An Introduction to Modern Analysis of the Conjugate Gradient Algorithm in Exact and Finite Precision

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 the conjugate gradient algorithm (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.

While the conjugate gradient algorithm has many nice theoretical properties, its behavior in finite precision can be *extremely* different than the behavior predicted by assuming exact arithmetic. Understanding what leads to these vastly different behaviors has been an active area of research since the 60s and 70s. My intention is to provide an overview of the conjugate gradient algorithm in exact precision, then introduce some of what is know about it in finite precision, and finally, present some modern research interests into the algorithm.

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,

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

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 $b, Ab, A^2b, \dots, A^{n-1}b$ (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 $v, Av, A^2v, A^3v, \dots$ 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 $j \le k$,

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

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^jv 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 $v, Av, \dots, A^kv$. Therefore, Aq_k will be a linear combination of $Av, A^2v, \dots, A^kv$. In particular, this means that $\text{span}\{q_1, \dots, q_k, Aq_k\} = \text{span}\{v, Av, \dots, A^kv\}$. 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_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,

$$q_{k+1} = c_1q_1 + c_2q_2 + \cdots + c_kq_k + c_{k+1}Aq_k$$

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

$$Aq_k = d_1q_1 + d_2q_2 + \cdots + d_{k+1}q_{k+1}$$

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).

Algorithm. (Lanczos)

```
procedure lanczos($A, v$)
  set $q_1 = v/\|v\|, \beta_0 = 0$
  for $k = 1, 2, \dots$:
    set $\tilde{q}_{k+1} = Aq_k - \beta_{k-1}q_{k-1}$
    set $\alpha_k = \langle \tilde{q}_{k+1}, q_k \rangle$
    set $\tilde{q}_{k+1} = \tilde{q}_{k+1} - \alpha_k q_k$
    set $\beta_k = \|\tilde{q}_{k+1}\|$
    set $q_{k+1} = \tilde{q}_{k+1}/\beta_k$
  end for
end procedure
```

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,

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

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 $\{b, Ab, A^2b, \ldots\}$ 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 $\{p_0, p_1, \ldots, p_{k-1}\}$.

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 $\{p_0, p_1, \ldots, p_{k-1}\}$, but instead can just keep x_k,r_k, and p_{k-1}. This is perhaps somewhat unsurprising give then we have seen that when A is symmetric we have a three term Lanczos recurrence.

