An Introduction to Conjugate Gradient

Tyler Chen

Motivation

Solving a linear system of equations $Ax = b$ is one of the most important tasks in modern science. A huge number of techniques and algorithms for dealing with more complex equations end up using linear approximations. As a result, applications such as weather forecasting, medical imaging, and training neural nets all require repeatedly solving linear systems to achieve the real world impact that we often take for granted. When A is symmetric and positive definite (if you don't remember what that means, don't worry, I have a refresher below), the Conjugate Gradient algorithm is a very popular choice for methods of solving $Ax = b$.

This popularity of CG is due to a couple factors. First, like most Krylov subspace methods, CG is *matrix free*. This means that you don't ever need to explicitly represent A as a matrix, you only need to be able to compute the product $v \mapsto Av$ for a given input vector v. For very large problems this means a big reduction in storage, and if A has some structure (eg, A comes from a DFT, difference/integral operator, is very sparse, etc.), it allows the algorithm to take advantage of fast matrix vector products. Second, CG only requires $\mathcal{O}(n)$ additional storage to run (as compared to $\mathcal{O}(n^2)$ that many other algorithms require). This can be very useful when the size of the system is very large as it reduces the communication costs of moving data in and out of memory/caches.

Measuring the accuracy of solutions

Perhaps the first question that should be asked about any numerical method is , *does it solve the intended problem?* In the case of solving linear systems, this means asking *does the output approximate the true solution?* If not, then there isn't much point using the method.

Let's quickly introduce the idea of the *error* and the *residual*. These quantities are both useful (in different ways) for measuring how close the approximate solution $\tilde{x}$ is to the true solution $x^* = A^{-1}b$.

The *error* is simply the difference between x and $\tilde{x}$. Taking the norm of this quantity gives us a scalar value which measures the distance between x and $\tilde{x}$. In some sense, this is perhaps the most natural way of measuring how close our approximate solution is to the true solution. 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. Thus, solving $Ax = b$ could be written as minimizing $\|x - x^*\| = \|x - A^{-1}b\|$.

Of course, since we are trying to compute x^*, it doesn't make sense for an algorithm to explicitly depend on x^*. The *residual* of $\tilde{x}$ is defined as $b - A\tilde{x}$. Again, $\|b - Ax^*\| = 0$, and since x^* is the only point where this is true, minimizing $\|b - Ax\|$ gives the true solution. The advantage is that we can easily compute the residual $b - A\tilde{x}$ once we have our numerical solution $\tilde{x}$, while there is not necessarily a good way to compute the error $x^* - x$.

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 $Ax = b$.

We would like to relax this problem in some way to make it "easier". One way to do this is to restrict the values that x can be. For instance, we can enforce that x comes from a smaller set of values which should make the problem of minimizing $\|b - Ax\|$ simpler (since there are less possibilities for x). 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 is quite likely that we will not longer be able to exactly solve $Ax = b$.

It then makes sense to try to balance the difficulty of the problems we have to solve at each step with the accuracy of the solutions they give. One way to do this is to start with an easy problem and get a very approximate solution, and then gradually increase the difficulty of the problem while refining the solution. If we do it in the right way, it's plausible that "increasing the difficulty" of the problem we are solving won't lead to extra work at each step, since we might be able to take advantage of having an approximate solution from a previous step.

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 ensure 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 work done)
3. eventually contain the true solution

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

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

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

First, these subspaces are relatively easy to construct because we can get a basis by repeatedly applying A to v. In fact, we can fairly easily construct an orthonormal basis for these spaces with the Arnoldi/Lanczos algorithms.

Therefore, if we can find a quantity which can be optimized over each direction of an orthonormal 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 from our derivation of CG, I think it is useful to relate polynomials with Krylov subspace methods early on, 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 $-c_0$ (which won't be zero since A is invertible) 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! (I think the coolest facts from linear algebra.) 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 is, the solution x^* to the system $Ax = b$ is a linear combination of $b, Ab, A^2b, \ldots, A^{n-1}b$ (i.e. $x^* \in \mathcal{K}_n(A, b)$). This observation is the motivation behind Krylov subspace methods. I might be useful to think of Krylov subspace methods 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).

The Arnoldi and Lanczos algorithms

The Arnoldi and Lanczos algorithms for computing an orthonormal basis for Krylov subspaces are, in one way or another, at the core of all Krylov subspace methods. Essentially, these algorithms are the Gram-Schmidt procedure applied to the vectors $v, Av, A^2v, A^3v, \ldots$ in a clever way.

