))

# Form and solve problem.
prob = cvx.Problem(obj, constraints)
prob.solve()  # Returns the optimal value.
print "status:", prob.status
print "optimal value", prob.value
print "optimal var", x.value, y.value
```

Nonetheless we have designed CVXPY so that using `from cvxpy import *` is generally safe for short scripts. The biggest catch is that the built-in `max` and `min` cannot be used on CVXPY expressions. Instead use the *CVXPY functions* `max_elemwise`, `max_entries`, `min_elemwise`, or `min_entries`.

The built-in `sum` can be used on lists of CVXPY expressions to add all the list elements together. Use the *CVXPY function* `sum_entries` to sum the entries of a single CVXPY matrix or vector expression.

Changing the problem

After you create a problem object, you can still modify the objective and constraints.

```
# Replace the objective.
prob.objective = Maximize(x + y)
print "optimal value", prob.solve()

# Replace the constraint (x + y == 1).
prob.constraints[0] = (x + y <= 3)
print "optimal value", prob.solve()
```

```
optimal value 1.0
optimal value 3.00000000006
```

Infeasible and unbounded problems

If a problem is infeasible or unbounded, the status field will be set to "infeasible" or "unbounded", respectively. The value fields of the problem variables are not updated.

```
from cvxpy import *

x = Variable()

# An infeasible problem.
prob = Problem(Minimize(x), [x >= 1, x <= 0])
prob.solve()
print "status:", prob.status
print "optimal value", prob.value

# An unbounded problem.
prob = Problem(Minimize(x))
prob.solve()
print "status:", prob.status
print "optimal value", prob.value
```

```
status: infeasible
optimal value inf
status: unbounded
optimal value -inf
```

Notice that for a minimization problem the optimal value is `inf` if infeasible and `-inf` if unbounded. For maximization problems the opposite is true.

Other problem statuses

If the solver called by CVXPY solves the problem but to a lower accuracy than desired, the problem status indicates the lower accuracy achieved. The statuses indicating lower accuracy are

- "optimal_inaccurate"
- "unbounded_inaccurate"
- "infeasible_inaccurate"

The problem variables are updated as usual for the type of solution found (i.e., optimal, unbounded, or infeasible).

If the solver completely fails to solve the problem, CVXPY throws a `SolverError` exception. If this happens you should try using other solvers. See the discussion of *Choosing a solver* for details.

CVXPY provides the following constants as aliases for the different status strings:

- `OPTIMAL`
- `INFEASIBLE`
- `UNBOUNDED`
- `OPTIMAL_INACCURATE`
- `INFEASIBLE_INACCURATE`
- `UNBOUNDED_INACCURATE`

For example, to test if a problem was solved successfully, you would use

```
prob.status == OPTIMAL
```

Vectors and matrices

Variables can be scalars, vectors, or matrices.

```
# A scalar variable.
a = Variable()

# Column vector variable of length 5.
x = Variable(5)

# Matrix variable with 4 rows and 7 columns.
A = Variable(4, 7)
```

You can use your numeric library of choice to construct matrix and vector constants. For instance, if `x` is a CVXPY Variable in the expression `A*x + b`, `A` and `b` could be Numpy ndarrays, SciPy sparse matrices, etc. `A` and `b` could even be different types.

Currently the following types may be used as constants:

- Numpy ndarrays
- Numpy matrices
- SciPy sparse matrices

Here's an example of a CVXPY problem with vectors and matrices:

```
# Solves a bounded least-squares problem.

from cvxpy import *
import numpy

# Problem data.
m = 10
n = 5
numpy.random.seed(1)
A = numpy.random.randn(m, n)
b = numpy.random.randn(m, 1)

# Construct the problem.
x = Variable(n)
objective = Minimize(sum_entries(square(A*x - b)))
constraints = [0 <= x, x <= 1]
prob = Problem(objective, constraints)

print "Optimal value", prob.solve()
print "Optimal var"
print x.value # A numpy matrix.
```

```
Optimal value 4.14133859146
Optimal var
[[ -2.76479783e-10]
 [  3.59742090e-10]
 [  1.34633378e-01]
 [  1.24978611e-01]
 [ -3.67846924e-11]]
```

Constraints

As shown in the example code, you can use `==`, `<=`, and `>=` to construct constraints in CVXPY. Equality and inequality constraints are elementwise, whether they involve scalars, vectors, or matrices. For example, together the constraints `0 <= x` and `x <= 1` mean that every entry of `x` is between 0 and 1.

If you want matrix inequalities that represent semi-definite cone constraints, see *Semidefinite matrices*. The section explains how to express a semi-definite cone inequality.

You cannot construct inequalities with `<` and `>`. Strict inequalities don't make sense in a real world setting. Also, you cannot chain constraints together, e.g., `0 <= x <= 1` or `x == y == 2`. The Python interpreter treats chained constraints in such a way that CVXPY cannot capture them. CVXPY will raise an exception if you write a chained constraint.

Parameters

Parameters are symbolic representations of constants. The purpose of parameters is to change the value of a constant in a problem without reconstructing the entire problem.

Parameters can be vectors or matrices, just like variables. When you create a parameter you have the option of specifying the sign of the parameter's entries (positive, negative, or unknown). The sign is unknown by default. The sign is used in *Disciplined Convex Programming*. Parameters can be assigned a constant value any time after they are created. The constant value must have the same dimensions and sign as those specified when the parameter was created.

```
# Positive scalar parameter.
m = Parameter(sign="positive")

