cvxpy_intro

January 26, 2023

1 Introduction to CVXPY

CVXPY is a Python-embedded modeling language for (disciplined) convex optimization problems. Much like CVX in MATLAB, it allows you to express the problem in a natural way that follows the math, instead of expressing the problem in a way that conforms to a specific solver's syntax.

Note: originally written by James Folberth, 2017. Some updates Sept 2018 by Stephen Becker, to work with current cvxpy (ver 1.0) – not all bugs are fixed though. Updated Jan 2021 and Jan 2023 to work with Google colab (which, as of Jan 25 2021, has cvxpy version `1.0.31` pre-installed; update Jan 2023, it has version `1.2.3`), mainly fixing size issues, like `(n,)` vs `(n,1)`.

CVXPY Homepage

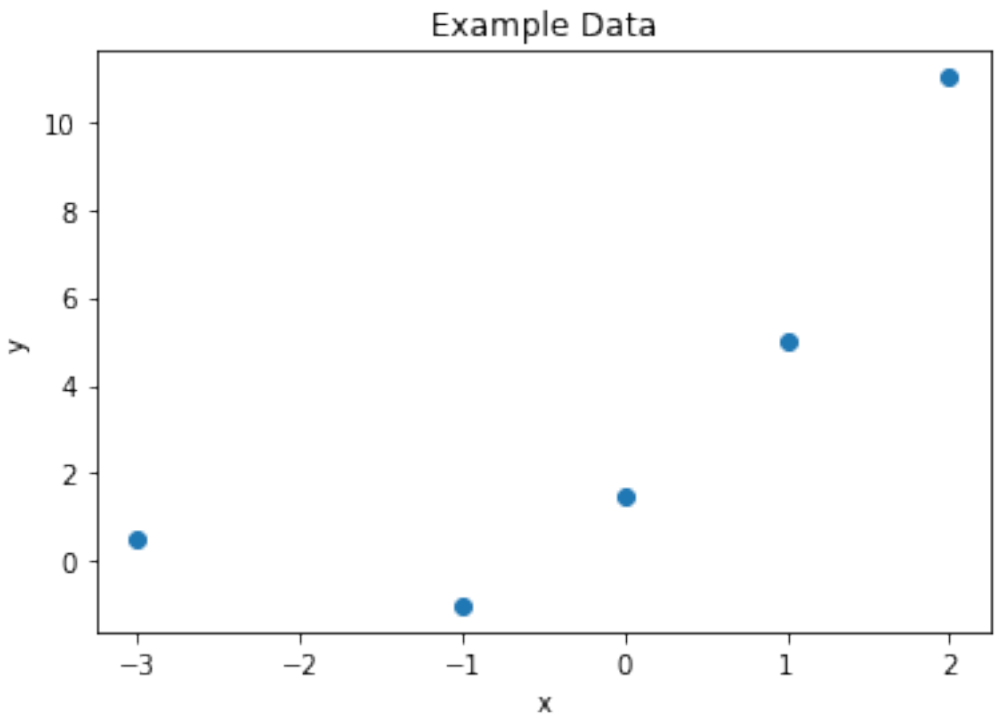
CVXPY Tutorial Documentation

CVXPY Examples - 2021 update: the CVXPY Examples now have google colab notebooks. Highly recommended! Look at those in addition to (or instead of) this notebook

[1]:
```
import numpy as np # we can use np.array to specify problem data
import matplotlib.pyplot as plt
%matplotlib inline
import cvxpy as cvx  # after import, can check version using cvx.__version__
```

1.1 Example: Least-Squares Curve Fitting

[3]:
```
x = np.array([-3,  -1, 0,   1, 2])
y = np.array([0.5, -1, 1.5, 5, 11])
plt.scatter(x,y)
plt.xlabel('x'); plt.ylabel('y'); plt.title('Example Data')
plt.show()
```



The data look like they follow a quadratic function. We can set up the following Vandermonde system and use unconstrained least-squares to estimate parameters for a quadratic function.

$$A = \begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \end{bmatrix}$$

Solving the following least-squares problem for β will give us parameters for a quadratic model:

$$\min_{\beta} \|A\beta - y\|_2$$

Note that we could easily solve this simple problem with a QR factorization (in MATLAB, `np.linalg.lstsq` in python/numpy).

