**CS294 Algorithms Toolkit** **Fall 2008**

# Lecture 9 ..ish Iterative Methods for Linear Systems:

*Lecturer: Satish Rao* *Scribe: Rao, for now*

**Disclaimer**: *These are rough notes, with some exercises.*

## 9.1 Iterative Methods for Linear Systems.

**Question: What's a linear system?**

Solve the equations

$$Ax = b.$$

**Question: How? How much time?**

Well, Gaussian elimination. Well (ignoring floating point issues) $O(n^3)$ time.

**Question: An iterative method?**

Well, we can do the following. Start at an arbitrary (perhaps random $x$), measure the error, and try to change $x$ to reduce the error.

We will be concerned with symettric matrices. Indeed, specifically matrices that arise as Laplacian matrices of graphs (these are symmetric matrices with negative weights on edges and positives on the diagonal that are more than the negative weights on the edges.) One can use linear algebra tricks to get somewhat more general matrices. But that is for another day.

**Question: How does one characterize a minimum of $f(x) = \frac{1}{2}x^T Ax - bx + c$?**

Take the gradient, and find out where it is 0.

Well, the gradient is

$$\frac{1}{2}(A + A^T)x - b = Ax - b.$$

We should set it to 0. Wow, this happens when $Ax = b$.

We note this is only true for matrices that are positive definite (all eigenvalues are positive). But so it is for "diagonally dominant" matrices.

**Question: How do we minimize?**

Gradient descent. That is, when at some $x_i$, and then go along direction that reduces the value, i.e., along the gradient direction which is $b - Ax_i$ (taking the negative to decrease the function.)

That is, we should set

$$x_{i+1} = x_i + \alpha_i r_i$$

where

$$r_i = b - Ax_i.$$

We term the $r_i$'s to be the residual. (We also define $e_i$ to be $x_i - x$, though we never get to see this quantity explictly.)

**Question: How far should we go? What should $\alpha_i$ be?**

Well, until we improve no longer. That is, when $f(x_{i+1})$ is minimal along this direction. Since $f$ is a quadratic, this is when the gradient is orthogonal to the line. The gradient at the $i+1$st step is

$$b - Ax_{i+1} = r_{i+1},$$

is the next residual.

We thus wish the residuals to be orthogonal. That is,

$$r_{i+1}^T r_i = 0$$

Expanding out, we get

$$\begin{aligned} r_{i+1} &= b - Ax_{i+1} && (9.1)\\ &= b - A(x_i + \alpha_i r_i) && (9.2)\\ &= r_i - \alpha_i A r_i && (9.3) \end{aligned}$$

If this is to be orthogonal ($r_i r_{i+1} = 0$), we get

$$\alpha_i = \frac{r_i^T r_i}{r_i^T A r_i}.$$

**Question: How does the error evolve?**

Well, it is how much we move toward $x$. Or, in particular,

$$e_{i+1} = e_i + \alpha_i r_i.$$

or

$$e_{i+1} = e_i + \alpha_i (A e_i).$$

**Question: Is it reducing?**

It is complicated to consider the $\alpha_i$. So, for intuition we consider a fixed $\alpha$.

Let's consider the eigenvectors of $A$, with eigenvalues $\lambda_j$. We can represent $e_i$ in this basis, with $a_j$ being the component along the $j$th eigenvector. The $a_j$ evolve by multiplying by $(1 - \alpha\lambda_j)$ in each step. Now, to minimize the magnitude of the maximum such multiplier, we set $\alpha = 2/(\lambda_{min} + \lambda_{max})$. In this case, the absolute value of the multplier is at most $(1 - 1/\kappa)$ where $\kappa$ is $\lambda_{max}/\lambda_{min}$.

That, is we get a constant factor of progress every $\kappa$ iterations.

**Question: Can we do better?**

The conjugate gradient method reduces this to $\sqrt{\kappa}$ (among other things, e.g., it also ensures that no more than $n$ iterations are required.)

**Question: What is preconditioning?**

We can actually solve the system.

$$B^{-1}Ax = B^{-1}b.$$

The iteration now breaks up into two parts.

From above, one step appears as follows

$$r_{i+1} = b - B^{-1}Ax_{i+1}$$

Instead, we could find $z = Ax_{i+1}$ and compute $y$

$$By = z,$$

to obtain

$$B^{-1}Ax_{i+1}.$$

**Question: So?**

Find $B$ such that $B^{-1}A$ has small condition number, and $B$ is easy to find a solution for (e.g., a tree, or recursively.)

**Question: An example?**

$B = A$, $B^{-1}A = I$, great condition number. Oops, may not be easy to invert (original problem.)

**Question: A tree?**

Easy to invert, but good condition number?

**Question: Since we want to make a matrix like $A$, how do we compare matrices?**

Compare eigenvalues, or perhaps as follows.