# Column vector parameter with unknown sign (by default).
c = Parameter(5)

# Matrix parameter with negative entries.
G = Parameter(4, 7, sign="negative")

# Assigns a constant value to G.
G.value = -numpy.ones((4, 7))
```

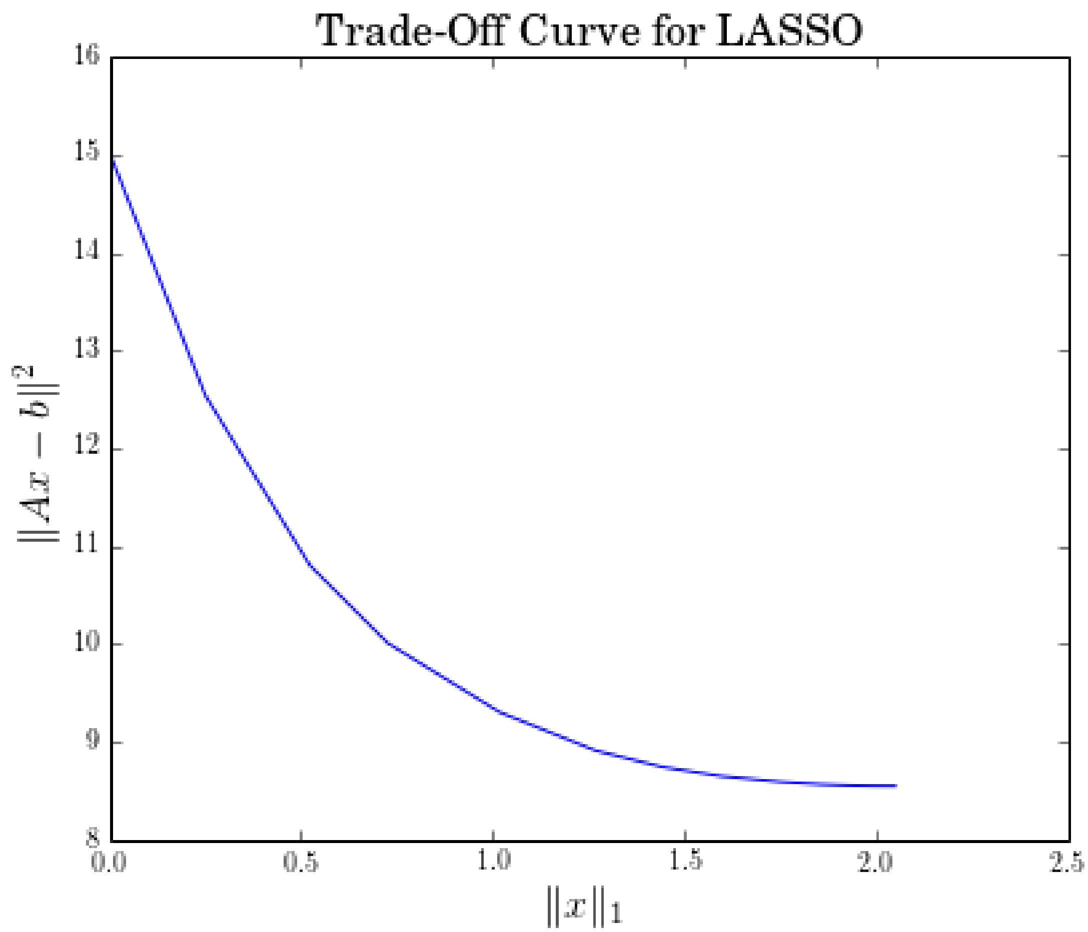
You can initialize a parameter with a value. The following code segments are equivalent:

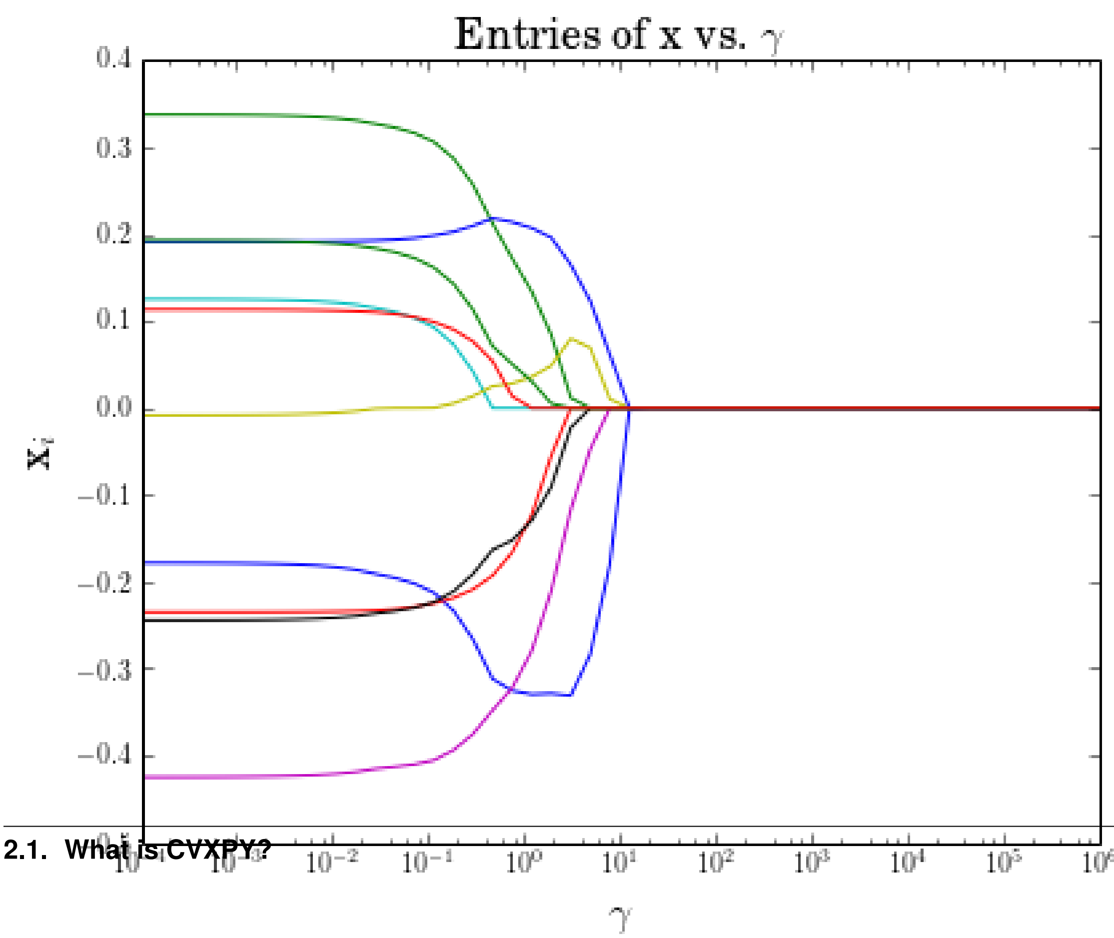
```
# Create parameter, then assign value.
rho = Parameter(sign="positive")
rho.value = 2

# Initialize parameter with a value.
rho = Parameter(sign="positive", value=2)
```

Computing trade-off curves is a common use of parameters. The example below computes a trade-off curve for a LASSO problem.

```
from cvxpy import *
import numpy
import matplotlib.pyplot as plt

# Problem data.
```

```
n = 15
m = 10
numpy.random.seed(1)
A = numpy.random.randn(n, m)
b = numpy.random.randn(n, 1)
# gamma must be positive due to DCP rules.
gamma = Parameter(sign="positive")

# Construct the problem.
x = Variable(m)
error = sum_squares(A*x - b)
obj = Minimize(error + gamma*norm(x, 1))
prob = Problem(obj)

# Construct a trade-off curve of ||Ax-b||^2 vs. ||x||_1
sq_penalty = []
l1_penalty = []
x_values = []
gamma_vals = numpy.logspace(-4, 6)
for val in gamma_vals:
    gamma.value = val
    prob.solve()
    # Use expr.value to get the numerical value of
    # an expression in the problem.
    sq_penalty.append(error.value)
    l1_penalty.append(norm(x, 1).value)
    x_values.append(x.value)

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.figure(figsize=(6,10))

# Plot trade-off curve.
plt.subplot(211)
plt.plot(l1_penalty, sq_penalty)
plt.xlabel(r'\|x\|_1', fontsize=16)
plt.ylabel(r'\|Ax-b\|^2', fontsize=16)
plt.title('Trade-Off Curve for LASSO', fontsize=16)

# Plot entries of x vs. gamma.
plt.subplot(212)
for i in range(m):
    plt.plot(gamma_vals, [xi[i,0] for xi in x_values])
plt.xlabel(r'\gamma', fontsize=16)
plt.ylabel(r'x_{i}', fontsize=16)
plt.xscale('log')
plt.title(r'\text{Entries of x vs. }\gamma', fontsize=16)

plt.tight_layout()
plt.show()
```


Trade-Off Curve for LASSO
$\|Ax - b\|^2$
$\|x\|_1$


Entries of x vs. γ
$\mathbf{x}_i$
γ

Trade-off curves can easily be computed in parallel. The code below computes in parallel the optimal x for each γ in the LASSO problem above.

```
from multiprocessing import Pool

# Assign a value to gamma and find the optimal x.
def get_x(gamma_value):
    gamma.value = gamma_value
    result = prob.solve()
    return x.value

# Parallel computation (set to 1 process here).
pool = Pool(processes = 1)
x_values = pool.map(get_x, gamma_vals)
```

Disciplined Convex Programming

Disciplined convex programming (DCP) is a system for constructing mathematical expressions with known curvature from a given library of base functions. CVXPY uses DCP to ensure that the specified optimization problems are convex.

This section of the tutorial explains the rules of DCP and how they are applied by CVXPY.

Visit dcp.stanford.edu for a more interactive introduction to DCP.

Expressions

Expressions in CVXPY are formed from variables, parameters, numerical constants such as Python floats and Numpy matrices, the standard arithmetic operators `+, -, *, /`, and a library of *functions*. Here are some examples of CVXPY expressions:

```
from cvxpy import *

# Create variables and parameters.
x, y = Variable(), Variable()
a, b = Parameter(), Parameter()

# Examples of CVXPY expressions.
3.69 + b/3
x - 4*a
sqrt(x) - min_elemwise(y, x - a)
max_elemwise(2.66 - sqrt(y), square(x + 2*y))
```

Expressions can be scalars, vectors, or matrices. The dimensions of an expression are stored as `expr.size`. CVXPY will raise an exception if an expression is used in a way that doesn't make sense given its dimensions, for example adding matrices of different size.

```
import numpy

X = Variable(5, 4)
A = numpy.ones((3, 5))

# Use expr.size to get the dimensions.
print "dimensions of X:", X.size
print "dimensions of sum_entries(X):", sum_entries(X).size
```

```
print "dimensions of A*X:", (A*X).size

# ValueError raised for invalid dimensions.
try:
    A + X
except ValueError, e:
    print e
```

```
dimensions of X: (5, 4)
dimensions of sum_entries(X): (1, 1)
dimensions of A*X: (3, 4)
Incompatible dimensions (3, 5) (5, 4)
```

CVXPY uses DCP analysis to determine the sign and curvature of each expression.

Sign

Each (sub)expression is flagged as *positive* (non-negative), *negative* (non-positive), *zero*, or *unknown*.

The signs of larger expressions are determined from the signs of their subexpressions. For example, the sign of the expression expr1*expr2 is

- Zero if either expression has sign zero.
- Positive if expr1 and expr2 have the same (known) sign.
- Negative if expr1 and expr2 have opposite (known) signs.
- Unknown if either expression has unknown sign.

The sign given to an expression is always correct. But DCP sign analysis may flag an expression as unknown sign when the sign could be figured out through more complex analysis. For instance, `x*x` is positive but has unknown sign by the rules above.

CVXPY determines the sign of constants by looking at their value. For scalar constants, this is straightforward. Vector and matrix constants with all positive (negative) entries are marked as positive (negative). Vector and matrix constants with both positive and negative entries are marked as unknown sign.

The sign of an expression is stored as `expr.sign`:

