Finding $A^{-1}\mathbf{b}$ with The Conjugate Gradient Method

Lee Fisher

October 16, 2021

Abstract

This document is a short explanation of how the conjugate gradient method can be used to quickly evaluate the inverse of symmetric positive definite matrix. We start by reframing the Poisson equation on an interval in terms of evaluating the inverse of an SPD matrix. We start with attempting to find a solution with a simple gradient descent and then we use more sophisticated linear algebra to make gradual improvements to the algorithm. We will include Matlab code snippets throughout the derivation to emphasize how the algorithm can be implemented, we also discuss limitations, and the convergence rate of this method. We conlcude with a brief discussion of preconditioning. You will find a full working example in this github folder.

1 Introduction

This is a method for solving a linear system of equations originally discovered by Hestenes and Stiefel in [3], this presentation of the method draws from [4], [2], and lecture notes from a graduate course on Numerical Analysis at UC Irvine [1].

There are many variations of the Conjugate Gradient method and also ways to apply it to matrices that are not symmetic prositive definite. This is a commonly used method to quickly solve a sparse linear system it is conceptually similar to the well known method of gradient descent; but by focusing the alternate geometry that the inner product associated to an SPD-matrix bestows on $\mathbb{R}^n$, we can in some cases drastically improve on the efficiency of the method of gradient descent.

2 Poisson's Equation on an Interval

Consider the following well known differential equation,

$$\begin{aligned} -u''(x) &= f(x) \text{ for } x \in (0,1) \\ u(0) &= g_0 \\ u(1) &= g_1. \end{aligned} \tag{1}$$

This may seem too simple, after all an analytic solution is easily available.

$$u(x) = -\int_0^x \int_0^t f(s)dsdt + C_{1,g} + C_{2,g}x. \tag{2}$$

This is nice and convenient, and the methods of numerical integration are fast and efficient. However this strategy will not work, or at least not nearly as easily in higher dimensions.

$$\begin{aligned} -\Delta u &= f(x) \text{ for } x \in \Omega \subset \mathbb{R}^n \\ u|_{\partial\Omega} &= g(x) \end{aligned} \tag{3}$$

Our strategy is not to solve by integrating but by matrix-ifying our equation. This, discretization strategy is quite powerful and it will work in any dimension; but focusing on the version is will provide enough generality to communicate all the important ideas.

We will replace (1) with an approximate problem. Consider a partition of the unit interval into equal pieces, each of width $h = 1/N$. When $u(x)$ is restricted to the endpoints of the partition, it can be

thought of as a vector with $n+1$ entries. So $\mathbf{u}_k = u(kh)$ for $k = 0, \ldots, N$. There is a nice way to estimate the derivative using only the points in our partition.

$$\lim_{\Delta x \to 0} \frac{u(kh + \Delta x) - 2u(kh) + u(kh - \Delta x)}{\Delta x^2} \approx \frac{\mathbf{u}_{k+1} - 2\mathbf{u}_k + \mathbf{u}_{k-1}}{h^2} = u''(kh) + O(h^2) \tag{4}$$

This error estimate can be verified by expanding $u(x)$ in Taylor series. From the last equality in (4) and a little algebra, we can rephrase (4) as an $(N-1) \times (N-1)$ linear system.

$$\begin{bmatrix} 2 & -1 & 0 & \ldots & 0 \\ -1 & 2 & -1 & \ldots & 0 \\ 0 & -1 & 2 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} h^2 f_1 - g_0 \\ h^2 f_2 \\ h^2 f_3 \\ \vdots \\ h^2 f_{N-1} - g_1 \end{bmatrix} \tag{5}$$

It is not hard to verify that the tridiagonal matrix on the left hand side really is symmetric and positive definite. It is straightforward to construct our matrix in MATLAB.

```
e = ones(N-1,1);
A = spdiags([-1*e 2*e -1*e], -1:1, N-1, N-1);
```

We will also intialize our problem by choosing $f(x) = \sinh(x)$, $g_0, g_1 = 0$, and $N = 100$. This gives us a consistent benchmark to judge the speed of the different algorithms.

```
N= 100;
h = 1/N;
pts = h:h:1-h;
f = h*h*sinh(pts);
g = [0,0];
f(1) = f(1) - g(1);
f(N-1) = f(N-1) - g(2);
```

The first algorithms we introduce are too slow and the later algorithms are too fast, so we will adjust the size of N as we go along.

3 The A-linear algebra.

We will focus on the abstract problem for the time being. We want to evalue to inverse of an SPD matrix A at a particular point.

$$A\mathbf{x} = \mathbf{b} \implies \mathbf{x}_* = A^{-1}\mathbf{b}. \qquad \text{We want to evaluate the inverse.} \tag{6}$$

$$A = A^t. \qquad A \text{ is symmetric.} \tag{7}$$

$$\text{For all } \mathbf{x} \neq 0,\ \langle \mathbf{x}, A\mathbf{x} \rangle = \mathbf{x}^t A \mathbf{x} > 0. \qquad A \text{ is postive definite.} \tag{8}$$

The goal is calculate $\mathbf{x}_*$, and if there is a trade off between finding an exact solution and finding an approximate solution quickly we will prefer the quick solution. The approach that is most well known is Gaussian elimination, and this will give the exact solution, but this process can be extremely, even infeasibly slow when unleased on a sufficiently large system. We introduce some terminology to clarify what we mean by approximate.

