Convex Optimization in Python with CVXPY

Steven Diamond Stephen Boyd
Stanford University

TCMM 2014, September 2014

Outline

Convex optimization problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \\ & Ax = b, \end{array}$$

with variable $x \in \mathbf{R}^n$

- objective and inequality constraints $f_0, \ldots, f_m$ are convex

  for all x, y, $\theta \in [0, 1]$,

  $$f_i(\theta x + (1-\theta)y) \leq \theta f_i(x) + (1-\theta) f_i(y)$$

  i.e., graphs of f_i curve upward
- equality constraints are linear

Why convex optimization?

- beautiful, fairly complete, and useful theory
- solution algorithms that work well in theory and practice
- **many applications** in
  - machine learning, statistics
  - control
  - signal, image processing
  - networking
  - engineering design
  - finance

. . . and many more

How do you solve a convex problem?

- use someone else's ('standard') solver (LP, QP, SOCP, ...)
  - easy, but your problem **must** be in a standard form
  - cost of solver development amortized across many users
- write your own (custom) solver
  - lots of work, but can take advantage of special structure
- use a convex modeling language
  - transforms user-friendly format into solver-friendly standard form
  - extends reach of problems solvable by standard solvers

Outline

Convex modeling languages

- long tradition of modeling languages for optimization
  - cf. AMPL, GAMS
- modeling languages for convex optimization
  - *e.g.*, CVX, YALMIP, CVXGEN, QCML
- function of a convex modeling language:
  - check/verify problem convexity
  - convert to standard form

Disciplined convex programming (DCP)

- system for constructing expressions with known curvature
  - constant, affine, nonnegative (convex), nonpositive (concave)
- expressions formed from
  - variables (curvature: affine, unknown sign)
  - constants (curvature: constant, known sign)
  - library of atoms with known curvature and sign (as function of their arguments)
- more at dcp.stanford.edu

Standard (conic) form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \in \mathcal{K} \end{array}$$

with variable $x \in \mathbf{R}^n$

- $\mathcal{K}$ is convex cone
  - $x \in \mathcal{K}$ is a generalized nonnegativity constraint
- linear objective, equality constraints
- special cases:
  - $\mathcal{K} = \mathbf{R}^n_+$: linear program (LP)
  - $\mathcal{K} = \mathbf{S}^n_+$: semidefinite program (SDP)
- general interface for solvers

Outline

Convex optimization

Convex modeling languages

CVXPY

Image in-painting

Trade-off curve, in parallel

Single commodity flow

Summary

CVXPY

a modeling language in Python for convex optimization

- translates from math to standard form used by solvers
- uses DCP to verify convexity
- open source all the way from the solvers
- supports parameterized problems
- mixes easily with general Python code, other libraries
- already used in many research projects and two classes
- over 7000 downloads on PyPi

CVXPY solvers

- all open source
- CVXOPT (Vandenberghe, Dahl, Andersen)
  - interior-point method
  - in Python
- ECOS (Domahidi)
  - interior-point method
  - compact, library-free C code
- SCS (O'Donoghue)
  - first-order method
  - native support of exponential cone
  - parallelism with OpenMP

CVXPY example

(constrained LASSO)

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 + \gamma\|x\|_1 \\ \text{subject to} & \mathbf{1}^T x = 0, \quad \|x\|_\infty \le 1 \end{array}$$

with variable $x \in \mathbf{R}^n$

```
from cvxpy import *
x = Variable(n)
cost = sum_squares(A*x-b) + gamma*norm(x,1)
obj = Minimize(cost)
constr = [sum_entries(x) == 0, norm(x,"inf") <= 1]
prob = Problem(obj, constr)
opt_val = prob.solve()
solution = x.value
```

Outline

Image in-painting

- guess pixel values in obscured/corrupted parts of image
- *total variation in-painting*: choose pixel values $x_{ij} \in \mathbf{R}^3$ to minimize
$$\mathrm{TV}(x) = \sum_{ij} \left\| \begin{bmatrix} x_{i+1,j} - x_{ij} \\ x_{i,j+1} - x_{ij} \end{bmatrix} \right\|_2$$
- a convex problem