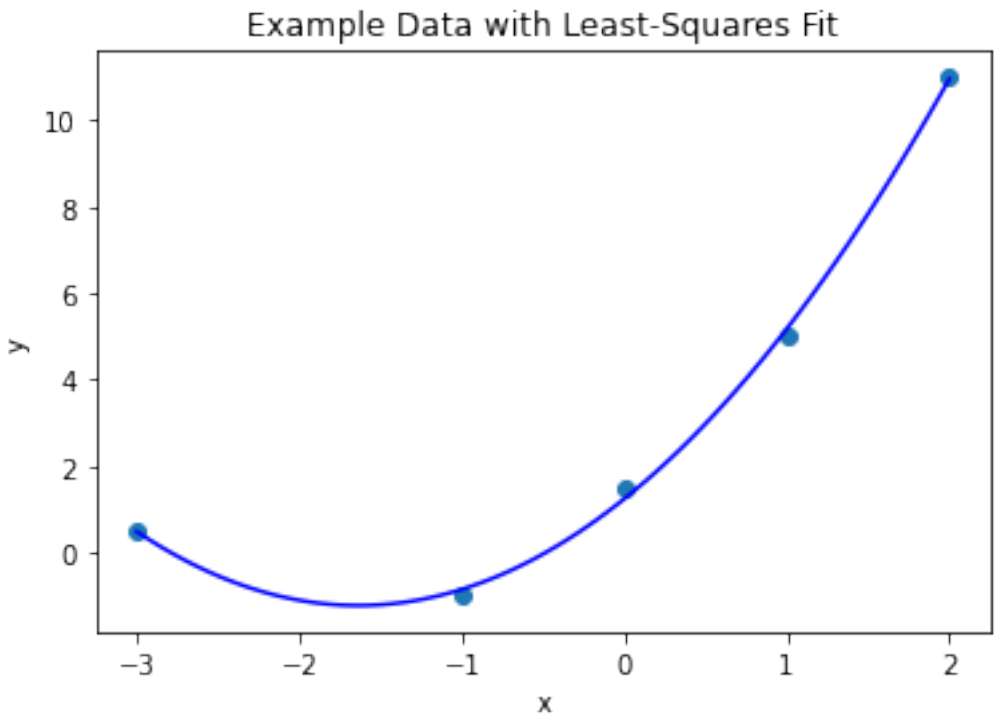
[5]:
```
A = np.column_stack((np.ones(5,), x, x**2))

# now setup and solve with CVXPY
beta = cvx.Variable(3)

# CVXPY's norm behaves like np.linalg.norm
# obj = cvx.Minimize(cvx.norm(A*beta-y)) # this syntax does not work
```

```
obj = cvx.Minimize(cvx.norm(A@beta-y))
prob = cvx.Problem(obj)

# Assuming the problem follows the DCP ruleset,
# CVXPY will select a solver and try to solve the problem.
# We can check if the problem is a disciplined convex program
# with prob.is_dcp().
prob.solve()

print("Problem status: ", prob.status)
print("Optimal value:  ", prob.value)
print("Optimal varriables beta:\n", beta.value)
```

```
Problem status:  optimal
Optimal value:   0.34962635220491034
Optimal varriables beta:
 [1.23858616 3.01656848 0.92157585]
```

Let's check the solution to see how we did:

[6]:
```
_beta = beta.value # get the optimal vars
_x = np.linspace(x.min(), x.max(), 100)
_y = _beta[0]*np.ones_like(_x) + _beta[1]*_x + _beta[2]*_x**2
plt.scatter(x,y)
plt.plot(_x,_y,'-b')
plt.xlabel('x'); plt.ylabel('y'); plt.title('Example Data with Least-Squares␣
 ↪Fit')
plt.show()
```



1.2 Example: ℓ_1-norm minimization

Consider the basis pursuit problem

$$\begin{array}{ll} \text{minimize} & \|x\|_1 \\ \text{subject to} & Ax = y. \end{array}$$

This is a least ℓ_1-norm problem that will hopefully yield a sparse solution x.

