A New Architecture for Optimization Modeling Frameworks

Matt Wytock, Steven Diamond, Felix Heide
and Stephen Boyd
Stanford University

November 14, 2016
Convex optimization problem

minimize \( f_0(x) \)
subject to \( f_i(x) \leq 0, \quad i = 1, \ldots, m \)
\( Ax = b, \)

with variable \( x \in \mathbb{R}^n \)

- 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
Convex modeling languages

- long tradition of modeling languages for optimization
  - AMPL, GAMS
- modeling languages for convex optimization
  - CVX, YALMIP, CVXGEN, CVXPY, Convex.jl, RCVX
- 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, convex, concave
- expressions formed from
  - variables
  - constants and parameters
  - library of functions with known curvature, monotonicity, sign
- basis of all convex modeling systems
- more at dcp.stanford.edu
The one rule that DCP is based on

\[ h(f_1(x), \ldots, f_k(x)) \] is convex when \( h \) is convex and for each \( i \)

- \( h \) is increasing in argument \( i \), and \( f_i \) is convex, or
- \( h \) is decreasing in argument \( i \), and \( f_i \) is concave, or
- \( f_i \) is affine

- there’s a similar rule for concave compositions
  (just swap convex and concave above)
Traditional architecture for optimization frameworks

Problem
Canonicalization
Standard form
Matrix stuffing
Sparse matrices
Solver
Solution
Standard (conic) form

minimize \( c^T x \)
subject to \( Ax = b \)
\( x \in \mathcal{K} \)

with variable \( x \in \mathbb{R}^n \)

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

- CVXOPT (Vandenberghe, Dahl, Andersen)
  - interior-point method
  - Python
- ECOS (Domahidi)
  - interior-point method
  - supports exponential cone
  - compact, library-free C code
- SCS (O’Donoghue)
  - first-order method
  - parallelism with OpenMP
  - GPU support
- others: GLPK, MOSEK, GUROBI, Cbc, Elemental, ...
- traditional architecture has been enormously successful
  - solvers based on interior point methods highly robust
  - solvers portable to new platforms with linear algebra libraries
    - BLAS, LAPACK, SuiteSparse, etc.
Drawbacks of traditional architecture

- for large problems, direct solutions to linear systems involving the $A$ matrix can be very expensive
- first-order methods (SCS) allow the use of indirect methods for linear solver subroutine
- but, representing all linear operators as sparse matrices can be inefficient
  - e.g., FFT-based convolution
- also, (most) existing solvers do not take advantage of modern platforms, e.g., GPUs, distributed
Graph-based architecture

1. Problem
2. Canonicalization
3. Standard form
4. Solver generation
5. Computation graph
6. Runtime execution
7. Solution
Computation graphs

- computation graph for $f(x, y) = x^2 + 2x + y$

- simple transformations produce computation graphs for function gradient and adjoint
  - key operations in first-order and indirect solvers
Computation graph frameworks

- huge momentum and engineering effort from deep learning community
  - TensorFlow, Theano, Caffe, Torch, ...
- support a wide variety of computational environments
  - CPU, GPU, distributed clusters, phones, ...
- given a computation graph, existing frameworks implement gradient descent
- for optimization, first-order and indirect solvers fit naturally
- limited support for sparse matrix factorizations, which are required by interior point methods, direct solvers
Generating solver graphs

- solver generation implemented with functions parameterized by graphs or graph generators
- e.g., conjugate gradient for solving linear system $Ax = b$

```python
def cg_solve(A, b, x_init, tol=1e-8):
    delta = tol*norm(b)
    def body(x, k, r_norm_sq, r, p):
        Ap = A(p)
        alpha = r_norm_sq / dot(p, Ap)
        x = x + alpha*p
        r = r - alpha*Ap
        r_norm_sq_prev = r_norm_sq
        r_norm_sq = dot(r, r)
        beta = r_norm_sq / r_norm_sq_prev
        p = r + beta*p
        return (x, k+1, r_norm_sq, r, p)
    def cond(x, k, r_norm_sq, r, p):
        return tf.sqrt(r_norm_sq) > delta
    r = b - A(x_init)
    loop_vars = (x_init, tf.constant(0), dot(r, r), r, r)
    return tf.while_loop(cond, body, loop_vars)[::3]
```
Software implementation and numerical examples

