Outline:

How to solve most convex optimization problems, IRL.

Polynomial time.

Convex problems we can solve in polynomial time.

To solve: $\min_x f(x)$ s.t. $g(x) \le 0$,
where f and g are convex.

Can turn this into an unstrained problem.

Last lecture we learned $\exists\ \lambda \ge 0$ s.t.
the solution is the global minimizer of

$$h(x) \stackrel{\text{def}}{=} f(x) + \lambda g(x).$$

So, can try to solve this unconstrained problem.

How choose λ?

If $\lambda = 0$, are just minimizing f.

If $\lambda \to \infty$, just minimize g

As vary λ from 0 to ∞, let x_λ be solution
$f(x_\lambda)$ gets bigger and $g(x_\lambda)$ gets smaller.

Would like to do binary search on λ to find $(x_\lambda) = 0$,
or rather $-\varepsilon \le g(x_\lambda) \le 0$, which we hope is good enough.

As we can not binary search on $(0, \infty)$
instead consider $(1-t) f(x) + t g(x)$ for t in $[0,1]$,
and binary search on t.
That is, $\lambda = \frac{t}{1-t}$.

Interior Point Methods (aka Barrier Methods)
(See Lauritzen §10.5)

α-release: Consider $\min_x f(x) + \varepsilon \frac{1}{-g(x)}$ s.t. $g(x) \leq 0$

Is constrained and looks more complicated.
But is useful if can start at an x_0 s.t. $g(x_0) < 0$

$-g(x)$ is concave, so $\frac{1}{-g(x)}$ is again convex,
But as $g(x) \to 0$ $\frac{1}{-g(x)} \to \infty$.
So any sort of local search (like gradient methods)
will stay inside region where $g(x_0) < 0$

Usual idea is to first solve it when ε is
big, and then lower ε. Each time using
previous solution as starting point.

β-release: Instead use $f(x) - \varepsilon \log(-g(x))$

As before, $-\log(-g(x))$ is convex for $g(x) \le 0$

And, $-\log(-g(x)) \to \infty$ as $g(x) \to 0$ (from below)

This combination works better in practice.

Version 1.0: Solve $\min f(x)$ s.t. $g_1(x), \dots, g_k(x) \le 0$

by $\min f(x) \underbrace{- \varepsilon \sum_{i=1}^{k} \log(-g_i(x))}_{\text{Barrier Function.}}$

Start with ε big.

Decrease ε on a multiplicative schedule, like

$\varepsilon \leftarrow \varepsilon(1 - 1/\sqrt{n})$

Don't need high accuracy solutions until $\varepsilon \to 0$.

Solve each problem starting from solution to previous.

Usually do solves using Newton's method, or a line search in Newton direction, instead of Gradient.

Newton: locally approximate F by Taylor expansion to second order:

$$F(x+\delta) \approx F(x) + \nabla F(x)^T \delta + \frac{1}{2} \delta^T (\nabla^2 F(x)) \delta$$

where $\nabla^2 F(x)$ is the Hessian - the matrix of $\frac{\partial F}{\partial x_i \partial x_j}$

The δ that minimizes this is $-(\nabla^2 F(x))^{-1} \nabla F(x)$

So, step there, or to a point along the line to it.

Solves quadratics in one step!
Usually needs very few steps.
But, each step requires solving a system of linear equations, which can be slow.

Karmarkar proved this method solves Linear Programs in polynomial time.
A lot of fast code is based on this.

Are extensions that work for many other nice convex optimization problems, such as those specified by generalized inequalities - when the cones are well understood.
(e.g. Positive Semidefinite Matrices)

Polynomial Time - what do we mean by this?

An algorithm runs in polynomial time if on inputs of size n, it always performs $\leq O(n^c)$ operations, for some fixed constant c.

For example, Gaussian Elimination of k-by-k matrices has $n = k^2$ numbers as input, and runs in time $\leq O(k^3) = O(n^{3/2})$.

There is an issue with how we measure input size.
To do this right, we should use bits.
This usually doesn't change much:

adding and comparing b-bit #s takes time $\leq O(b)$.
Multiplying two b bit #s takes time $\leq O(b \log b)$
[That's a recent breakthrough]

But, during LU factorization the #s we encounter require more bits.
If we want to do it exactly, we need to count those.

$(x_n = 3^{2^{n-1}})$

Problems are like: $x_1 = 3$, $x_i = x_{i-1}^2$ — grows very quickly.

Need to be careful if just want to approximate, or use floating point.

We will see that LU factorization is polynomial time, even when count bits.

Linear Programming is as well.

But, the # of arithmetic operations grows with the # of bits.

This is one difference between LU - which requires $\leq O(n^{3/2})$ arithmetic ops regardless of the numbers in the input, and LP, where for every known poly. time algorithm the # of arithmetic ops can grow with the # of bits in the input.

This is rarely an important distinction.

What's going on with LP is that the # of steps depends on the log of the condition number, and this can grow with the number of bits.

Approximately solving LP: given c, A, b, and $\varepsilon > 0$

We can find an x s.t. $Ax \leq b$

$$\text{and} \quad c^T x \geq c^T x_* - \varepsilon$$

In time polynomial in # bits used to write c, A, b, ε.

Once ε is small enough, we can round to an exact solution.
And, this is polynomial time, too.

In order for (exact) LP to be in polynomial time, the size of the answer must be bounded by a polynomial in the size of the input. We will verify that this holds.

