CVXPY Documentation

Release 1.0.11

Steven Diamond, Eric Chu, Stephen Boyd

Dec 26, 2018

Contents

Convex optimization, for everyone.

For the best support, join the CVXPY mailing list *and post your questions on* Stack Overflow.

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 with box constraints:

```
import cvxpy as cp
import numpy as np

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

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

# The optimal objective value 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; in addition to convex programming, CVXPY also supports a generalization of geometric programming.

For a guided tour of CVXPY, check out the *tutorial*. Browse the *library of examples* for applications to machine learning, control, finance, and more.

News.

- CVXPY v1.0.11 supports disciplined geometric programming, which lets you formulate geometric programs and log-log convex programs. See the *tutorial* for more information.
- Version 1.0 of CVXPY brings the API closer to NumPy and the architecture closer to software compilers, making it easy for developers to write custom problem transformations and target custom solvers. CVXPY 1.0 is not backwards compatible with previous versions of CVXPY. For more details, see *What's New in 1.0*, which includes instructions for migrating from previous versions of CVXPY.

CVXPY relies on the open source solvers ECOS, OSQP, and SCS. Additional solvers are supported, but must be installed separately. For background on convex optimization, see the book Convex Optimization by Boyd and Vandenberghe.

CVXPY was designed and implemented by Steven Diamond, with input and contributions from Stephen Boyd, Eric Chu, Akshay Agrawal, Robin Verschueren, and many others; it was inspired by the MATLAB package CVX.

CHAPTER 1

Install

Note: Version 1.0 of CVXPY is incompatible with previous versions in minor ways. See *What's New in 1.0* for how to update legacy code to a form that's compatible with 1.0.

1.1 Mac OS X and Linux

CVXPY supports both Python 2 and Python 3 on OS X and Linux. We recommend using Anaconda for installation, as we find that most users prefer to let Anaconda manage dependencies and environments for them. If you are comfortable with managing your own environment, you can instead install CVXPY with *pip*.

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

```
conda install -c conda-forge lapack
conda install -c cvxgrp cvxpy
```

3. Test the installation with `nose`.

```
conda install nose
nosetests cvxpy
```

1.2 Windows

CVXPY supports Python 2 (with Anaconda and pip) and Python 3 (with pip) on Windows. We recommend using Anaconda for installation, as we find that most users prefer to let Anaconda manage dependencies and environments for them. If you are comfortable with managing your own environment or need Python 3, you can instead install CVXPY with *pip*.

1. Download and install the latest version of Anaconda.
2. Download the Visual Studio C++ compiler for Python.

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

```
conda install -c conda-forge lapack
conda install -c cvxgrp cvxpy
```

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

1.3 Other Platforms

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

1.4 Pip

CVXPY can be installed on all platforms with pip. We recommend isolating your installation in a virtualenv. After activating the environment, simply execute:

```
pip install cvxpy
```

1.5 Install from source

CVXPY has the following dependencies:

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

To test the CVXPY installation, you additionally need Nose.

CVXPY automatically installs OSQP, ECOS, SCS, six, fastcache, and multiprocess. NumPy and SciPy will need to be installed manually, as will Swig . Once you've installed these dependencies:

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

```
python setup.py install
```

1.6 Install with CVXOPT support

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

1.7 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.

1.8 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.

1.9 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.

1.10 Install with XPRESS support

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

1.11 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.

1.12 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.

1.13 Install with CPLEX support

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

CHAPTER 2

Tutorial

2.1 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:

```
import cvxpy as cp

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

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

# Form objective.
obj = cp.Minimize((x - y)**2)

# Form and solve problem.
prob = cp.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.

2.1.1 Changing the problem

Problems are immutable, meaning they cannot be changed after they are created. To change the objective or constraints, create a new problem.

```
# Replace the objective.
prob2 = cp.Problem(cp.Maximize(x + y), prob.constraints)
print("optimal value", prob2.solve())

# Replace the constraint (x + y == 1).
constraints = [x + y <= 3] + prob2.constraints[1:]
prob3 = cp.Problem(prob2.objective, constraints)
print("optimal value", prob2.solve())
```

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

2.1.2 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.

```
import cvxpy as cp

x = cp.Variable()

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

# An unbounded problem.
prob = cp.Problem(cp.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.

2.1.3 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
```

2.1.4 Vectors and matrices

Variables can be scalars, vectors, or matrices, meaning they are 0, 1, or 2 dimensional.

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

# Vector variable with shape (5,).
x = cp.Variable(5)

# Matrix variable with shape (5, 1).
x = cp.Variable((5, 1))

# Matrix variable with shape (4, 7).
A = cp.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.

import cvxpy as cp
import numpy

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

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

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

```
  Optimal value 4.14133859146
  Optimal var
  [ -5.11480673e-21   6.30625742e-21   1.34643668e-01   1.24976681e-01
-4.79039542e-21]
```

2.1.5 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.

2.1.6 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 attributes such as the sign of the parameter's entries, whether the parameter is symmetric, etc. These attributes are used in *Disciplined Convex Programming* and are unknown unless specified. Parameters can be assigned a constant value any time after they are created. The constant value must have the same dimensions and attributes as those specified when the parameter was created.

```
# Positive scalar parameter.
m = cp.Parameter(nonneg=True)
```

(continues on next page)

(continued from previous page)

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

# Matrix parameter with negative entries.
G = cp.Parameter((4, 7), nonpos=True)

# 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 = cp.Parameter(nonneg=True)
rho.value = 2

# Initialize parameter with a value.
rho = cp.Parameter(nonneg=True, value=2)
```

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

```
import cvxpy as cp
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)
# gamma must be nonnegative due to DCP rules.
gamma = cp.Parameter(nonneg=True)

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

plt.rc('text', usetex=True)
```

(continues on next page)

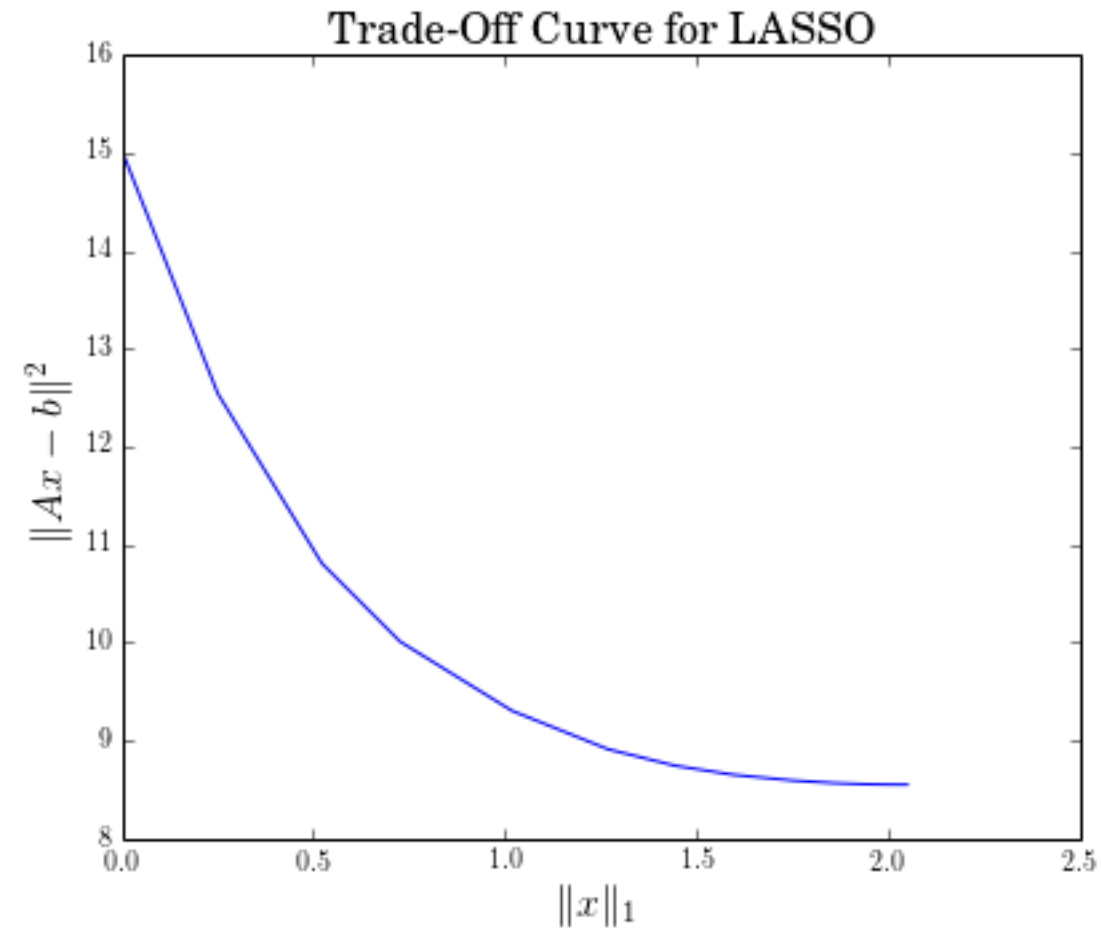
(continued from previous page)

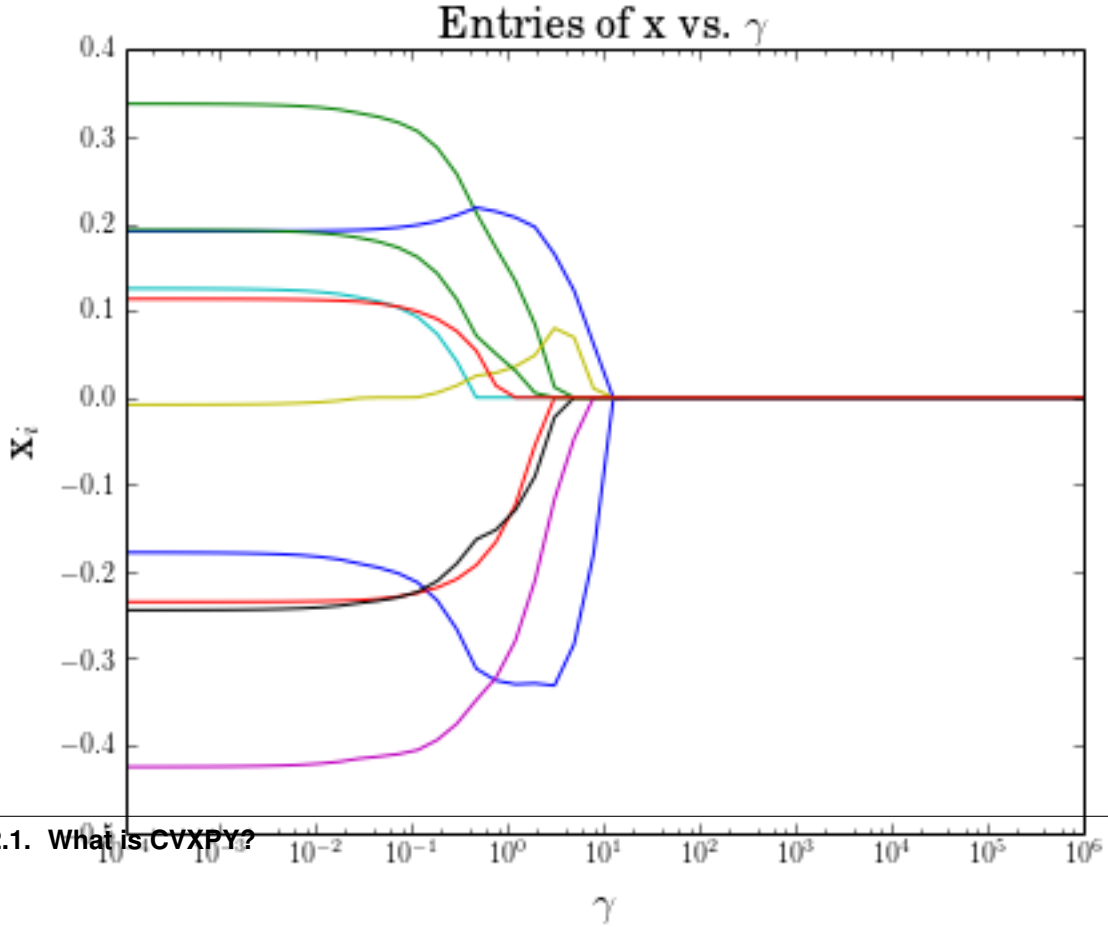
```
plt.rc('font', family='serif')
plt.figure(figsize=(6,10))

# 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] 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)
```

2.2 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.

2.2.1 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:

```
import cvxpy as cp

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

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

Expressions can be scalars, vectors, or matrices. The dimensions of an expression are stored as `expr.shape`. The total number of entries is given by `expr.size`, while the number of dimensions is given by `expr.ndim`. 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. The semantics for how shapes behave under arithmetic operations are the same as for NumPy ndarrays (except some broadcasting is banned).

```
import numpy

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

(continues on next page)

(continued from previous page)

```
# Use expr.shape to get the dimensions.
print("dimensions of X:", X.shape)
print("size of X:", X.size)
print("number of dimensions:", X.ndim)
print("dimensions of sum(X):", cp.sum(X).shape)
print("dimensions of A*X:", (A*X).shape)

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

```
dimensions of X: (5, 4)
size of X: 20
number of dimensions: 2
dimensions of sum(X): ()
dimensions of A*X: (3, 4)
Cannot broadcast dimensions (3, 5) (5, 4)
```

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

2.2.2 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 = cp.Variable()
a = cp.Parameter(nonpos=True)
c = numpy.array([1, -1])

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

```
sign of x: UNKNOWN
sign of a: NONPOSITIVE
sign of square(x): NONNEGATIVE
sign of c*a: UNKNOWN
```

2.2.3 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.

2.2.4 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 = cp.Variable()
a = cp.Parameter(nonneg=True)

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

(continues on next page)

(continued from previous page)

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

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

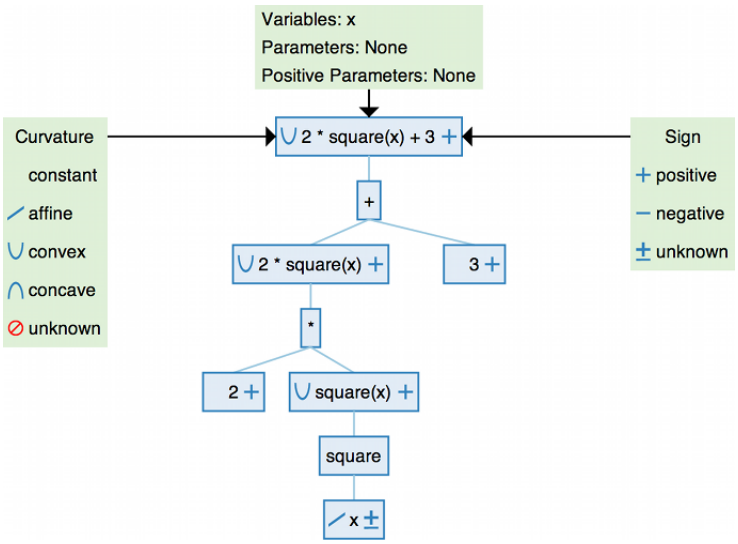
2.2.5 Infix operators

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 DCP only when one of the expressions is constant. `expr1/expr2` is DCP 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.

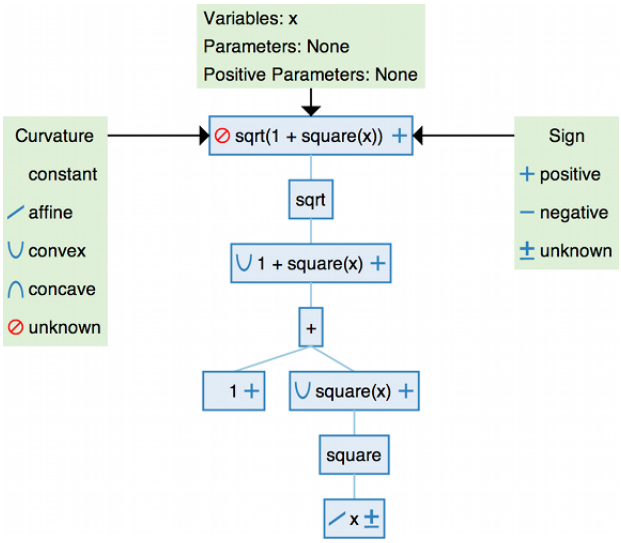
2.2.6 Example 1

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.