- based on CVXPY, a convex optimization modeling framework
- solves convex problems using TensorFlow
- implements a variant of SCS, a first-order method
- linear subproblems solved with conjugate gradient
- experiment platform details
  - 32-core Intel Xeon 2.2Ghz processor
  - nVidia Titan X GPU with 12GB RAM
Nonnegative deconvolution example

\[
\begin{align*}
\text{minimize} & \quad \| c \ast x - b \|_2 \\
\text{subject to} & \quad x \geq 0,
\end{align*}
\]

with variable \( x \in \mathbb{R}^n \), problem data \( c \in \mathbb{R}^n \), \( b \in \mathbb{R}^{2n-1} \)

```python
from cvxpy import *
from cvxflow import scs_tf
x = Variable(n)
f = norm(conv(c, x) - b, 2)
prob = Problem(Minimize(f), [x >= 0])
scs_tf.solve(prob)
```
Comparison on nonnegative deconvolution

![Graph comparing memory usage and GPU solve time for SCS Native and SCS TensorFlow across different input sizes.](image)

- **Memory usage (GB)**
  - SCS Native: 0.36, 0.9, 0.47, 1, 10.4
  - SCS TensorFlow: 0.36, 0.9, 0.47, 1, 1.3

- **GPU solve time (seconds)**
  - SCS Native: 2, 5.7, 2, 3.2, 105
  - SCS TensorFlow: 13, 3.2, 13, 5.7, 2

Input size:
- 100
- 1000
- 10000
Conclusions

- convex optimization is useful
- convex modeling languages make it easy
- graph-based architectures help it scale
- open source Python libraries available
  - cvxpy: cvxpy.org
  - cvxflow: github.com/cvxgrp/cvxflow
More details for nonnegative deconvolution

|                        | small      | medium     | large      |
|------------------------|------------|------------|------------|
| variables $n$          | 101        | 1001       | 10001      |
| constraints $m$        | 300        | 3000       | 30000      |
| nonzeros in $A$        | 9401       | 81601      | 6922001    |

**SCS native**

|                        | CPU        | GPU        | CPU        |
|------------------------|------------|------------|------------|
| solve time             | 0.1 secs   | 2.0 secs   | 2.2 secs   |
| solve time, GPU        | 2.0 secs   | 2.0 secs   | 2.0 secs   |
| matrix build time      | 0.01 secs  | 0.6 secs   | 0.6 secs   |
| memory usage           | 360 MB     | 470 MB     | 10.4 GB    |
| objective              | $1.38 \times 10^0$ | $4.57 \times 10^0$ | $1.41 \times 10^1$ |
| SCS iterations         | 380        | 100        | 160        |
| avg. CG iterations     | 8.44       | 2.95       | 3.01       |

**SCS TensorFlow**

|                        | CPU        | GPU        | CPU        |
|------------------------|------------|------------|------------|
| solve time             | 3.4 secs   | 5.7 secs   | 5.7 secs   |
| solve time, GPU        | 5.7 secs   | 3.2 secs   | 3.2 secs   |
| graph build time       | 0.8 secs   | 0.8 secs   | 0.9 secs   |
| memory usage           | 895 MB     | 984 MB     | 984 MB     |
| objective              | $1.38 \times 10^0$ | $4.57 \times 10^0$ | $1.41 \times 10^1$ |
| SCS iterations         | 480        | 100        | 160        |
| avg. CG iterations     | 2.75       | 2.00       | 2.00       |