```
x = Variable()
a = Parameter(sign="negative")
c = numpy.array([1, -1])

print "sign of x:", x.sign
print "sign of a:", a.sign
print "sign of square(x):", square(x).sign
print "sign of c*a:", (c*a).sign
```

```
sign of x: UNKNOWN
sign of a: NEGATIVE
sign of square(x): POSITIVE
sign of c*a: UNKNOWN
```

Curvature

Each (sub)expression is flagged as one of the following curvatures (with respect to its variables)

Curvature	Meaning
constant	$f(x)$ independent of x
affine	$f(\theta x + (1-\theta)y) = \theta f(x) + (1-\theta)f(y), \ \forall x, \ y, \ \theta \in [0,1]$
convex	$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y), \ \forall x, \ y, \ \theta \in [0,1]$
concave	$f(\theta x + (1-\theta)y) \geq \theta f(x) + (1-\theta)f(y), \ \forall x, \ y, \ \theta \in [0,1]$
unknown	DCP analysis cannot determine the curvature

using the curvature rules given below. As with sign analysis, the conclusion is always correct, but the simple analysis can flag expressions as unknown even when they are convex or concave. Note that any constant expression is also affine, and any affine expression is convex and concave.

Curvature rules

DCP analysis is based on applying a general composition theorem from convex analysis to each (sub)expression.

$f(expr_1, expr_2, ..., expr_n)$ is convex if f is a convex function and for each $expr_i$ one of the following conditions holds:

- f is increasing in argument i and $expr_i$ is convex.
- f is decreasing in argument i and $expr_i$ is concave.
- $expr_i$ is affine or constant.

$f(expr_1, expr_2, ..., expr_n)$ is concave if f is a concave function and for each $expr_i$ one of the following conditions holds:

- f is increasing in argument i and $expr_i$ is concave.
- f is decreasing in argument i and $expr_i$ is convex.
- $expr_i$ is affine or constant.

$f(expr_1, expr_2, ..., expr_n)$ is affine if f is an affine function and each $expr_i$ is affine.

If none of the three rules apply, the expression $f(expr_1, expr_2, ..., expr_n)$ is marked as having unknown curvature.

Whether a function is increasing or decreasing in an argument may depend on the sign of the argument. For instance, `square` is increasing for positive arguments and decreasing for negative arguments.

The curvature of an expression is stored as `expr.curvature`:

```
x = Variable()
a = Parameter(sign="positive")

print "curvature of x:", x.curvature
print "curvature of a:", a.curvature
print "curvature of square(x):", square(x).curvature
print "curvature of sqrt(x):", sqrt(x).curvature
```

```
curvature of x: AFFINE
curvature of a: CONSTANT
curvature of square(x): CONVEX
curvature of sqrt(x): CONCAVE
```

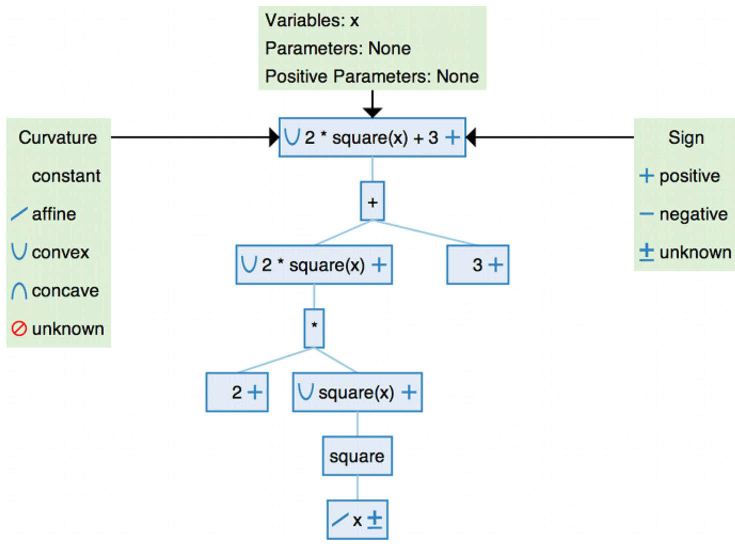
Infix operators

The infix operators `+, -, *, /` are treated exactly like functions. The infix operators + and - are affine, so the rules above are used to flag the curvature. For example, `expr1 + expr2` is flagged as convex if `expr1` and `expr2` are convex.

`expr1*expr2` is allowed only when one of the expressions is constant. If both expressions are non-constant, CVXPY will raise an exception. `expr1/expr2` is allowed only when `expr2` is a scalar constant. The curvature rules above apply. For example, `expr1/expr2` is convex when `expr1` is concave and `expr2` is negative and constant.

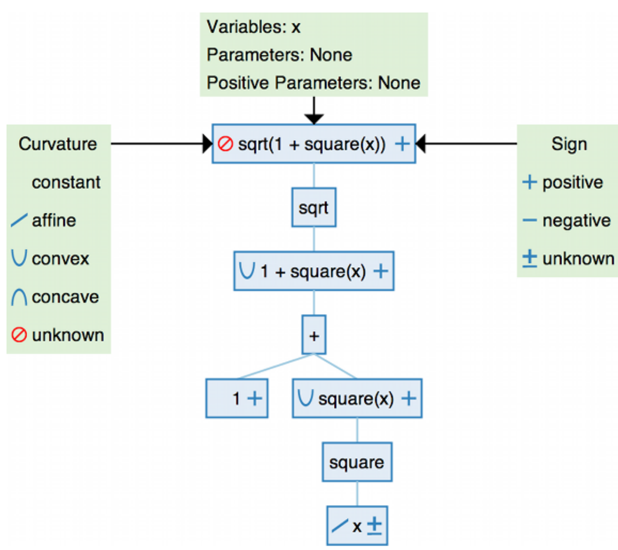
Example 1

DCP analysis breaks expressions down into subexpressions. The tree visualization below shows how this works for the expression `2*square(x) + 3`. Each subexpression is shown in a blue box. We mark its curvature on the left and its sign on the right.



Example 2

We'll walk through the application of the DCP rules to the expression `sqrt(1 + square(x))`.



The variable x has affine curvature and unknown sign. The square function is convex and non-monotone for arguments of unknown sign. It can take the affine expression x as an argument; the result square(x) is convex.

The arithmetic operator + is affine and increasing, so the composition 1 + square(x) is convex by the curvature rule for convex functions. The function sqrt is concave and increasing, which means it can only take a concave argument. Since 1 + square(x) is convex, sqrt(1 + square(x)) violates the DCP rules and cannot be verified as convex.

In fact, sqrt(1 + square(x)) is a convex function of x, but the DCP rules are not able to verify convexity. If the expression is written as norm(vstack(1, x), 2), the L2 norm of the vector $[1, x]$, which has the same value as sqrt(1 + square(x)), then it will be certified as convex using the DCP rules.

```
print "sqrt(1 + square(x)) curvature:",
print sqrt(1 + square(x)).curvature
print "norm(vstack(1, x), 2) curvature:",
print norm(vstack(1, x), 2).curvature
```

```
sqrt(1 + square(x)) curvature: UNKNOWN
norm(vstack(1, x), 2) curvature: CONVEX
```

DCP problems

A problem is constructed from an objective and a list of constraints. If a problem follows the DCP rules, it is guaranteed to be convex and solvable by CVXPY. The DCP rules require that the problem objective have one of two forms:

- Minimize(convex)
- Maximize(concave)

The only valid constraints under the DCP rules are

- affine == affine
- convex <= concave
- concave >= convex

You can check that a problem, constraint, or objective satisfies the DCP rules by calling object.is_dcp(). Here are some examples of DCP and non-DCP problems:

```
x = Variable()
y = Variable()

# DCP problems.
prob1 = Problem(Minimize(square(x - y)), [x + y >= 0])
prob2 = Problem(Maximize(sqrt(x - y)),
                [2*x - 3 == y,
                 square(x) <= 2])

print "prob1 is DCP:", prob1.is_dcp()
print "prob2 is DCP:", prob2.is_dcp()

# Non-DCP problems.

# A non-DCP objective.
prob3 = Problem(Maximize(square(x)))

print "prob3 is DCP:", prob3.is_dcp()
print "Maximize(square(x)) is DCP:", Maximize(square(x)).is_dcp()
```

```
# A non-DCP constraint.
prob4 = Problem(Minimize(square(x)), [sqrt(x) <= 2])

print "prob4 is DCP:", prob4.is_dcp()
print "sqrt(x) <= 2 is DCP:", (sqrt(x) <= 2).is_dcp()
```

```
prob1 is DCP: True
prob2 is DCP: True
prob3 is DCP: False
Maximize(square(x)) is DCP: False
prob4 is DCP: False
sqrt(x) <= 2 is DCP: False
```

CVXPY will raise an exception if you call `problem.solve()` on a non-DCP problem.