The Arnoldi algorithm

Recall that given a set of vectors $v_1, v_2, \dots, v_k$ the Gram-Schmidt procedure computes an orthonormal basis $q_1, q_2, \dots, q_k$ so that for all $j \leq k$,

$$\text{span}\{v_1, \dots, v_j\} = \text{span}\{q_1, \dots, q_j\}$$

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 (in practice, if you tried to do this, it wouldn't really work because eventually $A^j v$ and $A^{j+1}v$ will be nearly linearly dependent since this is essentially the power method). Instead, you can compute Aq_k in place of $A^{k+1}v$ once you have found an orthonormal basis $q_1, q_2, \dots, q_k$ spanning $v, Av, \dots, A^{k-1}v$.

If we assume that $\text{span}\{v, Av, \dots A^{k-1}v\} = \text{span}\{q_1, \dots, q_k\}$ then q_{k+1} can be written as a linear combination of $v, Av, \dots, A^k v$. Therefore, Aq_k will be a linear combination of $Av, A^2v, \dots, A^k v$. In particular, this means that $\text{span}\{q_1, \dots, q_k, Aq_k\} = \text{span}\{v, Av, \dots, A^k v\}$. Therefore, we will get exactly the same set of vectors by applying Gram-Schmidt to $\{v, Av, \dots, A^k v\}$ as if we compute Aq_k once we have computing q_k.

Since we obtain q_{k+1} by orthogonalizing Aq_k against $\{q_1, q_2, \dots, q_k\}$ then q_{k+1} is in the span of these vectors, there exist some c_i so that,

$$q_{k+1} = c_1 q_1 + c_2 q_2 + \cdots + c_k q_k + c_{k+1} A q_k$$

We can rearrange this (using new scalars d_i) to,

$$Aq_k = d_1 q_1 + d_2 q_2 + \cdots + d_{k+1} q_{k+1}$$

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). While I'm not going to derive them here, since 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) their explicit expressions can be easily written down.

Since Q is orthogonal then, $Q^* AQ = H$, so H and A are similar. This means that finding the eigenvalues and vectors of H will give us the eigenvalues and vectors of A. However, since H is upper Hessenburg, then solving the eigenproblem is easier than for a general matrix.

The Lancozs algorithm

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.

I will present a brief derivation for the method motivated by the three term recurrence above. Since we know that the q_j satisfy the three term recurrence, we would like the method to store as few of the q_j as possible (i.e. take advantage of the three term recurrence as opposed to the Arnoldi algorithm).

Suppose that we have q_j, q_{j-1}, and the coefficient β_{j-1}, and want expand the Krylov subspace to find q_{j+1} in a way that takes advantage of the three term recurrence. To do this we can expand the subspace by computing Aq_j and then orthogonalizing Aq_j against q_j and q_{j-1}. By the three term recurrence, Aq_j will be orthogonal to q_i for all $i \leq j-2$ so we do not need to explicitly orthogonalize against those vectors.

We orthogonalize,

$$\tilde{q}_{j+1} = Aq_j - \alpha_j q_j - \langle Aq_j, q_{j-1}\rangle q_{j-1}, \qquad \alpha_j = \langle Aq_j, q_j\rangle$$

and finally normalize,

$$q_{j+1} = \tilde{q}_{j+1}/\beta_j, \qquad \beta_j = \|\tilde{q}_{j+1}\|$$

Note that this is not the most "numerically stable" form of the algorithm, and care must be taken when implementing the Lanczos method in practice. We can improve stability slightly by using $Aq_j - \beta_{j-1}q_{j-1}$ instead of Aq_j when finding a vector in the next Krylov subspace. This allows us to ensure that we have orthogonalized q_{j+1} against q_j and q_{j-1} rather than just q_j. It also ensures that the tridiagonal matrix produces is symmetric in finite precision (since $\langle Aq_j, q_{j-1}\rangle$ may not be equal to β_j in finite precision).

