Lecture 22

Lecturer: Jonathan Kelner

1 Last time

Last time, we reduced solving sparse systems of linear equations $Ax = b$ where A is symmetric and positive definite to minimizing the quadratic form $f(x) = \frac{1}{2}x^T Ax - bx + c$.

The idea of steepest descent is to pick a point, find the direction of steepest decrease, step along that direction, and iterate until we get close to a zero. The appropriate directions turn out to be the residuals. We pick the step length to take us to the minimal f value along the line.

Each iteration requires only two matrix-vector multiplications. We can push this down to one by calculating the residuals as

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

This can allow floating point error to accumulate, though that can be fixed by occasionally calculating the residual using the original formula.

Today we'll talk about how to bound the error, and later how to get better performance.

2 Convergence Analysis of Steepest Descent

We study a very similar method that doesn't do the line search – it uses the same step length α at every iteration. Asymptotically, the best possible α and usual steepest descent have the same behavior.

Steepest descent doesn't depend on the basis at all, so we might as well pick the eigenbasis for analysis. The size of the error at each step certainly won't change if we switch bases, and it will be easier to see what's going on. For further cleanliness everything will be stated for 2x2 matrices – everything generalizes.

If we were trying to solve

$$\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

the exact solution would be $x_i = \frac{1}{\lambda_i} b$. Keep in mind that we're working in the eigenbasis to do the analysis, but the actual algorithm does not get to work in the basis, since finding the eigenbasis is as hard as inverting the matrx.

First, let's see what happens if we use $\alpha = 1$. Obviously this is not general, for example if the λs are greater than 2, but the algebra is enlightening.

Let's start with $x_0 = 0$, so the first residual $r_0 = b$, and for $i > 0$ the residual will be

$$\begin{aligned} r_i &= r_{i-1} - Ar_{i-1} \\ &= (1 - A)r_{i-1} \\ &= (1 - A)^i b \end{aligned}$$

where the last step follows from induction. Since $x_i = x_{i-1} + r_i = \sum_{k=0}^{i} r_k$, we can now write

$$x_k = \left[\sum_{i=0}^{k-1}(1-A)^i\right] b$$

But the sum $\sum_{i=0}^{k-1}(1-A)^i$ is just the first k terms of the taylor series $1/y$ around 1 – that is, x_k estimates $\frac{1}{\lambda_i}b$ using Taylor series!

For $\alpha \neq 1$ the computation is similar,

$$x_k = \left[\sum_{i=0}^{k-1}\alpha(1-\alpha A)^i\right] b$$

and we get another Taylor series approximation: $1/y$ around $1/\alpha$.

So how well can we choose α? We want the residuals to go to zero quickly. If we had 1×1 matrices, we could just set $\alpha = \frac{1}{\lambda}$ and get residual 0 in one step, but in general we need to choose α which works for different eigenvalues simultaneously.

Taylor series only converge well very near where you expand it; this gives some intuition for why the condition number should be related to the distance between $\lambda_{\max}$ and $\lambda_{\min}$. If these eigenvalues are far apart, then there is no α that works for all the eigenvalues.

We can bound the L_2 norm of the residual by bounding b_i by $||b||_2$ and taking the max of the multipliers

$$||r_k||_2 \leq \max_i|1-\alpha\lambda_i|^k||b||_2$$

So, we want to minimize

$$\max_i|1-\alpha\lambda_i|$$

Since the maximum will occur at either the largest or the smallest eigenvalue, the best we can do is to balance them and have $(1-\alpha\lambda_{\min}) = -(1-\alpha\lambda_{\min})$. This gives that the best α is the reciprocal of the midrange of the eigenvalues:

$$\alpha = \left(\tfrac{\lambda_{\min}+\lambda_{\max}}{2}\right)^{-1}$$

The resulting $\max_i|1-\alpha\lambda_i|$ is $1 - \frac{2}{\kappa+1}$ where $\kappa = \lambda_{\max}/\lambda_{\min}$, which we call the *condition number* of A. Note that κ is a ratio of eigenvalues, so it's unchanged by scaling the matrix.

From the bound for the L_2 norm, we can derive that the number of iterations grows linearly in κ. Now can we do better?

3 Conjugate Directions

Currently we are going to the minimal of f value along our search direction. As we saw in previous example, this can us to take a long zigzag path. What we would really like to do is go the length of the projection of x onto our search direction. If we could do that, then after i steps the error would be orthogonal to all previous search directions, and we'd be done with an $n \times n$ matrix after n iterations.

Suppose we have orthogonal directions $d_0, \ldots, d_{n-1}$ – the standard basis will do.

We have $x_{i+1} = x + i\alpha_i d_i$. We want $e_{i+1} \perp d_i$.

$$d_i^T e_{i+1} = d_i^T(e_i + \alpha_i d_i) = 0$$

which implies

$$\alpha_i = -\frac{d_i^T e_i}{d_i^T d_i}$$

The good news is we can compute everything except the e_i. The bad news is computing e_i is equivalent to finding x. Fortunately, a mild modification will make the calculation possible.