```
# A non-DCP problem.
prob = Problem(Minimize(sqrt(x)))

try:
    prob.solve()
except Exception as e:
    print e
```

```
Problem does not follow DCP rules.
```

Functions

This section of the tutorial describes the functions that can be applied to CVXPY expressions. CVXPY uses the function information in this section and the *DCP rules* to mark expressions with a sign and curvature.

Operators

The infix operators `+, -, *, /` are treated as functions. + and − are affine functions. `*` and `/` are affine in CVXPY because `expr1*expr2` is allowed only when one of the expressions is constant and `expr1/expr2` is allowed only when `expr2` is a scalar constant.

Indexing and slicing

All non-scalar expressions can be indexed using the syntax `expr[i, j]`. Indexing is an affine function. The syntax `expr[i]` can be used as a shorthand for `expr[i, 0]` when `expr` is a column vector. Similarly, `expr[i]` is shorthand for `expr[0, i]` when `expr` is a row vector.

Non-scalar expressions can also be sliced into using the standard Python slicing syntax. For example, `expr[i:j:k, r]` selects every kth element in column r of `expr`, starting at row i and ending at row j-1.

CVXPY supports advanced indexing using lists of indices or boolean arrays. The semantics are the same as NumPy (see NumPy advanced indexing). Any time NumPy would return a 1D array, CVXPY returns a column vector.

Transpose

The transpose of any expression can be obtained using the syntax `expr.T`. Transpose is an affine function.

Power

For any CVXPY expression `expr`, the power operator `expr**p` is equivalent to the function `power(expr, p)`.

Scalar functions

A scalar function takes one or more scalars, vectors, or matrices as arguments and returns a scalar.

Function	Meaning	Domain	Sign	Curvature	Monotonicity
geo_mean(x) geo_mean(x, p) $p \in \mathbf{R}^n_+$ $p \neq 0$	$x_1^{1/n} \cdots x_n^{1/n}$ $\left(x_1^{p_1} \cdots x_n^{p_n}\right)^{\frac{1}{\mathbf{1}^T p}}$	$x \in \mathbf{R}^n_+$	positive	concave	incr.
harmonic_mean(x)	$\frac{n}{\frac{1}{x_1} + \cdots + \frac{1}{x_n}}$	$x \in \mathbf{R}^n_+$	positive	concave	incr.
lambda_max(X)	$\lambda_{\max}(X)$	$X \in \mathbf{S}^n$	unknown	convex	None
lambda_min(X)	$\lambda_{\min}(X)$	$X \in \mathbf{S}^n$	unknown	concave	None
lambda_sum_largest(X, k) $k = 1, \ldots, n$	sum of k largest eigenvalues of X	$X \in \mathbf{S}^n$	unknown	convex	None
lambda_sum_smallest(X, k) $k = 1, \ldots, n$	sum of k smallest eigenvalues of X	$X \in \mathbf{S}^n$	unknown	concave	None
log_det(X)	$\log\left(\det(X)\right)$	$X \in \mathbf{S}^n_+$	unknown	concave	None
log_sum_exp(X)	$\log\left(\sum_{ij} e^{X_{ij}}\right)$	$X \in \mathbf{R}^{m \times n}$	unknown	convex	incr.
matrix_frac(x, P)	$x^T P^{-1} x$	$x \in \mathbf{R}^n$ $P \in \mathbf{S}^n_{++}$	positive	convex	None
max_entries(X)	$\max_{ij}\left\{X_{ij}\right\}$	$X \in \mathbf{R}^{m \times n}$	same as X	convex	incr.
min_entries(X)	$\min_{ij}\left\{X_{ij}\right\}$	$X \in \mathbf{R}^{m \times n}$	same as X	concave	incr.
mixed_norm(X, p, q)	$\left(\sum_k \left(\sum_l \lvert x_{k,l}\rvert^p\right)^{q/p}\right)^{1/q}$	$X \in \mathbf{R}^{n \times n}$	positive	convex	None
norm(x) norm(x, 2)	$\sqrt{\sum_i x_i^2}$	$X \in \mathbf{R}^n$	positive	convex	for $x_i \geq 0$ for $x_i \leq 0$
norm(X, "fro")	$\sqrt{\sum_{ij} X_{ij}^2}$	$X \in \mathbf{R}^{m \times n}$	positive	convex	for $X_{ij} \geq 0$ for $X_{ij} \leq 0$
norm(X, 1)	$\sum_{ij} \lvert X_{ij}\rvert$	$X \in \mathbf{R}^{m \times n}$	positive	convex	for $X_{ij} \geq 0$ for $X_{ij} \leq 0$
norm(X, "inf")	$\max_{ij}\{\lvert X_{ij}\rvert\}$	$X \in \mathbf{R}^{m \times n}$	positive	convex	for $X_{ij} \geq 0$ for $X_{ij} \leq 0$
norm(X, "nuc")	$\mathrm{tr}\left(\left(X^T X\right)^{1/2}\right)$	$X \in \mathbf{R}^{m \times n}$	positive	convex	None
norm(X) norm(X, 2)	$\sqrt{\lambda_{\max}\left(X^T X\right)}$	$X \in \mathbf{R}^{m \times n}$	positive	convex	None
pnorm(X, p) $p \geq 1$ or `p = 'inf'`	$\lVert X\rVert_p = \left(\sum_{ij} \lvert X_{ij}\rvert^p\right)^{1/p}$	$X \in \mathbf{R}^{m \times n}$	positive	convex	for $X_{ij} \geq 0$ for $X_{ij} \leq 0$
pnorm(X, p) $p < 1$, $p \neq 0$	$\lVert X\rVert_p = \left(\sum_{ij} X_{ij}^p\right)^{1/p}$	$X \in \mathbf{R}^{m \times n}_+$	positive	concave	incr.

Continued on next page

Table 2.1 – continued from previous page

Function	Meaning	Domain	Sign	Curvature	Monotonicity
quad_form(x, P) constant $P \in \mathbf{S}^n_+$	$x^T P x$	$x \in \mathbf{R}^n$	positive	convex	for $x_i \geq 0$ for $x_i \leq 0$
quad_form(x, P) constant $P \in \mathbf{S}^n_-$	$x^T P x$	$x \in \mathbf{R}^n$	negative	concave	for $x_i \geq 0$ for $x_i \leq 0$
quad_form(c, X) constant $c \in \mathbf{R}^n$	$c^T X c$	$X \in \mathbf{R}^{n \times n}$	depends on c, X	affine	depends on c
quad_over_lin(X, y)	$\left(\sum_{ij} X_{ij}^2\right) / y$	$x \in \mathbf{R}^n$ $y > 0$	positive	convex	for $X_{ij} \geq 0$ for $X_{ij} \leq 0$ decr. in y
sum_entries(X)	$\sum_{ij} X_{ij}$	$X \in \mathbf{R}^{m \times n}$	same as X	affine	incr.
sum_largest(X, k) $k = 1, 2, \ldots$	sum of k largest X_{ij}	$X \in \mathbf{R}^{m \times n}$	same as X	convex	incr.
sum_smallest(X, k) $k = 1, 2, \ldots$	sum of k smallest X_{ij}	$X \in \mathbf{R}^{m \times n}$	same as X	concave	incr.
sum_squares(X)	$\sum_{ij} X_{ij}^2$	$X \in \mathbf{R}^{m \times n}$	positive	convex	for $X_{ij} \geq 0$ for $X_{ij} \leq 0$
trace(X)	$\mathrm{tr}\left(X\right)$	$X \in \mathbf{R}^{n \times n}$	same as X	affine	incr.
tv(x)	$\sum_i \lvert x_{i+1} - x_i \rvert$	$x \in \mathbf{R}^n$	positive	convex	None
tv(X)	$\sum_{ij} \left\| \begin{bmatrix} X_{i+1,j} - X_{ij} \\ X_{i,j+1} - X_{ij} \end{bmatrix} \right\|_2$	$X \in \mathbf{R}^{m \times n}$	positive	convex	None
tv(X1,...,Xk)	$\sum_{ij} \left\| \begin{bmatrix} X^{(1)}_{i+1,j} - X^{(1)}_{ij} \\ X^{(1)}_{i,j+1} - X^{(1)}_{ij} \\ \vdots \\ X^{(k)}_{i+1,j} - X^{(k)}_{ij} \\ X^{(k)}_{i,j+1} - X^{(k)}_{ij} \end{bmatrix} \right\|_2$	$X^{(i)} \in \mathbf{R}^{m \times n}$	positive	convex	None

Clarifications

The domain $\mathbf{S}^n$ refers to the set of symmetric matrices. The domains $\mathbf{S}^n_+$ and $\mathbf{S}^n_-$ refer to the set of positive semi-definite and negative semi-definite matrices, respectively. Similarly, $\mathbf{S}^n_{++}$ and $\mathbf{S}^n_{--}$ refer to the set of positive definite and negative definite matrices, respectively.

For a vector expression `x`, `norm(x)` and `norm(x, 2)` give the Euclidean norm. For a matrix expression `X`, however, `norm(X)` and `norm(X, 2)` give the spectral norm.

The function `norm(X, "fro")` is called the Frobenius norm and `norm(X, "nuc")` the nuclear norm. The nuclear norm can also be defined as the sum of `X`'s singular values.

The functions `max_entries` and `min_entries` give the largest and smallest entry, respectively, in a single expression. These functions should not be confused with `max_elemwise` and `min_elemwise` (see *Elementwise functions*). Use `max_elemwise` and `min_elemwise` to find the max or min of a list of scalar expressions.

The function `sum_entries` sums all the entries in a single expression. The built-in Python `sum` should be used to add together a list of expressions. For example, the following code sums a list of three expressions:

```
expr_list = [expr1, expr2, expr3]
expr_sum = sum(expr_list)
```

Functions along an axis

The functions `sum_entries`, `norm`, `max_entries`, and `min_entries` can be applied along an axis. Given an `m` by `n` expression `expr`, the syntax `func(expr, axis=0)` applies `func` to each column, returning a 1 by `n` expression. The syntax `func(expr, axis=1)` applies `func` to each row, returning an `m` by 1 expression. For example, the following code sums along the columns and rows of a matrix variable:

```
X = Variable(5, 4)
col_sums = sum_entries(X, axis=0) # Has size (1, 4)
row_sums = sum_entries(X, axis=1) # Has size (5, 1)
```

Elementwise functions

These functions operate on each element of their arguments. For example, if `X` is a 5 by 4 matrix variable, then `abs(X)` is a 5 by 4 matrix expression. `abs(X)[1, 2]` is equivalent to `abs(X[1, 2])`.

Elementwise functions that take multiple arguments, such as `max_elemwise` and `mul_elemwise`, operate on the corresponding elements of each argument. For example, if `X` and `Y` are both 3 by 3 matrix variables, then `max_elemwise(X, Y)` is a 3 by 3 matrix expression. `max_elemwise(X, Y)[2, 0]` is equivalent to `max_elemwise(X[2, 0], Y[2, 0])`. This means all arguments must have the same dimensions or be scalars, which are promoted.

Function	Meaning	Domain	Sign	Curvature	Monotonicity
abs(x)	$\lvert x\rvert$	$x \in \mathbf{R}$	positive	convex	for $x \geq 0$ for $x \leq 0$
entr(x)	$-x \log(x)$	$x > 0$	unknown	concave	None
exp(x)	e^x	$x \in \mathbf{R}$	positive	convex	incr.
huber(x, M=1) $M \geq 0$	$\begin{cases} x^2 & \lvert x\rvert \leq M \\ 2M\lvert x\rvert - M^2 & \lvert x\rvert > M \end{cases}$	$x \in \mathbf{R}$	positive	convex	for $x \geq 0$ for $x \leq 0$
inv_pos(x)	$1/x$	$x > 0$	positive	convex	decr.
kl_div(x, y)	$x \log(x/y) - x + y$	$x > 0$ $y > 0$	positive	convex	None
log(x)	$\log(x)$	$x > 0$	unknown	concave	incr.
log1p(x)	$\log(x+1)$	$x > -1$	same as x	concave	incr.
logistic(x)	$\log(1+e^x)$	$x \in \mathbf{R}$	positive	convex	incr.
max_elemwise(x1, ..., xk)	$\max\{x_1, \ldots, x_k\}$	$x_i \in \mathbf{R}$	$\max(\mathrm{sign}(x_1))$	convex	incr.
min_elemwise(x1, ..., xk)	$\min\{x_1, \ldots, x_k\}$	$x_i \in \mathbf{R}$	$\min(\mathrm{sign}(x_1))$	concave	incr.
mul_elemwise(c, x) $c \in \mathbf{R}$	c*x	$x \in \mathbf{R}$	$\mathrm{sign}(cx)$	affine	depends on c
neg(x)	$\max\{-x, 0\}$	$x \in \mathbf{R}$	positive	convex	decr.
pos(x)	$\max\{x, 0\}$	$x \in \mathbf{R}$	positive	convex	incr.
power(x, 0)	1	$x \in \mathbf{R}$	positive	constant	
power(x, 1)	x	$x \in \mathbf{R}$	same as x	affine	incr.
power(x, p) $p = 2, 4, 8, \ldots$	x^p	$x \in \mathbf{R}$	positive	convex	for $x \geq 0$ for $x \leq 0$
power(x, p) $p < 0$	x^p	$x > 0$	positive	convex	decr.
power(x, p) $0 < p < 1$	x^p	$x \geq 0$	positive	concave	incr.
power(x, p) $p > 1,\ p \neq 2, 4, 8, \ldots$	x^p	$x \geq 0$	positive	convex	incr.
scalene(x, alpha, beta) alpha ≥ 0 beta ≥ 0	$\alpha \mathrm{pos}(x) + \beta \mathrm{neg}(x)$	$x \in \mathbf{R}$	positive	convex	for $x \geq 0$ for $x \leq 0$
sqrt(x)	$\sqrt{x}$	$x \geq 0$	positive	concave	incr.
square(x)	x^2	$x \in \mathbf{R}$	positive	convex	for $x \geq 0$ for $x \leq 0$

Vector/matrix functions

A vector/matrix function takes one or more scalars, vectors, or matrices as arguments and returns a vector or matrix.

Function	Meaning	Domain	Sign	Curvature	Monotonicity
bmat([[X11, ..., X1q], ..., [Xp1, ..., Xpq]])	$\begin{bmatrix} X^{(1,1)} & \cdots & X^{(1,q)} \\ \vdots & & \vdots \\ X^{(p,1)} & \cdots & X^{(p,q)} \end{bmatrix}$	$X^{(i,j)} \in \mathbf{R}^{m_i \times n_j}$	$\mathrm{sign}\left(\sum_{ij} X^{(i,j)}_{11}\right)$	affine	incr.
conv(c, x) $c \in \mathbf{R}^m$	$c * x$	$x \in \mathbf{R}^n$	$\mathrm{sign}\left(c_1 x_1\right)$	affine	depends on c
cumsum(X, axis=0)	cumulative sum along given axis.	$X \in \mathbf{R}^{m \times n}$	same as X	affine	incr.
diag(x)	$\begin{bmatrix} x_1 & & \\ & \ddots & \\ & & x_n \end{bmatrix}$	$x \in \mathbf{R}^n$	same as x	affine	incr.
diag(X)	$\begin{bmatrix} X_{11} \\ \vdots \\ X_{nn} \end{bmatrix}$	$X \in \mathbf{R}^{n \times n}$	same as X	affine	incr.
diff(X, k=1, axis=0) $k \in 0, 1, 2, \ldots$	kth order differences along given axis	$X \in \mathbf{R}^{m \times n}$	same as X	affine	incr.
hstack(X1, ..., Xk)	$\left[X^{(1)} \cdots X^{(k)}\right]$	$X^{(i)} \in \mathbf{R}^{m \times n_i}$	$\mathrm{sign}\left(\sum_i X^{(i)}_{11}\right)$	affine	incr.
kron(C, X) $C \in \mathbf{R}^{p \times q}$	$\begin{bmatrix} C_{11}X & \cdots & C_{1q}X \\ \vdots & & \vdots \\ C_{p1}X & \cdots & C_{pq}X \end{bmatrix}$	$X \in \mathbf{R}^{m \times n}$	$\mathrm{sign}\left(C_{11} X_{11}\right)$	affine	depends on C
reshape(X, n', m')	$X' \in \mathbf{R}^{m' \times n'}$	$X \in \mathbf{R}^{m \times n}$ $m'n' = mn$	same as X	affine	incr.
vec(X)	$x' \in \mathbf{R}^{mn}$	$X \in \mathbf{R}^{m \times n}$	same as X	affine	incr.
vstack(X1, ..., Xk)	$\begin{bmatrix} X^{(1)} \\ \vdots \\ X^{(k)} \end{bmatrix}$	$X^{(i)} \in \mathbf{R}^{m_i \times n}$	$\mathrm{sign}\left(\sum_i X^{(i)}_{11}\right)$	affine	incr.

Clarifications

The input to `bmat` is a list of lists of CVXPY expressions. It constructs a block matrix. The elements of each inner list are stacked horizontally and then the resulting block matrices are stacked vertically.

The output y of `conv(c, x)` has size $n + m - 1$ and is defined as $y[k] = \sum_{j=0}^{k} c[j]x[k-j]$.

The output x' of `vec(X)` is the matrix X flattened in column-major order into a vector. Formally, $x'_i = X_{i \bmod m, \lfloor i/m \rfloor}$.

The output X' of `reshape(X, m', n')` is the matrix X cast into an $m' \times n'$ matrix. The entries are taken from X in column-major order and stored in X' in column-major order. Formally, $X'_{ij} = \mathbf{vec}(X)_{m'j+i}$.

Advanced Features

This section of the tutorial covers features of CVXPY intended for users with advanced knowledge of convex optimization. We recommend Convex Optimization by Boyd and Vandenberghe as a reference for any terms you are unfamiliar with.

Dual variables

You can use CVXPY to find the optimal dual variables for a problem. When you call `prob.solve()` each dual variable in the solution is stored in the `dual_value` field of the constraint it corresponds to.