We can implement Lanczos iteration in numpy. Here we assume that we only want to output the diagonals of the tridiagonal matrix T, and don't need any of the vectors (this would be useful if we wanted to compute the eigenvalues of A, but not the eigenvectors).

```
def lanczos(A,q0,max_iter):
    alpha = np.zeros(max_iter)
    beta = np.zeros(max_iter)
    q_ = np.zeros(len(q0))
    q = q0/np.sqrt(q0@q0)

    for k in range(max_iter):
        qq = A@q-(beta[k-1]*q_ if k>0 else 0)
        alpha[k] = qq@q
        qq -= alpha[k]*q
        beta[k] = np.sqrt(qq@qq)
        q_ = np.copy(q)
        q = qq/beta[k]

return alpha,beta
```

A derivation of the Conjugate Gradient algorithm

There are many ways to view/derive the Conjugate Gradient algorithm. I'll derive the algorithm by directly minimizing by minimizing the A-norm of the error over successive Krylov subspaces, $\mathcal{K}_k(A, b)$, which I think is the most natural way to view the algorithm. My hope is that the derivation here provides an intuitive introduction to CG. Of course, what I think is a good way to present the topic won't match up exactly with every reader's own preference, so I highly recommend looking through some other resources as well. To me, this is of those topics where you have to go through the explanations a few times before you start to understand what is going on.

Linear algebra review

Before we get into the details, let's define some notation and review a few key concepts from linear algebra which we will rely on when deriving the CG algorithm.

- Any inner product $\langle \cdot, \cdot \rangle$ induces a norm $\| \cdot \|$ defined by $\|x\|^2 = \langle x, x \rangle$.
- For the rest of this piece we will denote the standard (Euclidian) inner product by $\langle \cdot, \cdot \rangle$ and the (Euclidian) norm by $\| \cdot \|$ or $\| \cdot \|_2$.
- A matrix A is positive definite if $\langle x, Ax \rangle > 0$ for all x.
- A symmetric positive definite matrix A naturally induces the inner product $\langle \cdot, \cdot \rangle_A$ defined by $\langle x, y \rangle_A = \langle x, Ay \rangle = \langle Ax, y \rangle$. The associated norm, called the A-norm will be denoted by $\|langle \cdot \|_A$ and is defined by,

$$\|x\|_A^2 = \langle x, x \rangle_A = \langle x, Ax \rangle = \|A^{1/2}x\|$$

- The point in a subspace V nearest to a point x is the projection of x onto V (where projection is done with the inner product and distance is measured with the induced norm). Given an orthonormal basis for V, this amounts to summing the projection of x onto each of the basis vectors.
- The k-th Krylov subspace generated by A and b is,

$$\mathcal{K}_k(A, b) = \text{span}\{b, Ab, \dots, A^{k-1}b\}$$

Minimizing the error

Now that we have that out of the way, let's begin our derivation. As stated above, we will minimize the A-norm of the error over successive Krylov subspaces generated by A and b. That is to say x_k will be the point so that,

$$\|e_k\|_A := \|x_k - x^*\|_A = \min_{x \in \mathcal{K}_k(A,b)} \|x - x^*\|_A, \qquad x^* = A^{-1}b$$

Since we are minimizing with respect to the A-norm, it will be useful to have an A-orthonormal basis for $\mathcal{K}_k(A, b)$. That is, a basis which is orthonormal in the A-inner

product. For now, let's just say we have such a basis, $\{p_0, p_1, \dots, p_{k-1}\}$, ahead of time. Since $x_k \in \mathcal{K}_k(A, b)$ we can write x_k in terms of this basis,

$$x_k = a_0 p_0 + a_1 p_1 + \cdots + a_{k-1} p_{k-1}$$

Note that we have $x_0 = 0$ and $e_k = x^* - x_k$. Then,

$$e_k = e_0 - a_0 p_0 - a_1 p_1 - \cdots - a_{k-1} p_{k-1}$$

By definition, the coefficients for x_k were chosen to minimize the A-norm of the error, $\|e_k\|_A$, over $\mathcal{K}_k(A, b)$. Therefore, e_k has zero component in each of the directions $\{p_0, p_1, \dots, p_{k-1}\}$. In particular, that means that $a_j p_j$ cancels exactly with e_0 in the direction of p_j.

We now make an important observation. Namely, that the coefficients do not depend on k. Therefore, since the coefficients $a'_0, a'_1, \dots, a'_{k-2}$ of x_{k-1} were chosen in exactly the same way as the coefficients for x_k, then $a_0 = a'_0, a_1 = a'_1, \dots, a_{k-2} = a'_{k-2}$.

