A Brief Introduction to CVXPY

2019 Winter Camp at PHBS

Xianhua Peng
Associate Professor
PHBS, Peking University

January 2019

Table of contents

Overview

Overview

- How to solve such a problem in Python?

$$
\begin{aligned}
V = \min_{\{x,y\}} & (x-y)^2 \\
s.t. \quad & x+y=1 \\
& x-y \geq 1
\end{aligned}
$$

Answer: $x = 1, y = 0$

- Use CVSPY package:

```
1  import cvxpy as cp
```

Overview

- Step 1: Create two scalar optimization variables:

```
x = cp.Variable()
y = cp.Variable()
```

- Step 2: Form objective function and constraints

```
obj_expr = (x - y)**2
obj = cp.Minimize(obj_expr)
constraints = [x + y == 1, x - y >= 1]
```

Overview

- Step 3: Form the minimization problem

```
prob = cp.Problem(obj, constraints)
```

- Step 4: Solve the problem

```
V = prob.solve()
```

Overview

- Step 5: Display your result

```
1  print('V=', V)
2  print('x=', x.value)
3  print('y=', y.value)
```

```
V = 1.0
x = 1.0
y = 1.5700862132409 83e-22
```

Atoms and Expression

The Structure of CVXPY package

- In CVXPY package, all the documented functions and classes are imported under `cvxpy` namespace, that means you can use them by simply type `cp.balabala`.
- There are 6 sections in CVXPY packages:
  - **atoms**
  - **expressions**
  - **constraints**
  - **problem**
  - **reductions**
  - **transforms**
- You can find more details in

  `https://www.cvxpy.org/api_reference/cvxpy.html`

Atom and Expression: Express Formulas

- The expression power of CVXPY is based on **atoms** and **expression**.
- **expression:** Expression tree is a collection of mathematical expressions linked together by one or more atoms, which are encoded as instances of the `Expression` class, and each Leaf in a tree is a Variable, Parameter, or Constant.
- **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 object contains information about its domain, sign, curvature, log-log curvature, and monotonicity.

Expressions Class

- In CVXPY, `Expression` objects are the instances of class `cvxpy.expressions.expression.Expression`.
- For more attributes of `Expression` class, please refer to the source codes here:

`https://www.cvxpy.org/_modules/cvxpy/expressions/expression.html#Expression`

Expressions: Variable domain, name value

- Find the **domain** and **name** of a variable:

```
print(x.domain, x.name())
```

```
[] var0
```

The domain of x is empty (full set), and the name of x in CVXPY is `var0`.

- Get the **value** of an expression:

```
obj_expr.value
```

```
array(1.)
```

The value of our objective function after minimization process is close to zero. Also, the value of an expression is stored in a `numpy` array.

Expressions: Variable curvature and sign

- Find the **curvature** of an expression:

```
obj_expr.curvature
```

```
'CONVEX'
```

Thus, our objective function is convex.

- Get the **sign** of an expression:

```
obj_expr.sign
```

```
'NONNEGATIVE'
```

It seems our objective function $(x - y)^2$ is non-negative.

Expressions: Variable gradient

- Find the **gradient** of the expression w.r.t. each variable:

```
1  obj_expr.grad
```

```
{Variable(()): 2.0, Variable(()): -2.0}
```

We can see that the gradient of objective function $(x - y)^2$ is $(2, -2)$. However, it is not marked clearly to which variable the partial derivative is.

Expressions: Rename variables

- Trick: name variables. CVXPY name variables automatically from `var0`, `var1` up to the latest one, this may cause some confusions when you check your expression. You may rename your variables in this way

```
x._name = 'x'
y._name = 'y'
```

Now, you may print your objective function:

```
print(obj_expr)
```

```
power(x + -y, 2)
```

Expressions: Vector/matrix variables

- Declare a vector or matrix variable (or parameter, constant)

```
var_b = cp.Variable((2,1))
```

Here we declare a variable of matrix form with shape $(2, 1)$.

- Example:

$$\min_{\beta}(y - X\beta)^T(y - X\beta)$$

```
import numpy as np
X = np.random.normal(0,1,(100,2))
Y = X @ np.array([[2],[1]]) + \
np.random.normal(0,1,(100,1))
```

From the data generation process, we know $\beta = (2, 1)^T$.