2.2.7 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:",
      cp.sqrt(1 + cp.square(x)).curvature)
print("norm(hstack([1, x]), 2) curvature:",
      cp.norm(cp.hstack([1, x]), 2).curvature)
```

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

2.2.8 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 = cp.Variable()
y = cp.Variable()

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

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

# Non-DCP problems.

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

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

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

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

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

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

2.3 Atomic Functions

This section of the tutorial describes the atomic 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.

2.3.1 Operators

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

Note that in CVXPY, `expr1 * expr2` denotes matrix multiplication when `expr1` and `expr2` are matrices; if you're running Python 3, you can alternatively use the `@` operator for matrix multiplication. Regardless of your Python version, you can also use the function *matmul* to multiply two matrices. To multiply two arrays or matrices elementwise, use *multiply*.

Indexing and slicing

Indexing in CVXPY follows exactly the same semantics as NumPy ndarrays. For example, if `expr` has shape `(5,)` then `expr[1]` gives the second entry. More generally, `expr[i:j:k]` selects every kth element of `expr`, starting at `i` and ending at `j-1`. If `expr` is a matrix, then `expr[i:j:k]` selects rows, while `expr[i:j:k, r:s:t]` selects both rows and columns. Indexing drops dimensions while slicing preserves dimenions. For example,

```
x = cvxpy.Variable(5)
print("0 dimensional", x[0].shape)
print("1 dimensional", x[0:1].shape)
```

```
0 dimensional: ()
1 dimensional: (1,)
```

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)`.

2.3.2 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)* $p \in \mathbf{R}^n_+$ $p \neq 0$	$x_1^{1/n} \cdots x_n^{1/n}$ $(x_1^{p_1} \cdots x_n^{p_n})^{\frac{1}{\mathbf{1}^T p}}$	$x \in \mathbf{R}^n_+$	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) $k = 1, \ldots, n$ eigenvalues of X	sum of k largest				
	$X \in \mathbf{S}^n$	unknown	convex	None	

Continued on next page

Table 1 – continued from previous page

Function	Meaning	Domain	Sign	Curvature	Monotonicity
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(X)	$\max_{ij}\{X_{ij}\}$	$X \in \mathbf{R}^{m \times n}$	same as X	convex	incr.
min(X)	$\min_{ij}\{X_{ij}\}$	$X \in \mathbf{R}^{m \times n}$	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)	$\sqrt{\sum_i \lvert x_i\rvert^2}$	$X \in \mathbf{R}^n$	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.
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(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.

Continued on next page

Table 1 – continued from previous page

Function	Meaning	Domain	Sign	Curvature	Monotonicity
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)	$\mathrm{tr}\,(X)$	$X \in \mathbf{R}^{n \times n}$	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` and `min` give the largest and smallest entry, respectively, in a single expression. These functions should not be confused with `maximum` and `minimum` (see *Elementwise functions*). Use `maximum` and `minimum` to find the max or min of a list of scalar expressions.

The CVXPY function `sum` 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)
```

2.3.3 Functions along an axis

The functions `sum`, `norm`, `max`, and `min` can be applied along an axis. Given an `m` by `n` expression `expr`, the syntax `func(expr, axis=0, keepdims=True)` applies `func` to each column, returning a 1 by `n` expression. The syntax `func(expr, axis=1, keepdims=True)` applies `func` to each row, returning an `m` by 1 expression. By default `keepdims=False`, which means dimensions of length 1 are dropped. For example, the following code sums along the columns and rows of a matrix variable:

```
X = cvxpy.Variable((5, 4))
col_sums = cvxpy.sum(X, axis=0, keepdims=True) # Has size (1, 4)
col_sums = cvxpy.sum(X, axis=0) # Has size (4,)
row_sums = cvxpy.sum(X, axis=1) # Has size (5,)
```

2.3.4 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 `maximum` and `multiply`, operate on the corresponding elements of each argument. For example, if `X` and `Y` are both 3 by 3 matrix variables, then `maximum(X, Y)` is a 3 by 3 matrix expression. `maximum(X, Y)[2, 0]` is equivalent to `maximum(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)	$\|x\|$	$x \in \mathbf{R}$	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) $M \geq 0$	$\begin{cases} x^2 & \|x\| \leq M \\ 2M\|x\| - M^2 & \|x\| > M \end{cases}$	$x \in \mathbf{R}$	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.
maximum(x, y)	$\max\{x, y\}$	$x, y \in \mathbf{R}$	depends on x,y	convex	incr.
minimum(x, y)	$\min\{x, y\}$	$x, y \in \mathbf{R}$	depends on x,y	concave	incr.
multiply(c, x) $c \in \mathbf{R}$	c*x	$x \in \mathbf{R}$	$\text{sign}(cx)$	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	$\alpha\text{pos}(x) + \beta\text{neg}(x)$	$x \in \mathbf{R}$	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 $x \leq 0$

2.3.5 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]])	$\begin{bmatrix} X^{(1,1)} & \cdots & X^{(1,q)} \\ \vdots & & \vdots \\ X^{(p,1)} & \cdots & X^{(p,q)} \end{bmatrix}$	$X^{(i,j)} \in \mathbf{R}^{m_i \times n_j}$	$\text{sign}\left(\sum_{ij} X_{11}^{(i,j)}\right)$	affine	incr.
conv(c, x) $c \in \mathbf{R}^m$	$c * x$	$x \in \mathbf{R}^n$	$\text{sign}\,(c_1 x_1)$	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])	$\begin{bmatrix} X^{(1)} \cdots X^{(k)} \end{bmatrix}$	$X^{(i)} \in \mathbf{R}^{m \times n_i}$	$\text{sign}\left(\sum_i X_{11}^{(i)}\right)$	affine	incr.
kron(C, X) $C \in \mathbf{R}^{p \times q}$	$\begin{bmatrix} C_{11}X & \cdots & C_{1q}X \\ \vdots & & \vdots \\ C_{p1}X & \cdots & C_{pq}X \end{bmatrix}$	$X \in \mathbf{R}^{m \times n}$	$\text{sign}\,(C_{11}X_{11})$	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])	$\begin{bmatrix} X^{(1)} \\ \vdots \\ X^{(k)} \end{bmatrix}$	$X^{(i)} \in \mathbf{R}^{m_i \times n}$	$\text{sign}\left(\sum_i X_{11}^{(i)}\right)$	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, $X'_{ij} = \mathbf{vec}(X)_{m'j+i}$.

2.4 Disciplined Geometric Programming

Disciplined geometric programming (DGP) is an analog of DCP for *log-log convex* functions, that is, functions of positive variables that are convex with respect to the geometric mean instead of the arithemetic mean.

While DCP is a ruleset for constructing convex programs, DGP is a ruleset for log-log convex programs (LLCPs), which are problems that are convex after the variables, objective functions, and constraint functions are replaced with their logs, an operation that we refer to as a *log-log* transformation. Every geometric program (GP) and generalized geometric program (GGP) is an LLCP, but there are LLCPs that are neither GPs nor GGPs.

CVXPY lets you form and solve DGP problems, just as it does for DCP problems. For example, the following code solves a simple geometric program,

```
import cvxpy as cp

# DGP requires Variables to be declared positive via `pos=True`.
x = cp.Variable(pos=True)
y = cp.Variable(pos=True)
z = cp.Variable(pos=True)

objective_fn = x * y * z
constraints = [
  4 * x * y * z + 2 * x * z <= 10, x <= 2*y, y <= 2*x, z >= 1]
problem = cp.Problem(cp.Maximize(objective_fn), constraints)
problem.solve(gp=True)
print("Optimal value: ", problem.value)
print("x: ", x.value)
print("y: ", y.value)
print("z: ", z.value)
```

and it prints the below output.

```
Optimal value: 1.9999999938309496
x: 0.9999999989682057
y: 1.999999974180587
z: 1.0000000108569758
```

Note that to solve DGP problems, you must pass the option `gp=True` to the `solve()` method.

This section explains what DGP is, and it shows how to construct and solve DGP problems using CVXPY. At the end of the section are tables listing all the atoms that can be used in DGP problems, similar to the tables presented in the section on *DCP atoms*.

For an in-depth reference on DGP, see our accompanying paper. For interactive code examples, check out our *notebooks*.

Note: DGP is a recent addition to CVXPY. If you have feedback, please file an issue or make a pull request on Github.

2.4.1 Log-log curvature

Just as every Expression in CVXPY has a curvature (constant, affine, convex, concave, or unknown), every Expression also has a log-log curvature.

A function $f : D \subseteq \mathbf{R}^n_{++} \to \mathbf{R}$ is said to be log-log convex if the function $F(u) = \log f(e^u)$, with domain $\{u \in \mathbf{R}^n : e^u \in D\}$, is convex (where $\mathbf{R}^n_{++}$ denotes the set of positive reals and the logarithm and exponential are meant elementwise); the function F is called the log-log transformation of f. The function f is log-log concave if F is concave, and it is log-log affine if F is affine.

Every log-log affine function has the form

$$f(x) = cx_1^{a_1}x_2^{a_2}\ldots x_n^{a_n}$$

where x is in $\mathbf{R}^n_{++}$, the a_i are real numbers, and c is a positive scalar. In the context of geometric programming, such a function is called a monomial function. A sum of monomials, known as a posynomial function in geometric programming, is a log-log convex function; A table of all the atoms with known log-log curvature is presented at the end of this page.

In the below table, F is the log-log transformation of f, $u = \log x$, and $v = \log y$, where x and y are in the domain of f

Log-Log Curvature	Meaning
log-log constant	F is a constant (so f is a positive constant)
log-log affine	$F(\theta u + (1-\theta)v) = \theta F(u) + (1-\theta)F(v),\ \forall u,\ v,\ \theta \in [0,1]$
log-log convex	$F(\theta u + (1-\theta)v) \leq \theta F(u) + (1-\theta)F(v),\ \forall u,\ v,\ \theta \in [0,1]$
log-log concave	$F(\theta u + (1-\theta)v) \geq \theta F(u) + (1-\theta)F(v),\ \forall u,\ v,\ \theta \in [0,1]$
unknown	DGP analysis cannot determine the curvature

CVXPY's log-log curvature analysis can flag Expressions as unknown even when they are log-log convex or log-log concave. Note that any log-log constant expression is also log-log affine, and any log-log affine expression is log-log convex and log-log concave.

The log-log curvature of an Expression is stored in its `.log_log_curvature` attribute. For example, running the following script

```
import cvxpy as cp

x = cp.Variable(pos=True)
y = cp.Variable(pos=True)

constant = cp.Constant(2.0)
monomial = constant * x * y
posynomial = monomial + (x ** 1.5) * (y ** -1)
reciprocal = posynomial ** -1
unknown = reciprocal + posynomial

print(constant.log_log_curvature)
print(monomial.log_log_curvature)
print(posynomial.log_log_curvature)
print(reciprocal.log_log_curvature)
print(unknown.log_log_curvature)
```

prints the following output.

```
LOG-LOG CONSTANT
LOG-LOG AFFINE
LOG-LOG CONVEX
LOG-LOG CONCAVE
UNKNOWN
```

You can also check the log-log curvature of an Expression by calling the methods `is_log_log_constant()`, `is_log_log_affine()`, `is_log_log_convex()`, `is_log_log_concave()`. For example, `posynomial.is_log_log_convex()` would evaluate to `True`.

2.4.2 Log-log curvature rules

For an Expression to have known log-log curvature, all of the Constants, Variables, and Parameters it refers to must be elementwise positive. A Constant is positive if its numerical value is positive. Variables and Parameters are positive only if the keyword argument `pos=True` is supplied to their constructors (e.g., `x = cvxpy.Variable(shape=(), pos=True)`). To summarize, when formulating a DGP problem, *all Constants should be elementwise positive, and all Variables and Parameters must be constructed with the attribute* `pos=True`.

DGP analysis is exactly analogous to DCP analysis. It is based on a library of atoms (functions) with known monotonicity and log-log curvature and a a single composition rule. The library of atoms is presented at the end of this page; the composition rule is stated below.

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

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

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

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

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

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

If an Expression satisfies the composition rule, we colloquially say that the Expression "is DGP." You can check whether an Expression is DGP by calling the method `is_dgp()`. For example, the assertions in the following code block will pass.

```
import cvxpy as cp

x = cp.Variable(pos=True)
y = cp.Variable(pos=True)

monomial = 2.0 * constant * x * y
posynomial = monomial + (x ** 1.5) * (y ** -1)

assert monomial.is_dgp()
assert posynomial.is_dgp()
```

An Expression is DGP precisely when it has known log-log curvature, which means at least one of the methods `is_log_log_constant()`, `is_log_log_affine()`, `is_log_log_convex()`, `is_log_log_concave()` will return `True`.

2.4.3 DGP problems

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

- Minimize(log-log convex)

- Maximize(log-log concave)

The only valid constraints under the DGP rules are

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

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

```
import cvxpy as cp

# DGP requires Variables to be declared positive via `pos=True`.
x = cp.Variable(pos=True)
y = cp.Variable(pos=True)
z = cp.Variable(pos=True)

objective_fn = x * y * z
constraints = [
  4 * x * y * z + 2 * x * z <= 10, x <= 2*y, y <= 2*x, z >= 1]
assert objective_fn.is_log_log_concave()
assert all(constraint.is_dgp() for constraint in constraints)
problem = cp.Problem(cp.Maximize(objective_fn), constraints)
assert problem.is_dgp()

# All Variables must be declared as positive for an Expression to be DGP.
w = cp.Variable()
objective_fn = w * x * y
assert not objective_fn.is_dgp()
problem = cp.Problem(cp.Maximize(objective_fn), constraints)
assert not problem.is_dgp()
```

CVXPY will raise an exception if you call `problem.solve(gp=True)` on a non-DGP problem.

2.4.4 DGP atoms

This section of the tutorial describes the DGP atom library, that is, the atomic functions with known log-log curvature and monotonicity. CVXPY uses the function information in this section and the DGP rules to mark expressions with a log-log curvature. Note that every DGP expression is positive.

Infix operators

The infix operators `+, *, /` are treated as atoms. The operators `*` and `/` are log-log affine functions. The operator `+` is log-log convex in both its arguments.

Note that in CVXPY, `expr1 * expr2` denotes matrix multiplication when `expr1` and `expr2` are matrices; if you're running Python 3, you can alternatively use the `@` operator for matrix multiplication. Regardless of your Python version, you can also use the *matmul atom* to multiply two matrices. To multiply two arrays or matrices elementwise, use the *multiply atom*. Finally, to take the product of the entries of an Expression, use the *prod atom*.

Transpose

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

Power

For any CVXPY expression `expr`, the power operator `expr**p` is equivalent to the function `power(expr, p)`. Taking powers is a log-log affine function.

Scalar functions

A scalar function takes one or more scalars, vectors, or matrices as arguments and returns a scalar. Note that several of these atoms may be applied along an axis; see the API reference or the *DCP atoms tutorial* for more information.

