Introduction to Conjugate Gradient

Tyler Chen

This is the first piece from a series on topics relating to the Conjugate Gradient algorithm. I have split up the content into the following pages:

- Introduction to Linear Systems/Krylov subspaces
- Derivation of CG
- Error bounds for CG and the Remez Algorithm
- Finite precision CG

All of the pages are available as a single pdf document.

Linear Systems

Solving a linear system of equations $Ax = b$ is one of the most important tasks in modern science. Applications such as weather forecasting, medical imaging, and training neural nets all require repeatedly solving linear systems.

Loosely speaking, methods for linear systems can be separated into two categories: direct methods and iterative methods. Direct methods such as Gaussian elimination manipulate the entries of the matrix A in order to compute the solution $x = A^{-1}b$. On the other hand, iterative methods generate a sequence $x_0, x_1, x_2, \ldots$ of approximations to the true solution $x^* = A^{-1}b$, where hopefully each iterate is a better approximation to the true solution.

At first glance, it may seem that direct methods are better. After all, after a known number of steps you get the exact solution. In many cases this is true, especially when the matrix A is dense and relatively small. The main drawback to direct methods is that they are not able to easily take advantage of sparsity. That means that even if A has some nice structure and a lot of entries are zero, direct methods will take the same amount of time and storage to compute the solution as if A were dense. This is where iterative methods come in. Often times iterative methods require only that the product $x \mapsto Ax$ be able to be computed. If A is sparse the product can be done cheaply, and if A has some known structure, a you might not even need to construct A. Such methods are aptly called “matrix free”.

The rest of this series gives an introduction to the analysis of Conjugate Gradient, a commonly used iterative method for solving $Ax = b$ when A is symmetric positive definite. My intention is not to provide a rigorous explanation of the topic, but rather to provide some (hopefully useful) intuition about where this method comes from and how it works in practice. I assume some linear algebra background (roughly at the level of a first undergrad course in linear algebra).

If you are a bit rusty on your linear algebra I suggest taking a look at the Khan Academy videos. For a more rigorous and much broader treatment of iterative methods, I suggest Anne Greenbaum's book on the topic. Finally, for a relatively recent overview of modern analysis of the Lanczos and Conjugate Gradient methods I suggest Gerard Meurant and Zdenek Strakos's report.

Measuring the accuracy of solutions

Perhaps the first question that should be asked about an iterative method is, "Does the sequence of approximate solutions $x_0, x_1, x_2, \ldots$ converges to the true solution? If this sequence doesn't converge to the true solution (or something close to the true solution), then it won't be very useful in solving $Ax = b$.

Let's quickly introduce the idea of the *error* and the *residual* of an approximate solution x_k. These are both useful measures of how close x_k is to $x^* = A^{-1}b$. The *error* is simply the difference between x and x_k. Taking the norm of this quantity gives us a scalar value which measures the distance between x and x_k. In fact, when we say the sequence $x_0, x_1, x_2, \ldots$ converges to x_*, we mean that the scalar sequence,$\|x^* - x_0\|, \|x^* - x_1\|, \|x^* - x_2\|, \ldots$ converges to zero (where $\|\cdot\|$ is the norm associated with some metric space). Thus, solving $Ax = b$ could be written as minimizing $\|x - x^*\| = \|x - A^{-1}b\|$ for some norm $\|\cdot\|$.

Of course, since we are trying to compute x^*, we don't want our algorithm to depend on x^*. The *residual* of x_k is defined as $b - Ax_k$. Again $b - Ax^* = 0$ so minimizing $\|b - Ax\|$ will give the true solution. The advantage here is of course that we can easily compute $b - Ax_k$ once we have x_k.

Krylov subspaces

From the previous section, we know that minimizing $\|b - Ax\|$ will give the solution x^*. Unfortunately, this problem is just as hard as solving $Ax = b$. However, if we restrict x to come from a smaller set of values, then the problem become simpler. For instance, if we say that $x = cy$ for some fixed vector y, then this is a scalar minimization problem. Of course, by restricting what values we choose for x it might not be possible to exactly solve $Ax = b$.

We would like to somehow balance how easy the problems we have to solve with how accurate the solutions they are. One way to do this is to start with an easy problem and get a coarse problem, and then gradually increase the difficulty of the problem while refining the solution.

Suppose we have a sequence of subspaces $V_0 \subset V_1 \subset V_2 \subset \cdots V_m$. Then we can construct a sequence of iterates, $x_0 \in V_0, x_1 \in V_1, \ldots$. If at each step we make sure that x_k minimizes $\|b - Ax\|$ over V_k, then the norm of the residuals will decrease (because $V_k \subset V_{k+1}$).

Ideally this sequences of subspaces would:

1. be easy to construct
2. be easy to optimize over (given the previous iterate)
3. eventually contain the true solution

We now formally introduce Krylov subspaces, and show that they satisfy these properties.