When forall vectors $x$, we have

$$x^t Nx \geq x^T Mx,$$

we say $N$ is bigger or equal to $M$.

**Question: Is this interesting?**

Sure, if we have this in both directions, the matrices are the same.

If $M$ is bigger than 0 the matrix is positive definite. The notion holds for addition. The $i$th eigenvalue of $N$ is larger than the $i$th eigenvalue of $M$.

**Question: What happens for $B^{-1}A$?**

If $B$ is bigger than $A$ it decreases the maximum eigenvalue to one intuitively. If $B$ is smaller than $A$ it increase the least towards one.

(Caveat: this is not quite correct. It may not be symetric even. One actually needs to define the combined condition number of the system. But, it follows this intuition.)

**Question: Find $B$ that is close to $A$**

That is, $B$ is smaller than $A$ and $kB$ is bigger than $A$. The resulting condition number of $B^{-1}A$ will be $O(k)$.

We will now discuss graphs (and their associated Laplacian) matrices. Both will be discussed using the same symbol.

**Question: For laplacians, what is a subgraph of $A$?**

It will always be less than $A$ (as a matrix).

**Question: What might be bigger than $A$?**

Consider an edge $(u, v)$ and a path between $u$ and $v$. We have that the matrix corresponding to the edge is less than the matrix corresponding to the path where each edge is multiplied by the length of the path.

**Question: Why?**

The laplacian of an edge $(1, k)$ multiplied by a vector yields $(x_1 - x_k)^2$, while the laplacian of a path $(1, 2)(2, 3) \dots (k - 1, k)$ yiels $\sum_i (x_i - x_{i+1})^2$. Multiplying the later by $k$ dominates the former. (Cauchy-Schwartz, or just think about it.)

**Question: Let's consider a spanning tree of $A$?**

Each edge in $A$ can be mapped to a path in $T$. Now, step by step, we can multiply the edges in a path by the length of the path. The resulting graph is "larger" than $A$.

**Question: What is the maximum number that multiplies any edge?**

The total stretch over all edges in $A$ when mapped to paths in $T$. Is this small?

There is a theorem that states it can be done with average stretch $O(\log^2 n)$. Or total stretch $O(m \log^2 n)$.

**Question: Thus, what is the condition number of $B^{-1}A$?**

$O(m \log^2 n)$.

With conjugate gradients, we can get an $\tilde{O}(m^{1.5})$ ($\tilde{O}(\cdot)$ drops log factors) algorithm.

**Question: Can we do better?**

We can reduce the total stretch by clustering $t$ subtrees and adding $s$ (either $t^2$ or more cleverly $t \log t$) edges between subtrees. This can reduce the stretch by a factor of $t$.

**Question: Solving $By = z$?**

Now, we need to deal with the additional non-tree edges. We can eliminate in time $O(s^3)$. Thus, setting $t = m^{1/6}$, it costs no more per iteration. And we get an $m^{1/12}$ improvement in running time. Not the best tradeoff, but hopefully you get the idea.

**Question: Can we do better?**

For planar graphs, the number of edges is $O(t)$. Now, if the resulting graph $B$ is a tree with $O(t)$ edges. By eliminating leaves and degree two nodes, one gets an $O(t)$ node graph. By choosing $t = n/3\beta$, the number of remaining nodes divides by $\beta$. Moreover, the condition number of the $B^{-1}A$ is $O(1)$. Thus, a constant number of iterations of the outermost loop suffices, and the work decreases geometrically in inner loops. One needs only solve the inner loop to some precision to make progress in the outer loop.

**Question: The inner loops need to run a few iterations to get error down, how does this pan out?**

The idea is that the number of iterations is $\sqrt{\gamma}$. The outer needs $\sqrt{3\beta}$ iterations. The next level needs this many in each step, but it is on a graph that is $n/\beta$ size so the total work at this level is smaller by a factor of $1/\sqrt{\beta}$.

**Question: How about general graphs?**

We could sparsify from the beginning and get a graph with $O(n \log^c n)$ edges and proceed as above. And sparsifying at the "tree" plus other edges leaves $O(t \log^c n)$ edges. Thus, choosing $t = n/\beta \log^c n$ reduces the subproblem to one of size $n/\beta$. Here the number of iterations is at around $\sqrt{\beta \log^c n}$ in the outer and grows by this factor with each recursive call. Then again, the size shrinks by $1/\beta$ at each recursive call. This, when $\beta >> \log^c n$, we get a geometric series.

**Question: Did I pull a fast one?**

Yes, I spoke only of unweighted graphs, and the general graph solution produces weighted graphs. But this works in the natural manner (generalizing the notion of stretch in trees, choosing max weight edges between subtrees to choose, and a few more tricks.)