Expressions: Vector/matrix variables

- Using formula $\beta = (X^T X)^{-1} X^T y$

```
b = np.linalg.solve(X.T @ X, X.T @ Y)
print(b)
```

```
[[2.22091534]
 [0.97792947]]
```

- Using CVXPY:

```
obj = cp.Minimize(cp.sum_squares(Y - X @ var_b
prob = cp.Problem(obj)
V = prob.solve()
var_b.value
```

```
array([[2.22091534],
       [0.97792947]])
```

Expressions: Overloaded operators in variables

- Expression class overloads many operators, so that you can use them in a compact way. For example,

```
1 add_xy = x.__add__(y)
2 print(add_xy)
```

```
var0 + var1
```

  All those overwritten operators are decorated with `__`, such as `expr1.__mul__(expr2)`.

- For more APIs in `Variable` class, please refer to `https://www.cvxpy.org/api_reference/cvxpy.expressions.html`

Expressions: Parameters

- Parameters: Since `Problem` object in CVXPY is immutable, we can declare X and Y as instances of `Parameter` class rather than constants.

```
1  X = cp.Parameter((100, 2))
2  Y = X @ np.array([[2],[1]]) + \
3       np.random.normal(0,1,(100,1))
4
5  X.value = np.random.normal(0, 1, (100, 2))
```

In the line 5, you have to initialize the value of parameter X before using it.

Atom

- In the above example, you have already seen a CVXPY atom object: `"cp.sum_squares"`, generating another expression by applying on the expression `"Y - X @ var_b"`. The meaning of this atom is actually:

$$cp.sum\_squares(x) = \sum_{i=1}^{n} x_i^2$$

- We can devide the atoms in CVXPY into 5 categories:
  - **Operators**: such as $+$, $-$, $*$, $**$, @, / and indexing operations.
  - **Scalar functions**: atoms return a scalar.
  - **Operations along an axis**: Functions `sum`, `norm`, `max`, and `min` applied along an axis.
  - **Elementwise functions**: atoms operate on each element of their arguments.
  - **Vector/matrix functions**: atoms returns a vector/matrix.

Atom

- Examples: minimize the infinite norm of a vector (a problem with min-max structre)

$$\min_{\beta} \max_{i} \{(y_i - x_i\beta)^2\}_{i=1}^{n}$$

where x_i denotes the i-th row of matrix X, y_i is the i-th element of Y.

```
obj_expr = cp.norm((Y - X @ var_b)**2, 'inf')
obj = cp.Minimize(obj_expr)
prob = cp.Problem(obj)
V = prob.solve(); print(var_b.value)
```

```
array([[2.056103  ],
       [0.86418431]])
```

Atom

- Examples: Quadratic form of positive definite matrix

$$V = \min_{\beta} \beta^T Q \beta$$

$$s.t.\ \beta \geq 0$$

Suppose

$$Q = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
beta = cp.Variable((3, 1))
Q = np.array([[2, 0, 0],
              [0, 3, 0],
              [0, 0, 1]])
```

Atom

Continue...

```
obj_expr = cp.quad_form(beta, Q)
obj = cp.Maximize(obj_expr)
prob = cp.Problem(obj, [beta >= 0])
V = prob.solve(); print(beta.value)
```

Answer:

```
[[0.]
 [0.]
 [0.]]
```

- For more information of atoms in CVXPY, please refer to `https://www.cvxpy.org/api_reference/cvxpy.atoms.`

Disciplined Convex Programming

Sign Of Expression

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 $expr_1 * expr_2$ is

- Zero if either expression has sign zero
- Positive if $expr_1$ and $expr_2$ have the same (known) sign
- Negative if $expr_1$ and $expr_2$ have opposite (known) signs
- Unknown if either expression has unknown sign

Curvature

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

Table 1: Different kinds of curvature

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

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

Curvature rules

$f(expr_1, expr_2, ..., expr_n)$ 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 for each $expr_i$ is affine.

Example 1

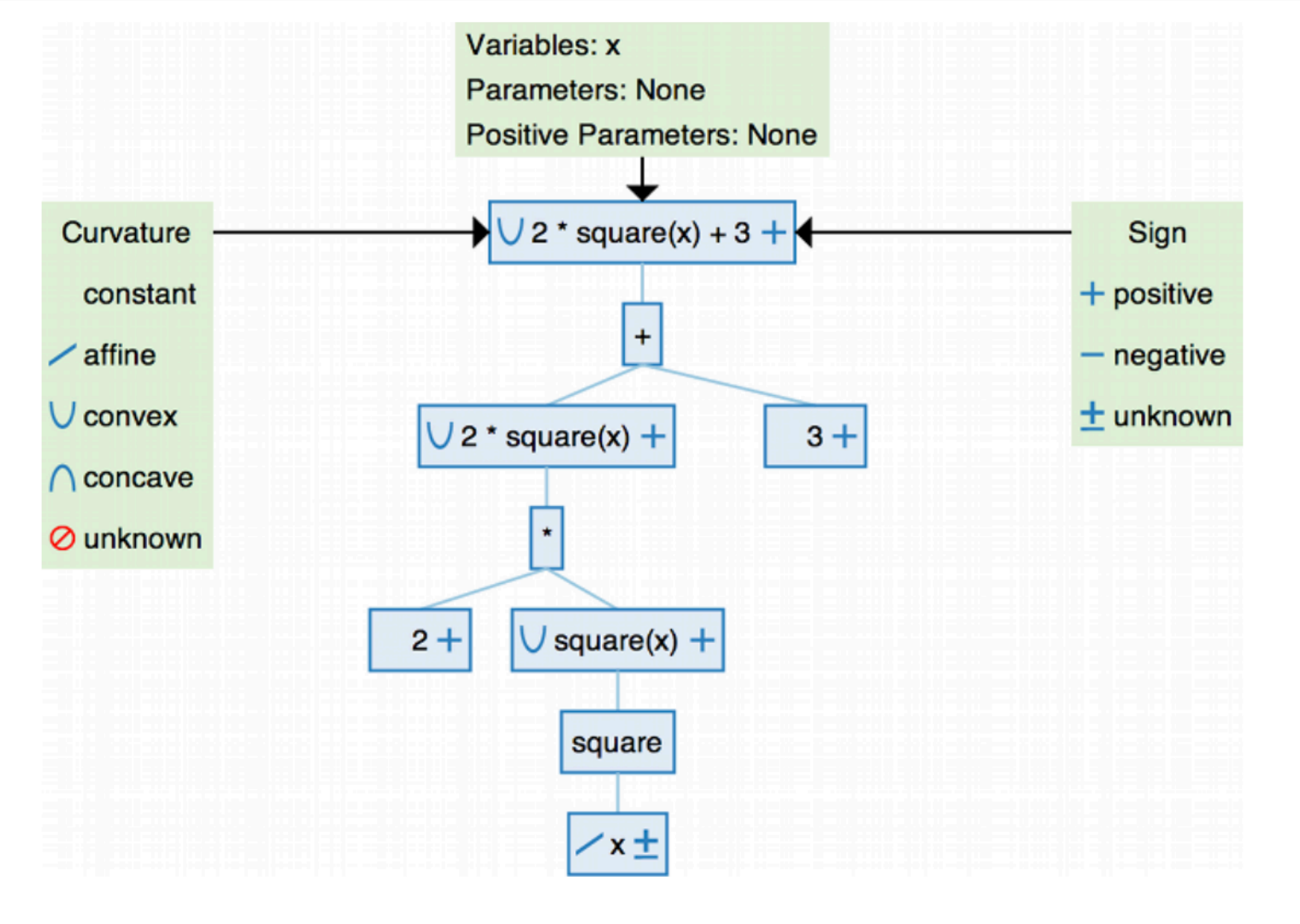

Figure 1: The expression 2*square(x) + 3

Example 2

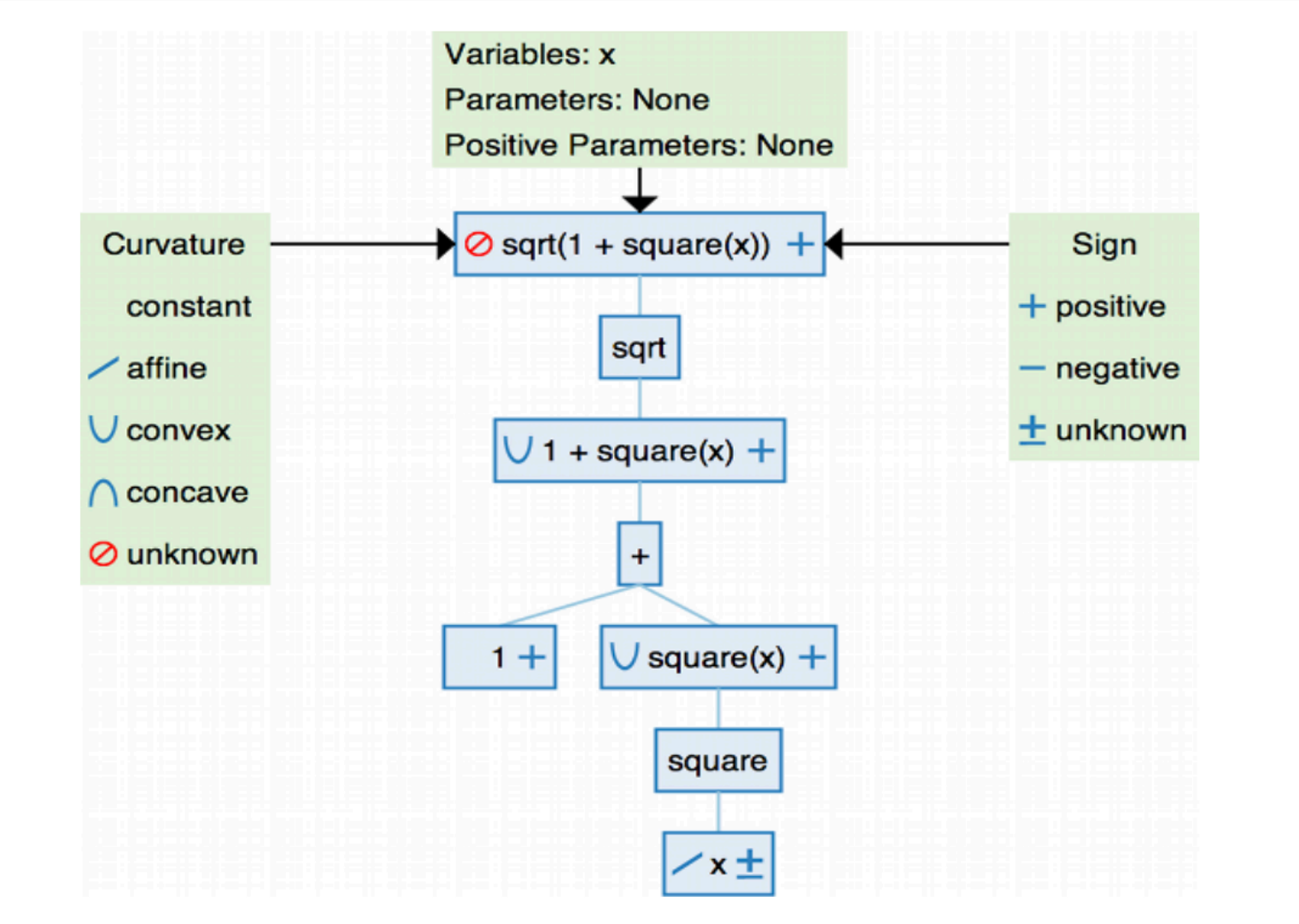

Figure 2: The expression sqrt(1 + square(x))

Constraints

Constraint

A **constraint** is an equality or inequality that restricts the domain of an optimization problem.

- Non-positive
- Equality or zero
- Positive semidefinite

Constraint examples

The preferred way of creating a constraint is through operator overloading. Symply write:

Non-positive

