Domain Specific Languages for Convex Optimization

Stephen Boyd

joint work with E. Chu, J. Mattingley, M. Grant

Electrical Engineering Department, Stanford University

ROKS 2013, Leuven, 9 July 2013

Outline

Convex optimization

Constructive convex analysis

Cone representation

Canonicalization

Parser/solvers and parser/generators

Conclusions

Outline

Convex optimization problem — standard form

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

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

Convex optimization problem — 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)
- the modern canonical form

Why convex optimization?

- beautiful, fairly complete, and useful theory

Why convex optimization?

- beautiful, fairly complete, and useful theory
- solution algorithms that work well in theory and practice
  - convex optimization is **actionable**

Why convex optimization?

- beautiful, fairly complete, and useful theory
- solution algorithms that work well in theory and practice
  - convex optimization is **actionable**
- **many applications** in
  - control
  - combinatorial optimization
  - signal and image processing
  - communications, networks
  - circuit design
  - machine learning, statistics
  - 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
- transform your problem into a standard form, and use a standard solver
  - extends reach of problems solvable by standard solvers
- **this talk:** methods to formalize and automate last approach

Outline

How can you tell if a problem is convex?

approaches:

- use basic definition, first or second order conditions, *e.g.*, $\nabla^2 f(x) \succeq 0$
- via convex calculus: construct f using
  - library of basic functions that are convex
  - calculus rules or transformations that preserve convexity

Convex functions: Basic examples

- x^p ($p \geq 1$ or $p \leq 0$), $-x^p$ ($0 \leq p \leq 1$)
- e^x, $-\log x$, $x \log x$
- $a^T x + b$
- $x^T P x$ ($P \succeq 0$)
- $\|x\|$ (any norm)
- $\max(x_1, \ldots, x_n)$

Convex functions: Less basic examples

- $x^T x/y$ $(y > 0)$, $x^T Y^{-1} x$ $(Y \succ 0)$
- $\log(e^{x_1} + \cdots + e^{x_n})$
- $-\log \Phi(x)$ (Φ is Gaussian CDF)
- $\log\det X^{-1}$ $(X \succ 0)$
- $\lambda_{\max}(X)$ $(X = X^T)$

Calculus rules

- **nonnegative scaling**: f convex, $\alpha \geq 0 \implies \alpha f$ convex
- **sum**: f, h convex $\implies f + g$ convex
- **affine composition**: f convex $\longrightarrow f(Ax + b)$ convex
- **pointwise maximum**: $f_1, \ldots, f_m$ convex $\implies \max_i f_i(x)$ convex
- **partial minimization**: $f(x, y)$ convex $\implies \inf_y f(x, y)$ convex
- **composition**: h convex increasing, f convex $\implies h(f(x))$ convex

Examples

from basic functions and calculus rules, we can show convexity of ...

- piecewise-linear function: $\max_{i=1,\ldots,k}(a_i^T x + b_i)$
- ℓ_1-regularized least-squares cost: $\|Ax - b\|_2^2 + \lambda\|x\|_1$, with $\lambda \geq 0$
- sum of largest k elements of x: $x_{[1]} + \cdots + x_{[k]}$

A general composition rule

$h(f_1(x), \ldots, f_k(x))$ 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
- this one rule subsumes most of the others
- in turn, it can be derived from the partial minimization rule

Constructive convexity verification

- start with function given as **expression**
- build parse tree for expression
  - leaves are variables or constants/parameters
  - nodes are functions of children, following general rule
- tag each subexpression as convex, concave, affine, constant
  - variation: tag subexpression signs, use for monotonicity
    e.g., $(\cdot)^2$ is increasing if its argument is nonnegative
- sufficient (but not necessary) for convexity

Example

for $x < 1$, $y < 1$

$$\frac{(x-y)^2}{1-\max(x,y)}$$

is convex

- (leaves) x, y, and 1 are affine expressions
- $\max(x,y)$ is convex; $x - y$ is affine
- $1 - \max(x,y)$ is concave
- function u^2/v is convex, monotone decreasing in v for $v > 0$
  hence, convex with $u = x - y$, $v = 1 - \max(x,y)$

Example

- $f(x) = \sqrt{1+x^2}$ is convex
- but cannot show this using constructive convex analysis
  - (leaves) 1 is constant, x is affine
  - x^2 is convex
  - $1+x^2$ is convex
  - but $\sqrt{1+x^2}$ **doesn't match general rule**
- writing $f(x) = \|(1,x)\|_2$, however, works
  - $(1,x)$ is affine
  - $\|(1,x)\|_2$ is convex

Disciplined convex programming (DCP)

- framework for describing convex optimization problems
- based on constructive convex analysis
- sufficient but not necessary for convexity
- basis for several domain specific languages and tools for convex optimization

Disciplined convex program: Structure

a DCP has

- zero or one **objective**, with form
  - minimize {scalar convex expression} or
  - maximize {scalar concave expression}

- zero or more **constraints**, with form
  - {convex expression} <= {concave expression} or
  - {concave expression} >= {convex expression} or
  - {affine expression} == {affine expression}