We now have an objective, $\|x\|_1$, and an equality constraint $Ax = y$.

[7]:
```
# make a bogus sparse solution and RHS
m = 200; n = 100;
A = np.random.randn(m,n)
_x = np.zeros((n,1))  # just using this notation to show that this is something␣
 ↪we're going to pretend we don't actually have
# _x = np.zeros(n) # better, fewer headaches later, but adjust line 8 too
_k = 10
_I = np.random.permutation(n)[0:_k]
_x[_I] = np.random.randn(_k,1)
y = np.dot(A,_x)  # this is (200,1), as is A.dot(_x)

# Here's an essential step, change from (200,1) to (200,) size
#  if we defined _x=np.zeros((n,1))
```

```
y = y.ravel()  # https://www.geeksforgeeks.org/differences-flatten-ravel-numpy/

x = cvx.Variable(n)   # a bit like sympy.  Shape is (n,) not (n,1)

# Even though the cvx.norm function behaves very similarly to
# the np.linalg.norm function, we CANNOT use the np.linalg.norm
# function on CVXPY objects.  If we do, we'll probably get a strange
# error message.
obj = cvx.Minimize(cvx.norm(x,1))

# specify a list of constraints
# constraints = [ A*x == y ]  # A*x is (200,), as is A@x.  This is OK
constraints = [ A@x == y ]

# constraints = [ A.dot(x) == y ] # No, not OK. This is (200,100). CVXPY issue.

# specify and solve the problem
prob = cvx.Problem(obj, constraints)
prob.solve(verbose=True) # let's see the underlying solver's output

print("Problem status: ", prob.status)
print("Optimal value:  ", prob.value)

print("True nonzero inds:      ", sorted(_I))
print("Recovered nonzero inds: ", sorted(np.where(abs(x.value) > 1e-14)[0]))
# Note: we cannot access "x", we need to do "x.value"
#   (also, turn _x to right shape)
err = np.linalg.norm(x.value -_x.ravel())
print(f'Norm of error, ||x-x_est|| is {err:e}')
```

```
===============================================================================
                                     CVXPY
                                     v1.2.3
===============================================================================
(CVXPY) Jan 26 08:03:07 PM: Your problem has 100 variables, 1 constraints, and 0
parameters.
(CVXPY) Jan 26 08:03:07 PM: It is compliant with the following grammars: DCP,
DQCP
(CVXPY) Jan 26 08:03:07 PM: (If you need to solve this problem multiple times,
but with different data, consider using parameters.)
(CVXPY) Jan 26 08:03:07 PM: CVXPY will first compile your problem; then, it will
invoke a numerical solver to obtain a solution.
-------------------------------------------------------------------------------
                                  Compilation
-------------------------------------------------------------------------------
(CVXPY) Jan 26 08:03:07 PM: Compiling problem (target solver=ECOS).
(CVXPY) Jan 26 08:03:07 PM: Reduction chain: Dcp2Cone -> CvxAttr2Constr ->
ConeMatrixStuffing -> ECOS
```

```
(CVXPY) Jan 26 08:03:07 PM: Applying reduction Dcp2Cone
(CVXPY) Jan 26 08:03:07 PM: Applying reduction CvxAttr2Constr
(CVXPY) Jan 26 08:03:07 PM: Applying reduction ConeMatrixStuffing
(CVXPY) Jan 26 08:03:07 PM: Applying reduction ECOS
(CVXPY) Jan 26 08:03:07 PM: Finished problem compilation (took 5.566e-02
seconds).
-----------------------------------------------------------------------------
                                Numerical solver
-----------------------------------------------------------------------------
(CVXPY) Jan 26 08:03:07 PM: Invoking solver ECOS  to obtain a solution.
-----------------------------------------------------------------------------
                                    Summary
-----------------------------------------------------------------------------
(CVXPY) Jan 26 08:03:07 PM: Problem status: optimal
(CVXPY) Jan 26 08:03:07 PM: Optimal value: 8.683e+00
(CVXPY) Jan 26 08:03:07 PM: Compilation took 5.566e-02 seconds
(CVXPY) Jan 26 08:03:07 PM: Solver (including time spent in interface) took
3.980e-02 seconds
Problem status:  optimal
Optimal value:   8.683258916729915
True nonzero inds:       [13, 20, 23, 37, 44, 56, 60, 80, 83, 96]
Recovered nonzero inds:  [13, 20, 23, 37, 44, 56, 60, 80, 83, 96]
Norm of error, ||x-x_est|| is 1.339427e-15
```