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
```

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,

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

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,

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

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 $U^*U = UU^* = I$.

Error bounds from minimax polynomials

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, since $U^*U = I$,

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

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,

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

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 vectors. In finite precision, we cannot have exact orthogonality, so our induction based arguments no longer hold. Even so, both methods are widely used in practice, which suggests that there are subtle forces at play allowing the algorithms to work.

In exact arithmetic, CG will obtain the exact solution in at most n steps, and the error at step k can be bounded by the size of the degree k minimax polynomial on the eigenvalues of A. In finite precision, the loss of orthogonality leads to two easily observable effects: delayed convergence, and reduced final accuracy. The following figure shows both of these phenomena for various precisions.

Delay of convergence

In our derivation of the conjugate gradient algorithm, we used the A-orthogonality of successive search directions to show that we only needed to optimize over the current search direction. In finite precision, once orthogonality is lost, we not longer know that minimizing along a given search direction will also result in a solution which is optimal in the previous search directions. As such, a conjugate gradient algorithm in finite precision will end up doing "redundant" optimization.

Loss of attainable accuracy

Even if we knew the solution x^* to the system $Ax = b$, it's unlikely that we could represent it in finite precision.

$\epsilon\|A\|$ - different variants converge to worse levels

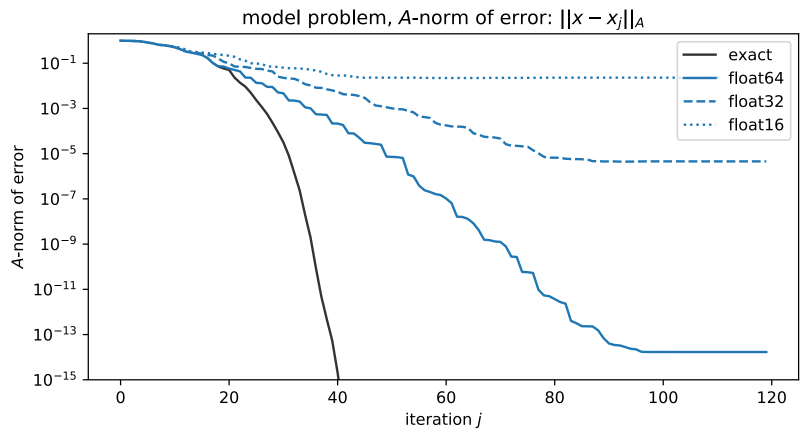

Figure 1: Convergence of conjugate gradient in various precisions. Note that the computation would finish in at most 48 steps in exact arithmetic.

Updated vs. true residual

In exact precision, the updated residual r_k was equal to the true residual $b - Ax_k$. In finite precision, this is not longer the case. Interestingly, even in finite precision, the updated residual (of many variants) keeps decreasing far below machine precision, until it eventually underflows. **Do we know why or have citations about this?**

The analysis of Greenbaum

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 in his 1971 PhD thesis. An analysis of similar importance 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).

The analysis from this paper is quite involved, and does not provides sufficient conditions for good convergence. However, necessary conditions of similar strength are not known, so it is not clear whether the conditions from the analysis can be relaxed significantly or not. In essence, Greenbaum showed that, in finite precision, a "good" conjugate gradient algorithm applied to the matrix A will behave like exact conjugate gradient applied to a larger matrix $\widehat{T}$ with eigenvalues near those of A.

Thus, while the error bounds derived based on the minimax polynomial on the spectrum of A no longer hold in exact arithmetic, bounds of a similar form can be obtained by finding the minimax polynomial on the union of small intervals about the eigenvalues of

A. In particular, the bound from Chebyshev polynomials will not be significantly affect, as the condition number of the larger matrix will be nearly identical to that of A.

As before, the Remez algorithm can be used to compute the minimax polynomial of a given degree on the union of intervals.

Finding the larger matrix

A constructive algorithm for finding an extended matrix $\hat{T}$ where exact CG behaves like finite precision CG on A was presented in [greenbaum_89]. The algorithm is given the Lanczos vectors and coefficients from the finite precision computation, and extends the tridiagonal matrix T by slightly perturbing a new, exact, Lanczos recurrence so that it terminates. The resulting larger tridiagonal matrix $\hat{T}$, which contains T in the top left block, will have eigenvalues near those of A (assuming T came from a "good" implementation).

An explanation of the algorithm is given in the appendix of [greenbaum_liu_chen_19], and an jupyter notebook is available here, and two Python implementations of the algorithm are available in the same Github repository.

Some conditions for the analysis

I have brushed what a "good" conjugate gradient implementation means. In some sense this is still not known, since there has been no analysis providing both necessary and sufficient conditions for convergence to behave in a certain way. That said, the conditions given in [greenbaum_89] are sufficient, and should be discussed.

We have already seen that 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 α_j, β_j 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

We know that in exact arithmetic, each of these properties is satisfied exactly. In fact, in exact arithmetic, all Lanczos vectors are orthogonal. However, in finite precision it is well known that the Lancozs vectors eventually lose orthogonality. Paige described the way in which this orthogonality is lost, and proved that (what is now) the standard Lanczos implementation satisfies the three properties above. Greenbaum extended Paige's analysis, which is why her results use the same conditions.

As it turns out, nobody has actually ever proved that any of the Conjugate Variant methods used in practice actually satisfy these conditions. In our paper, "On the Convergence of Conjugate Gradient Variants in Finite Precision Arithmetic" we analyze some variants in terms of these quantities, and try to provide rounding error analysis which will explain why these properties are or are not satisfied for each variant.