We can then write,

$$x_k = x_{k-1} + a_{k-1} p_{k-1}$$

and

$$e_k = e_{k-1} - a_{k-1} p_{k-1}$$

Now that we have explicitly written x_k in terms of an update to x_{k-1} this is starting to look like an iterative method!

Let's compute an explicit representation of the coefficient a_{k-1}. As previously noted, we have chosen x_k to minimize $\|e_k\|_A$ over $\mathcal{K}_k(A, b)$. Therefore, the component of e_k in each of the directions $p_0, p_1, \dots, p_{k-1}$ must be zero. That is, $\langle e_k, p_j \rangle = 0$ for all $i = 0, 1, \dots, k-1$.

$$0 = \langle e_k, p_{k-1} \rangle_A = \langle e_{k-1}, p_{k-1} \rangle - a_{k-1} \langle p_{k-1}, p_{k-1} \rangle_A$$

Thus

$$a_{k-1} = \frac{\langle e_{k-1}, p_{k-1} \rangle_A}{\langle p_{k-1}, p_{k-1} \rangle_A}$$

This expression might look like a bit of a roadbock, since if we knew the initial error $e_0 = x^* - 0$ then we would know the solution to the original system! However, we have been working with the A-inner product so we can write,

$$A e_{k-1} = A(x^* - x_{k-1}) = b - A x_{k-1} = r_{k-1}$$

Therefore, we can compute a_{k-1} as,

$$a_{k-1} = \frac{\langle r_{k-1}, p_{k-1} \rangle}{\langle p_{k-1}, A p_{k-1} \rangle}$$

Finding the Search Directions

At this point we are almost done. The last thing to do is understand how to update p_k. The first thing we might try would be to do something like Gram-Schmidt on $\{b, Ab, A^2b, ...\}$ to get the p_k, i.e. Arnoldi iteration in the inner product induced by A. This will work fine if you take some care with the exact implementation. However, since A is symmetric we might hope to be able to use some short recurrence, which turns out to be the case.

Since $r_k = b - Ax_k$ and $x_k \in \mathcal{K}_k(A, b)$, then $r_k \in \mathcal{K}_{k+1}(A, b)$. Thus, we can obtain p_k by A-orthogonalizing r_k against $\{p_0, p_1, \dots, p_{k-1}\}$.

Recall that e_k is A-orthogonal to $\mathcal{K}_k(A, b)$. That is, for $j \leq k - 1$,

$$\langle e_k, A^j b \rangle_A = 0$$

Therefore, noting that $Ae_k = r_k$, for $j \leq k - 2$,

$$\langle r_k, A^j b \rangle_A = 0$$

That is, r_k is A-orthogonal to $\mathcal{K}_{k-1}(A, b)$. In particular, this means that, for $j \leq k - 2$,

$$\langle r_k, p_j \rangle_A = 0$$

That means that to obtain p_k we really only need to A-orthogonalize r_k against p_{k-1}! That is,

$$p_k = r_k + b_k p_{k-1}, \qquad b_k = -\frac{\langle r_k, p_{k-1} \rangle_A}{\langle p_{k-1}, p_{k-1} \rangle_A}$$

The immediate consequence is that we do not need to save the entire basis $\{p_0, p_1, \dots, p_{k-1}\}$, but instead can just keep x_k,r_k, and p_{k-1}. **expand on this**!!

Putting it all together

We are now essentially done! In practice, people generally use the following equivalent formulas for a_{k-1} and b_k,

$$a_{k-1} = \frac{\langle r_{k-1}, r_{k-1} \rangle}{\langle p_{k-1}, Ap_{k-1} \rangle}, \qquad b_k = \frac{\langle r_k, r_k \rangle}{\langle r_{k-1}, r_{k-1} \rangle}$$

The first people to discover this algorithm Magnus Hestenes and Eduard Stiefel who independently developed it around 1952. As such, the standard implementation is attributed to them.