Example

- 512×512 color image
- denote corrupted pixels with $K \in \{0, 1\}^{512 \times 512}$
  - $K_{ij} = 1$ if pixel value is known
  - $K_{ij} = 0$ if unknown
- $X_{\text{corr}} \in \mathbf{R}^{512 \times 512 \times 3}$ is corrupted image

Image in-painting CVXPY code

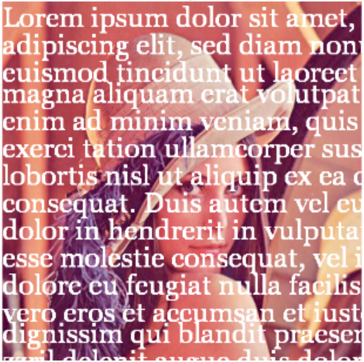
```
from cvxpy import *
variables = []
constr = []
for i in range(3):
    X = Variable(512, 512)
    variables += [X]
    constr += [mul_elemwise(K, X - X_corr[:,:,i]) == 0]

prob = Problem(Minimize(tv(*variables)), constr)
prob.solve(solver=SCS)
```

Example

Original

Corrupted

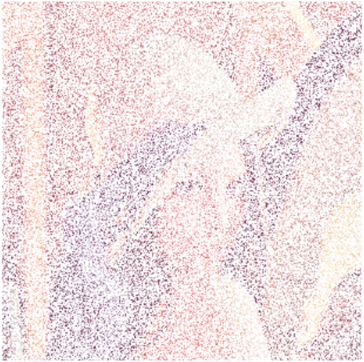


Example

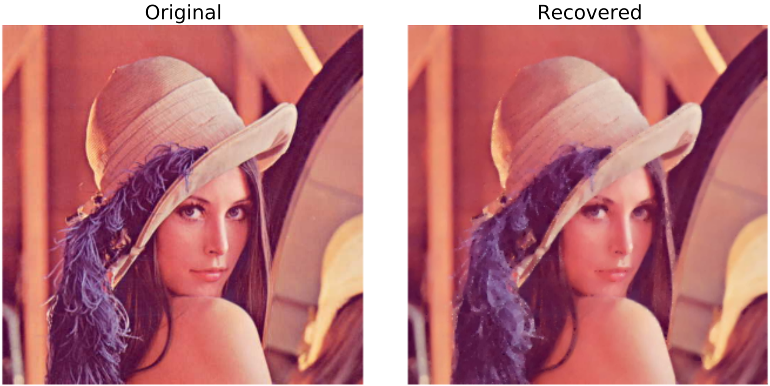
Original

Recovered

Example (80% of pixels removed)



Example (80% of pixels removed)



Outline

Parameters

- symbolic representations of constants
- fixed sign and dimensions
- change value of constant without rebuilding problem

Parameter syntax

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

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

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

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

LASSO in CVXPY

(LASSO)

$$\text{minimize} \quad \|Ax - b\|_2^2 + \gamma\|x\|_1$$

with variable $x \in \mathbf{R}^n$

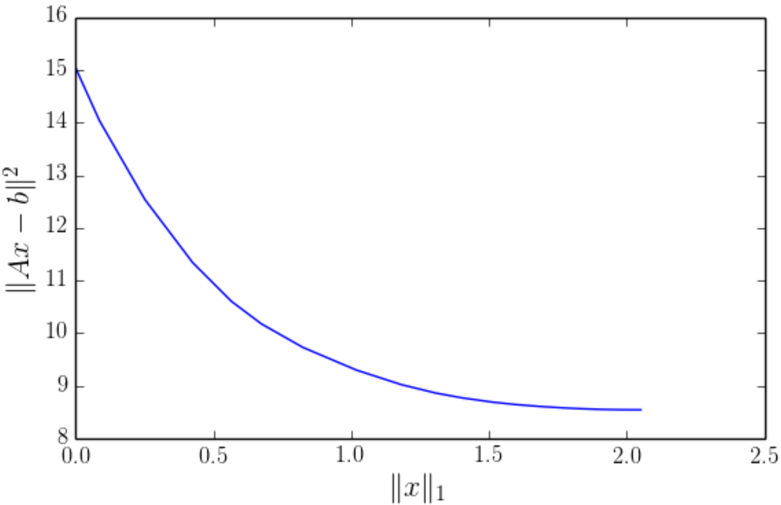
```
x = Variable(n)
gamma = Parameter(sign="positive")
error = sum_squares(A*x-b)
regularization = gamma*norm(x,1)
prob = Problem(Minimize(error + regularization))
```