Disciplined convex program: Expressions

- expressions formed from
  - **variables**,
  - **constants/parameters**,
  - and **functions** from a library
- library functions have known convexity, monotonicity, and sign properties
- all subexpressions match general composition rule

Disciplined convex program

- a valid DCP is
  - convex-by-construction (cf. posterior convexity analysis)
  - 'syntactically' convex (can be checked 'locally')
- convexity depends only on attributes of library functions, and not their meanings
  - *e.g.*, could swap $\sqrt{\cdot}$ and $\sqrt[4]{\cdot}$, or $\exp \cdot$ and $(\cdot)_+$, since their attributes match

Outline

Cone representation

(*Nesterov, Nemirovsky*)

cone representation of (convex) function f:

- $f(x)$ is optimal value of cone program

$$\begin{array}{ll} \text{minimize} & c^T x + d^T y + e \\ \text{subject to} & A \left[\begin{array}{c} x \\ y \end{array} \right] = b, \quad \left[\begin{array}{c} x \\ y \end{array} \right] \in \mathcal{K} \end{array}$$

  - cone program in (x, y), we but minimize only over y
- *i.e.*, we define f by partial minimization of cone program

Examples

- $f(x) = -(xy)^{1/2}$ is optimal value of SDP

$$\begin{array}{ll} \text{minimize} & -t \\ \text{subject to} & \begin{bmatrix} x & t \\ t & y \end{bmatrix} \succeq 0 \end{array}$$

  with variable t

- $f(x) = x_{[1]} + \cdots + x_{[k]}$ is optimal value of LP

$$\begin{array}{ll} \text{minimize} & \mathbf{1}^T\lambda - k\nu \\ \text{subject to} & x + \nu\mathbf{1} = \lambda - \mu \\ & \lambda \succeq 0, \quad \mu \succeq 0 \end{array}$$

  with variables λ, μ, ν

SDP representations

Nesterov, Nemirovsky, and others have worked out SDP representations for many functions, *e.g.*,

- x^p, $p \geq 1$ rational
- $-(\det X)^{1/n}$
- $\sum_{i=1}^k \lambda_i(X)$ $(X = X^T)$
- $\|X\| = \sigma_1(X)$ $(X \in \mathbf{R}^{m \times n})$
- $\|X\|_* = \sum_i \sigma_i(X)$ $(X \in \mathbf{R}^{m \times n})$

some of these representations are not obvious . . .

Outline

Canonicalization

- start with problem in DCP form, with cone representable library functions
- **automatically** transform to equivalent cone program

Canonicalization: How it's done

- for each (non-affine) library function $f(x)$ appearing in parse tree, with cone representation

$$\begin{array}{ll} \text{minimize} & c^T x + d^T y + e \\ \text{subject to} & A \begin{bmatrix} x \\ y \end{bmatrix} = b, \quad \begin{bmatrix} x \\ y \end{bmatrix} \in \mathcal{K} \end{array}$$

  - add new variable y, and constraints above
  - replace $f(x)$ with affine expression $c^T x + d^T y + e$

- yields problem with linear equality and cone constaints
- DCP ensures equivalence of resulting cone program

Outline

Parser/solvers and parser/generators

- **parser/solver** (CVX, YALMIP)
  - canonicalize problem *instance* (with numeric parameters)
  - solve using cone program solver

Parser/solvers and parser/generators

- **parser/solver** (CVX, YALMIP)
  - canonicalize problem *instance* (with numeric parameters)
  - solve using cone program solver

- **parser/generator** (CVXGEN, QCML)
  - canonicalize problem *family* (with symbolic parameters)
  - generate mapping from original problem to cone program
  - connect to generic (QCML) or custom (CVXGEN) cone program solver

Example

- constrained least-squares problem with ℓ_1 regularization

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 + \lambda\|x\|_1 \\ \text{subject to} & \|x\|_\infty \leq 1 \end{array}$$

  - variable $x \in \mathbf{R}^n$
  - constants/parameters A, b, $\lambda > 0$

CVX

- parser/solver (M. Grant)
- embedded in Matlab; targets multiple cone solvers

- CVX specification for example problem:

```
cvx_begin
  variable x(n)    % declare vector variable
  minimize (sum(square(A*x-b,2)) + lambda*norm(x,1))
  subject to norm(x,inf) <= 1
cvx_end
```

- here A, b, λ are **constants**

Some functions in the CVX library

function	meaning	attributes
`norm(x, p)`	$\|x\|_p$, $p \geq 1$	cvx
`square(x)`	x^2	cvx
`square_pos(x)`	$(x_+)^2$	cvx, nondecr
`pos(x)`	x_+	cvx, nondecr
`sum_largest(x,k)`	$x_{[1]} + \cdots + x_{[k]}$	cvx, nondecr
`sqrt(x)`	$\sqrt{x}$, $x \geq 0$	ccv, nondecr
`inv_pos(x)`	$1/x$, $x > 0$	cvx, nonincr
`max(x)`	$\max\{x_1, \ldots, x_n\}$	cvx, nondecr
`quad_over_lin(x,y)`	x^2/y, $y > 0$	cvx, nonincr in y
`lambda_max(X)`	$\lambda_{\max}(X)$, $X = X^T$	cvx
`huber(x)`	$\begin{cases} x^2, & \|x\| \leq 1 \\ 2\|x\| - 1, & \|x\| > 1 \end{cases}$	cvx