Algorithm. (Hestenes and Stiefel Conjugate Gradient)

procedure HSCG(A, b, x_0)
  set $r_0 = b - Ax_0, \nu_0 = \langle r_0, r_0 \rangle, p_0 = r_0, s_0 = Ar_0,$
    $a_0 = \nu_0 / \langle p_0, s_0 \rangle$
  for $k = 1, 2, \ldots$ **:**
    set $x_k = x_{k-1} + a_{k-1} p_{k-1}$
      $r_k = r_{k-1} - a_{k-1} p_{k-1}$
    set $\nu_k = \langle r_k, r_k \rangle$, **and** $b_k = \nu_k / \nu_{k-1}$
    set $p_k = r_k + b_k p_{k-1}$
    set $s_k = Ap_k$
    set $\mu_k = \langle p_k, s_k \rangle$, **and** $a_k = \nu_k / \mu_k$
  end for
end procedure

This can be easily implemented in numpy. Note that we use f for the right hand side vector to avoid conflict with the coefficient b.

```
def cg(A,f,max_iter):
    x = np.zeros(len(f)); r = np.copy(f); p = np.copy(r); s=A@p
    nu = r @ r; a = nu/(p@s); b = 0
    for k in range(1,max_iter):
        x += a*p
        r -= a*s

        nu_ = nu
        nu = r@r
        b = nu/nu_

        p = r + b*p
        s = A@p

        a = nu/(p@s)

    return x
```

Conjugate Gradient is Lanczos in Disguise

It's perhaps not so surprising that the Conjugate Gradient and Lanczos algorithms are closely related. After all, they are both Krylov subspace methods for symmetric matrices.

More precisely, the Lanczos algorithm will produce an orthonormal basis for $\mathcal{K}_k(A, b)$, $k = 0, 1, \ldots$ if we initialize with initial vector $r_0 = b$. We also know that the Conjugate Gradient residuals form an orthogonal basis for for these spaces, which means that the Lanczos vectors must be scaled versions of the Conjugate Gradient residuals.

This relationship provides a way of transferring research about the Lanczos algorithm to be to CG, and visa versa. In fact, the analysis of finite precision CG done by Greenbaum requires viewing CG in terms of the Lanczos recurrence.

In case you're just looking for the punchline, the Lanczos coefficients and vectors can be obtained from the Conjugate Gradient algorithm by the following relationship,

$$q_{j+1} \equiv (-1)^j \frac{r_j}{\|r_j\|}, \qquad \beta_j \equiv \frac{\|r_j\|}{\|r_{j-1}\|} \frac{1}{a_{j-1}}, \qquad \alpha_j \equiv \left(\frac{1}{a_{j-1}} + \frac{b_j}{a_{j-2}} \right)$$

Note that the indices are offset, because the Lanczos algorithm is started with initial vector q_1.

Derivation

The derivation of the above result is so much difficult as it is tedious. Before you read my derivation, I would highly recommend trying to derive it on your own, since it's a good chance to improve your familiarity with both algorithms. While I hope that my derivation is not too hard to follow, there are definitely other ways to arrive at the same result, and often to really start to understand something you have to work it out on your own.

Recall that the three term Lanczos recurrence is,

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

In each iteration of CG we update,

$$r_j = r_{j-1} - a_{j-1} A p_{j-1}, \qquad p_j = r_j + b_j p_{j-1}$$

Thus, substituting the expression for p_{j-1} we find,

$$\begin{aligned} r_j &= r_{j-1} - a_{j-1} A (r_{j-1} + b_{j-1} p_{j-2}) \\ &= r_{j-1} - a_{j-1} A r_{j-1} - a_{j-1} b_{j-1} A p_{j-2} \end{aligned}$$

Now, rearranging our equation for r_{j-1} we have that $Ap_{j-2} = (r_{j-2} - r_{j-1})/a_{j-2}$. Therefore,

$$r_j = r_{j-1} - a_{j-1} A r_{j-1} - \frac{a_{j-1} b_{j-1}}{a_{j-2}} (r_{j-2} - r_{j-1})$$

At this point we've found a three term recurrence for r_j, which is a hopeful sign that we are on the right track. We know that the Lanczos vectors are orthonormal and that the recurrence is symmetric, so we'll keep massaging our CG relation to try to get it into that form.

First, let's rearrange terms and regroup them so that the indices and matrix multiply match up with the Lanczos recurrence. This gives,

$$Ar_{j-1} = \left(\frac{1}{a_{j-1}} + \frac{b_{j-1}}{a_{j-2}}\right) r_{j-1} - \frac{b_{j-1}}{a_{j-2}} r_{j-2} - \frac{1}{a_{j-1}} r_j$$

Now, we normalize our residuals as $z_j = r_{j-1}/\|r_{j-1}\|$ so that $r_{j-1} = \|r_{j-1}\| z_j$. Plugging these in gives,

$$\|r_{j-1}\| Az_j = \|r_{j-1}\| \left(\frac{1}{a_{j-1}} + \frac{b_{j-1}}{a_{j-2}}\right) z_j - \|r_{j-2}\| \frac{b_{j-1}}{a_{j-2}} z_j - \|r_j\| \frac{1}{a_{j-1}} z_{j+1}$$