Definition 3.1 (Error)**.** *If $\mathbf{x}$ is a approximation to $\mathbf{x}_*$ then we will define respectively the error vector and the error of the approximation as*

$$\epsilon_{\mathbf{x}} = \mathbf{x}_* - \mathbf{x} \tag{9}$$

$$\|\epsilon_{\mathbf{x}}\| = \|\mathbf{x}_* - \mathbf{x}\| \tag{10}$$

Definition 3.2 (Residual)**.** *We say that a vector $\mathbf{x}$ is an approximation to $\mathbf{x}_*$ then we say the residual of the approximation is,*

$$\mathbf{r}_{\mathbf{x}} = \mathbf{b} - A\mathbf{x}. \tag{11}$$

We will omit the subscripts when the meaning is clear from context. Intuitively the error simply tells us how far our approximation misses the mark. Unfortunately we can't get our hands on the error exactly because that would require knowing $\mathbf{x}_*$. The residual will work as an auxillary, more tangible, measurement of the error. We can use the fact that $\mathbf{b} = A\mathbf{x}_*$ to see this more clearly.

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} = A\mathbf{x}_* - A\mathbf{x} = A(\mathbf{x}_* - \mathbf{x}) = A\epsilon. \tag{12}$$

This is a reasonable substutite, it is straightforward to compare the size of the error to the size of the residual.

$$\|\mathbf{r}\| = \sqrt{\langle A\epsilon, A\epsilon\rangle} = \|\epsilon\|\sqrt{\left\langle A\frac{\epsilon_x}{\|\epsilon\|}, A\frac{\epsilon}{\|\epsilon\|}\right\rangle} \leq \|\epsilon\|\sqrt{\sup_{\|\mathbf{u}\|=1}\langle A\mathbf{u}, A\mathbf{u}\rangle} = \|\epsilon\| \cdot \|A\| \tag{13}$$

This is what we will work with to measure error but this has some limitations. We assume that calculating $\|A\|$ is an expensive task, so without knowledge of the norm it could be difficult to judge whether the resolvent actually corresponds to a good approximation. If $\|A\|$ is large then it is possible that a large resolvent could actually correspond to a good approximation and if the norm is small we would have the opposite problem. This first issue is solved by making sure to judge the quality of $\|r\|$ by how it compares to the size of $\|b\|$.

There is another issue however. Since we assumed that A is SPD this implies that it has only positive real eigenvalues, an equivalent statement is to say that the image of a sphere in $\mathbb{R}^n$ under A will be an ellipsoid. If this ellipsoid is highly eccentric, so the largest axis is much longer than the shortest axis, (i.e. the largest eigenvalue is much larger than the smallest eigenvalue) then the resolvent can only work as a crude estimate of the error. We call these problem causing matrices *ill-conditioned* and we will cover how to deal with them in more detail in section 5, see [3] §18 for an explicit pathological example.

Our strategy is to use an iterative process to create a sequence of approximations, $\mathbf{x}_0, \mathbf{x}_1, \ldots$ and a corresponding sequence of resolvents $\mathbf{r}_0, \mathbf{r}_1, \ldots$ we will halt the process once the norm of the resolvent becomes small, and when A is well-conditioned this approach is going to work. For simplicity we will use the convention that $\mathbf{x}_0 = 0$ and $\mathbf{r}_0 = \mathbf{b}$.

Definition 3.3 (Geometry of A.)**.** *We define the A-inner product and the A-norm respectively*

$$\langle \mathbf{u}, \mathbf{v}\rangle_A = \langle \mathbf{u}, A\mathbf{v}\rangle = \langle A\mathbf{u}, \mathbf{v}\rangle \tag{14}$$

$$\|\mathbf{u}\|_A = \sqrt{\langle \mathbf{u}, A\mathbf{u}\rangle} \tag{15}$$

Since A is SPD, $\langle\cdot,\cdot\rangle_A$ really is an inner product and $\|\cdot\|_A$ really is a norm. The analogy continues though, here is more terminology to discuss the geometry associated to A.

Definition 3.4 (Conjugate Vectors)**.** $\mathbf{u}$ *and* $\mathbf{v}$ *are conjugate if they are A-orthogonal, that is* $\langle \mathbf{u}, \mathbf{v}\rangle_A = 0$.