Function	Meaning	Domain	Log-log curvature	Monotonicity
geo_mean(x) *geo_mean(x, p)* $p \in \mathbf{R}^n_+$ $p \neq 0$	$x_1^{1/n} \cdots x_n^{1/n}$ $(x_1^{p_1} \cdots x_n^{p_n})^{\frac{1}{\mathbf{1}^T p}}$	$x \in \mathbf{R}^n_+$	log-log affine	incr.
harmonic_mean(x)	$\frac{n}{\frac{1}{x_1}+\cdots+\frac{1}{x_n}}$	$x \in \mathbf{R}^n_+$	log-log concave	incr.
max(X)	$\max_{ij}\{X_{ij}\}$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log convex	incr.
min(X)	$\min_{ij}\{X_{ij}\}$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log concave	incr.
norm(x) norm(x, 2)	$\sqrt{\sum_i \lvert x_i\rvert^2}$	$X \in \mathbf{R}^n_{++}$	log-log convex	incr.
norm(X, "fro")	$\sqrt{\sum_{ij} X_{ij}^2}$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log convex	incr.
norm(X, 1)	$\sum_{ij}\lvert X_{ij}\rvert$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log convex	incr.
norm(X, "inf")	$\max_{ij}\{\lvert X_{ij}\rvert\}$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log convex	incr.
pnorm(X, p) $p \geq 1$ or `p = 'inf'`	$\lVert X\rVert_p = \left(\sum_{ij}\lvert X_{ij}\rvert^p\right)^{1/p}$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log convex	incr.
pnorm(X, p) $0 < p < 1$	$\lVert X\rVert_p = \left(\sum_{ij} X_{ij}^p\right)^{1/p}$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log convex	incr.
prod(X)	$\prod_{ij} X_{ij}$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log affine	incr.
quad_form(x, P)	$x^T P x$	$x \in \mathbf{R}^n$, $P \in \mathbf{R}^{n\times n}_{++}$	log-log convex	incr.
quad_over_lin(X, y)	$\left(\sum_{ij} X_{ij}^2\right)/y$	$x \in \mathbf{R}^n_{++}$ $y > 0$	log-log convex	in X_{ij} decr. in y
sum(X)	$\sum_{ij} X_{ij}$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log convex	incr.
sum_squares(X)	$\sum_{ij} X_{ij}^2$	$X \in \mathbf{R}^{m\times n}_{++}$	log-log convex	incr.
trace(X)	$\mathrm{tr}\,(X)$	$X \in \mathbf{R}^{n\times n}_{++}$	log-log convex	incr.
pf_eigenvalue(X)	spectral radius of X	$X \in \mathbf{R}^{n\times n}_{++}$	log-log convex	incr.

Elementwise functions

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

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

Function	Meaning	Domain	Curvature	Monotonicity
diff_pos(x, y)	$x - y$	$0 < y < x$	log-log concave	incr. in x decr. in y
entr(x)	$-x\log(x)$	$0 < x < 1$	log-log concave	None
exp(x)	e^x	$x > 0$	log-log convex	incr.
log(x)	$\log(x)$	$x > 1$	log-log concave	incr.
maximum(x, y)	$\max\{x, y\}$	$x, y > 0$	log-log convex	incr.
minimum(x, y)	$\min\{x, y\}$	$x, y > 0$	log-log concave	incr.
multiply(x, y)	$x * y$	$x, y > 0$	log-log affine	incr.
one_minus_pos(x)	$1 - x$	$0 < x < 1$	log-log concave	decr.
power(x, 0)	1	$x > 0$	constant	constant
power(x, p)	x	$x > 0$	log-log affine	for $p > 0$ for $p < 0$
sqrt(x)	$\sqrt{x}$	$x > 0$	log-log affine	incr.
square(x)	x^2	$x > 0$	log-log affine	incr.

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	Curvature	Monotonicity
bmat([[X11,...,X1q], ..., [Xp1,...,Xpq]])	$\begin{bmatrix} X^{(1,1)} & \cdots & X^{(1,q)} \\ \vdots & & \vdots \\ X^{(p,1)} & \cdots & X^{(p,q)} \end{bmatrix}$	$X^{(i,j)} \in \mathbf{R}_{++}^{m_i \times n_j}$	log-log affine	incr.
diag(x)	$\begin{bmatrix} x_1 & & \\ & \ddots & \\ & & x_n \end{bmatrix}$	$x \in \mathbf{R}_{++}^n$	log-log affine	incr.
diag(X)	$\begin{bmatrix} X_{11} \\ \vdots \\ X_{nn} \end{bmatrix}$	$X \in \mathbf{R}_{++}^{n \times n}$	log-log affine	incr.
eye_minus_inv(X)	$(I - X)^{-1}$	$X \in \mathbf{R}_{++}^{n \times n}, \lambda_{\text{pf}}(X) < 1$	log-log convex	incr.
hstack([X1, ..., Xk])	$\begin{bmatrix} X^{(1)} \cdots X^{(k)} \end{bmatrix}$	$X^{(i)} \in \mathbf{R}_{++}^{m \times n_i}$	log-log affine	incr.
matmul(X, Y)	XY	$X \in \mathbf{R}_{++}^{m \times n}, Y \in \mathbf{R}_{++}^{n \times p}$	log-log convex	incr.
resolvent(X)	$(sI - X)^{-1}$	$X \in \mathbf{R}_{++}^{n \times n}, \lambda_{\text{pf}}(X) < s$	log-log convex	incr.
reshape(X, (n', m'))	$X' \in \mathbf{R}^{m' \times n'}$	$X \in \mathbf{R}_{++}^{m \times n}$ $m'n' = mn$	log-log affine	incr.
vec(X)	$x' \in \mathbf{R}^{mn}$	$X \in \mathbf{R}_{++}^{m \times n}$	log-log affine	incr.
vstack([X1, ..., Xk])	$\begin{bmatrix} X^{(1)} \\ \vdots \\ X^{(k)} \end{bmatrix}$	$X^{(i)} \in \mathbf{R}_{++}^{m_i \times n}$	log-log affine	incr.

2.5 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.

2.5.1 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.

```
import cvxpy as cp

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

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

# Form objective.
obj = cp.Minimize((x - y)**2)

# Form and solve problem.
prob = cp.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.

2.5.2 Attributes

Variables and parameters can be created with attributes specifying additional properties. For example, `Variable(nonneg=True)` is a scalar variable constrained to be nonnegative. Similarly, `Parameter(nonpos=True)` is a scalar parameter constrained to be nonpositive. The full constructor for *Leaf* (the parent class of *Variable* and *Parameter*) is given below.

Leaf(*shape=None*, *name=None*, *value=None*, *nonneg=False*, *nonpos=False*, *symmetric=False*, *diag=False*, *PSD=False*, *NSD=False*, *boolean=False*, *integer=False*)

Creates a Leaf object (e.g., Variable or Parameter). Only one attribute can be active (set to True).

Parameters

- **shape** (*tuple or int*) – The variable dimensions (0D by default). Cannot be more than 2D.
- **name** (*str*) – The variable name.
- **value** (*numeric type*) – A value to assign to the variable.
- **nonneg** (*bool*) – Is the variable constrained to be nonnegative?
- **nonpos** (*bool*) – Is the variable constrained to be nonpositive?
- **symmetric** (*bool*) – Is the variable constrained to be symmetric?
- **hermitian** (*bool*) – Is the variable constrained to be Hermitian?
- **diag** (*bool*) – Is the variable constrained to be diagonal?
- **complex** (*bool*) – Is the variable complex valued?
- **imag** (*bool*) – Is the variable purely imaginary?
- **PSD** (*bool*) – Is the variable constrained to be symmetric positive semidefinite?
- **NSD** (*bool*) – Is the variable constrained to be symmetric negative semidefinite?
- **boolean** (*bool or list of tuple*) – Is the variable boolean (i.e., 0 or 1)? True, which constrains the entire variable to be boolean, False, or a list of indices which should be constrained as boolean, where each index is a tuple of length exactly equal to the length of shape.
- **integer** (*bool or list of tuple*) – Is the variable integer? The semantics are the same as the boolean argument.

The `value` field of Variables and Parameters can be assigned a value after construction, but the assigned value must satisfy the object attributes. A Euclidean projection onto the set defined by the attributes is given by the *project* method.

```
p = Parameter(nonneg=True)
try:
    p.value = -1
except Exception as e:
    print(e)

print("Projection:", p.project(-1))
```

```
Parameter value must be nonnegative.
Projection: 0.0
```

A sensible idiom for assigning values to leaves is *leaf.value = leaf.project(val)*, ensuring that the assigned value satisfies the leaf's properties. A slightly more efficient variant is *leaf.project_and_assign(val)*, which projects and assigns the value directly, without additionally checking that the value satisfies the leaf's properties. In most cases `project` and checking that a value satisfies a leaf's properties are cheap operations (i.e., $O(n)$), but for symmetric positive semidefinite or negative semidefinite leaves, the operations compute an eigenvalue decomposition.

Many attributes, such as nonnegativity and symmetry, can be easily specified with constraints. What is the advantage then of specifying attributes in a variable? The main benefit is that specifying attributes enables more fine-grained DCP analysis. For example, creating a variable `x` via `x = Variable(nonpos=True)` informs the DCP analyzer that `x` is nonpositive. Creating the variable `x` via `x = Variable()` and adding the constraint `x >= 0` separately does not provide any information about the sign of `x` to the DCP analyzer.

2.5.3 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 `Variable((n, n), PSD=True)` 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 = cp.Variable((100, 100), PSD=True)

# You can use X anywhere you would use
# a normal CVXPY variable.
obj = cp.Minimize(cp.norm(X) + cp.sum(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 $z \in \mathcal{R}^n$. In other words, $X + X^T$ is positive semidefinite. 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 `Variable((n, n), symmetric=True)` to create an `n` by `n` variable constrained to be symmetric. The difference between specifying that a variable is symmetric via attributes and adding the constraint `X == X.T` is that attributes are parsed for DCP information and a symmetric variable is defined over the (lower dimensional) vector space of symmetric matrices.

2.5.4 Mixed-integer programs

In mixed-integer programs, certain variables are constrained to be boolean (i.e., 0 or 1) or integer valued. You can construct mixed-integer programs by creating variables with the attribute that they have only boolean or integer valued entries:

```
# Creates a 10-vector constrained to have boolean valued entries.
x = cp.Variable(10, boolean=True)

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

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

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

2.5.5 Complex valued expressions

By default variables and parameters are real valued. Complex valued variables and parameters can be created by setting the attribute `complex=True`. Similarly, purely imaginary variables and parameters can be created by setting the attributes `imag=True`. Expressions containing complex variables, parameters, or constants may be complex valued. The functions `is_real`, `is_complex`, and `is_imag` return whether an expression is purely real, complex, or purely imaginary, respectively.

```
# A complex valued variable.
x = cp.Variable(complex=True)
# A purely imaginary parameter.
p = cp.Parameter(imag=True)

print("p.is_imag() = ", p.is_imag())
print("(x + 2).is_real() = ", (x + 2).is_real())
```

```
p.is_imag() = True
(x + 2).is_real() = False
```

The top-level expressions in the problem objective and inequality constraints must be real valued, but subexpressions may be complex. Arithmetic and all linear atoms are defined for complex expressions. The nonlinear atoms `abs` and all norms except `norm(X, p)` for `p < 1` are also defined for complex expressions. All atoms whose domain is symmetric matrices are defined for Hermitian matrices. Similarly, the atoms `quad_form(x, P)` and `matrix_frac(x, P)` are defined for complex `x` and Hermitian `P`. Lastly, equality and postive semidefinite constraints are defined for complex expressions.

The following additional atoms are provided for working with complex expressions:

- `real(expr)` gives the real part of `expr`.
- `imag(expr)` gives the imaginary part of `expr` (i.e., `expr = real(expr) + 1j*imag(expr)`).
- `conj(expr)` gives the complex conjugate of `expr`.
- `expr.H` gives the Hermitian (conjugate) transpose of `expr`.

2.5.6 Transforms

Transforms provide additional ways of manipulating CVXPY objects beyond the atomic functions. For example, the *`indicator`* transform converts a list of constraints into an expression representing the convex function that takes value 0 when the constraints hold and ∞ when they are violated.

```
x = cp.Variable()
constraints = [0 <= x, x <= 1]
expr = cp.indicator(constraints)
x.value = .5
print("expr.value = ", expr.value)
x.value = 2
print("expr.value = ", expr.value)
```

```
expr.value = 0.0
expr.value = inf
```

The full set of transforms available is discussed in *Transforms*.

2.5.7 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.

2.5.8 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*, *gp=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.
- **gp** (*bool, optional*) – If True, parses the problem as a disciplined geometric program instead of a disciplined convex program.

- **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, OSQP, and SCS. Many other solvers can be called by CVXPY if installed separately. The table below shows the types of problems the supported solvers can handle.

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

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 can both handle all problems (except mixed-integer programs). ECOS_BB is called for mixed-integer LPs and SOCPs. If the problem is a QP, CVXPY will use OSQP.

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 = cp.Variable(2)
obj = cp.Minimize(x[0] + cp.norm(x, 1))
constraints = [x >= 2]
prob = cp.Problem(obj, constraints)

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

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

```
optimal value with OSQP: 6.0
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
optimal value with CPLEX: 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', 'OSQP', 'CPLEX']
```

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=cp.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
```

Solving disciplined geometric programs

TODO see XXX

2.5.9 Solver stats

When the `solve` method is called on a problem object and a solver is invoked, the problem object records the optimal value, the values of the primal and dual variables, and several solver statistics. We have already discussed how to view the optimal value and variable values. The solver statistics are accessed via the `problem.solver_stats` attribute, which returns a *`SolverStats`* object. For example, `problem.solver_stats.solve_time` gives the time it took the solver to solve the problem.

2.5.10 Warm start

When solving the same problem for multiple values of a parameter, many solvers can exploit work from previous solves (i.e., warm start). For example, the solver might use the previous solution as an initial point or reuse cached matrix factorizations. Warm start is enabled by default and controlled with the `warm_start` solver option. The code below shows how warm start can accelerate solving a sequence of related least-squares problems.

```
import cvxpy as cp
import numpy

# Problem data.
```

(continues on next page)

(continued from previous page)

```
m = 2000
n = 1000
numpy.random.seed(1)
A = numpy.random.randn(m, n)
b = cp.Parameter(m)

# Construct the problem.
x = cp.Variable(n)
prob = cp.Problem(cp.Minimize(cp.sum_squares(A*x - b)),
                  [x >= 0])

b.value = numpy.random.randn(m)
prob.solve()
print("First solve time:", prob.solve_time)

b.value = numpy.random.randn(m)
prob.solve(warm_start=True)
print("Second solve time:", prob.solve_time)
```

```
First solve time: 11.14
Second solve time: 2.95
```

The speed up in this case comes from caching the KKT matrix factorization. If `A` were a parameter, factorization caching would not be possible and the benefit of warm start would only be a good initial point.

Setting solver options

The OSQP, 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=cp.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
```

(continues on next page)

(continued from previous page)

```
    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 is the complete list of solver options.

OSQP options:

See OSQP documentation.

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.

'save_file' The name of a file where MOSEK will save the problem just before optimization. Refer to MOSEK documentation for a list of supported file formats. File format is chosen based on the extension.

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).

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'`)

CPLEX options:

'cplex_params' a dictionary where the key-value pairs are composed of parameter names (as used in the CPLEX Python API) and parameter values. For example, to set the advance start switch parameter (i.e., CPX_PARAM_ADVIND), use "advance" for the parameter name. For the data consistency checking and modeling assistance parameter (i.e., CPX_PARAM_DATACHECK), use "read.datacheck" for the parameter name, and so on.

'cplex_filename' a string specifying the filename to which the problem will be written. For example, use "model.lp", "model.sav", or "model.mps" to export to the LP, SAV, and MPS formats, respectively.

2.5.11 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 OSQP arguments.
data = prob.get_problem_data(cvx.OSQP)

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

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

(continues on next page)

(continued from previous page)

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

# Get SCS arguments.
data = prob.get_problem_data(cvx.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(cvx.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(cvx.ECOS, solver_output)
```

2.5.12 Reductions

CVXPY uses a system of **reductions** to rewrite problems from the form provided by the user into the standard form that a solver will accept. A reduction is a transformation from one problem to an equivalent problem. Two problems are equivalent if a solution of one can be converted efficiently to a solution of the other. Reductions take a CVXPY Problem as input and output a CVXPY Problem. The full set of reductions available is discussed in *Reductions*.

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 *Machine Learning* section is a tutorial covering convex methods in machine learning.
- The *Advanced* and *Advanced Applications* sections contains more complex examples aimed at experts in convex optimization.
- The *Disciplined Geometric Programming* section contains an interactive tutorial on *disciplined geometric programming* and various examples of DGP problems.

3.1 Basic Examples

- Total variation in-painting [.ipynb]
- Control
- Portfolio optimization
- Worst-case risk analysis
- Optimal advertising
- Model fitting

3.2 Machine Learning

- Ridge regression [.py] [.ipynb]
- Lasso regression [.py] [.ipynb]
- SVM classifier with regularization
- Huber regression

- Quantile regression

3.3 Advanced

- Object-oriented convex optimization [.ipynb]
- Consensus optimization [.ipynb]
- Method of multipliers [.ipynb]

3.4 Advanced Applications

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

3.5 Disciplined Geometric Programming

- DGP fundamentals [.ipynb]
- Maximizing the volume of a box [.ipynb]
- Power control [.ipynb]
- Perron-Frobenius matrix completion [.ipynb]
- Rank-one nonnegative matrix factorization [.ipynb]

CHAPTER 4

API Documentation

CVXPY is designed to be intuitive enough so that it may be used without consulting an API reference; the tutorials will suffice in acquainting you with our software. Nonetheless, we include here an API reference for those who are comfortable reading technical documentation.