Thus, dividing through by $\|r_{j-1}\|$ we have,

$$Az_j = \left(\frac{1}{a_{j-1}} + \frac{b_{j-1}}{a_{j-2}}\right) z_j - \frac{\|r_{j-2}\|}{\|r_{j-1}\|} \frac{b_{j-1}}{a_{j-2}} z_{j-1} - \frac{\|r_j\|}{\|r_{j-1}\|} \frac{1}{a_{j-1}} z_{j+1}$$

This looks close, but the coefficients for the last two terms have the same formula in the Lanczos recurrence. However, recall that $b_j = \langle r_j, r_j\rangle / \langle r_{j-1}, r_{j-1}\rangle = \|r_j\|^2 / \|r_{j-1}\|^2$. Thus,

$$Az_j = \left(\frac{1}{a_{j-1}} - \frac{b_{j-1}}{a_{j-2}}\right) z_j - \frac{\|r_{j-1}\|}{\|r_{j-2}\|} \frac{1}{a_{j-2}} z_{j-1} - \frac{\|r_j\|}{\|r_{j-1}\|} \frac{1}{a_{j-1}} z_{j+1}$$

We're almost there! While we have the correct for the recurrence, the coefficients from the Lanczos method are always positive. This means that our z_j have the wrong signs. Fixing this gives the relationship,

$$q_{j+1} \equiv (-1)^j \frac{r_j}{\|r_j\|}, \qquad \beta_j \equiv \frac{\|r_j\|}{\|r_{j-1}\|} \frac{1}{a_{j-1}}, \qquad \alpha_j \equiv \left(\frac{1}{a_{j-1}} + \frac{b_j}{a_{j-2}}\right)$$

Error Bounds for the Conjugate Gradient Algorithm

This page is a work in progress.

In our derivation of the Conjugate Gradient method, we minimized the A-norm of the error over sucessive Krylov subspaces. Ideally we would like to know how quickly this method converge. That is, how many iterations are needed to reach a specified level of accuracy.

Linear algebra review

- The 2-norm of a symmetric positive definite matrix is the largest eigenvalue of the matrix
- The 2-norm is submultiplicative. That is, $\|A\|\|B\| \leq \|AB\|$
- A matrix U is called unitary if $U^*U = UU^* = I$.

Polynomial error bounds

Previously we have show that,

$$e_k \in e_0 + \text{span}\{p_0, p_1, \dots, p_{k-1}\} = e_0 + \mathcal{K}_k(A, b)$$

Observing that $r_0 = Ae_0$ we find that,

$$e_k \in e_0 + \text{span}\{Ae_0, A^2e_0, \dots, A^ke_0\}$$

Thus, we can write,

$$\|e_k\|_A = \min_{p \in \mathcal{P}_k} \|p(A)e_0\|_A, \qquad \mathcal{P}_k = \{p : p(0) = 1, \deg p \leq k\}$$

Since $A^{1/2}p(A) = p(A)A^{1/2}$ we can write,

$$\|p(A)e_0\|_A = \|A^{1/2}p(A)e_0\| = \|p(A)A^{1/2}e_0\|$$

Now, using the submultiplicative property of the 2-norm,

$$\|p(A)A^{1/2}e_0\| \leq \|p(A)\|\|A^{1/2}e_0\| = \|p(A)\|\|e_0\|_A$$

Since A is positive definite, it is diagonalizable as $U\Lambda U^*$ where U is unitary and Λ is the diagonal matrix of eigenvalues of A. Thus,

$$A^k = (U\Lambda U^*)^k = U\Lambda^k U^*$$

We can then write $p(A) = Up(\Lambda)U^*$ where $p(\Lambda)$ has diagonal entries $p(\lambda_i)$. Therefore, using the *unitary invariance* property of the 2-norm,

$$\|p(A)\| = \|Up(\Lambda)U^*\| = \|p(\Lambda)\|$$

Now, since the 2-norm of a symmetric matrix is the magnitude of the largest eigenvalue,

$$\|p(\Lambda)\| = \max_i |p(\lambda_i)|$$

Finally, putting everything together we have,

$$\frac{\|e_k\|_A}{\|e_0\|_A} \leq \min_{p \in \mathcal{P}_k} \max_i |p(\lambda_i)|$$

Since the inequality we obtained from the submultiplicativity of the 2-norm is tight, this bound is also tight in the sense that for a fixed k there exists an initial error e_0 so that equality holds.