CVXGEN

- parser/generator (J. Mattingley)
- domain specific input
- emits flat C source that solves problem family
- goal:
  - spend (perhaps much) time generating code
  - save (hopefully much) time solving problem instances

CVXGEN specification

- CVXGEN specification for example problem:

```
parameters
  lambda positive
  A(m,n)
  b(m)
end

variables
  x(n)
end

minimize
  sum(square(A*x - b)) + lambda*norm1(x)
subject to
  norm_inf(x) <= 1
end
```

- here A, b, λ are **symbolic parameters**

Sample solve times for CVXGEN generated code

(on quad-core 3.4GHz Xeon with 16GB of RAM)

problem	vars	constrs	SDPT3 (ms)	CVXGEN (ms)
`portfolio`	110	111	350	0.4
`svm`	111	200	510	0.6
`generator`	286	620	470	1.5
`battery`	144	289	310	0.3

Quadratic cone modeling language (QCML)

- parser/generator (E. Chu)
- domain specific input; parser embedded in Python
- targets CVXOPT in Python
- can generate source code for several targets
- goal: seamless transition from prototyping to code generation

QCML specification

- full Python source

```
from qcml import QCML
p = QCML()     # QCML parser object
p.parse("""    # QCML begin
  dimensions m n
  parameters A(m,n) b(m)
  parameter lambda positive
  variable x(n)
  minimize sum(square(A*x - b)) + lambda*norm1(x)
    norm_inf(x) <= 1
""")            # QCML end
# canonicalize the problem
p.canonicalize()
```

Using QCML as parser/solver

- once canonicalized, create a Python solver

```
p.codegen("cvxopt") # creates Python source code
f = p.solver        # bytecode for solver function
```

Using QCML as parser/solver

- once canonicalized, create a Python solver

```
p.codegen("cvxopt") # creates Python source code
f = p.solver        # bytecode for solver function
```

- `f` is a Python function mapping parameters into solutions

```
sol = f(params)     # solution for problem instance
```

  - `params` is a dictionary holding parameter values
  - `sol` is a dictionary holding optimal value, solver status, ...

Using QCML as parser/solver

- once canonicalized, create a Python solver

```
p.codegen("cvxopt") # creates Python source code
f = p.solver        # bytecode for solver function
```

- f is a Python function mapping parameters into solutions

```
sol = f(params)     # solution for problem instance
```

  - params is a dictionary holding parameter values
  - sol is a dictionary holding optimal value, solver status, ...

- combine canonicalize, codegen, and solver

```
sol = p.solve(params)
```

- recreates CVX-like functionality

Using QCML as parser/generator

- once canonicalized, create external source code

```
p.codegen("ecos")   # creates C solver source code
```

Using QCML as parser/generator

- once canonicalized, create external source code

```
p.codegen("ecos")   # creates C solver source code
```

- generates folder with
  - C source that maps problem parameters into SOCP
  - C source that maps SOCP solution into problem solution
  - Makefile
- links with external solver, in this case, ECOS

Using QCML as parser/generator

- once canonicalized, create external source code

```
p.codegen("ecos")    # creates C solver source code
```

- generates folder with
  - C source that maps problem parameters into SOCP
  - C source that maps SOCP solution into problem solution
  - Makefile
- links with external solver, in this case, ECOS
- recreates CVXGEN-like functionality

Using QCML as parser/generator

- once canonicalized, create external source code

```
p.codegen("ecos")   # creates C solver source code
```

- generates folder with
  - C source that maps problem parameters into SOCP
  - C source that maps SOCP solution into problem solution
  - Makefile
- links with external solver, in this case, ECOS

- recreates CVXGEN-like functionality
- (eventually) target custom deployment context
  - embedded systems, GPGPU, clusters, ...

Outline

Conclusions

- DCP is a formalization of constructive convex analysis
  - simple method to certify problem as convex
  - basis of several domain specific languages for convex optimization
- parser/solvers make rapid prototyping easy
- parser/generators yield solvers that
  - are extremely fast
  - can be embedded in real-time applications
- hybrid solution unifies prototyping and deployment

References

- *Disciplined Convex Programming* (Grant, Boyd, Ye)
- *Graph Implementations for Nonsmooth Convex Programs* (Grant, Boyd)
- *Automatic Code Generation for Real-Time Convex Optimization* (Mattingley, Boyd)
- *Code Generation for Embedded Second-Order Cone Programming* (Chu, Parikh, Domahidi, Boyd)

- CVX (Grant, Boyd)
- CVXGEN (Mattingley, Boyd)
- QCML (Chu, Boyd)