For loop style trade-off curve

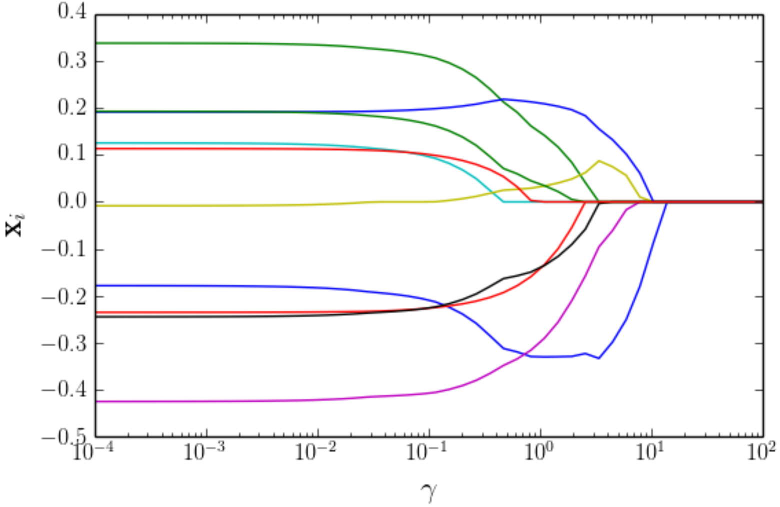
compute a trade-off curve by updating parameter gamma

```
x_values = []
for val in numpy.logspace(-4, 2):
    gamma.value = val
    prob.solve()
    x_values.append(x.value)
```

Trade-off curve for LASSO



Entries of x versus γ: (regularization path)



Parallel style trade-off curve

```
# Use tools for parallelism in standard library.
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 with N processes.
pool = Pool(processes = N)
x_values = pool.map(get_x, numpy.logspace(-4, 2))
```

Performance

- Lasso with $A \in \mathbf{R}^{1000 \times 500}$, 100 values of γ
- single thread time for one LASSO: 4 seconds
- performance using solver SCS:

	For loop	4 processes	32 processes
4 core MacBook Pro	403 sec	147 sec	136 sec
32 cores, Intel Xeon	619 sec	175 sec	56 sec

Outline

Convex optimization

Convex modeling languages

CVXPY

Image in-painting

Trade-off curve, in parallel

Single commodity flow

Summary

Single commodity flow

- directed graph with p nodes, n edges
- flow f_i on edge i
- external source/sink flow s_j at node j
- single commodity flow problem:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} \phi_i(f_i) + \sum_{j=1}^{p} \psi_j(s_j), \\ \text{subject to} & \text{zero net flow at each node} \end{array}$$

  - variables are f_i, s_j
  - ϕ_i convex flow cost functions
  - ψ_j convex source cost functions
  - can include constraints in ϕ_i, ψ_j

Matrix representation

- node incidence matrix $A \in \mathbf{R}^{p \times n}$

$$A_{ij} = \begin{cases} +1 & \text{edge } i \text{ leaves node } j \\ -1 & \text{edge } i \text{ enters node } j \\ 0 & \text{otherwise.} \end{cases}$$

- zero net flow at each node: $Af = s$
- final problem:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} \phi_i(f_i) + \sum_{j=1}^{p} \psi_j(s_j), \\ \text{subject to} & Af = s \end{array}$$

Object-oriented representation

- node object includes source, cost, source/net flow constraints
- edge object includes flow, cost, flow constraints
- solve the problem:

```
cost = sum([object.cost for object in nodes + edges])
obj = Minimize(cost)
constraints = []
for object in nodes + edges:
    constraints += object.constraints()
Problem(obj, constraints).solve()
```