All of the documented classes and functions are imported into the `cvxpy` namespace; this means that they can be used by simply writing `cvxpy.symbol`, where `symbol` is the name of your class or function of choice, so long as you import the `cvxpy` package in your python source file.

The documentation is grouped five sections: *atoms*, *constraints*, *expressions*, *problems*, and *reductions*. The atoms section documents the classes implementing atomic mathematical functions, like `exp`, `log`, and `sqrt`; the constraints section documents the constraints that can be imposed upon variables; the expressions section documents the classes implementing mathematical expression trees, including the *`Variable`* and *`Parameter`* classes; the problem section documents the *`Problem`* class and other related classes; the reductions section documents principled operations that convert problems from a particular form to another equivalent form; and the transforms section documents additional operations available for manipulating CVXPY objects;

4.1 Atoms

Atoms are mathematical functions that can be applied to *`Expression`* instances. Applying an atom to an expression yields another expression. Atoms and compositions thereof are precisely the mechanisms that allow you to build up mathematical expression trees in CVXPY.

Every atom is tagged with information about its domain, sign, curvature, log-log curvature, and monotonicity; this information lets atom instances reason about whether or not they are DCP or DGP. See the *Atomic Functions* page for a compact, accessible summary of each atom's attributes.

4.1.1 Affine Atoms

All of the atoms listed here are affine in their arguments.

AddExpression

class cvxpy.atoms.affine.add_expr.**AddExpression**(*arg_groups*)

Bases: cvxpy.atoms.affine.affine_atom.AffAtom

The sum of any number of expressions.

MulExpression

class cvxpy.atoms.affine.binary_operators.**MulExpression**(*lh_exp*, *rh_exp*)

Bases: cvxpy.atoms.affine.binary_operators.BinaryOperator

Matrix multiplication.

The semantics of multiplication are exactly as those of NumPy's matmul function, except here multiplication by a scalar is permitted. MulExpression objects can be created by using the '*' operator of the Expression class.

Parameters

- **lh_exp** (Expression) – The left-hand side of the multiplication.

- **rh_exp** (Expression) – The right-hand side of the multiplication.

DivExpression

class cvxpy.atoms.affine.binary_operators.**DivExpression**(*lh_exp*, *rh_exp*)
Bases: cvxpy.atoms.affine.binary_operators.BinaryOperator

Division by scalar.

Can be created by using the / operator of expression.

Bmat

cvxpy.atoms.affine.bmat.**bmat**(*block_lists*)
Constructs a block matrix.

Takes a list of lists. Each internal list is stacked horizontally. The internal lists are stacked vertically.

Parameters **block_lists** (*list of lists*) – The blocks of the block matrix.

Returns The CVXPY expression representing the block matrix.

Return type CVXPY expression

conv

class cvxpy.atoms.affine.conv.**conv**(*lh_expr*, *rh_expr*)
Bases: cvxpy.atoms.affine.affine_atom.AffAtom

1D discrete convolution of two vectors.

The discrete convolution c of vectors a and b of lengths n and m, respectively, is a length-$(n+m-1)$ vector where

$$c_k = \sum_{i+j=k} a_i b_j, \quad k = 0, \ldots, n+m-2.$$

Parameters

- **lh_expr** (Constant) – A constant 1D vector or a 2D column vector.
- **rh_expr** (Expression) – A 1D vector or a 2D column vector.

cumsum

class cvxpy.atoms.affine.cumsum.**cumsum**(*expr*, *axis=0*)
Bases: cvxpy.atoms.affine.affine_atom.AffAtom, cvxpy.atoms.axis_atom.AxisAtom

Cumulative sum.

expr
The expression being summed.

Type CVXPY expression

axis
The axis to sum across if 2D.

Type int

diag

cvxpy.atoms.affine.diag.**diag**(*expr*)

Extracts the diagonal from a matrix or makes a vector a diagonal matrix.

Parameters **expr** (*Expression or numeric constant*) – A vector or square matrix.

Returns An Expression representing the diagonal vector/matrix.

Return type *Expression*

diff

cvxpy.atoms.affine.diff.**diff**(*x*, *k=1*, *axis=0*)

Vector of kth order differences.

Takes in a vector of length n and returns a vector of length n-k of the kth order differences.

diff(x) returns the vector of differences between adjacent elements in the vector, that is

[x[2] - x[1], x[3] - x[2], . . .]

diff(x, 2) is the second-order differences vector, equivalently diff(diff(x))

diff(x, 0) returns the vector x unchanged

hstack

cvxpy.atoms.affine.hstack.**hstack**(*arg_list*)

Horizontal concatenation of an arbitrary number of Expressions.

Parameters **arg_list** (*list of Expression*) – The Expressions to concatenate.

index

class cvxpy.atoms.affine.index.**index**(*expr*, *key*, *orig_key=None*)

Bases: cvxpy.atoms.affine.affine_atom.AffAtom

Indexing/slicing into an Expression.

CVXPY supports NumPy-like indexing semantics via the Expression class' overloading of the `[]` operator. This is a low-level class constructed by that operator, and it should not be instantiated directly.

Parameters

- **expr** (Expression) – The expression indexed/sliced into.
- **key** – The index/slicing key (i.e. expr[key[0],key[1]]).

kron

class cvxpy.atoms.affine.kron.**kron**(*lh_expr*, *rh_expr*)

Bases: cvxpy.atoms.affine.affine_atom.AffAtom

Kronecker product.

matmul

cvxpy.atoms.affine.binary_operators.**matmul**(*lh_exp*, *rh_exp*)
Matrix multipliction.

multiply

class cvxpy.atoms.affine.binary_operators.**multiply**(*lh_expr*, *rh_expr*)
Bases: *cvxpy.atoms.affine.binary_operators.MulExpression*

Multiplies two expressions elementwise.

promote

cvxpy.atoms.affine.promote.**promote**(*expr*, *shape*)
Promote a scalar expression to a vector/matrix.

Parameters

- **expr** (Expression) – The expression to promote.
- **shape** (*tuple*) – The shape to promote to.

Raises ValueError – If expr is not a scalar.

reshape

class cvxpy.atoms.affine.reshape.**reshape**(*expr*, *shape*)
Bases: cvxpy.atoms.affine.affine_atom.AffAtom

Reshapes the expression.

Vectorizes the expression then unvectorizes it into the new shape. The entries are reshaped and stored in column-major order, also known as Fortran order.

Parameters

- **expr** (Expression) – The expression to promote.
- **shape** (*tuple*) – The shape to promote to.

sum

cvxpy.atoms.affine.sum.**sum**(*expr*, *axis=None*, *keepdims=False*)
Sum the entries of an expression.

Parameters

- **expr** (Expression) – The expression to sum the entries of.
- **axis** (*int*) – The axis along which to sum.
- **keepdims** (*bool*) – Whether to drop dimensions after summing.

trace

class cvxpy.atoms.affine.trace.**trace**(*expr*)
Bases: cvxpy.atoms.affine.affine_atom.AffAtom

The sum of the diagonal entries of a matrix.

Parameters **expr** (Expression) – The expression to sum the diagonal of.

transpose

class cvxpy.atoms.affine.transpose.**transpose**(*expr*, *axes=None*)
Bases: cvxpy.atoms.affine.affine_atom.AffAtom

Transpose an expression.

NegExpression

class cvxpy.atoms.affine.unary_operators.**NegExpression**(*expr*)
Bases: cvxpy.atoms.affine.unary_operators.UnaryOperator

Negation of an expression.

upper_tri

class cvxpy.atoms.affine.upper_tri.**upper_tri**(*expr*)
Bases: cvxpy.atoms.affine.affine_atom.AffAtom

The vectorized strictly upper triagonal entries.

vec

cvxpy.atoms.affine.vec.**vec**(*X*)
Flattens the matrix X into a vector in column-major order.

Parameters **X** (*Expression or numeric constant*) – The matrix to flatten.

Returns An Expression representing the flattened matrix.

Return type *Expression*

vstack

cvxpy.atoms.affine.vstack.**vstack**(*arg_list*)
Wrapper on vstack to ensure list argument.

4.1.2 cvxpy.atoms.elementwise package

All of the atoms listed here operate elementwise on expressions. For example, *exp* exponentiates each entry of expressions that are supplied to it.

abs

class cvxpy.atoms.elementwise.abs.**abs**(*x*)
Bases: cvxpy.atoms.elementwise.elementwise.Elementwise

Elementwise absolute value

entr

class cvxpy.atoms.elementwise.entr.**entr**(*x*)
Bases: cvxpy.atoms.elementwise.elementwise.Elementwise

Elementwise $-x \log x$.

exp

class cvxpy.atoms.elementwise.exp.**exp**(*x*)
Bases: cvxpy.atoms.elementwise.elementwise.Elementwise

Elementwise e^x.

huber

class cvxpy.atoms.elementwise.huber.**huber**(*x*, *M=1*)
Bases: cvxpy.atoms.elementwise.elementwise.Elementwise

The Huber function

$$\text{Huber}(x, M) = \begin{cases} 2M|x| - M^2 & \text{for } |x| \geq |M| \\ |x|^2 & \text{for } |x| \leq |M|. \end{cases}$$

M defaults to 1.

Parameters

- **x** (Expression) – The expression to which the huber function will be applied.
- **M** (Constant) – A scalar constant.

inv_pos

cvxpy.atoms.elementwise.inv_pos.**inv_pos**(*x*)
x^{-1} for $x > 0$.

kl_div

class cvxpy.atoms.elementwise.kl_div.**kl_div**(*x*, *y*)
Bases: cvxpy.atoms.elementwise.elementwise.Elementwise

$x \log(x/y) - x + y$

log

class cvxpy.atoms.elementwise.log.**log**(*x*)
Bases: cvxpy.atoms.elementwise.elementwise.Elementwise

Elementwise $\log x$.

log1p

class cvxpy.atoms.elementwise.log1p.**log1p**(*x*)
Bases: *cvxpy.atoms.elementwise.log.log*

Elementwise $\log(1 + x)$.

logistic

class cvxpy.atoms.elementwise.logistic.**logistic**(*x*)
Bases: cvxpy.atoms.elementwise.elementwise.Elementwise

$\log(1 + e^x)$

This is a special case of log(sum(exp)) that is evaluates to a vector rather than to a scalar which is useful for logistic regression.

maximum

class cvxpy.atoms.elementwise.maximum.**maximum**(*arg1*, *arg2*, **args*)
Bases: cvxpy.atoms.elementwise.elementwise.Elementwise

Elementwise maximum of a sequence of expressions.

minimum

cvxpy.atoms.elementwise.minimum.**minimum**(*arg1*, *arg2*, **args*)
Elementwise minimum of a sequence of expressions.

neg

cvxpy.atoms.elementwise.neg.**neg**(*x*)
Alias for -minimum{x, 0}.

pos

cvxpy.atoms.elementwise.pos.**pos**(*x*)
Alias for maximum{x,0}.

power

class cvxpy.atoms.elementwise.power.**power**(*x*, *p*, *max_denom=1024*)

Bases: cvxpy.atoms.elementwise.elementwise.Elementwise

Elementwise power function $f(x) = x^p$.

If `expr` is a CVXPY expression, then `expr**p` is equivalent to `power(expr, p)`.

Specifically, the atom is given by the cases

$p = 0$	$f(x) = 1$	constant, positive
$p = 1$	$f(x) = x$	affine, increasing, same sign as x
$p = 2, 4, 8, \ldots$	$f(x) = \lvert x\rvert^p$	convex, signed monotonicity, positive
$p < 0$	$f(x) = \begin{cases} x^p & x > 0 \\ +\infty & x \leq 0 \end{cases}$	convex, decreasing, positive
$0 < p < 1$	$f(x) = \begin{cases} x^p & x \geq 0 \\ -\infty & x < 0 \end{cases}$	concave, increasing, positive
$p > 1,\ p \neq 2, 4, 8, \ldots$	$f(x) = \begin{cases} x^p & x \geq 0 \\ +\infty & x < 0 \end{cases}$	convex, increasing, positive.

Note: Generally, `p` cannot be represented exactly, so a rational, i.e., fractional, **approximation** must be made.

Internally, `power` computes a rational approximation to `p` with a denominator up to `max_denom`. The resulting approximation can be found through the attribute `power.p`. The approximation error is given by the attribute `power.approx_error`. Increasing `max_denom` can give better approximations.

When `p` is an `int` or `Fraction` object, the approximation is usually **exact**.

Note: The final domain, sign, monotonicity, and curvature of the `power` atom are determined by the rational approximation to `p`, **not** the input parameter `p`.

For example,

```
>>> from cvxpy import Variable, power
>>> x = Variable()
>>> g = power(x, 1.001)
>>> g.p
Fraction(1001, 1000)
>>> g
Expression(CONVEX, POSITIVE, (1, 1))
```

results in a convex atom with implicit constraint $x \geq 0$, while

```
>>> g = power(x, 1.0001)
>>> g.p
1
>>> g
Expression(AFFINE, UNKNOWN, (1, 1))
```

results in an affine atom with no constraint on `x`.

- When $p > 1$ and `p` is not a power of two, the monotonically increasing version of the function with full domain,

$$f(x) = \begin{cases} x^p & x \geq 0 \\ 0 & x < 0 \end{cases}$$

  can be formed with the composition `power(pos(x), p)`.

- The symmetric version with full domain,

$$f(x) = |x|^p$$

  can be formed with the composition `power(abs(x), p)`.

  Parameters

  - **x** (*cvx.Variable*) –
  - **p** (*int, float, or Fraction*) – Scalar power.
  - **max_denom** (*int*) – The maximum denominator considered in forming a rational approximation of `p`.

scalene

`cvxpy.atoms.elementwise.scalene.`**scalene**(*x*, *alpha*, *beta*)
Alias for `alpha*pos(x) + beta*neg(x)`.

sqrt

`cvxpy.atoms.elementwise.sqrt.`**sqrt**(*x*)
The square root of an expression.

square

`cvxpy.atoms.elementwise.square.`**square**(*x*)
The square of an expression.

4.1.3 Other Atoms

The atoms listed here are neither affine nor elementwise.

- *cummax*
- *diff_pos*
- *eye_minus_inv*
- *geo_mean*
- *harmonic_mean*
- *lambda_max*
- *lambda_min*

- *lambda_sum_largest*
- *lambda_sum_smallest*
- *log_det*
- *log_sum_exp*
- *matrix_frac*
- *max*
- *min*
- *mixed_norm*
- *norm*
- *norm1*
- *norm2*
- *norm_inf*
- *normNuc*
- *one_minus_pos*
- *pf_eigenvalue*
- *pnorm*
- *Pnorm*
- *prod*
- *quad_form*
- *quad_over_lin*
- *resolvent*
- *sigma_max*
- *sum_largest*
- *sum_smallest*
- *sum_squares*
- *tv*

cummax

class cvxpy.atoms.cummax.**cummax**(*x*, *axis=None*)

Bases: cvxpy.atoms.axis_atom.AxisAtom

$\max_{i,j}\{X_{i,j}\}$.

diff_pos

eye_minus_inv

class cvxpy.atoms.eye_minus_inv.**eye_minus_inv**(*X*)

Bases: *cvxpy.atoms.atom.Atom*

The unity resolvent of a positive matrix, $(I - X)^{-1}$.

For an elementwise positive matrix X, this atom represents

$$(I - X)^{-1},$$

and it enforces the constraint that the spectral radius of X is at most 1.

This atom is log-log convex.

Parameters **X** (*cvxpy.Expression*) – A positive square matrix.

geo_mean

class cvxpy.atoms.geo_mean.**geo_mean**(*x*, *p=None*, *max_denom=1024*)

Bases: *cvxpy.atoms.atom.Atom*

The (weighted) geometric mean of vector `x`, with optional powers given by `p`:

$$(x_1^{p_1} \cdots x_n^{p_n})^{\frac{1}{\mathbf{1}^T p}}$$

The powers `p` can be a `list`, `tuple`, or `numpy.array` of nonnegative `int`, `float`, or `Fraction` objects with nonzero sum.