```
from cvxpy import *

# Create two scalar optimization variables.
x = Variable()
y = Variable()

# Create two constraints.
constraints = [x + y == 1,
               x - y >= 1]

# Form objective.
obj = Minimize(square(x - y))

# Form and solve problem.
prob = Problem(obj, constraints)
prob.solve()

# The optimal dual variable (Lagrange multiplier) for
# a constraint is stored in constraint.dual_value.
print "optimal (x + y == 1) dual variable", constraints[0].dual_value
print "optimal (x - y >= 1) dual variable", constraints[1].dual_value
print "x - y value:", (x - y).value
```

```
optimal (x + y == 1) dual variable 6.47610300459e-18
optimal (x - y >= 1) dual variable 2.00025244976
x - y value: 0.999999986374
```

The dual variable for `x - y >= 1` is 2. By complementarity this implies that `x - y` is 1, which we can see is true. The fact that the dual variable is non-zero also tells us that if we tighten `x - y >= 1`, (i.e., increase the right-hand side), the optimal value of the problem will increase.

Semidefinite matrices

Many convex optimization problems involve constraining matrices to be positive or negative semidefinite (e.g., SDPs). You can do this in CVXPY in two ways. The first way is to use `Semidef(n)` to create an `n` by `n` variable constrained to be symmetric and positive semidefinite. For example,

```
# Creates a 100 by 100 positive semidefinite variable.
X = Semidef(100)

# You can use X anywhere you would use
# a normal CVXPY variable.
obj = Minimize(norm(X) + sum_entries(X))
```

The second way is to create a positive semidefinite cone constraint using the >> or << operator. If `X` and `Y` are `n` by `n` variables, the constraint `X >> Y` means that $z^T(X - Y)z \geq 0$, for all $z \in \mathcal{R}^n$. The constraint does not require that `X` and `Y` be symmetric. Both sides of a postive semidefinite cone constraint must be square matrices and affine.

The following code shows how to constrain matrix expressions to be positive or negative semidefinite (but not necessarily symmetric).

```
# expr1 must be positive semidefinite.
constr1 = (expr1 >> 0)

# expr2 must be negative semidefinite.
constr2 = (expr2 << 0)
```

To constrain a matrix expression to be symmetric, simply write

```
# expr must be symmetric.
constr = (expr == expr.T)
```

You can also use `Symmetric(n)` to create an `n` by `n` variable constrained to be symmetric.

Mixed-integer programs

In mixed-integer programs, certain variables are constrained to be boolean or integer valued. You can construct mixed-integer programs using the `Bool` and `Int` constructors. These take the same arguments as the `Variable` constructor, and they return a variable constrained to have only boolean or integer valued entries.

The following code shows the `Bool` and `Int` constructors in action:

```
# Creates a 10-vector constrained to have boolean valued entries.
x = Bool(10)

# expr1 must be boolean valued.
constr1 = (expr1 == x)

# Creates a 5 by 7 matrix constrained to have integer valued entries.
Z = Int(5, 7)

# expr2 must be integer valued.
constr2 = (expr2 == Z)
```

Problem arithmetic

For convenience, arithmetic operations have been overloaded for problems and objectives. Problem arithmetic is useful because it allows you to write a problem as a sum of smaller problems. The rules for adding, subtracting, and multiplying objectives are given below.

```
# Addition and subtraction.

Minimize(expr1) + Minimize(expr2) == Minimize(expr1 + expr2)

Maximize(expr1) + Maximize(expr2) == Maximize(expr1 + expr2)

Minimize(expr1) + Maximize(expr2) # Not allowed.

Minimize(expr1) - Maximize(expr2) == Minimize(expr1 - expr2)

# Multiplication (alpha is a positive scalar).

alpha*Minimize(expr) == Minimize(alpha*expr)

alpha*Maximize(expr) == Maximize(alpha*expr)
```

```
-alpha*Minimize(expr) == Maximize(-alpha*expr)

-alpha*Maximize(expr) == Minimize(-alpha*expr)
```

The rules for adding and multiplying problems are equally straightforward:

```
# Addition and subtraction.

prob1 + prob2 == Problem(prob1.objective + prob2.objective,
                         prob1.constraints + prob2.constraints)

prob1 - prob2 == Problem(prob1.objective - prob2.objective,
                         prob1.constraints + prob2.constraints)

# Multiplication (alpha is any scalar).

alpha*prob == Problem(alpha*prob.objective, prob.constraints)
```

Note that the + operator concatenates lists of constraints, since this is the default behavior for Python lists. The in-place operators +=, -=, and *= are also supported for objectives and problems and follow the same rules as above.

Solve method options

The `solve` method takes optional arguments that let you change how CVXPY solves the problem. Here is the signature for the `solve` method:

solve (*solver=None*, *verbose=False*, ***kwargs*)
Solves a DCP compliant optimization problem.

Parameters

- **solver** (*str, optional*) – The solver to use.
- **verbose** (*bool, optional*) – Overrides the default of hiding solver output.
- **kwargs** – Additional keyword arguments specifying solver specific options.

Returns The optimal value for the problem, or a string indicating why the problem could not be solved.

We will discuss the optional arguments in detail below.

Choosing a solver

CVXPY is distributed with the open source solvers ECOS, ECOS_BB, CVXOPT, and SCS. CVXPY also supports GLPK and GLPK_MI via the CVXOPT GLPK interface, CBC, MOSEK, GUROBI, and Elemental. The table below shows the types of problems the solvers can handle.

	LP	SOCP	SDP	EXP	MIP
CBC	X				X
GLPK	X				
GLPK_MI	X				X
Elemental	X	X			
ECOS	X	X		X	
ECOS_BB	X	X		X	X
GUROBI	X	X			X
MOSEK	X	X	X		
CVXOPT	X	X	X	X	
SCS	X	X	X	X	

A special solver LS is also available. It is unable to solve any of the problem types in the table above, but it recognizes and solves linearly constrained least squares problems very quickly.

Here EXP refers to problems with exponential cone constraints. The exponential cone is defined as

$$\{(x, y, z) \mid y > 0, y \exp(x/y) \leq z\} \cup \{(x, y, z) \mid x \leq 0, y = 0, z \geq 0\}.$$

You cannot specify cone constraints explicitly in CVXPY, but cone constraints are added when CVXPY converts the problem into standard form.

By default CVXPY calls the solver most specialized to the problem type. For example, ECOS is called for SOCPs. SCS and CVXOPT can both handle all problems (except mixed-integer programs). CVXOPT is preferred by default. For many problems SCS will be faster, though less accurate. ECOS_BB is called for mixed-integer LPs and SOCPs. If the problem has a quadratic objective function and equality constraints only, CVXPY will use LS.

You can change the solver called by CVXPY using the `solver` keyword argument. If the solver you choose cannot solve the problem, CVXPY will raise an exception. Here's example code solving the same problem with different solvers.