Any proof that a given method satisfies these properties would constitute a significant theoretical advancement in the understanding of the conjugate gradient algorithm in finite precision. Similarly, finding weaker conditions which provide some sort of convergence guarantees for a finite precision CG implementation would also be of significant importance.

Communication Hiding Conjugate Gradient Algorithms

So far, all we have considered are error bounds in terms of the number of iterations. However, in practice, what we really care about is how long a computation takes. A natural way to try and speed up an algorithm is through parallelization, and many variants of the conjugate gradient algorithm have been introduced to try and take advantage of high performance computers. However, while these variants are all equivalent in exact arithmetic, they perform operations in different orders. Based on our discussion about conjugate gradient in finite precision, it should be too big of a surprise that the variants all behave differently.

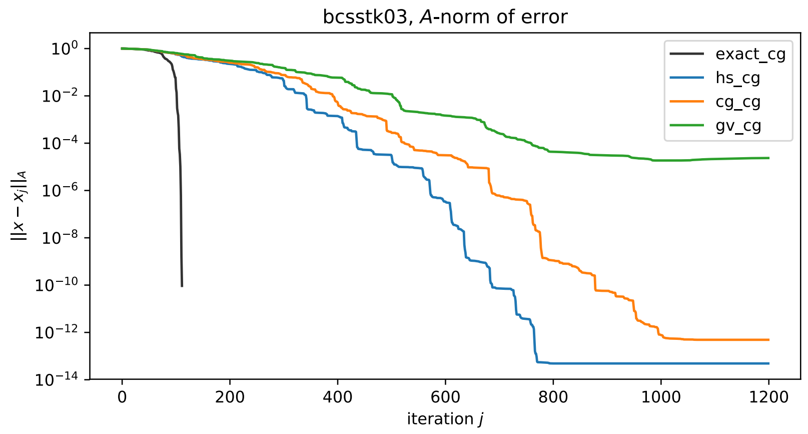

Figure 2: Convergence of different variants in finite precision. Note that the computation would finish in at most 112 steps in exact arithmetic.

The DOE supercomputer “Summit” is able to compute 122 petaflops per second. That’s something like 10^{16} floating point operations every second! Unfortunately, on high performance machines, conjugate gradient often perform orders of magnitude fewer floating point operations per second than the computer is theoretically capable of.

The reason for this is *communication*.

Traditionally, analysis of algorithms has been done in terms of counting the number of operations computed, and the amount of storage used. However, an important factor in the real world performance is the time it takes to move data around. Even on a single core computer, the cost of moving a big matrix off the hard drive into memory can be much more significant than the floating point operations done. There is naturally a lot of interest in reducing the communication costs of various algorithms, and conjugate gradient certainly has a lot of room for improvement.

Parallel computers work by having different processors work on different parts of a computation at the same time. Naturally, many things can be sped up this way, but like in our example, doubling the computing power doesn't necessarily mean doubling the speed of the computation. In many numerical algorithms, "global communication" is one of the main causes of latency. Loosely speaking, global communication means that all processors working on a larger task must finish with their subtask and report on the result before the computation can proceed. This means that even if we can distribute a computation to many processors, the time it takes to move the data required for those computations will eventually limit how effective adding more processors is. So, a 1000x increase in processor power won't necessarily cut the computation time to 1/1000.

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 $s_k = Ap_k$
    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 of dense vectors 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 can be much higher.

There are multiply ways to address the communication bottleneck in CG. The two main approaches are "hiding" communication, and "avoiding" communication. Communication hiding algorithm such as as pipelined CG introduce auxiliary vectors so that the inner products can be computed at the same time, allowing the communication to be overlapped with other computations. On the other hand, communication avoiding algorithms such as s-step CG compute iterations in blocks of size s, reducing the synchronization costs by a factor around s.

In this piece I will be talking about some common communicating hiding methods. If you're interested in communication avoiding methods, or want more information about communication hiding methods, Erin Carson has very useful slides.

Overlapping inner products

We would like to be able to reduce the number of points in the algorithm a global communication is required. 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. This means there are two global communications per iteration.

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 $w_k = Ar_k$. However, this still requires two global communication points each iteration.

We now note that,

$$\begin{aligned}\mu_k = \langle p_k, s_k \rangle &= \langle r_k + b_k p_{k-1}, w_k + b_k s_{k-1} \rangle \\ &= \langle r_k, w_k \rangle + b_k \langle p_{k-1}, w_k \rangle + b_k \langle r_k, s_{k-1} \rangle + b_k^2 \langle p_{k-1}, s_{k-1} \rangle\end{aligned}$$

Now, since $w_k = Ar_k$ and $s_{k-1} = Ap_{k-1}$, then $\langle p_{k-1}, w_k \rangle = \langle r_k, s_{k-1} \rangle$. Thus,

$$\mu_k = \langle r_k, w_k \rangle + 2b_k \langle r_k, s_k \rangle + b_k^2 \mu_{k-1}$$

Notice now that $\langle r_k, w_k \rangle$, and $\langle r_k, s_k \rangle$ can both be overlapped with $\nu_k = \langle r_k, r_k \rangle$. Thus, using this coefficient formula there is only a single global synchronization per iteration. However, this came at the cost of having to compute an additional inner product.

Now, using our recurrence for r_k,

$$\langle r_k, s_{k-1} \rangle = \langle p_k - b_k p_{k-1}, s_{k-1} \rangle = -b_k \langle p_{k-1}, s_{k-1} \rangle = -b_k \mu_{k-1}$$

Thus, defining $\eta_k = \langle w_k, r_k \rangle$ and canceling terms,

$$\mu_k = \langle w_k, r_k \rangle - b_k^2 \mu_{k-1} = \eta_k - (b_k/a_k)\nu_k$$

This variant is known as Chronopoulos and Gear conjugate gradient.

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 $w_k = Ar_k$
    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 $a_k = \nu_k / (\eta_k - (b_k/a_{k-1})\nu_k)$
        $p_k = r_k + b_k p_{k-1}$
        $s_k = w_k + b_k s_{k-1}$
  end for
end procedure
```