If not specified, `p` defaults to a vector of all ones, giving the **unweighted** geometric mean

$$x_1^{1/n} \cdots x_n^{1/n}.$$

The geometric mean includes an implicit constraint that $x_i \geq 0$ whenever $p_i > 0$. If $p_i = 0$, x_i will be unconstrained.

The only exception to this rule occurs when `p` has exactly one nonzero element, say, `p_i`, in which case `geo_mean(x, p)` is equivalent to `x_i` (without the nonnegativity constraint). A specific case of this is when $x \in \mathbf{R}^1$.

Note: Generally, `p` cannot be represented exactly, so a rational, i.e., fractional, **approximation** must be made.

Internally, `geo_mean` immediately computes an approximate normalized weight vector $w \approx p/\mathbf{1}^T p$ and the `geo_mean` atom is represented as

$$x_1^{w_1} \cdots x_n^{w_n},$$

where the elements of `w` are `Fraction` objects that sum to exactly 1.

The maximum denominator used in forming the rational approximation is given by `max_denom`, which defaults to 1024, but can be adjusted to modify the accuracy of the approximations.

The approximating `w` and the approximation error can be found through the attributes `geo_mean.w` and `geo_mean.approx_error`.

Examples

The weights `w` can be seen from the string representation of the `geo_mean` object, or through the `w` attribute.

```
>>> from cvxpy import Variable, geo_mean, Problem, Maximize
>>> x = Variable(3, name='x')
>>> print(geo_mean(x))
geo_mean(x, (1/3, 1/3, 1/3))
>>> g = geo_mean(x, [1, 2, 1])
>>> g.w
(Fraction(1, 4), Fraction(1, 2), Fraction(1, 4))
```

Floating point numbers with few decimal places can sometimes be represented exactly. The approximation error between `w` and `p/sum(p)` is given by the `approx_error` attribute.

```
>>> import numpy as np
>>> x = Variable(4, name='x')
>>> p = np.array([.12, .34, .56, .78])
>>> g = geo_mean(x, p)
>>> g.w
(Fraction(1, 15), Fraction(17, 90), Fraction(14, 45), Fraction(13, 30))
>>> g.approx_error
0.0
```

In general, the approximation is not exact.

```
>>> p = [.123, .456, .789, .001]
>>> g = geo_mean(x, p)
>>> g.w
(Fraction(23, 256), Fraction(341, 1024), Fraction(295, 512), Fraction(1, 1024))
>>> 1e-4 <= g.approx_error <= 1e-3
True
```

The weight vector `p` can contain combinations of `int`, `float`, and `Fraction` objects.

```
>>> from fractions import Fraction
>>> x = Variable(4, name='x')
>>> g = geo_mean(x, [.1, Fraction(1,3), 0, 2])
>>> print(g)
geo_mean(x, (3/73, 10/73, 0, 60/73))
>>> g.approx_error <= 1e-10
True
```

Sequences of `Fraction` and `int` powers can often be represented **exactly**.

```
>>> p = [Fraction(1,17), Fraction(4,9), Fraction(1,3), Fraction(25,153)]
>>> x = Variable(4, name='x')
>>> print(geo_mean(x, p))
geo_mean(x, (1/17, 4/9, 1/3, 25/153))
```

Terms with a zero power will not have an implicit nonnegativity constraint.

```
>>> p = [1, 0, 1]
>>> x = Variable(3, name='x')
>>> obj = Maximize(geo_mean(x,p))
>>> constr = [sum(x) <= 1, -1 <= x, x <= 1]
>>> val = Problem(obj, constr).solve()
>>> x = np.array(x.value).flatten()
>>> print(x)
[ 1. -1.  1.]
```

Parameters

- **x** (Variable) – A column or row vector whose elements we will take the geometric mean of.
- **p** (Sequence (list, tuple, ...) of int, float, or Fraction objects) – A vector of weights for the weighted geometric mean

  When p is a sequence of int and/or Fraction objects, w can often be an **exact** representation of the weights. An exact representation is sometimes possible when p has float elements with only a few decimal places.
- **max_denom** (*int*) – The maximum denominator to use in approximating p/sum(p) with geo_mean.w. If w is not an exact representation, increasing max_denom **may** offer a more accurate representation, at the cost of requiring more convex inequalities to represent the geometric mean.

w

A rational approximation of p/sum(p).

Type tuple of Fractions

approx_error

The error in approximating p/sum(p) with w, given by $\|p/\mathbf{1}^T p - w\|_\infty$

Type float

harmonic_mean

cvxpy.atoms.harmonic_mean.**harmonic_mean**(*x*)

The harmonic mean of x.

Parameters **x** (*Expression or numeric*) – The expression whose harmonic mean is to be computed. Must have positive entries.

Returns

$$\frac{n}{\left(\sum_{i=1}^n x_i^{-1}\right)},$$

where n is the length of x.

Return type *Expression*

lambda_max

class cvxpy.atoms.lambda_max.**lambda_max**(*A*)

Bases: *cvxpy.atoms.atom.Atom*

Maximum eigenvalue; $\lambda_{\max}(A)$.

lambda_min

cvxpy.atoms.lambda_min.**lambda_min**(*X*)

Minimum eigenvalue; $\lambda_{\min}(A)$.

lambda_sum_largest

cvxpy.atoms.lambda_sum_largest.**lambda_sum_largest**(*X*, *k*)
Sum of the largest k eigenvalues.

lambda_sum_smallest

cvxpy.atoms.lambda_sum_smallest.**lambda_sum_smallest**(*X*, *k*)
Sum of the largest k eigenvalues.

log_det

class cvxpy.atoms.log_det.**log_det**(*A*)
Bases: *cvxpy.atoms.atom.Atom*

$\log\det A$

log_sum_exp

class cvxpy.atoms.log_sum_exp.**log_sum_exp**(*x*, *axis=None*, *keepdims=False*)
Bases: cvxpy.atoms.axis_atom.AxisAtom

$\log \sum_i e^{x_i}$

matrix_frac

cvxpy.atoms.matrix_frac.**matrix_frac**(*X*, *P*)
tr X.T*P^-1*X

max

class cvxpy.atoms.max.**max**(*x*, *axis=None*, *keepdims=False*)
Bases: cvxpy.atoms.axis_atom.AxisAtom

$\max_{i,j}\{X_{i,j}\}$.

min

cvxpy.atoms.min.**min**(*x*, *axis=None*, *keepdims=False*)
$\min i, j\{X_{i,j}\}$.

mixed_norm

cvxpy.atoms.mixed_norm.**mixed_norm**(*X*, *p=2*, *q=1*)
Lp,q norm; $(\sum_k(\sum_l|x_{k,l}|^p)^{q/p})^{1/q}$.

Parameters

- **X** (*Expression or numeric constant*) – The matrix to take the l_{p,q} norm of.
- **p** (*int or str, optional*) – The type of inner norm.
- **q** (*int or str, optional*) – The type of outer norm.

Returns An Expression representing the mixed norm.

Return type *Expression*

norm

cvxpy.atoms.norm.**norm**(*x*, *p=2*, *axis=None*)

Wrapper on the different norm atoms.

Parameters

- **x** (*Expression or numeric constant*) – The value to take the norm of.
- **p** (*int or str, optional*) – The type of norm.

Returns An Expression representing the norm.

Return type *Expression*

norm1

class cvxpy.atoms.norm1.**norm1**(*expr*, *axis=None*, *keepdims=False*)

Bases: cvxpy.atoms.axis_atom.AxisAtom

norm2

norm_inf

class cvxpy.atoms.**norm_inf**(*expr*, *axis=None*, *keepdims=False*)

Bases: cvxpy.atoms.axis_atom.AxisAtom

normNuc

class cvxpy.atoms.norm_nuc.**normNuc**(*A*)

Bases: *cvxpy.atoms.atom.Atom*

Sum of the singular values.

one_minus_pos

class cvxpy.atoms.**one_minus_pos**(*x*)

Bases: *cvxpy.atoms.atom.Atom*

The difference $1 - x$ with domain $\{x : 0 < x < 1\}$.

This atom is log-log concave.

x [*Expression*] An Expression.

pf_eigenvalue

class cvxpy.atoms.pf_eigenvalue.**pf_eigenvalue**(*X*)
Bases: *cvxpy.atoms.atom.Atom*

The Perron-Frobenius eigenvalue of a positive matrix.

For an elementwise positive matrix X, this atom represents its spectral radius, i.e., the magnitude of its largest eigenvalue. Because X is positive, the spectral radius equals its largest eigenvalue, which is guaranteed to be positive.

This atom is log-log convex.

Parameters **X** (*cvxpy.Expression*) – A positive square matrix.

pnorm

cvxpy.atoms.pnorm.**pnorm**(*x*, *p=2*, *axis=None*, *keepdims=False*, *max_denom=1024*)
Factory function for a mathematical p-norm.

Parameters **p** (*numeric type or string*) – The type of norm to construct; set this to np.inf or 'inf' to construct an infinity norm.

Returns A norm1, norm_inf, or Pnorm object.

Return type *Atom*

Pnorm

class cvxpy.atoms.pnorm.**Pnorm**(*x*, *p=2*, *axis=None*, *keepdims=False*, *max_denom=1024*)
The vector p-norm, for p not equal to 1 or infinity.

If given a matrix variable, pnorm will treat it as a vector, and compute the p-norm of the concatenated columns. Only accepts p values that are not equal to 1 or infinity; the norm1 and norm_inf classes handle those norms.

For $p > 1$, the p-norm is given by

$$\|x\|_p = \left(\sum_i |x_i|^p\right)^{1/p},$$

with domain $x \in \mathbf{R}^n$.

For $p < 1,\ p \neq 0$, the p-norm is given by

$$\|x\|_p = \left(\sum_i x_i^p\right)^{1/p},$$

with domain $x \in \mathbf{R}^n_+$.

- Note that the "p-norm" is actually a **norm** only when $p > 1$. For these cases, it is convex.
- The expression is not defined when $p = 0$.
- Otherwise, when $p < 1$, the expression is concave, but it is not a true norm.

Note: Generally, p cannot be represented exactly, so a rational, i.e., fractional, **approximation** must be made.

Internally, pnorm computes a rational approximation to the reciprocal $1/p$ with a denominator up to max_denom. The resulting approximation can be found through the attribute pnorm.p. The approximation

error is given by the attribute `pnorm.approx_error`. Increasing `max_denom` can give better approximations.

When `p` is an `int` or `Fraction` object, the approximation is usually **exact**.

Parameters

- **x** (*cvxpy.Variable*) – The value to take the norm of.
- **p** (*int, float, or Fraction*) – We require that $p > 1$, but $p \neq \infty$. See the norm1 and norm_inf classes for these norms, or use the pnorm function wrapper to instantiate them.

max_denom [int] The maximum denominator considered in forming a rational approximation for `p`.

axis [0 or 1] The axis to apply the norm to.

Returns An Expression representing the norm.

Return type *Expression*

prod

cvxpy.atoms.prod.**prod**(*expr*, *axis=None*, *keepdims=False*)

Multiply the entries of an expression.

The semantics of this atom are the same as np.prod.

This atom is log-log affine, but it is neither convex nor concave.

Parameters

- **expr** (*Expression or list[Expression, Numeric]*) – The expression to multiply the entries of, or a list of Expressions and numeric types.
- **axis** (*int*) – The axis along which to take the product; ignored if *expr* is a list.
- **keepdims** (*bool*) – Whether to drop dimensions after taking the product; ignored if *expr* is a list.

quad_form

cvxpy.atoms.quad_form.**quad_form**(*x*, *P*)

Alias for $x^T P x$.

quad_over_lin

class cvxpy.atoms.quad_over_lin.**quad_over_lin**(*x*, *y*)

Bases: *cvxpy.atoms.atom.Atom*

$(sum_{ij} X_{ij}^2)/y$

resolvent

cvxpy.atoms.eye_minus_inv.**resolvent**(*X*, *s*)
The resolvent of a positive matrix, $(sI - X)^{-1}$.

For an elementwise positive matrix X and a positive scalar s, this atom computes

$$(sI - X)^{-1},$$

and it enforces the constraint that the spectral radius of X/s is at most 1.

This atom is log-log convex.

Parameters

- **X** (*cvxpy.Expression*) – A positive square matrix.
- **s** (*cvxpy.Expression or numeric*) – A positive scalar.

sigma_max

class cvxpy.atoms.sigma_max.**sigma_max**(*A*)
Bases: *cvxpy.atoms.atom.Atom*

Maximum singular value.

sum_largest

class cvxpy.atoms.sum_largest.**sum_largest**(*x*, *k*)
Bases: *cvxpy.atoms.atom.Atom*

Sum of the largest k values in the matrix X.

sum_smallest

cvxpy.atoms.sum_smallest.**sum_smallest**(*x*, *k*)
Sum of the smallest k values.

sum_squares

cvxpy.atoms.sum_squares.**sum_squares**(*expr*)
The sum of the squares of the entries.

Parameters **expr** (Expression) – The expression to take the sum of squares of.

Returns An expression representing the sum of squares.

Return type *Expression*

tv

cvxpy.atoms.total_variation.**tv**(*value*, **args*)
Total variation of a vector, matrix, or list of matrices.

Uses L1 norm of discrete gradients for vectors and L2 norm of discrete gradients for matrices.

Parameters

- **value** (*Expression or numeric constant*) – The value to take the total variation of.
- **args** (*Matrix constants/expressions*) – Additional matrices extending the third dimension of value.

Returns An Expression representing the total variation.

Return type *Expression*

4.1.4 Atom

class cvxpy.atoms.atom.**Atom**(**args*)

Bases: *cvxpy.expressions.expression.Expression*

Abstract base class for atoms.

domain
A list of constraints describing the closure of the region where the expression is finite.

grad
Gives the (sub/super)gradient of the expression w.r.t. each variable.

Matrix expressions are vectorized, so the gradient is a matrix. None indicates variable values unknown or outside domain.

Returns A map of variable to SciPy CSC sparse matrix or None.

is_atom_affine()
Is the atom affine?

is_atom_concave()
Is the atom concave?

is_atom_convex()
Is the atom convex?

is_atom_log_log_affine()
Is the atom log-log affine?

is_atom_log_log_concave()
Is the atom log-log concave?

is_atom_log_log_convex()
Is the atom log-log convex?

is_decr(*idx*)
Is the composition non-increasing in argument idx?

is_incr(*idx*)
Is the composition non-decreasing in argument idx?

4.2 Constraints

A constraint is an equality or inequality that restricts the domain of an optimization problem. CVXPY has five types of constraints: non-positive, equality or zero, positive semidefinite, second-order cone, and exponential cone. The *vast* majority of users will need only create constraints of the first three types. Additionally, most users need not know anything more about constraints other than how to create them. The constraint APIs do nonetheless provide methods

that advanced users may find useful; for example, some of the APIs allow you to inspect dual variable values and residuals.

4.2.1 Constraint

class cvxpy.constraints.constraint.**Constraint**(*args*, *constr_id=None*)

Bases: cvxpy.utilities.canonical.Canonical

The base class for constraints.

A constraint is an equality, inequality, or more generally a generalized inequality that is imposed upon a mathematical expression or a list of thereof.

Parameters

- **args** (*list*) – A list of expression trees.
- **constr_id** (*int*) – A unique id for the constraint.

is_dcp()

Checks whether the constraint is DCP.

Returns True if the constraint is DCP, False otherwise.

Return type bool

value(*tolerance=1e-08*)

Checks whether the constraint violation is less than a tolerance.

Parameters **tolerance** (*float*) – The absolute tolerance to impose on the violation.

Returns True if the violation is less than tolerance, False otherwise.

Return type bool

Raises ValueError – If the constrained expression does not have a value associated with it.

violation()

The numeric residual of the constraint.

The violation is defined as the distance between the constrained expression's value and its projection onto the domain of the constraint:

$$||\Pi(v) - v||_2^2$$

where v is the value of the constrained expression and Π is the projection operator onto the constraint's domain .