```
# Solving a problem with different solvers.
x = Variable(2)
obj = Minimize(x[0] + norm(x, 1))
constraints = [x >= 2]
prob = Problem(obj, constraints)

# Solve with ECOS.
prob.solve(solver=ECOS)
print "optimal value with ECOS:", prob.value

# Solve with ECOS_BB.
prob.solve(solver=ECOS_BB)
print "optimal value with ECOS_BB:", prob.value

# Solve with CVXOPT.
prob.solve(solver=CVXOPT)
print "optimal value with CVXOPT:", prob.value

# Solve with SCS.
prob.solve(solver=SCS)
print "optimal value with SCS:", prob.value

# Solve with GLPK.
prob.solve(solver=GLPK)
print "optimal value with GLPK:", prob.value

# Solve with GLPK_MI.
prob.solve(solver=GLPK_MI)
```

```
print "optimal value with GLPK_MI:", prob.value

# Solve with GUROBI.
prob.solve(solver=GUROBI)
print "optimal value with GUROBI:", prob.value

# Solve with MOSEK.
prob.solve(solver=MOSEK)
print "optimal value with MOSEK:", prob.value

# Solve with Elemental.
prob.solve(solver=ELEMENTAL)
print "optimal value with Elemental:", prob.value

# Solve with CBC.
prob.solve(solver=CBC)
print "optimal value with CBC:", prob.value
```

```
optimal value with ECOS: 5.99999999551
optimal value with ECOS_BB: 5.99999999551
optimal value with CVXOPT: 6.00000000512
optimal value with SCS: 6.00046055789
optimal value with GLPK: 6.0
optimal value with GLPK_MI: 6.0
optimal value with GUROBI: 6.0
optimal value with MOSEK: 6.0
optimal value with Elemental: 6.0000044085242727
optimal value with CBC: 6.0
```

Use the `installed_solvers` utility function to get a list of the solvers your installation of CVXPY supports.

```
print installed_solvers()
```

```
['CBC', 'CVXOPT', 'MOSEK', 'GLPK', 'GLPK_MI', 'ECOS_BB', 'ECOS', 'SCS', 'GUROBI',
↪'ELEMENTAL', 'LS']
```

Viewing solver output

All the solvers can print out information about their progress while solving the problem. This information can be useful in debugging a solver error. To see the output from the solvers, set `verbose=True` in the solve method.

```
# Solve with ECOS and display output.
prob.solve(solver=ECOS, verbose=True)
print "optimal value with ECOS:", prob.value
```

```
ECOS 1.0.3 - (c) A. Domahidi, Automatic Control Laboratory, ETH Zurich, 2012-2014.

It     pcost         dcost        gap      pres     dres     k/t      mu       step     IR
 0   +0.000e+00   +4.000e+00   +2e+01   2e+00    1e+00    1e+00    3e+00    N/A      1 1 -
 1   +6.451e+00   +8.125e+00   +5e+00   7e-01    5e-01    7e-01    7e-01    0.7857   1 1 1
 2   +6.788e+00   +6.839e+00   +9e-02   1e-02    8e-03    3e-02    2e-02    0.9829   1 1 1
 3   +6.828e+00   +6.829e+00   +1e-03   1e-04    8e-05    3e-04    2e-04    0.9899   1 1 1
 4   +6.828e+00   +6.828e+00   +1e-05   1e-06    8e-07    3e-06    2e-06    0.9899   2 1 1
 5   +6.828e+00   +6.828e+00   +1e-07   1e-08    8e-09    4e-08    2e-08    0.9899   2 1 1
```

```
OPTIMAL (within feastol=1.3e-08, reltol=1.5e-08, abstol=1.0e-07).
Runtime: 0.000121 seconds.

optimal value with ECOS: 6.82842708233
```

Setting solver options

The ECOS, ECOS_BB, MOSEK, CBC, CVXOPT, and SCS Python interfaces allow you to set solver options such as the maximum number of iterations. You can pass these options along through CVXPY as keyword arguments.

For example, here we tell SCS to use an indirect method for solving linear equations rather than a direct method.

```
# Solve with SCS, use sparse-indirect method.
prob.solve(solver=SCS, verbose=True, use_indirect=True)
print "optimal value with SCS:", prob.value
```

```
----------------------------------------------------------------------------
    SCS v1.0.5 - Splitting Conic Solver
    (c) Brendan O'Donoghue, Stanford University, 2012
----------------------------------------------------------------------------
Lin-sys: sparse-indirect, nnz in A = 13, CG tol ~ 1/iter^(2.00)
EPS = 1.00e-03, ALPHA = 1.80, MAX_ITERS = 2500, NORMALIZE = 1, SCALE = 5.00
Variables n = 5, constraints m = 9
Cones:  linear vars: 6
    soc vars: 3, soc blks: 1
Setup time: 2.78e-04s
----------------------------------------------------------------------------
 Iter | pri res | dua res | rel gap | pri obj | dua obj | kap/tau | time (s)
----------------------------------------------------------------------------
     0| 4.60e+00  5.78e-01       nan      -inf       inf       inf  3.86e-05
    60| 3.92e-05  1.12e-04  6.64e-06  6.83e+00  6.83e+00  1.41e-17  9.51e-05
----------------------------------------------------------------------------
Status: Solved
Timing: Total solve time: 9.76e-05s
    Lin-sys: avg # CG iterations: 1.00, avg solve time: 2.24e-07s
    Cones: avg projection time: 4.90e-08s
----------------------------------------------------------------------------
Error metrics:
|Ax + s - b|_2 / (1 + |b|_2) = 3.9223e-05
|A'y + c|_2 / (1 + |c|_2) = 1.1168e-04
|c'x + b'y| / (1 + |c'x| + |b'y|) = 6.6446e-06
dist(s, K) = 0, dist(y, K*) = 0, s'y = 0
----------------------------------------------------------------------------
c'x = 6.8284, -b'y = 6.8285
============================================================================
optimal value with SCS: 6.82837896975
```

Here's the complete list of solver options.

ECOS options:

'max_iters' maximum number of iterations (default: 100).

'abstol' absolute accuracy (default: 1e-7).

'reltol' relative accuracy (default: 1e-6).

'feastol' tolerance for feasibility conditions (default: 1e-7).

'abstol_inacc' absolute accuracy for inaccurate solution (default: 5e-5).

'reltol_inacc' relative accuracy for inaccurate solution (default: 5e-5).

'feastol_inacc' tolerance for feasibility condition for inaccurate solution (default: 1e-4).

ECOS_BB options:

'mi_max_iters' maximum number of branch and bound iterations (default: 1000)

'mi_abs_eps' absolute tolerance between upper and lower bounds (default: 1e-6)

'mi_rel_eps' relative tolerance, (U-L)/L, between upper and lower bounds (default: 1e-3)

MOSEK options:

'mosek_params' A dictionary of MOSEK parameters. Refer to MOSEK's Python or C API for details. Note that if parameters are given as string-value pairs, parameter names must be of the form `'MSK_DPAR_BASIS_TOL_X'` as in the C API. Alternatively, Python enum options like `'mosek.dparam.basis_tol_x'` are also supported.

CVXOPT options:

'max_iters' maximum number of iterations (default: 100).

'abstol' absolute accuracy (default: 1e-7).

'reltol' relative accuracy (default: 1e-6).

'feastol' tolerance for feasibility conditions (default: 1e-7).

'refinement' number of iterative refinement steps after solving KKT system (default: 1).

'kktsolver' The KKT solver used. The default, "chol", does a Cholesky factorization with preprocessing to make A and [A; G] full rank. The "robust" solver does an LDL factorization without preprocessing. It is slower, but more robust.

SCS options:

'max_iters' maximum number of iterations (default: 2500).

'eps' convergence tolerance (default: 1e-3).

'alpha' relaxation parameter (default: 1.8).

'scale' balance between minimizing primal and dual residual (default: 5.0).

'normalize' whether to precondition data matrices (default: True).

'use_indirect' whether to use indirect solver for KKT sytem (instead of direct) (default: True).

'warm_start' whether to initialize the solver with the previous solution (default: False). The use case for warm start is solving the same problem for multiple values of a parameter.

CBC options:

Cut-generation through CGL

General remarks:

- some of these cut-generators seem to be buggy (observed problems with AllDifferentCuts, RedSplitCuts, LandPCuts, PreProcessCuts)
- a few of these cut-generators will generate noisy output even if `'verbose=False'`

The following cut-generators are available: `GomoryCuts`, `MIRCuts`, `MIRCuts2`, `TwoMIRCuts`, `ResidualCapacityCuts`, `KnapsackCuts` `FlowCoverCuts`, `CliqueCuts`, `LiftProjectCuts`, AllDifferentCuts, OddHoleCuts, RedSplitCuts, LandPCuts, PreProcessCuts, ProbingCuts, SimpleRoundingCuts.

'CutGenName' if cut-generator is activated (e.g. 'GomoryCuts=True')

Getting the standard form

If you are interested in getting the standard form that CVXPY produces for a problem, you can use the get_problem_data method. Calling get_problem_data(solver) on a problem object returns a dict of the arguments that CVXPY would pass to that solver. If the solver you choose cannot solve the problem, CVXPY will raise an exception.