Computing the optimal p is not trivial, but an algorithm called the Remez algorithm can be used to compute it.

Let $L \subset \mathbb{R}$ be some closed set. The *minimax polynomial of degree* k on L is the polynomial satisfying,

$$\min_{p \in \mathcal{P}_k} \max_{x \in L} |p(x)|, \qquad \mathcal{P}_k = \{p : p(0) = 1, \deg p \leq k\}$$

Chebyshev bounds

The minimax polynomial on the eigenvalues of A is a bit tricky to work with. Although we can find it using the Remez algorithm, this is somewhat tedious, and requires knowledge of the whole spectrum of A. We would like to come up with a bound which depends on less information about A. One way to obtain such a bound is to expand the set on which we are looking for the minimax polynomial.

To this end, let $\mathcal{I} = [\lambda_{\text{min}}, \lambda_{\text{max}}]$. Then, since $\lambda_i \in \mathcal{I}$,

$$\min_{p \in \mathcal{P}_k} \max_i |p(\lambda_i)| \leq \min_{p \in \mathcal{P}_k} \max_{x \in \mathcal{I}} |p(x)|$$

The right hand side requires that we know the largest and smallest eigenvalues of A, but doesn't require any of the ones between. This means it can be useful in practice, since we can easily compute the top and bottom eignevalues with the power method.

The polynomials satisfying the right hand side are called the *Chebyshev Polynomials* and can be easily written down with a simple recurrence relation. If $\mathcal{I} = [-1, 1]$ then the relation is,

$$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x), \qquad T_0 = 1, \qquad T_1 = x$$

For $\mathcal{I} \neq [-1, 1]$, the above polynomials are simply stretched and shifted to the interval in question.

Let $\kappa = \lambda_{\text{max}}/\lambda_{\text{min}}$ (this is called the condition number). Then, from properties of these polynomials,

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

The Conjugate Gradient Algorithm in Finite Precision

This page is a work in progress.

A key component of our derivations of the Lanczos and Conjugate Gradient methods was the orthogonality of certain basis vectors. In finite precision, our induction based arguments no longer hold, and so it's reasonable to expect the algorithms will fail. That

said, since you're reading about these methods, they must somehow still be usable in practice. This turns out to be the case, and both methods are widely used for eignevalue problems and solving linear systems.

The first major progress in the analysis of the Lanczos algorithm was done by Chris Paige, who characterized the behavior of the method in finite precision. A similarly important analysis of Conjugate Gradient was done by Anne Greenbaum in her 1989 paper, "Behavior of slightly perturbed Lanczos and conjugate-gradient recurrences". A big takeaway from Greenbaum's analysis is that the error bound from the Chevyshev polynomials still holds in finite precision (to a close approximation). My goal here is to present the highlights of that paper.

The results

Remez Algorithm

Some conditions for the analysis

CG is doing the Lanczos algorithm in disguise. In particular, normalizing the residuals from CG gives the vectors q_j produced by the Lanczos algorithm, and combing the CG constants in the right way gives the coefficients for the three term Lanczos recurrence.

The analysis by Greenbaum requires that the finite precision Conjugate Gradient algorithm (viewed as the Lanczos algorithm) satisfy a few properties. Namely,

- the three term Lanczos recurrence is well satisfied
- the Lanczos vectors have norm close to one
- successive Lanczos vectors are nearly orthogonal

As it turns out, nobody has actually ever proved that any of the Conjugate Variant methods used in practice actually satisfy these conditions (although some do numerically satisfy the conditions).

Communication Avoiding Conjugate Gradient Algorithms

One of the main drawbacks to Conjugate gradient in a high performance setting is .....

Communication bottlenecks in CG

Recall the standard Hestenes and Stifel CG implementation. In the below description, every block of code after a "**set**" must wait for the output from the previous block. Much of the algorithm is scalar and vector updates which are relatively cheap (in terms of floating

point operations and communication). The most expensive computations each iteration are the matrix vector product, and the two inner products.