Returns The residual value.

Return type NumPy.ndarray

Raises `ValueError` – If the constrained expression does not have a value associated with it.

4.2.2 NonPos

class cvxpy.constraints.nonpos.**NonPos**(*expr*, *constr_id=None*)

Bases: *cvxpy.constraints.constraint.Constraint*

A constraint of the form $x \leq 0$.

The preferred way of creating a `NonPos` constraint is through operator overloading. To constrain an expression `x` to be non-positive, simply write `x <= 0`; to constrain `x` to be non-negative, write `x >= 0`. The former creates a `NonPos` constraint with `x` as its argument, while the latter creates one with `-x` as its argument. Strict inequalities are not supported, as they do not make sense in a numerical setting.

Parameters

- **expr** (`Expression`) – The expression to constrain.
- **constr_id** (*int*) – A unique id for the constraint.

dual_value

The value of the dual variable.

Type NumPy.ndarray

is_dcp()

A non-positive constraint is DCP if its argument is convex.

shape

The shape of the constrained expression.

Type int

size

The size of the constrained expression.

Type int

value(*tolerance=1e-08*)

Checks whether the constraint violation is less than a tolerance.

Parameters **tolerance** (*float*) – The absolute tolerance to impose on the violation.

Returns True if the violation is less than `tolerance`, False otherwise.

Return type bool

Raises `ValueError` – If the constrained expression does not have a value associated with it.

violation()

The numeric residual of the constraint.

The violation is defined as the distance between the constrained expression's value and its projection onto the domain of the constraint:

$$||\Pi(v) - v||_2^2$$

where v is the value of the constrained expression and Π is the projection operator onto the constraint's domain .

Returns The residual value.

Return type NumPy.ndarray

Raises `ValueError` – If the constrained expression does not have a value associated with it.

4.2.3 Zero

class cvxpy.constraints.zero.**Zero**(*expr*, *constr_id=None*)
Bases: *cvxpy.constraints.constraint.Constraint*

A constraint of the form $x = 0$.

The preferred way of creating a `Zero` constraint is through operator overloading. To constrain an expression `x` to be zero, simply write `x == 0`. The former creates a `Zero` constraint with `x` as its argument.

is_dcp()
A zero constraint is DCP if its argument is affine.

value(*tolerance=1e-08*)
Checks whether the constraint violation is less than a tolerance.

Parameters **tolerance** (*float*) – The absolute tolerance to impose on the violation.

Returns True if the violation is less than `tolerance`, False otherwise.

Return type bool

Raises `ValueError` – If the constrained expression does not have a value associated with it.

violation()
The numeric residual of the constraint.

The violation is defined as the distance between the constrained expression's value and its projection onto the domain of the constraint:

$$||\Pi(v) - v||_2^2$$

where v is the value of the constrained expression and Π is the projection operator onto the constraint's domain .

Returns The residual value.

Return type NumPy.ndarray

Raises `ValueError` – If the constrained expression does not have a value associated with it.

4.2.4 PSD

class cvxpy.constraints.psd.**PSD**(*expr*, *constr_id=None*)
Bases: *cvxpy.constraints.constraint.Constraint*

A constraint of the form $\frac{1}{2}(X + X^T) \succcurlyeq_{S_n^+} 0$

Applying a `PSD` constraint to a two-dimensional expression `X` constrains its symmetric part to be positive semidefinite: i.e., it constrains `X` to be such that

$$z^T(X + X^T)z \geq 0,$$

for all z.

The preferred way of creating a `PSD` constraint is through operator overloading. To constrain an expression `X` to be PSD, write `X >> 0`; to constrain it to be negative semidefinite, write `X << 0`. Strict definiteness constraints are not provided, as they do not make sense in a numerical setting.

Parameters

- **expr** (*Expression.*) – The expression to constrain; *must* be two-dimensional.
- **constr_id** (*int*) – A unique id for the constraint.

is_dcp()
A PSD constraint is DCP if the constrained expression is affine.

value(*tolerance=1e-08*)
Checks whether the constraint violation is less than a tolerance.

Parameters **tolerance** (*float*) – The absolute tolerance to impose on the violation.

Returns True if the violation is less than `tolerance`, False otherwise.

Return type bool

Raises `ValueError` – If the constrained expression does not have a value associated with it.

violation()
The numeric residual of the constraint.

The violation is defined as the distance between the constrained expression's value and its projection onto the domain of the constraint:

$$||\Pi(v) - v||_2^2$$

where v is the value of the constrained expression and Π is the projection operator onto the constraint's domain .

Returns The residual value.

Return type NumPy.ndarray

Raises `ValueError` – If the constrained expression does not have a value associated with it.

4.2.5 SOC

class cvxpy.constraints.second_order.**SOC**(*t*, *X*, *axis=0*, *constr_id=None*)
Bases: *cvxpy.constraints.constraint.Constraint*

A second-order cone constraint for each row/column.

Assumes `t` is a vector the same length as `X`'s columns (rows) for `axis == 0` (`1`).

t
The scalar part of the second-order constraint.

X
A matrix whose rows/columns are each a cone.

axis
Slice by column 0 or row 1.

is_dcp()
An SOC constraint is DCP if each of its arguments is affine.

value(*tolerance=1e-08*)
Checks whether the constraint violation is less than a tolerance.

Parameters **tolerance** (*float*) – The absolute tolerance to impose on the violation.

Returns True if the violation is less than `tolerance`, False otherwise.

Return type bool

Raises `ValueError` – If the constrained expression does not have a value associated with it.

violation()
The numeric residual of the constraint.

The violation is defined as the distance between the constrained expression's value and its projection onto the domain of the constraint:

$$||\Pi(v) - v||_2^2$$

where v is the value of the constrained expression and Π is the projection operator onto the constraint's domain .

Returns The residual value.

Return type NumPy.ndarray

Raises `ValueError` – If the constrained expression does not have a value associated with it.

4.2.6 ExpCone

class cvxpy.constraints.exponential.**ExpCone**(*x*, *y*, *z*, *constr_id=None*)
Bases: cvxpy.constraints.nonlinear.NonlinearConstraint

A reformulated exponential cone constraint.

Operates elementwise on x, y, z.

Original cone:

$$K = \{(x, y, z) \mid y > 0, ye^{x/y} <= z\} \cup \{(x, y, z) \mid x \leq 0, y = 0, z \geq 0\}$$

Reformulated cone:

$$K = \{(x, y, z) \mid y, z > 0, y\log(y) + x \leq y\log(z)\} \cup \{(x, y, z) \mid x \leq 0, y = 0, z \geq 0\}$$

Parameters

- **x** (Variable) – x in the exponential cone.
- **y** (Variable) – y in the exponential cone.
- **z** (Variable) – z in the exponential cone.

is_dcp()
An exponential constraint is DCP if each argument is affine.

value(*tolerance=1e-08*)
Checks whether the constraint violation is less than a tolerance.

Parameters **tolerance** (*float*) – The absolute tolerance to impose on the violation.

Returns True if the violation is less than `tolerance`, False otherwise.

Return type bool

Raises `ValueError` – If the constrained expression does not have a value associated with it.

violation()
The numeric residual of the constraint.

The violation is defined as the distance between the constrained expression's value and its projection onto the domain of the constraint:

$$||\Pi(v) - v||_2^2$$

where v is the value of the constrained expression and Π is the projection operator onto the constraint's domain .

Returns The residual value.

Return type NumPy.ndarray

Raises `ValueError` – If the constrained expression does not have a value associated with it.

4.3 Expressions

CVXPY represents mathematical objects as *expression trees*. An expression tree is a collection of mathematical expressions linked together by one or more atoms. Expression trees are encoded as instances of the *Expression* class, and each *Leaf* in a tree is a *Variable*, *Parameter*, or `Constant`.

- *Expression*
- *Leaf*
- *Variable*
- *Parameter*
- *Constant*

4.3.1 Expression

class cvxpy.expressions.expression.**Expression**
Bases: cvxpy.utilities.canonical.Canonical

A mathematical expression in a convex optimization problem.

Overloads many operators to allow for convenient creation of compound expressions (e.g., the sum of two expressions) and constraints.

T
The transpose of the expression.

Type *Expression*

__add__(*other*)
Expression : Sum two expressions.

__div__(*other*)
Expression : One expression divided by another.

__eq__(*other*)
Equality : Creates a constraint `self == other`.

__ge__(*other*)
NonPos : Creates an inequality constraint.

__le__(*other*)
Inequality : Creates an inequality constraint `self <= other`.

__lshift__(*other*)
PSD : Creates a negative semidefinite inequality.

__matmul__(*other*)
Expression : Matrix multiplication of two expressions.

__mul__(*other*)
Expression : The product of two expressions.

__pow__(*power*)
Raise expression to a power.

Parameters **power** (*float*) – The power to which to raise the expression.

Returns The expression raised to `power`.

Return type *Expression*

__radd__(*other*)
Expression : Sum two expressions.

__rdiv__(*other*)
Expression : Called for Number / Expression.

__rlshift__(*other*)
PSD : Creates a negative semidefinite inequality.

__rmatmul__(*other*)
Expression : Called for matrix @ Expression.

__rmul__(*other*)
Expression : Called for Number * Expression.

__rrshift__(*other*)
PSD : Creates a positive semidefinite inequality.

__rshift__(*other*)
PSD : Creates a positive semidefinite inequality.

__rsub__(*other*)
Expression : The difference of two expressions.

__rtruediv__(*other*)
Expression : Called for Number / Expression.

__sub__(*other*)
Expression : The difference of two expressions.

__truediv__(*other*)
Expression : One expression divided by another.

curvature
The curvature of the expression.

Type str

domain
The constraints describing the closure of the region where the expression is finite.

Type list

grad
Gives the (sub/super)gradient of the expression w.r.t. each variable.

Matrix expressions are vectorized, so the gradient is a matrix.

Returns A map of variable to SciPy CSC sparse matrix; None if a variable value is missing.

Return type dict

is_affine()
Is the expression affine?

is_concave()
Is the expression concave?

is_constant()
Is the expression constant?

is_convex()
Is the expression convex?

is_dcp()
Checks whether the Expression is DCP.

Returns True if the Expression is DCP, False otherwise.

Return type bool

is_dgp()
Checks whether the Expression is log-log DCP.

Returns True if the Expression is log-log DCP, False otherwise.

Return type bool

is_log_log_affine()
Is the expression affine?

is_log_log_concave()
Is the expression log-log concave?

is_log_log_convex()
Is the expression log-log convex?

is_nonneg()
Is the expression positive?

is_nonpos()
Is the expression negative?

is_zero()
Is the expression all zero?

name()
str : The string representation of the expression.

ndim
The number of dimensions in the expression's shape.

Type int

shape
The expression dimensions.

Type tuple

sign
The sign of the expression.

Type str

size
The number of entries in the expression.

Type int

value
The numeric value of the expression.

Type NumPy.ndarray or None

4.3.2 Leaf

class cvxpy.expressions.leaf.**Leaf**(*shape*, *value=None*, *nonneg=False*, *nonpos=False*, *complex=False*, *imag=False*, *symmetric=False*, *diag=False*, *PSD=False*, *NSD=False*, *hermitian=False*, *boolean=False*, *integer=False*, *sparsity=None*, *pos=False*, *neg=False*)

Bases: *cvxpy.expressions.expression.Expression*

A leaf node of an expression tree; i.e., a Variable, Constant, or Parameter.

A leaf may carry *attributes* that constrain the set values permissible for it. Leafs can have no more than one attribute, with the exception that a leaf may be both `nonpos` and `nonneg` or both `boolean` in some indices and `integer` in others.

An error is raised if a leaf is assigned a value that contradicts one or more of its attributes. See the `project` method for a convenient way to project a value onto a leaf's domain.

Parameters

- **shape** (*tuple or int*) – The leaf dimensions. Either an integer n for a 1D shape, or a tuple where the semantics are the same as NumPy ndarray shapes. **Shapes cannot be more than 2D**.
- **value** (*numeric type*) – A value to assign to the leaf.
- **nonneg** (*bool*) – Is the variable constrained to be nonnegative?
- **nonpos** (*bool*) – Is the variable constrained to be nonpositive?
- **complex** (*bool*) – Is the variable complex valued?
- **symmetric** (*bool*) – Is the variable symmetric?
- **diag** (*bool*) – Is the variable diagonal?
- **PSD** (*bool*) – Is the variable constrained to be positive semidefinite?
- **NSD** (*bool*) – Is the variable constrained to be negative semidefinite?
- **Hermitian** (*bool*) – Is the variable Hermitian?
- **boolean** (*bool or list of tuple*) – Is the variable boolean? True, which constrains the entire Variable to be boolean, False, or a list of indices which should be constrained as boolean, where each index is a tuple of length exactly equal to the length of shape.

- **integer** (*bool or list of tuple*) – Is the variable integer? The semantics are the same as the boolean argument.
- **sparsity** (*list of tuplewith*) – Fixed sparsity pattern for the variable.
- **pos** (*bool*) – Is the variable positive?
- **neg** (*bool*) – Is the variable negative?

T
The transpose of the expression.

Type *Expression*

ndim
The number of dimensions in the expression's shape.

Type int

project(*val*)
Project value onto the attribute set of the leaf.

A sensible idiom is `leaf.value = leaf.project(val)`.

Parameters **val** (*numeric type*) – The value assigned.

Returns The value rounded to the attribute type.

Return type numeric type

project_and_assign(*val*)
Project and assign a value to the variable.

shape
The dimensions of the expression.

Type tuple

size
The number of entries in the expression.

Type int

value
The numeric value of the parameter.

Type NumPy.ndarray or None

4.3.3 Variable

class cvxpy.expressions.variable.**Variable**(*shape=()*, *name=None*, *var_id=None*, ***kwargs*)

Bases: *cvxpy.expressions.leaf.Leaf*

The optimization variables in a problem.

T
The transpose of the expression.

Type *Expression*

name()
str : The name of the variable.

ndim
The number of dimensions in the expression's shape.

Type int

project(*val*)
Project value onto the attribute set of the leaf.

A sensible idiom is `leaf.value = leaf.project(val)`.

Parameters **val** (*numeric type*) – The value assigned.

Returns The value rounded to the attribute type.

Return type numeric type

project_and_assign(*val*)
Project and assign a value to the variable.

shape
The dimensions of the expression.

Type tuple

size
The number of entries in the expression.

Type int

value
The numeric value of the parameter.

Type NumPy.ndarray or None

4.3.4 Parameter

class cvxpy.expressions.constants.parameter.**Parameter**(*shape=()*, *name=None*, *value=None*, ***kwargs*)

Bases: *cvxpy.expressions.leaf.Leaf*

Parameters in optimization problems.

Parameters are constant expressions whose value may be specified after problem creation. The only way to modify a problem after its creation is through parameters. For example, you might choose to declare the hyperparameters of a machine learning model to be Parameter objects; more generally, Parameters are useful for computing trade-off curves.

T
The transpose of the expression.

Type *Expression*

ndim
The number of dimensions in the expression's shape.

Type int

project(*val*)
Project value onto the attribute set of the leaf.

A sensible idiom is `leaf.value = leaf.project(val)`.

Parameters **val** (*numeric type*) – The value assigned.

Returns The value rounded to the attribute type.

Return type numeric type

project_and_assign(*val*)
Project and assign a value to the variable.

shape
The dimensions of the expression.

Type tuple

size
The number of entries in the expression.

Type int

value
The numeric value of the parameter.

Type NumPy.ndarray or None

4.3.5 Constant

class cvxpy.expressions.constants.**Constant**(*value*)
Bases: *cvxpy.expressions.leaf.Leaf*

A constant value.

Raw numerical constants (Python primite types, NumPy ndarrays, and NumPy matrices) are implicitly cast to constants via Expression operator overloading. For example, if `x` is an expression and `c` is a raw constant, then `x + c` creates an expression by casting `c` to a Constant.

T
The transpose of the expression.

Type *Expression*

ndim
The number of dimensions in the expression's shape.

Type int

shape
Returns the (row, col) dimensions of the expression.

size
The number of entries in the expression.

Type int

value
The numeric value of the constant.