The k-th Krylov subspace generated by a square matrix A and a vector v is defined to be,

$$\mathcal{K}_k(A, v) = \operatorname{span}\{v, Av, \ldots, A^{k-1}v\}$$

First, these subspaces are relatively easy to construct because we can get them by repeatedly applying A to v. In fact, we can fairly easily construct an orthonormal basis for these spaces (discussed below).

Therefore, if we have a quantity which can be optimized over each direction of an orthonomal basis independently, then optimizing over these expanding subspaces will be easy because we only need to optimize in a single new direction at each step.

We now show that $\mathcal{K}_k(A, b)$ will eventually contain our solution by the time $k = n$. While this result comes about naturally later from our description of some algorithms, I think it is useful to immediately relate polynomials with Krylov subspace methods as the two are intimately related.

Suppose A has characteristic polynomial,

$$p_A(t) = \det(tI - A) = c_0 + c_1 t + \cdots + c_{n-1}t^{n-1} + t^n$$

It turns out that $c_0 = (-1)^n \det(A)$ so that c_0 is nonzero if A is invertible.

The Cayley-Hamilton Theorem states that a matrix satisfies its own characteristic polynomial. This means,

$$0 = p_A(A) = c_0 I + c_1 A + \cdots c_{n+1}A^{n-1} + A^n$$

Moving the identity term to the left and dividing by $-c_0$ we can write,

$$A^{-1} = -(c_1/c_0)I - (c_2/c_0)A - \cdots - (1/c_0)A^{n-1}$$

This says that A^{-1} can be written as a polynomial in A! In particular,

$$x = A^{-1}b = -(c_1/c_0)b - (c_2/c_0)Ab - \cdots - (1/c_0)A^{n-1}b$$

That means that the solution to $Ax = b$ is a linear combination of $b, Ab, A^2b, \dots, A^{n-1}b$. This observation is the motivation behind Krylov subspace methods. Thus, Krylov subspace methods can be viewed as building low degree polynomial approximations to $A^{-1}b$ using powers of A times b (in fact Krylov subspace methods can be used to approximate $f(A)b$ where f is any function).

Finally, we note that $x = A^{-1}b \in \mathcal{K}_n(A, b)$.

Arnoldi and Lanczos Algorithms

The Arnoldi algorithm for computing an orthonormal basis for Krylov subspaces is at the core of most Krylov subspace methods. Essentially, the Arnoldi algorithm is the Gram-Schmidt procedure applied to the vectors $v, Av, A^2v, A^3v, \dots$ in a clever way.

Recall that given a set of vectors $v_0, v_1, \dots, v_k$ the Gram-Schmidt procedure computes an orthonormal basis $q_0, q_1, \dots, q_k$ so that for all $j \leq k$,

$$\text{span}\{v_0, \dots, v_j\} = \text{span}\{q_0, \dots, q_j\}$$

The trick behind the Arnoldi algorithm is the fact that you do not need to construct the whole set $v, Av, A^2v, \dots$ ahead of time. Instead, you can compute Aq_j in place of $A^{j+1}v$ once you have found an orthonormal basis $q_0, q_1, \dots, q_j$ spanning $v, Av, \dots, A^jv$.

If we assume that $\text{span}\{v, Av, \dots A^jv\} = \text{span}\{q_0, \dots, q_j\}$ then q_j can be written as a linear combination of $v, Av, \dots, A^jv$. Therefore, Aq_j will be a linear combination of $Av, A^2v, \dots, A^{j+1}v$. In particular, this means that $\text{span}\{q_0, \dots, q_j, Aq_j\} = \text{span}\{v, Av, \dots, A^{j+1}v\}$. Therefore, we will get exactly the same set of vectors by applying Gram-Schmidt to $\{v, Av, \dots, A^kv\}$ as if we compute Aq_j once we have computing q_j.

Since we obtain q_{j+1} by orthogonalizing Aq_j against $\{q_0, q_1, \dots, q_j\}$ then q_{j+1} is in the span of these vectors. That means we can can some c_i so that,

$$q_{j+1} = c_0q_0 + c_1q_1 + \cdots + c_jq_j + c_{j+1}Aq_j$$

If we rewrite using new scalars d_i we have,

$$Aq_j = d_0q_0 + d_1q_1 + \cdots + d_{j+1}q_{j+1}$$

This can be written in matrix form as,

$$AQ = QH$$

where H is "upper Hessenburg" (like upper triangular but the first subdiagonal also has nonzero entries). In fact, while we have not showed it, the entries of H

come directly from the Arnoldi algorithm (just like how the entries of R in a QR factorization can be obtained from Gram Schmidt).

When A is Hermetian, then $Q^*AQ = H$ is also Hermetian. Since H is upper Hessenburg and Hermitian, it must be tridiagonal! This means that the q_j satisfy a three term recurrence,

$$Aq_j = \beta_{j-1}q_{j-1} + \alpha_j q_j + \beta_j q_{j+1}$$

where $\alpha_1, \dots, \alpha_n$ are the diagonal entries of T and $\beta_1, \dots, \beta_{n-1}$ are the off diagonal entries of T. The Lanczos algorithm is an efficient way of computing this decomposition.