Algorithm. (Hestenes and Stiefel Conjugate Gradient)

procedure HSCG(A, b, x_0)
  set $r_0 = b - Ax_0, \nu_0 = \langle r_0, r_0 \rangle, p_0 = r_0, s_0 = Ar_0,$
    $a_0 = \nu_0 / \langle p_0, s_0 \rangle$
  for $k = 1, 2, \ldots$ **:**
    set $x_k = x_{k-1} + a_{k-1} p_{k-1}$
      $r_k = r_{k-1} - a_{k-1} s_{k-1}$
    set $\nu_k = \langle r_k, r_k \rangle,$ **and** $b_k = \nu_k / \nu_{k-1}$
    set $p_k = r_k + b_k p_{k-1}$
    set $s_k = A p_k$
    set $\mu_k = \langle p_k, s_k \rangle,$ **and** $a_k = \nu_k / \mu_k$
  end for
end procedure

A matrix vector product requires $\mathcal{O}(\text{nnz})$ (number of nonzero) floating point operations, while an inner product requires $\mathcal{O}(n)$ operations. For many applications of CG, the number of nonzero entries is something like kn, where k relatively small. In these cases, the cost of floating point arithmetic for a matrix vector product and an inner product is roughly the same. On the other hand, the communication costs for the inner products are much lower.

matvec is usually sparse

inner products require "all reduce" i.e. collect information back from all the different processors/nodes would like to be able to overlap these as much as possible

Overlapping inner products

We would like to be able to *overlap* as many of the heavy computations as possible. However, in the current form, we need to wait for each of the previous computations before we are able to do a matrix vector product or an inner product.

Using our recurrences we can write,

$$s_k = Ap_k = A(r_k + b_k p_{k-1}) = Ar_k + b_k s_{k-1}$$

If we define the axillary vector $w_k = Ar_k$, in exact arithmetic using this formula for s_k will be equivalent to the original formula for s_k. However, we can now compute w_k as soon as we have r_k. Therefore, the computation of $\nu_k = \langle r_k, r_k \rangle$ can be overlapped with the computation of $w_k = Ar_k$.

These coefficient formulas seem to work better that CGCG..

Algorithm. (Chronopoulos and Gear Conjugate Gradient)

procedure CGCG(A, b, x_0)
  set $r_0 = b - Ax_0, \nu_0 = \langle r_0, r_0 \rangle, p_0 = r_0, s_0 = Ar_0,$
    $a_0 = \nu_0 / \langle p_0, s_0 \rangle$
  for $k = 1, 2, \ldots$ **:**
    set $x_k = x_{k-1} + a_{k-1} p_{k-1}$
      $r_k = r_{k-1} - a_{k-1} s_{k-1}$
    set $w_k = Ar_k$
      $\nu_k = \langle r_k, r_k \rangle,$ **and** $b_k = \nu_k / \nu_{k-1}$
    set $\eta_k = \langle r_k, w_k \rangle,$ **and** $a_k = \nu_k / (\eta_k - (b_k / a_{k-1}) \nu_k)$
      $p_k = r_k + b_k p_{k-1}$
      $s_k = w_k + b_k s_{k-1}$
  end for
end procedure

Algorithm. (Ghysels and Vanroose Conjugate Gradient)

procedure CGCG(A, b, x_0)
  set $r_0 = b - Ax_0, \nu_0 = \langle r_0, r_0 \rangle, p_0 = r_0, s_0 = Ar_0,$
    $w_0 = s_0, u_0 = Aw_0, a_0 = \nu_0 / \langle p_0, s_0 \rangle$
  for $k = 1, 2, \ldots$ **:**
    set $x_k = x_{k-1} + a_{k-1} p_{k-1}$
      $r_k = r_{k-1} - a_{k-1} s_{k-1}$
      $w_k = w_{k-1} - a_{k-1} u_{k-1}$
    set $\nu_k = \langle r_k, r_k \rangle,$ **and** $b_k = \nu_k / \nu_{k-1}$
      $\eta_k = \langle r_k, w_k \rangle,$ **and** $a_k = \nu_k / (\eta_k - (b_k / a_{k-1}) \nu_k)$
      $t_k = Aw_k$
    set $p_k = r_k + b_k p_{k-1}$
      $s_k = w_k + b_k s_{k-1}$
      $u_k = b_k u_{k-1}$
  end for
end procedure

fda

fdas

Current research on CG and related Krylov subspace methods

Avoiding communication

- CGCG, GVCG
- s-step methods

Computing matrix functions

- $f(A)b$ for functions other that $f(x) = x^{-1}$