Type NumPy.ndarray or None

4.4 Problems

The *Problem* class is the entry point to specifying and solving optimization problems. Each *cvxpy.problems.problem.Problem* instance encapsulates an optimization problem, i.e., an objective and a set of constraints, and the *solve()* method either solves the problem encoded by the instance, returning the optimal value and setting

variables values to optimal points, or reports that the problem was in fact infeasible or unbounded. You can construct a problem, solve it, and inspect both its value and the values of its variables like so:

```
problem = Problem(Minimize(expression), constraints)
problem.solve()
if problem.status not in ["infeasible", "unbounded"]:
    # Otherwise, problem.value is inf or -inf, respectively.
    print "Optimal value: %s" % problem.value
for variable in problem.variables():
    print "Variable %s: value %s" % (variable.name(), variable.value)
```

Problems are **immutable**, except through the specification of *Parameter* values. This means that you **cannot** modify a problem's objective or constraints after you have created it. If you find yourself wanting to add a constraint to an existing problem, you should instead create a new problem using, for example, the following idiom:

```
problem = Problem(Minimize(expression), constraints)
problem = Problem(problem.objective, problem.constraints + new_constraints)
```

Most users need not know anything about the *Problem* class except how to instantiate it, how to solve problem instances (*solve()*), and how to query the solver results (*status* and *value*).

Information about the size of a problem instance and statistics about the most recent solve invocation are captured by the *SizeMetrics* and *SolverStats* classes, respectively, and can be accessed via the *size_metrics()* and *solver_stats()* properties of the *Problem* class.

4.4.1 Minimize

class cvxpy.problems.objective.**Minimize**(*expr*)

Bases: cvxpy.problems.objective.Objective

An optimization objective for minimization.

Parameters **expr** (Expression) – The expression to minimize. Must be a scalar.

Raises ValueError – If expr is not a scalar.

is_dcp()

The objective must be convex.

is_dgp()

The objective must be log_log_convex.

4.4.2 Maximize

class cvxpy.problems.objective.**Maximize**(*expr*)

Bases: cvxpy.problems.objective.Objective

An optimization objective for maximization.

Parameters **expr** (Expression) – The expression to maximize. Must be a scalar.

Raises ValueError – If expr is not a scalar.

is_dcp()
The objective must be concave.

is_dgp()
The objective must be log-log concave.

4.4.3 SizeMetrics

class cvxpy.problems.problem.**SizeMetrics**(*problem*)
Bases: object

Reports various metrics regarding the problem.

num_scalar_variables
The number of scalar variables in the problem.

Type integer

num_scalar_data
The number of scalar constants and parameters in the problem. The number of constants used across all matrices, vectors, in the problem. Some constants are not apparent when the problem is constructed: for example, The sum_squares expression is a wrapper for a quad_over_lin expression with a constant 1 in the denominator.

Type integer

num_scalar_eq_constr
The number of scalar equality constraints in the problem.

Type integer

num_scalar_leq_constr
The number of scalar inequality constraints in the problem.

Type integer

max_data_dimension
The longest dimension of any data block constraint or parameter.

Type integer

max_big_small_squared
The maximum value of (big)(small)^2 over all data blocks of the problem, where (big) is the larger dimension and (small) is the smaller dimension for each data block.

Type integer

4.4.4 SolverStats

class cvxpy.problems.problem.**SolverStats**(*results_dict*, *solver_name*)
Bases: object

Reports some of the miscellaneous information that is returned by the solver after solving but that is not captured directly by the Problem instance.

solve_time
The time (in seconds) it took for the solver to solve the problem.

Type double

setup_time
The time (in seconds) it took for the solver to setup the problem.

Type double

num_iters
The number of iterations the solver had to go through to find a solution.

Type int

4.4.5 Problem

class cvxpy.problems.problem.**Problem**(*objective*, *constraints=None*)
Bases: cvxpy.utilities.canonical.Canonical

A convex optimization problem.

Problems are immutable, save for modification through the specification of Parameter

Parameters

- **objective** (Minimize *or* Maximize) – The problem's objective.
- **constraints** (*list*) – The constraints on the problem variables.

atoms()
Accessor method for atoms.

Returns A list of the atom types in the problem; note that this list contains classes, not instances.

Return type list of Atom

constants()
Accessor method for parameters.

Returns A list of the constants in the problem.

Return type list of Constant

constraints
A shallow copy of the problem's constraints.

Note that constraints cannot be reassigned, appended to, or otherwise modified after creation, except through parameters.

get_problem_data(*solver*)
Returns the problem data used in the call to the solver.

When a problem is solved, a chain of reductions, called a *SolvingChain*, compiles it to some low-level representation that is compatible with the targeted solver. This method returns that low-level representation.

For some solving chains, this low-level representation is a dictionary that contains exactly those arguments that were supplied to the solver; however, for other solving chains, the data is an intermediate representation that is compiled even further by libraries other than CVXPY.

A solution to the equivalent low-level problem can be obtained via the data by invoking the solve_via_data method of the returned solving chain, a thin wrapper around the code external to CVXPY that further

processes and solves the problem. Invoke the unpack_results method to recover a solution to the original problem.

Parameters **solver** (*str*) – The solver the problem data is for.

Returns

- *dict or object* – lowest level representation of problem
- *SolvingChain* – The solving chain that created the data.
- *list* – The inverse data generated by the chain.

is_dcp()
Does the problem satisfy DCP rules?

is_dgp()
Does the problem satisfy DGP rules?

is_qp()
Is problem a quadratic program?

objective
The problem's objective.

Note that the objective cannot be reassigned after creation, and modifying the objective after creation will result in undefined behavior.

Type *Minimize* or *Maximize*

parameters()
Accessor method for parameters.

Returns A list of the parameters in the problem.

Return type list of *Parameter*

classmethod register_solve(*name*, *func*)
Adds a solve method to the Problem class.

Parameters

- **name** (*str*) – The keyword for the method.
- **func** (*function*) – The function that executes the solve method. This function must take as its first argument the problem instance to solve.

size_metrics
Information about the problem's size.

Type *SizeMetrics*

solve(**args*, ***kwargs*)
Solves the problem using the specified method.

Populates the *.status', '.value*

Parameters

- **solver** (*str, optional*) – The solver to use. For example, 'ECOS', 'SCS', or 'OSQP'.
- **verbose** (*bool, optional*) – Overrides the default of hiding solver output.
- **gp** (*bool, optional*) – If True, parses the problem as a disciplined geometric program instead of a disciplined convex program.

- **solver_specific_opts** (*dict, optional*) – A dict of options that will be passed to the specific solver. In general, these options will override any default settings imposed by cvxpy.
- **method** (*function, optional*) – A custom solve method to use.

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

Return type float

Raises

- DCPError – Raised if the problem is not DCP and *gp* is False.
- DGPError – Raised if the problem is not DGP and *gp* is True.
- SolverError – Raised if no suitable solver exists among the installed solvers, or if an unanticipated error is encountered.

solver_stats
Information returned by the solver.

Type *SolverStats*

status
The status from the last time the problem was solved; one of optimal, infeasible, or unbounded.

Type str

unpack_results(*solution*, *chain*, *inverse_data*)
Updates the problem state given the solver results.

Updates problem.status, problem.value and value of primal and dual variables.

Parameters

- **solution** (*object*) – The solution returned by applying the chain to the problem and invoking the solver on the resulting data.
- **chain** (SolvingChain) – A solving chain that was used to solve the problem.
- **inverse_data** (*list*) – The inverse data returned by applying the chain to the problem.

value
The value from the last time the problem was solved (or None if not solved).

Type float

variables()
Accessor method for variables.

Returns A list of the variables in the problem.

Return type list of *Variable*

4.5 Reductions

A *Reduction* is a transformation from one problem to an equivalent problem. Two problems are equivalent if a solution of one can be converted to a solution of the other with no more than a moderate amount of effort. CVXPY uses reductions to rewrite problems into forms that solvers will accept.

Reductions allow CVXPY to simplify problems and target different categories of solvers (quadratic program solvers and conic solvers are two examples of solver categories). Appropriating terminology from software compilers, we classify reductions as either middle-end reductions or back-end reductions. A reduction that simplifies a source problem without regard to the targeted solver is called a *middle-end reduction*, whereas a reduction that takes a source problem and converts it to a form acceptable to a category of solvers is called a *back-end reduction*. Each solver (along with the mode in which it is invoked) is called a *back-end* or *target*.

The majority of users will not need to know anything about the reduction API; indeed, most users need not even know that reductions exist. But those who wish to extend CVXPY or contribute to it may find the API useful, as might those who are simply curious to learn how CVXPY works.

4.5.1 Middle-End Reductions

The reductions listed here are not specific to a back end (solver); they can be applied regardless of whether you wish to target, for example, a quadratic program solver or a conic solver.

Complex2Real

class cvxpy.reductions.complex2real.complex2real.**Complex2Real**(*problem=None*)

Bases: *cvxpy.reductions.reduction.Reduction*

Lifts complex numbers to a real representation.

accepts(*problem*)

States whether the reduction accepts a problem.

Parameters **problem** (Problem) – The problem to check.

Returns True if the reduction can be applied, False otherwise.

Return type bool

apply(*problem*)

Applies the reduction to a problem and returns an equivalent problem.

Parameters **problem** (Problem) – The problem to which the reduction will be applied.

Returns

- *Problem or dict* – An equivalent problem, encoded either as a Problem or a dict.
- *InverseData, list or dict* – Data needed by the reduction in order to invert this particular application.

invert(*solution*, *inverse_data*)

Returns a solution to the original problem given the inverse_data.

Parameters

- **solution** (Solution) – A solution to a problem that generated the inverse_data.
- **inverse_data** – The data encoding the original problem.

Returns A solution to the original problem.

Return type *Solution*

CvxAttr2Constr

class cvxpy.reductions.cvx_attr2constr.**CvxAttr2Constr**(*problem=None*)
Bases: *cvxpy.reductions.reduction.Reduction*

Expand convex variable attributes into constraints.

accepts(*problem*)
States whether the reduction accepts a problem.

Parameters **problem** (Problem) – The problem to check.

Returns True if the reduction can be applied, False otherwise.

Return type bool

apply(*problem*)
Applies the reduction to a problem and returns an equivalent problem.

Parameters **problem** (Problem) – The problem to which the reduction will be applied.

Returns

- *Problem or dict* – An equivalent problem, encoded either as a Problem or a dict.
- *InverseData, list or dict* – Data needed by the reduction in order to invert this particular application.

invert(*solution*, *inverse_data*)
Returns a solution to the original problem given the inverse_data.

Parameters

- **solution** (Solution) – A solution to a problem that generated the inverse_data.
- **inverse_data** – The data encoding the original problem.

Returns A solution to the original problem.

Return type *Solution*

Dgp2Dcp

class cvxpy.reductions.dgp2dcp.dgp2dcp.**Dgp2Dcp**(*problem=None*)
Bases: cvxpy.reductions.canonicalization.Canonicalization

Reduce DGP problems to DCP problems.

This reduction takes as input a DGP problem and returns an equivalent DCP problem. Because every (generalized) geometric program is a DGP problem, this reduction can be used to convert geometric programs into convex form.

Example

```
>>> import cvxpy as cp
>>>
>>> x1 = cp.Variable(pos=True)
>>> x2 = cp.Variable(pos=True)
>>> x3 = cp.Variable(pos=True)
>>>
>>> monomial = 3.0 * x_1**0.4 * x_2 ** 0.2 * x_3 ** -1.4
>>> posynomial = monomial + 2.0 * x_1 * x_2
>>> dgp_problem = cp.Problem(cp.Minimize(posynomial), [monomial == 4.0])
>>>
>>> dcp2cone = cvxpy.reductions.Dcp2Cone()
>>> assert not dcp2cone.accepts(dgp_problem)
>>>
>>> gp2dcp = cvxpy.reductions.Dgp2Dcp(dgp_problem)
>>> dcp_problem = gp2dcp.reduce()
>>>
>>> assert dcp2cone.accepts(dcp_problem)
>>> dcp_probem.solve()
>>>
>>> dgp_problem.unpack(gp2dcp.retrieve(dcp_problem.solution))
>>> print(dgp_problem.value)
>>> print(dgp_problem.variables())
```

accepts (*problem*)
A problem is accepted if it is DGP.

apply (*problem*)
Converts a DGP problem to a DCP problem.

invert (*solution*, *inverse_data*)
Returns a solution to the original problem given the inverse_data.

Parameters

- **solution** (Solution) – A solution to a problem that generated the inverse_data.
- **inverse_data** – The data encoding the original problem.

Returns A solution to the original problem.

Return type *Solution*

EvalParams

class cvxpy.reductions.eval_params.**EvalParams**(*problem=None*)
Bases: *cvxpy.reductions.reduction.Reduction*

Replaces symbolic parameters with their constant values.

accepts (*problem*)
States whether the reduction accepts a problem.

Parameters **problem** (Problem) – The problem to check.

Returns True if the reduction can be applied, False otherwise.

Return type bool

apply(*problem*)
Replace parameters with constant values.

Parameters **problem** (Problem) – The problem whose parameters should be evaluated.

Returns A new problem where the parameters have been converted to constants.

Return type *Problem*

Raises ParameterError – If the problem has unspecified parameters (i.e., a parameter whose value is None).

invert(*solution*, *inverse_data*)
Returns a solution to the original problem given the inverse_data.

FlipObjective

class cvxpy.reductions.flip_objective.**FlipObjective**(*problem=None*)
Bases: *cvxpy.reductions.reduction.Reduction*

Flip a minimization objective to a maximization and vice versa.

accepts(*problem*)
States whether the reduction accepts a problem.

Parameters **problem** (Problem) – The problem to check.

Returns True if the reduction can be applied, False otherwise.

Return type bool

apply(*problem*)
$\max(f(x)) = -\min(-f(x))$

Parameters **problem** (Problem) – The problem whose objective is to be flipped.

Returns

- *Problem* – A problem with a flipped objective.
- *list* – The inverse data.

invert(*solution*, *inverse_data*)
Map the solution of the flipped problem to that of the original.

Parameters

- **solution** (Solution) – A solution object.
- **inverse_data** (*list*) – The inverse data returned by an invocation to apply.

Returns A solution to the original problem.

Return type *Solution*

4.5.2 Back-End Reductions

The reductions listed here are specific to the choice of back end, i.e., solver. Currently, we support two types of back ends: conic solvers and quadratic program solvers. When a problem is solved through the solve() method, CVXPY attempts to find the best back end for your problem. The Dcp2Cone reduction converts DCP-compliant problems into conic form, while the Qp2SymbolicQp converts problems with quadratic, piecewise affine objectives, affine equality constraints, and piecewise-linear inequality constraints into a form that is closer to what is accepted by solvers. The

problems output by both reductions must be passed through another sequence of reductions, not documented here, before they are ready for to be solved.

- *Dcp2Cone*
- *Qp2SymbolicQp*

Dcp2Cone

class cvxpy.reductions.dcp2cone.dcp2cone.**Dcp2Cone**(*problem=None*)
Bases: cvxpy.reductions.canonicalization.Canonicalization

Reduce DCP problems to a conic form.

This reduction takes as input (minimization) DCP problems and converts them into problems with affine objectives and conic constraints whose arguments are affine.

accepts(*problem*)
A problem is accepted if it is a minimization and is DCP.

apply(*problem*)
Converts a DCP problem to a conic form.

Qp2SymbolicQp

class cvxpy.reductions.qp2quad_form.qp2symbolic_qp.**Qp2SymbolicQp**(*problem=None*)
Bases: cvxpy.reductions.canonicalization.Canonicalization

Reduces a quadratic problem to a problem that consists of affine expressions and symbolic quadratic forms.

accepts(*problem*)
Problems with quadratic, piecewise affine objectives, piecewise-linear constraints inequality constraints, and affine equality constraints are accepted.

apply(*problem*)
Converts a QP to an even more symbolic form.

- *Solution*
- *Reduction*
- *Chain*
- *SolvingChain*

4.5.3 Solution

class cvxpy.reductions.solution.**Solution**(*status*, *opt_val*, *primal_vars*, *dual_vars*, *attr*)
Bases: object

A solution to an optimization problem.

status
The status code.

Type str

opt_val
The optimal value.

Type float

primal_vars
A map from variable ids to optimal values.

Type dict of id to NumPy ndarray

dual_vars
A map from constraint ids to dual values.