So far we've been talking about orthogonality relative to the standard inner product. There's no real reason to do this, and in fact it will be more convenient to work with the inner product $||x||_A^2 = x^T Ax$, instead of $x^T Ix$ as we have been. Geometrically, this unwarps the isolines of the quadratic form into perfect circles.

We can think of this as a change of basis: $x' = A^{1/2}x$, though not for computation – pretty much the only way to get the square root of A would be to retrieve the eigenvalues, which would defeat the purpose.

Suppose we have A-orthogonal search directions (d_i) – now the unit basis won't do, but suppose for the moment we have magically acquired search directions.

Again, $x_{i+1} = x_i + \alpha_i d_i$. We want $e_{i+1} \perp_A d_i$.

$$d_i^T A e_{i+1} = d_i^T A(e_i + \alpha d_i) = 0$$

which implies

$$\alpha_i = -\frac{d_i^T A e_i}{d_i^T A d_i}$$

But Ae_i is just r_i, which we do know how to compute. Yay.

4 Conjugate Gram-Schmidt

Conjugate directions is insufficient for our purposes because we might not have time to do n iterations. We'll settle for a crude answer, but we need it very fast.

Also, as mentioned before, we don't have search directions. You may recall the Gram-Schmidt process for orthogonalizing a set of vectors from a previous class. Does it work for A-orthogonality? Certainly; see page 5 of slides on Conjugate Gram-Schmidt.

The problem is that Conjugate Gram-Schmidt is still too slow. The crucial change we made to the algorithm is requiring each direction to be orthogonal to *all* previous search directions. While this gave us good convergence, it means we have to subtract off the projection into each of the previous directions, which means that we have to remember what the previous directions were. This incurs both time and space cost. We need a more sophisticated way to find the directions.

5 Conjugate Gradients

The trick is to choose the linearly independent vectors we feed to Gram-Schmidt very carefully. We will generate these vectors on the go. Define $D_i = \text{span}(d_0, \ldots, d_{i-1})$.

The property that we leverage is that after i steps, Conjugate Directions finds a point in $x_0 + D_i$ – in fact, the one that minimizes the size of the error $||e_i||_A = (e_i^T A e_i)^{1/2}$.

Let the input to Gram-Schmidt be (u_i), and define U_i analogously to D_i. By construction, $x_i \in x_0 + D_i = x_0 + U_i$, and e_i will be A-orthogonal to $D_i = U_i$.

We choose the magic inputs $u_i = r_i$. Since $r_{i+1} = -Ae_{i+1}$, by definition r_{i+1} is plain old orthogonal to D_{i+1} (and $D_i, D_{i-1}, \ldots$). Also, $r_{i+1} = r_i - \alpha_i A d_i$, so $D_{i+1} = D_i \cup AD_i$. Putting the two together, r_{i+1} is A-orthogonal to D_i.

Thus, r_{i+1} only A-projects onto the d_i component of D_{i+1}. There's only one thing to subtract off, so only one or two A-dot products are needed per iteration again, as in steepest descent. We no longer need to remember all the previous search directions, just the very last one, so we've fixed the space complexity as well.

The algorithm is given on a slide on page 6.

6 Convergence Analysis of Conjugate Gradients

After i iterations, the error is

$$e_i = \left(I + \sum_{j=1}^{i} \psi_j A^j \right) e_0$$

where the ψ's are some mess of α's and β's. Thus we can think of conjugate gradients at the ith step as finding these best possible coefficients for an ith degree polynomial $P_i(\lambda)$ to make the A-norm of the error small.

$$||e_i||_A^2 \leq \min_{P_i} \max_{\lambda \in \Lambda(A)} [P_i(\lambda)]^2 ||e_0||_A^2$$

Any sequence of i-degree polynomials which are 1 at 0 will give bounds on the error; we want ones which are small for every eigenvalue $\lambda \in \Lambda(A)$. This should remind you of the analysis of steepest descent, but Taylor Series are not the right choice here – they're designed to work around a point, while we want polynomials which will work at every eigenvalue. We can modify the magic polynomials from lecture 6 to work here. Recall that Chebyshev polynomials have the property of being 1 at 1, and small for some $[0, l]$ where $l < 1$ is a parameter. We want polynomials which are 1 at 0 and small in $[\lambda_{\min}, \lambda_{\max}]$. This allows us to bound the error (measured in the A-inner product) at the ith iteration as

$$||e_i||_A \leq 2 \left(1 - \frac{2}{\sqrt{k} + 1} \right)^i ||e_0||_A$$

so the number iterations grows with the square root of κ, which is way better than the linear performance of steepest descent.

Note that the algorithm isn't actually computing any Chebyshev polynomials – it uses the best polynomial, which is at least as good. Also, notice that if we knew the range of the eigenvalues to begin with, we could skip to designing a polynomial to estimate A^{-1}. Conjugate gradients magically finds the best polynomial without explicitly knowing these values.

18.409 Topics in Theoretical Computer Science: An Algorithmist's Toolkit
Fall 2009