Chapter VIII
Conclusions

- What we did
- What remains to be done

Lecture 33
Summary and open problems

---

some final thoughts and conclusions
(& stuff we didn't cover)

- refining a statement from lecture 1
- ellipsoid algorithm vs. IP methods
- choosing an algorithm:
  - run-time vs. code-time tradeoff
  - existing vs. custom software
- exploiting structure
- non-convex problems

Refining a statement from lecture 1

"convex problems are easy to solve"

. . . **if** you can evaluate functions and subgradients (or gradients & Hessians) efficiently

example: yield maximization

$$\text{maximize } \mathbf{Prob}(x + v \in C)$$

- variable is x
- C convex
- v random variable with log-concave probability density

a convex problem, but evaluating function & derivatives usually requires Monte-Carlo simulation

Ellipsoid algorithm vs. IP methods

ellipsoid algorithm

- "old workhorse" (slow, reliable, easy to use)
- easy to code — usually if you can compute $f(x)$ you can compute a $g \in \partial f(x)$ with little additional effort
- easy to understand (one page analysis)
- works for nondifferentiable functions
- often best for FTTA situations (First Time To Answer), at least for small problems
- for any given problem, there's always a better algorithm!

interior-point methods

- fast, especially for larger problems
- handles very many (but not all) convex problems
- harder to code (and debug, maintain) than ellipsoid method
- algorithm parameter(s) can affect performance (but not hard to tune)

What algorithm to use?

important factors:

- problem size
- how many instances to solve? can range from 1 to very very large
- run-time efficiency
- coding time & effort: ranges from zero (code exists) to ...
- existing codes

time to answer =

$$\text{code-time} + \#\text{instances} \times \text{run-time}$$

examples:

- LP: use existing, highly optimized code
  (unless you need to and can exploit engineering structure)
- small problem, few instances, no existing code, just need answer: ellipsoid
- real-time embedded code — probably IP or custom

Using existing software

existing software:

- LP (many, sophisticated)
- QP (fewer)
- SOCP, SDP (some)
- others soon (entropy, GP, ...)

some tricks for using existing software:

- **embed** or **transform** problem, *e.g.*,
  - Chebychev via LP
  - matrix norm via SDP
  - add variables so problem becomes sparse
- **approximate** problem, *e.g.*,
  - approximate nonlinear convex functions as piecewise linear
  - yields LP

lose on efficiency, possibly accuracy; gain in code time some useful websites

- `http://plato.la.asu.edu/guide.html`
  (decision tree for optimization software)
- `http://www.mcs.anl.gov/home/otc/Guide`
  (NEOS guide of optimization software)
- `http://gams.cam.nist.gov`
  (guide to available mathematical software)

Custom software

disadvantage: code development effort

advantage: can exploit particular problem structure

- sparsity (speeds up linear algebra), *e.g.*, $v = -(\nabla^2 f)^{-1}\nabla f$
- many other types of structure, *e.g.*, Toeplitz/FFT
- can find good primal & dual starting points, *e.g.*, heuristic or approximate solution, warm start

speedup can be dramatic (orders of magnitude)

Exploiting structure

main effort in IP methods: computing Newton step

example. LP (n variables, m constraints)

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

Hessian, gradient of $tc^T x - \sum_{i=1}^{m} \log(b_i - a_i^T x)$:

$$H = A^T D^{-2} A, \quad g = tc - A^T D^{-1} \mathbf{1}, \quad D = \textbf{diag}\,(b - Ax)$$

- if A sparse, then (often) H sparse, hence can solve $Hv = -g$ via sparse Cholesky, etc.
- if can evaluate Aw and $A^T y$ fast, can use CG to solve $Hv = -g$
- if n moderate (say, < 1000), m large
  - use structure to form H, g fast
  - solve $Hv = -g$ using dense Cholesky

Non-convex optimization problems

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

– *theory*: duality results are weaker; optimality conditions are necessary but not sufficient

– *global* minimum is very hard to compute

– most convex optimization methods (*e.g.*, Newton's method, SUMT) still apply, but find only local minimum

Robust optimization

Robust optimization is one important application of convex optimization

- Usually NP-hard problems
- Using "linear-fractional representations", can make those problem **quadratic**
- Then apply a standard Lagrange relaxation approach
- Yields guaranteed bounds on the original problem, in polynomial-time
- Important point is to **exploit structure** of perturbation

  (otherwise, overly pessimistic)

relaxation method:

- Forget uncertainty, formulate, or approximate, the "nominal problem" via SDP
- Apply Lagrange relaxation for robust SDP

Applications

Of convex optimization:

- Not always easy to recognize
- "think in matrix terms!"
- easy to solve

Of robust optimization:

- Easy to recognize
- hard to solve
- Any convex opt application can be made robust (with some extra computer cost)

Challenges

- SDP and GP . . .
  Is there a broader structure??
- Estimate and improve quality of relaxations
- Large scale problems
- Sparse, low-rank solutions to SDP
- SDP in stochastic programming, optimal control

Acknowledgments

Many thanks to S. Boyd (Stanford) & L. Vandenberghe (UCLA)

Provided **LOTS** of nice stuff:

- inspiration
- most of these lecture notes
- reader
- sources, code
- etc