We first need to do it for systems of linear equations, like $Ax=b$. (following Korte, § 4.1)

Since we are talking about exact solutions, we will assume that each real number is a rational. We won't use floating point, because we want $3x=1$ to be solvable, and you can't write $1/3$ in floating point.

You can write an integer between 0 and 2^b-1 using b bits. But, you first have to announce how many bits you will use. And, how many bits you will use to do that...

The solution is to use Elias' Ω-codes.

A simpler solution uses $1+\lceil \log_2 b \rceil + b$ bits.

We will just use the fact that an integer x can be represented with $\leq 1+2\lceil \log_2 |x| \rceil$ bits.

Let's define $\mathrm{size}(x)$ to be the # of bits we need to write x.

For $x = y/z$ where y and z are integers,
$\text{size}(x) \le \text{size}(y) + \text{size}(z)$.

For integers $y_1, \dots, y_n$,
$\text{size}(\prod_i y_i) \le \sum_i \text{size}(y_i)$, because

$$\lceil \log \prod_i y_i \rceil \le \sum_i \lceil \log y_i \rceil$$

For rationals $x_i = y_i/z_i$,
$\text{size}(\prod_i x_i) \le \sum_i \text{size}(x_i)$

because $\prod_i x_i = \dfrac{\prod_i y_i}{\prod_i z_i}$

$\text{Size}(\sum_i x_i) \le 2 \sum_i \text{size}(x_i)$.

<u>proof</u> $\sum_i x_i = \dfrac{\sum_{i=1}^{n} y_i \sum_{j \ne i} z_i}{\prod_{j=1}^{n} z_i}$

which has size $\le \sum_{j=1}^{n} \text{size}(i) \le \sum_i \text{size}(x_i)$
for the denominator

and the numerator has absolute value at most
$(\sum_i |y_i|) |\prod z_i|$

which again has size $\le \sum_i \text{size}(x_i)$

We represent an n-vector $(x_1, \ldots, x_n)$ by writing n followed by $x_1 \ldots x_n$. So, its size is at most $\text{size}(n) + \sum_i \text{size}(x_i)$.

<u>Lem</u> For an n-by-n matrix A,

$$\text{size}(\det(A)) \leq 2\,\text{size}(A)$$

<u>proof</u> Recall $\det(A) = \sum_{\text{perms } \pi} (-1)^{\text{sgn}(\pi)} \prod_{i=1}^{n} A(i, \pi(i))$

Write $A(i,j) = y_{i,j} / z_{i,j}$ for integers.
Then $\det(A) = p/q$ is rational with denominator $q \leq \prod (z_{i,j})$, and

$$|\det(A)| \leq \prod_{i,j} (1 + |y_{i,j}|)$$

As $\text{size}(1 + |A_{i,j}|) \leq 2\,\text{size}(A_{i,j})$,
the result follow from the product formula above.

Lem If $Ax = b$, then $\text{size}(x) \leq 4n(\text{size}(A) + \text{size}(b))$.

proof Cramer's rule says that $x(i) = \dfrac{\det(A_i)}{\det(A)}$,

where A_i is the matrix obtained by replacing the ith column of A by b.

So, $\text{size}(x(i)) \leq 2(\text{size}(A) + (\text{size}(A) + \text{size}(b)))$

And, $\text{size}(x) \leq \text{size}(n) + 4n\,\text{size}(A) + 2n\,\text{size}(b)$

$\leq 4n(\text{size}(A) + \text{size}(b))$

One can push this further to confirm that LU factorization is polynomial time.

<u>Thm</u> The LP $\max c^T x$ s.t. $Ax \le b$ has a solution vector x_* of size $\le 4d(\text{size}(A) + \text{size}(b))$ for n-by-d A.

<u>proof sketch</u> If there is a strictly feasible ~~x_0~~, then there exists a set S of d constraints such that $A(S, \cdot)\, x = b(S)$. We then apply the previous lemma.

Approximate solutions to convex optimization problems.

Say we want to solve $\min_x f(x)$ s.t. $x \in C$ for convex f and C.

Want an ε-approx solution: x s.t. $\exists\, x_*$ s.t. $\|x - x_*\| \le \varepsilon$

Need to be able to tell if $x \in C$.
Call a function that does this a "Membership Oracle"

and we count how many times we call it.

A "separation oracle" returns a separating hyperplane when $x \notin C$, and is even more useful.

Also need to be able to evaluate f.
This is all simpler if $f(x) = c^T x$.
If not, need to worry about a "Function Oracle" and the size of numbers it returns.
One detail: assume can ask for low precision:
on input f, δ it returns a rational r
s.t. $|r - f(x)| \leq \delta$.

We will just count # of times need to evaluate f, so whole algorithm is polynomial if f is.

To use C, we need to know $x_0 \in C$ and numbers r and R s.t. $B(x_0, r) \subseteq C \subseteq B(x_0, R)$

Then, are algorithms that give an ε-approx solution in time polynomial in
$\log(1/\varepsilon)$, $\log(R/r)$, $\text{size}(x_0)$, and n.

See Ellipsoid Algorithm in Grötschel-Lovász-Schrijver. '93
or Random Walk approach of Bertsimas-Vempala '04
or Recent paper of Lee-Sidford-Wong '15

One can even weaken the notions to approximate membership, etc.