Node object

```
class Node(object):
    def __init__(self, cost):
        self.source = Variable()
        self.cost = cost(self.source)
        self.edge_flows = []

    def constraints(self):
        """The constraint net flow == 0."""
        net_flow = sum(self.edge_flows) + self.source
        return [net_flow == 0]
```

Edge object

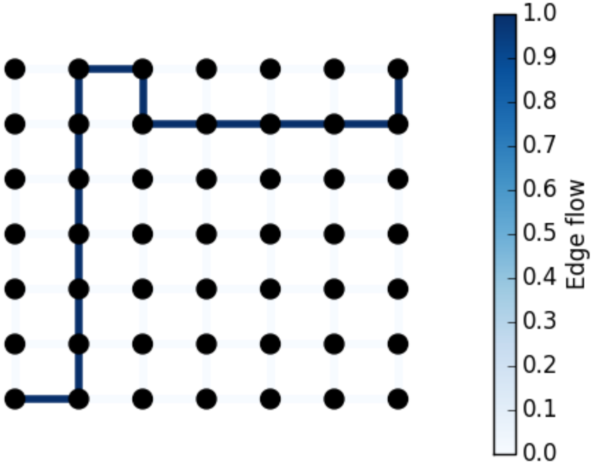
```
class Edge(object):
    def __init__(self, cost):
        self.flow = Variable()
        self.cost = cost(self.flow)

    def connect(self, in_node, out_node):
        """Connects two nodes via the edge."""
        in_node.edge_flows.append(-self.flow)
        out_node.edge_flows.append(self.flow)
```

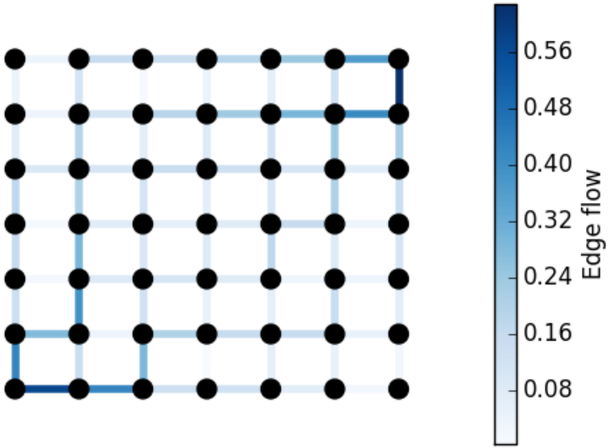
Example

- 7-by-7 grid of nodes
- 1 unit of flow sent from source s_1 to sink s_n:
  - $s_1 = +1$
  - $s_n = -1$
  - $s_i = 0$ for $i = 2, \dots, n-1$
- flow cost $\phi_i(f_i) = w_i \left(|f_i| + \lambda f_i^2\right)$
  - weights $w_i > 0$ randomly chosen

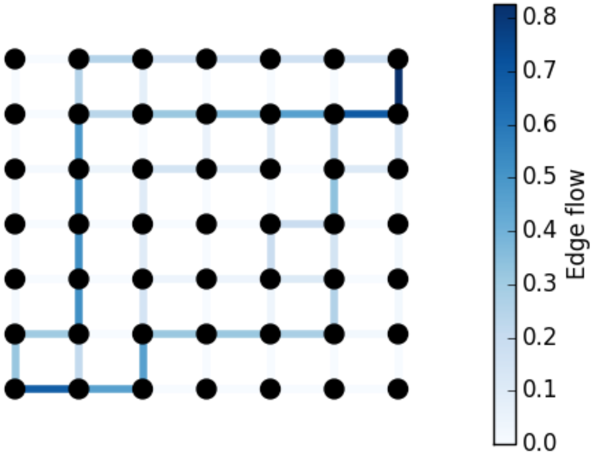
Shortest path ($\lambda = 0$)



Diffusion ($\lambda = +\infty$)



Diffusion with sparsity ($\lambda = 1$)



Outline

Summary

- convex optimization is easy with CVXPY
- mixes well with high level Python
  - parallelism
  - object oriented design
- building block for
  - distributed optimization
  - nonconvex optimization

Future work

- not just for prototyping
- speed and scalability
- abstract linear operators