However, while we have only a single global communication per iteration, the matrix vector product must still be computed before we are able to compute the inner products. To allow the matrix vector product to be overlapped with the inner products, we again introduce auxiliary vectors. Observe that,

$$w_k = Ar_k = A(r_{k-1} - a_{k-1}s_{k-1}) = Ar_{k-1} - a_{k-1}As_{k-1} = w_{l-1} - a_{k-1}As_{k-1}$$

Now, define $u_k = As_k$ and note that,

$$u_k = As_k = A(w_k + b_k s_{k-1}) = Aw_k + b_k As_{k-1} = Aw_k + b_k u_{k-1}$$

That's it. For convenience we define $t_k = Aw_k$. Then, the matrix vector product Aw_k can occur as soon as we have computed w_k. This variant is known as either Ghysels and Vanroose conjugate gradient or pipelined conjugate gradient.

Algorithm. (Ghysels and Vanroose (pipelined) conjugate gradient)

```
procedure GVCG($A, b, x_0$)
  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 $p_k = r_k + b_k p_{k-1}$
        $s_k = w_k + b_k s_{k-1}$
        $u_k = t_k + b_k u_{k-1}$
  end for
end procedure
```

Recently, [cite ppl] have developed a “deep pipelined” conjugate gradient, which essentially introduces even more auxiliary vectors to allow for more overlapping.

As mentioned above,

Current research on Conjugate Gradient and related Krylov subspace methods

Avoiding communication

- CGCG, GVCG
- s-step methods

Computing matrix functions

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