```
# Get ECOS arguments.
data = prob.get_problem_data(ECOS)

# Get ECOS_BB arguments.
data = prob.get_problem_data(ECOS_BB)

# Get CVXOPT arguments.
data = prob.get_problem_data(CVXOPT)

# Get SCS arguments.
data = prob.get_problem_data(SCS)
```

After you solve the standard conic form problem returned by get_problem_data, you can unpack the raw solver output using the unpack_results method. Calling unpack_results(solver, solver_output) on a problem will update the values of all primal and dual variables as well as the problem value and status.

For example, the following code is equivalent to solving the problem directly with CVXPY:

```
# Get ECOS arguments.
data = prob.get_problem_data(ECOS)
# Call ECOS solver.
solver_output = ecos.solve(data["c"], data["G"], data["h"],
                           data["dims"], data["A"], data["b"])
# Unpack raw solver output.
prob.unpack_results(ECOS, solver_output)
```

CHAPTER 3

Examples

These examples show many different ways to use CVXPY. The *Basic Examples* section shows how to solve some common optimization problems in CVXPY. The *Advanced Examples* and *Advanced Applications* sections contains more complex examples aimed at experts in convex optimization.

Basic Examples

- Total variation in-painting
- Control
- SVM classifier with regularization
- Portfolio optimization
- Worst-case risk analysis
- Optimal advertising
- Huber regression
- Quantile regression
- Model fitting

Advanced Examples

- Object-oriented convex optimization
- Consensus optimization
- Method of multipliers

Advanced Applications

- Allocating interdiction effort to catch a smuggler
- Antenna array design
- Channel capacity
- Computing a sparse solution of a set of linear inequalities
- Entropy maximization
- Fault detection
- Filter design
- Fitting censored data
- L1 trend filtering
- Nonnegative matrix factorization
- Optimal parade route
- Optimal power and bandwidth allocation in a Gaussian broadcast channel
- Power assignment in a wireless communication system
- Predicting NBA game wins
- Robust Kalman filtering for vehicle tracking
- Sizing of clock meshes
- Sparse covariance estimation for Gaussian variables
- Water filling

CHAPTER 4

FAQ

Where can I get help with CVXPY?

You can post questions about how to use CVXPY on the CVXPY mailing list. If you've found a bug in CVXPY or have a feature request, create an issue on the CVXPY Github issue tracker.

Where can I learn more about convex optimization?

The book Convex Optimization by Boyd and Vandenberghe is available for free online and has extensive background on convex optimization. To learn more about disciplined convex programming, visit the DCP tutorial website.

How do I know which version of CVXPY I'm using?

To check which version of CVXPY you have installed, run the following code snippet in the Python prompt:

```
import cvxpy
print cvxpy.__version__
```

What do I do if I get a `DCPError` exception?

The problems you solve in CVXPY must follow the rules of disciplined convex programming (DCP). DCP is like a type system for optimization problems. For more about DCP, see the *DCP tutorial section* or the DCP tutorial website.

How do I find DCP errors?

You can test whether a problem, objective, constraint, or expression satisfies the DCP rules by calling `object.is_dcp()`. If the function returns `False`, there is a DCP error in that object.

What do I do if I get a `SolverError` exception?

Sometimes solvers encounter numerical issues and fail to solve a problem, in which case CVXPY raises a `SolverError`. If this happens to you, try using different solvers on your problem, as discussed in the "Choosing a solver" section of *Advanced Features*. If the solver CVXOPT fails, try using the solver option `kktsolver=ROBUST_KKTSOLVER`.

What solvers does CVXPY support?

See the "Solve method options" section in *Advanced Features* for a list of the solvers CVXPY supports. If you would like to use a solver CVXPY does not support, make a feature request on the CVXPY Github issue tracker.

What are the differences between CVXPY's solvers?

The solvers support different classes of problems and occupy different points on the Pareto frontier of speed, accuracy, and open source vs. closed source. See the "Solve method options" section in *Advanced Features* for details.

What do I do if I get "Exception: Cannot evaluate the truth value of a constraint"?

This error likely means you are chaining constraints (e.g., writing an expression like `0 <= x <= 1`) or using the built-in Python `max` and `min` functions on CVXPY expressions. It is not possible for CVXPY to correctly handle these use cases, so CVXPY throws an (admittedly cryptic) exception.

What do I do if I get "RuntimeError: maximum recursion depth exceeded"?

See this thread on the mailing list.

Can I use NumPy functions on CVXPY objects?

No, you can only use CVXPY functions on CVXPY objects. If you use a NumPy function on a CVXPY object, it will probably fail in a confusing way.

Can I use SciPy sparse matrices with CVXPY?

Yes, though you need to be careful. SciPy sparse matrices do not support operator overloading to the extent needed by CVXPY. (See this Github issue for details.) You can wrap a SciPy sparse matrix as a CVXPY constant, however, and then use it normally with CVXPY:

```
# Wrap the SciPy sparse matrix A as a CVXPY constant.
A = Constant(A)
# Use A normally in CVXPY expressions.
expr = A*x
```

How do I constrain a CVXPY matrix expression to be positive semidefinite?

See *Advanced Features*.

How do I create variables with special properties, such as boolean or symmetric variables?

See *Advanced Features*.

How do I create a variable that has multiple special properties, such as boolean and symmetric?

Create one variable with each desired property, and then set them all equal by adding equality constraints. CVXPY 1.0 will have a more elegant solution.

How do I create complex variables?

You must represent complex variables using real variables, as described in this Github issue. We hope to add complex variables soon.

How do I create variables with more than 2 dimensions?

You must mimic the extra dimensions using a dict, as described in this Github issue.

How does CVXPY work?

The algorithms and data structures used by CVXPY are discussed in this paper.

How do I cite CVXPY?

If you use CVXPY for published work, we encourage you to cite the software. Use the following BibTeX citation:

```
@article{cvxpy,
  author       = {Steven Diamond and Stephen Boyd},
  title        = {{CVXPY}: A {P}ython-Embedded Modeling Language for Convex␣
↪Optimization},
  journal      = {Journal of Machine Learning Research},
  note         = {To appear},
  url          = {http://stanford.edu/~boyd/papers/pdf/cvxpy_paper.pdf},
  year         = {2016},
}
```

CHAPTER 5

Citing CVXPY

If you use CVXPY for published work, we encourage you to cite the accompanying JMLR MLOSS paper.

Use the following BibTeX citation:

```
@article{cvxpy,
  author  = {Steven Diamond and Stephen Boyd},
  title   = {{CVXPY}: A {P}ython-Embedded Modeling Language for Convex Optimization},
  journal = {Journal of Machine Learning Research},
  year    = {2016},
  volume  = {17},
  number  = {83},
  pages   = {1--5},
}
```

CHAPTER 6

How to Help

We welcome all contributors to CVXPY. You don't need to be an expert in convex optimization to help out!

The cvxpy mailing list is for CVXPY users. Join this mailing list if you have questions about CVXPY or want to track CVXPY's progress.

If you're interested in contributing to CVXPY, join the Gitter chat to talk with developers.

We use GitHub to track our source code and for tracking and discussing issues. The open issues are a rough list of what needs to be done in developing CVXPY.

CHAPTER 7

Related Projects

CVXPY is part of a larger ecosystem of optimization software. We list here the optimization packages most relevant to CVXPY users.

Modeling frameworks

- DCCP is a CVXPY extension for modeling and solving difference of convex problems.
- NCVX is a CVXPY extension for modeling and solving problems with convex objectives and decision variables from a nonconvex set.
- cvxstoc is a CVXPY extension that makes it easy to code and solve stochastic optimization problems, i.e., convex optimization problems that involve random variables.
- SnapVX is a Python-based convex optimization solver for problems defined on graphs.
- CVX is a MATLAB-embedded modeling language for convex optimization problems. CVXPY is based on CVX.
- Convex.jl is a Julia-embedded modeling language for convex optimization problems. Convex.jl is based on CVXPY and CVX.
- CVXcanon is a C++ package that factors out the common operations that modeling languages like CVXPY, CVX, and Convex.jl perform.
- GPkit is a Python package for defining and manipulating geometric programming (GP) models.
- PICOS is a user-friendly python interface to many linear and conic optimization solvers.

Solvers

- ECOS is an open-source C library for solving convex second-order and exponential cone programs.
- CVXOPT is an open-source Python package for convex optimization.

- SCS is an open-source C library for solving large-scale convex cone problems.
- Elemental is an open-source C++ library for distributed-memory dense and sparse-direct linear algebra and optimization.
- GLPK is an open-source C library for solving linear programs and mixed integer linear programs.
- GUROBI is a commercial solver for mixed integer second-order cone programs.
- MOSEK is a commercial solver for mixed integer second-order cone programs and semidefinite programs.

CHAPTER 8

CVXPY Short Course

Convex optimization is easy using CVXPY! We have developed a short course that teaches how to use Python and CVXPY, explains the basics of convex optimization, and covers a variety of applications.

Visit the short course home page for further details:

- Short course home page
- Course overview slides

CHAPTER 9

License

modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "{}" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright 2017 Steven Diamond

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either

Index

S