1.3 Example: Relaxation of Boolean LP

Consider the Boolean linear program

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \preceq b \\ & x_i \in \{0,1\}, \quad i = 1, ..., n. \end{array}$$

Note: the generalized inequality $\preceq$ is just element-wise $\leq$ on vectors.

This is not a convex problem, but we can relax it to a linear program and hope that a solution to the relaxed, convex problem is "close" to a solution to the original Boolean LP. A relaxation of the Boolean LP is the following LP:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \preceq b \\ & \mathbf{0} \preceq x \preceq \mathbf{1}. \end{array}$$

The relaxed solution x^{rlx} can be used to guess a Boolean point $\hat{x}$ by rounding based on a threshold $t \in [0, 1]$:

$$\hat{x}_i = \begin{cases} 1 & x_i^{\text{rlx}} \geq t \\ 0 & \text{otherwise,} \end{cases}$$

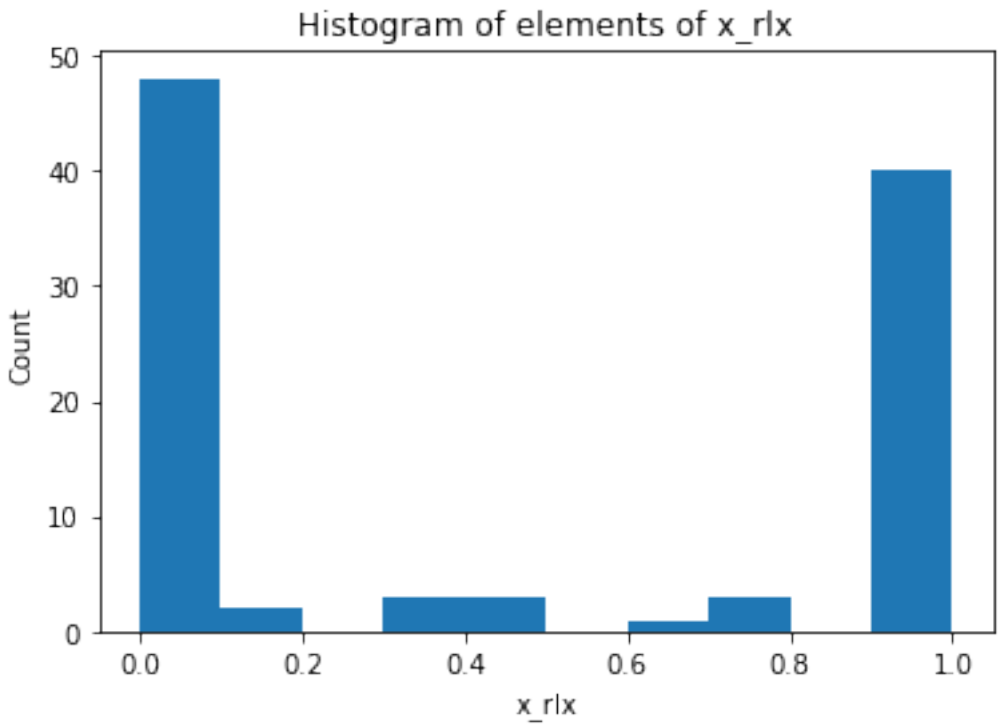
for $i = 1, ..., n$. However, the Boolean point $\hat{x}$ might not satisfy $Ax \preceq b$ (i.e., $\hat{x}$ might be infeasible).