Type dict of id to NumPy ndarray

attr
Miscelleneous information propagated up from a solver.

Type dict

4.5.4 Reduction

class cvxpy.reductions.reduction.**Reduction**(*problem=None*)
Bases: object

Abstract base class for reductions.

A reduction is an actor that transforms a problem into an equivalent problem. By equivalent we mean that there exists a mapping between solutions of either problem: if we reduce a problem A to another problem B and then proceed to find a solution to B, we can convert it to a solution of A with at most a moderate amount of effort.

A reduction that is instantiated with a non-None problem offers two key methods: *reduce* and *retrieve*. The *reduce()* method converts the problem the reduction was instantiated with to an equivalent problem. The *retrieve()* method takes as an argument a Solution for the equivalent problem and returns a Solution for the problem owned by the reduction.

Every reduction offers three low-level methods: accepts, apply, and invert. The accepts method of a particular reduction specifies the types of problems that it is applicable to; the apply method takes a problem and reduces it to an equivalent form, and the invert method maps solutions from reduced-to problems to their problems of provenance.

problem [Problem] A problem owned by this reduction; possibly None.

__init__(*problem=None*)
Construct a reduction for reducing *problem*.

If *problem* is not None, then a subsequent invocation of *reduce()* will reduce *problem* and return an equivalent one.

accepts(*problem*)
States whether the reduction accepts a problem.

Parameters **problem** (Problem) – The problem to check.

Returns True if the reduction can be applied, False otherwise.

Return type bool

apply(*problem*)
Applies the reduction to a problem and returns an equivalent problem.

Parameters **problem** (Problem) – The problem to which the reduction will be applied.

Returns

- *Problem or dict* – An equivalent problem, encoded either as a Problem or a dict.
- *InverseData, list or dict* – Data needed by the reduction in order to invert this particular application.

invert(*solution*, *inverse_data*)
Returns a solution to the original problem given the inverse_data.

Parameters

- **solution** (Solution) – A solution to a problem that generated the inverse_data.
- **inverse_data** – The data encoding the original problem.

Returns A solution to the original problem.

Return type *Solution*

reduce()
Reduces the owned problem to an equivalent problem.

Returns An equivalent problem, encoded either as a Problem or a dict.

Return type *Problem* or dict

Raises ValueError – If this Reduction was constructed without a Problem.

retrieve(*solution*)
Retrieves a solution to the owned problem.

Parameters **solution** (Solution) – A solution to the problem emitted by *reduce()*.

Returns A solution to the owned problem.

Return type *Solution*

Raises ValueError – If *self.problem* is None, or if *reduce()* was not previously called.

4.5.5 Chain

class cvxpy.reductions.chain.**Chain**(*problem=None*, *reductions=[]*)
Bases: *cvxpy.reductions.reduction.Reduction*

A logical grouping of multiple reductions into a single reduction.

reductions
A list of reductions.

Type list[*Reduction*]

accepts(*problem*)
A problem is accepted if the sequence of reductions is valid.

In particular, the i-th reduction must accept the output of the i-1th reduction, with the first reduction (self.reductions[0]) in the sequence taking as input the supplied problem.

Parameters **problem** (Problem) – The problem to check.

Returns True if the chain can be applied, False otherwise.

Return type bool

apply(*problem*)
Applies the chain to a problem and returns an equivalent problem.

Parameters **problem** (Problem) – The problem to which the chain will be applied.

Returns

- *Problem or dict* – The problem yielded by applying the reductions in sequence, starting at self.reductions[0].
- *list* – The inverse data yielded by each of the reductions.

invert (*solution*, *inverse_data*)
Returns a solution to the original problem given the inverse_data.

4.5.6 SolvingChain

class cvxpy.reductions.solvers.solving_chain.**SolvingChain**(*problem=None*, *reductions=[]*)

Bases: *cvxpy.reductions.chain.Chain*

A reduction chain that ends with a solver.

Parameters **reductions** (*list[Reduction]*) – A list of reductions. The last reduction in the list must be a solver instance.

reductions
A list of reductions.

Type list[*Reduction*]

solver
The solver, i.e., reductions[-1].

Type Solver

solve (*problem*, *warm_start*, *verbose*, *solver_opts*)
Solves the problem by applying the chain.

Applies each reduction in the chain to the problem, solves it, and then inverts the chain to return a solution of the supplied problem.

Parameters

- **problem** (Problem) – The problem to solve.
- **warm_start** (*bool*) – Whether to warm start the solver.
- **verbose** (*bool*) – Whether to enable solver verbosity.
- **solver_opts** (*dict*) – Solver specific options.

Returns **solution** – A solution to the problem.

Return type *Solution*

solve_via_data (*problem*, *data*, *warm_start*, *verbose*, *solver_opts*)
Solves the problem using the data output by the an apply invocation.

The semantics are:

```
data, inverse_data = solving_chain.apply(problem)
solution = solving_chain.invert(solver_chain.solve_via_data(data, ...))
```

which is equivalent to writing

```
solution = solving_chain.solve(problem, ...)
```

Parameters

- **problem** (Problem) – The problem to solve.
- **data** (*map*) – Data for the solver.
- **warm_start** (*bool*) – Whether to warm start the solver.
- **verbose** (*bool*) – Whether to enable solver verbosity.
- **solver_opts** (*dict*) – Solver specific options.

Returns The information returned by the solver; this is not necessarily a Solution object.

Return type raw solver solution

4.6 Transforms

Transforms provide additional ways of manipulating CVXPY objects beyond the atomic functions. While atomic functions operate only on expressions, transforms may also take Problem, Objective, or Constraint objects as input.

4.6.1 Scalarize

The *scalarize* transforms convert a list of objectives into a single objective, for example a weighted sum. All scalarizations are monotone in each objective, which means that optimizing over the scalarized objective always returns a Pareto-optimal point with respect to the original list of objectives. Moreover, all points on the Pareto curve except for boundary points can be attained given some weighting of the objectives.

scalarize.**weighted_sum**(*weights*)
Combines objectives as a weighted sum.

Parameters

- **objectives** – A list of Minimize/Maximize objectives.
- **weights** – A vector of weights.

Returns A Minimize/Maximize objective.

scalarize.**max**(*weights*)
Combines objectives as max of weighted terms.

Parameters

- **objectives** – A list of Minimize/Maximize objectives.
- **weights** – A vector of weights.

Returns A Minimize objective.

scalarize.**log_sum_exp**(*weights*, *gamma=1*)
Combines objectives as log_sum_exp of weighted terms.

The objective takes the form log(sum_{i=1}^n exp(gamma*weights[i]*objectives[i]))/gamma

As gamma goes to 0, log_sum_exp approaches weighted_sum. As gamma goes to infinity, log_sum_exp approaches max.

Parameters

- **objectives** – A list of Minimize/Maximize objectives.
- **weights** – A vector of weights.
- **gamma** – Parameter interpolating between weighted_sum and max.

Returns A Minimize objective.

scalarize.**targets_and_priorities**(*priorities*, *targets*, *limits=None*, *off_target=1e-05*)

Combines objectives with penalties within a range between target and limit.

Each Minimize objective i has value

priorities[i]*objectives[i] when objectives[i] >= targets[i]

+infinity when objectives[i] > limits[i]

Each Maximize objective i has value

priorities[i]*objectives[i] when objectives[i] <= targets[i]

+infinity when objectives[i] < limits[i]

Parameters

- **objectives** – A list of Minimize/Maximize objectives.
- **priorities** – The weight within the trange.
- **targets** – The start (end) of penalty for Minimize (Maximize)
- **limits** – The hard end (start) of penalty for Minimize (Maximize)
- **off_target** – Penalty outside of target.

Returns A Minimize/Maximize objective.

4.6.2 Other

Here we list other available transforms.

class cvxpy.transforms.**indicator**(*constraints*, *err_tol=0.001*)

An expression representing the convex function I(constraints) = 0 if constraints hold, +infty otherwise.

Parameters

- **constraints** (*list*) – A list of constraint objects.
- **err_tol** – A numeric tolerance for determining whether the constraints hold.

transforms.**linearize**()

Returns an affine approximation to the expression computed at the variable/parameter values.

Gives an elementwise lower (upper) bound for convex (concave) expressions that is tight at the current variable/parameter values. No guarantees for non-DCP expressions.

If f and g are convex, the objective f - g can be (heuristically) minimized using the implementation below of the convex-concave method:

```
for iters in range(N):
    Problem(Minimize(f - linearize(g))).solve()
```

Returns None if cannot be linearized.

Parameters **expr** – An expression.

Returns An affine expression or None.

transforms.**partial_optimize**()

CHAPTER 5

FAQ

5.1 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.

5.2 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.

5.3 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__
```

5.4 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.

5.5 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.

5.6 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`.

5.7 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.

5.8 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.

5.9 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.

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

See this thread on the mailing list.

5.11 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.

5.12 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
```

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

See *Advanced Features*.

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

See *Advanced Features*.

5.15 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.

5.16 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.

5.17 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.

5.18 Why does it take so long to compile my Problem?

In general, you should vectorize CVXPY expressions whenever possible if you care about performance (e.g., write A * x == b instead of a_i * x == b_i for every row a_i of A). Consult this IPython notebook for details.

5.18.1 How does CVXPY work?

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

5.19 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 6

Citing CVXPY

If you use CVXPY for published work, we encourage you to cite the accompanying JMLR MLOSS paper and the JCD paper. Please use the following BibTeX citations:

```
@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},
}
```

```
@article{cvxpy_rewriting,
  author  = {Akshay Agrawal, Robin Verschueren, Steven Diamond and Stephen Boyd},
  title   = {A Rewriting System for Convex Optimization Problems},
  journal = {Journal of Control and Decision},
  year    = {2018},
  volume  = {5},
  number  = {1},
  pages   = {42--60},
}
```

If you use CVXPY's support for disciplined geometric programming, please cite the accompanying paper and use following BibTeX citation:

```
@article{dgp,
  author  = {Akshay Agrawal, Steven Diamond and Stephen Boyd},
  title   = {Disciplined Geometric Programming},
  journal = {arXiv},
  archivePrefix = {arXiv},
  eprint = {1812.04074},
  primaryClass = {math.OC},
```

(continues on next page)

(continued from previous page)

```
  year    = {2018},
}
```

CHAPTER 7

Contributing

We welcome all contributors to CVXPY. You don't need to be an expert in convex optimization to help out! To get started as a CVXPY developer, install CVXPY from *source*. Contributions to CVXPY are made through pull requests or feature branches to our Github repository. For project ideas, consult our Github issues page, which lists both bugs and larger feature requests.

When navigating the source, keep in mind that CVXPY is a work in progress. The only APIs guaranteed to be stable are those discussed in the API documentation. All other APIs are subject to change without warning. Files and symbols that are certain to change in the future are marked "DEPRECATED" in the source.

Finally, feel free to introduce yourself on the cvxpy mailing list, which is for users and developers alike. Use this mailing list to introduce yourself. You can also contact the project leads Steven Diamond, Akshay Agrawal, and Stephen Boyd directly.

CHAPTER 8

Related Projects

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

8.1 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.
- cvxcore 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.

8.2 Solvers

- OSQP is an open-source C library for solving convex quadratic programs.
- 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 9

What's New in 1.0

CVXPY 1.0 includes a major rewrite of the CVXPY internals, as well as a number of changes to the user interface. We first give an overview of the changes, before diving into the details. We only cover changes that might be of interest to users.

We have created a script to convert code using CVXPY 0.4.11 into CVXPY 1.0, available here.

9.1 Overview

- Disciplined geometric programming (DGP): Starting with version 1.0.11, CVXPY lets you formulate and solve log-log convex programs, which generalize both traditional geometric programs and generalized geometric programs. To get started with DGP, check out *the tutorial* and consult the accompanying paper.
- Reductions: CVXPY 1.0 uses a modular system of *reductions* to convert problems input by the user into the format required by the solver, which makes it easy to support new standard forms, such as quadratic programs, and more advanced user inputs, such as problems with complex variables. See *Reductions* and the accompanying paper for further details.
- Attributes: Variables and parameters now support a variety of attributes that describe their symbolic properties, such as nonnegative or symmetric. This unifies the treatment of symbolic properties for variables and parameters and replaces specialized variable classes such as `Bool` and `Semidef`.
- NumPy compatibility: CVXPY's interface has been changed to resemble NumPy as closely as possible, including support for 0D and 1D arrays.
- Transforms: The new transform class provides additional ways of manipulating CVXPY objects, beyond the atomic functions. While atomic functions operate only on expressions, transforms may also take Problem, Objective, or Constraint objects as input.

9.2 Reductions

A reduction is a transformation from one problem to an equivalent problem. Two problems are equivalent if a solution of one can be converted to a solution of the other with no more than a moderate amount of effort. CVXPY uses reductions to rewrite problems into forms that solvers will accept. The practical benefit of the reduction based framework is that CVXPY 1.0 supports quadratic programs as a target solver standard form in addition to cone programs, with more standard forms on the way. It also makes it easy to add generic problem transformations such as converting problems with complex variables into problems with only real variables.

9.3 Attributes

Attributes describe the symbolic properties of variables and parameters and are specified as arguments to the constructor. For example, `Variable(nonneg=True)` creates a scalar variable constrained to be nonnegative. Attributes replace the previous syntax of special variable classes like `Bool` for boolean variables and `Semidef` for symmetric positive semidefinite variables, as well as specification of the sign for parameters (e.g., `Parameter(sign='positive')`). Concretely, write

- `Variable(shape, boolean=True)` instead of `Bool(shape)`.
- `Variable(shape, integer=True)` instead of `Int(shape)`.
- `Variable((n, n), PSD=True)` instead of `Semidef(n)`.
- `Variable((n, n), symmetric=True)` instead of `Symmetric(n)`.
- `Variable(shape, nonneg=True)` instead of `NonNegative(shape)`.
- `Parameter(shape, nonneg=True)` instead of `Parameter(shape, sign='positive')`.
- `Parameter(shape, nonpos=True)` instead of `Parameter(shape, sign='negative')`.

See *Attributes* for a complete list of supported attributes. More attributes will be added in the future.

9.4 NumPy Compatibility

The following interface changes have been made to make CVXPY more compatible with NumPy syntax:

- The `value` field of CVXPY expressions now returns NumPy ndarrays instead of NumPy matrices.
- The dimensions of CVXPY expressions are given by the `shape` field, while the `size` field gives the total number of entries. In CVXPY 0.4.11 and earlier, the `size` field gave the dimensions and the `shape` field did not exist.
- The dimensions of CVXPY expressions are no longer always 2D. 0D and 1D expressions are possible. We will add support for arbitrary ND expressions in the future. The number of dimensions is given by the `ndim` field.
- The shape argument of the `Variable`, `Parameter`, and `reshape` constructors must be a tuple. Instead of writing, `Parameter(2, 3)` to create a parameter of shape `(2, 3)`, you must write `Parameter((2, 3))`.
- Indexing and other operations can map 2D expressions down to 1D or 0D expressions. For example, if `X` has shape `(3, 2)`, then `X[:,0]` has shape `(3,)`. CVXPY behavior follows NumPy semantics in all cases, with the exception that broadcasting only works when one argument is 0D.
- Several CVXPY atoms have been renamed:
  - `mul_elemwise` to `multiply`

  - `max_entries` to `max`
  - `sum_entries` to `sum`
  - `max_elemwise` to `maximum`
  - `min_elemwise` to `minimum`

- Due to the name changes, we now strongly recommend against importing CVXPY using the syntax `from cvxpy import *`.
- The `vstack` and `hstack` atoms now take lists as input. For example, write `vstack([x, y])` instead of `vstack(x, y)`.

9.5 Transforms

Transforms provide additional ways of manipulating CVXPY objects beyond the atomic functions. For example, the `indicator` transform converts a list of constraints into an expression representing the convex function that takes value 0 when the constraints hold and ∞ when they are violated. See *Transforms* for a full list of the new transforms.

CHAPTER 10

CVXPY Short Course

Convex optimization is simple 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 11

License

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other

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

express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER 12

Versions

Version 1.0 is the latest stable release; it is the official, supported version of CVXPY. Previous versions will not be maintained, but you can access their archival documentation.

- *1.0*
- 0.4.11

Index

Symbols

A

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z