```
x <= 0
```

Zero

```
x == 0
```

PSD

```
x >> 0
```

Problems

Problem

The **Problem class** is to describe a convex optimization problem. **Problem instance** encapsulates an optimization problem, an **objective** and a set of **constraints**, and the **solve()** method could solves the problem encoded by the instance.

Example

```
1    problem = Problem(Minimize(expression),
2                      constraints)
3    problem.solve()
```

Infeasible 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.

Infeasible problems

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

Output

```
status: infeasible
optimal value inf
```

Unbounded problems

If a problem is unbounded, the status field of the problem is set "unbounded", The value fields of the problem variables are not updated.

Unbounded problems

```
prob = cp.Problem(cp.Minimize(x))
prob.solve()
print("status:", prob.status)
print("optimal value", prob.value)
```

Output

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

Advanced Features

Dual variables

You can use CVXPY to find the optimal dual variables for a problem.

Example

```
constraints = [x + y == 1,
               x - y >= 1]
...
constraints[0].dual_value
constraints[1].dual_value
```

Choosing a solver

CVXPY is distributed with the open source solvers ECOS, OSQP, and SCS. Many other solvers can be called by CVXPY if installed separately.

Example

```
...
# 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)
```

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**, textbftolerance. You can pass these options along through CVXPY as keyword arguments.

Example

```
print("Optimal value",
      prob.solve(solver=cp.ECOS,
                 max_iters=100,
                 abstol=1e-7))
```