From Boyd and Vandenberghe: > You can think of x_i as a job we either accept or decline, and $-c_i$ as the (positive) revenue we generate if we accept job i. We can think of $Ax \preceq b$ as a set of limits on m resources. A_{ij}, which is positive, is the amount of resource i consumed if we accept job j; b_i, which is positive, is the amount of recourse i available.

[11]:
```
m = 300; n = 100;
A = np.random.rand(m,n)
# b = A.dot(np.ones((n,1)))/2.
# c = -np.random.rand(n,1)
b = A.dot(np.ones((n)))/2.
c = -np.random.rand(n)

x_rlx = cvx.Variable(n)
obj = cvx.Minimize(c.T*x_rlx) # deprecated
# obj = cvx.Minimize(c.T@x_rlx) # preferred way
constraints = [ A@x_rlx <= b,
                0 <= x_rlx,
                x_rlx <= 1 ]

prob = cvx.Problem(obj, constraints)
prob.solve()

print("Problem status: ", prob.status)
print("Optimal value:  ", prob.value)

plt.hist(x_rlx.value)
plt.xlabel('x_rlx'); plt.ylabel('Count')
plt.title('Histogram of elements of x_rlx')
plt.show()
```

```
/usr/local/lib/python3.8/dist-packages/cvxpy/expressions/expression.py:593:
UserWarning:
This use of ``*`` has resulted in matrix multiplication.
Using ``*`` for matrix multiplication has been deprecated since CVXPY 1.1.
    Use ``*`` for matrix-scalar and vector-scalar multiplication.
    Use ``@`` for matrix-matrix and matrix-vector multiplication.
    Use ``multiply`` for elementwise multiplication.
This code path has been hit 4 times so far.

  warnings.warn(msg, UserWarning)
```

```
Problem status:  optimal
Optimal value:   -32.781741310790224
```



1.4 Example: Minimum Volume Ellipsoid

Sometimes an example is particularly hard and we might need to adjust solver options, or use a different solver.

Consider the problem of finding the minimum volume ellipsoid (described by the matrix A and vector b) that covers a finite set of points $\{x_i\}_{i=1}^n$ in $\mathbb{R}^2$. The MVE can be found by solving

$$\begin{array}{ll} \text{maximize} & \log(\det(A)) \\ \text{subject to} & \|Ax_i + b\| \leq 1, \quad i = 1, ..., n. \end{array}$$

To allow CVXPY to see that the problem conforms to the DCP ruleset, we should use the function `cvx.log_det(A)` instead of something like `log(det(A))`.

[13]:
```
# Generate some data
np.random.seed(271828) # solver='CVXOPT' reaches max_iters
m = 2; n = 50
x = np.random.randn(m,n)

# A = cvx.Variable(2,2)  # This is old notation, doesn't work anymore
A = cvx.Variable((2,2))
b = cvx.Variable(2)
obj = cvx.Maximize(cvx.log_det(A))
# constraints = [ cvx.norm(A*x[:,i] + b) <= 1 for i in range(n) ]
```

```
constraints = [ cvx.norm(A@x[:,i] + b) <= 1 for i in range(n) ]

prob = cvx.Problem(obj, constraints)
#prob.solve(solver='CVXOPT', verbose=True) # progress stalls
#prob.solve(solver='CVXOPT', kktsolver='robust', verbose=True) # progress still␣
 ↪stalls
prob.solve(solver='SCS', verbose=False) # seems to work, although it's not super␣
 ↪accurate

# plot the ellipse and data
angles = np.linspace(0, 2*np.pi, 200)
rhs = np.row_stack((np.cos(angles) - b.value[0], np.sin(angles) - b.value[1]))
ellipse = np.linalg.solve(A.value, rhs)

plt.scatter(x[0,:], x[1,:])
plt.plot(ellipse[0,:].T, ellipse[1,:].T)
plt.xlabel('Dimension 1'); plt.ylabel('Dimension 2')
plt.title('Minimum Volume Ellipsoid')
plt.show()
```