Definition 3.5 (A-Projection.)**.** *If S is a subspace of $\mathbb{R}^n$ we can define the A-projection of* $\mathbf{u}$ *onto S as the unique vector in S which satisfies the following:*

$$\langle \mathrm{Proj}_S^A \mathbf{u}, \mathbf{v}\rangle_A = \langle \mathbf{u}, \mathbf{v}\rangle_A \text{ for all } \mathbf{v} \in S. \tag{16}$$

This definition is nice, but it does not tell us exactly *how* to calculate the A projection. Thankfully this is completely analogous computing orthogonal projections in $\mathbb{R}^n$ in general. Suppose $S = \mathrm{span}\{\mathbf{v}_1, \ldots \mathbf{v}_k\}$, and $\{\mathbf{v}_i\}$ are mutually conjugate then

$$\mathrm{Proj}_S^A \mathbf{u} = \sum_{i=1}^{k} \frac{\langle \mathbf{u}, \mathbf{v}_i\rangle_A}{\langle \mathbf{v}_i, \mathbf{v}_i\rangle_A}\mathbf{v}_i \tag{17}$$

These concepts are crucial to the method. Suppose that we had a collection of n mutually conjugate vectors $\mathbf{p}_1, \mathbf{p}_2, \ldots \mathbf{p}_n$. Then we could make short work of our problem. There would be some real numbers $\{\alpha_i\}_{i=1,\ldots n}$ such that,

$$\mathbf{x}_* = \sum_{i=1}^{n} \alpha_i \mathbf{p}_i \tag{18}$$

But since the $\mathbf{p}_i$ are mutually conjugate, calculating the α_i is simple. We can take the A-inner product of $\mathbf{x}_*$ with $\mathbf{p}_j$,

$$\langle \mathbf{p}_j, \mathbf{x}_*\rangle_A = \sum_{i=1}^{n} \alpha_i \langle \mathbf{p}_j, \mathbf{p}_i\rangle_A = \alpha_j \langle \mathbf{p}_j, \mathbf{p}_j\rangle_A. \tag{19}$$

But on the other hand we have that,

$$\langle \mathbf{p}_j, \mathbf{x}_*\rangle_A = \langle \mathbf{p}_j, A\mathbf{x}_*\rangle = \langle \mathbf{p}_j, \mathbf{b}\rangle. \tag{20}$$

We simply solve for the α_j and find our solution.

$$\mathbf{x}_* = \sum_{i=1}^{n} \frac{\langle \mathbf{p}_i, \mathbf{b} \rangle}{\langle \mathbf{p}_i, \mathbf{p}_i \rangle_A} \mathbf{p}_i. \tag{21}$$

Of course this is unrealistic, but it is a good conceptual starting point. Computing and storing the entire conjugate basis for $\mathbb{R}^n$ is going to be slow and also require a lot of space. We will be able to reformulate the idea in terms of an iterative process. This will work similarly to gradient descent, but instead of strictly following the flow of the gradient we will impose the condition that the residual of the next step is A-orthogonal to the subspace spanned by the previous residuals.

4 The Conjugate Gradient Method

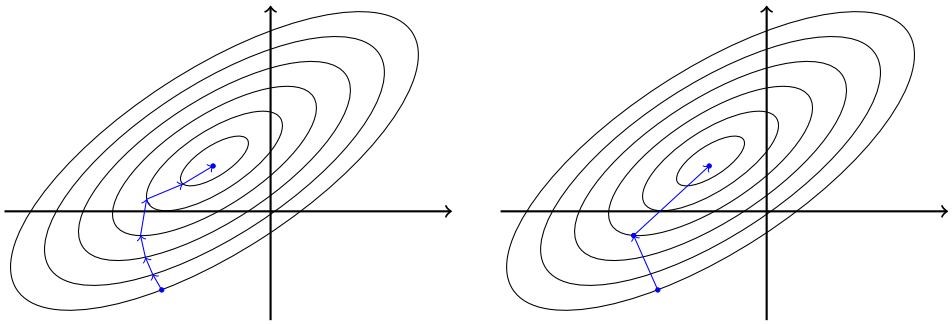

Figure 1: Gradient descent vs. conjugate gradient method.

Notice we can rephrase problem (6) in terms of minimizing a convex function.

$$\varphi(\mathbf{v}) = \tfrac{1}{2}\|\mathbf{v}\|_A^2 - \langle \mathbf{b}, \mathbf{v} \rangle \tag{22}$$

$$A\mathbf{x} = \mathbf{b} \iff \mathbf{x} = \operatorname*{arg\,min}_{\mathbf{v} \in \mathbb{R}^n} f(\mathbf{v}) \tag{23}$$

For this simple convex function we have a nice expression for the gradient.

$$-\nabla\varphi(\mathbf{x}) = \mathbf{b} - A\mathbf{x} = \mathbf{r}_\mathbf{x}. \tag{24}$$

The negative gradient at $\mathbf{x}$ is simply the corresponding residual. Applying gradient descent is going to be our first attempt at solving the problem. We will use α for the step size.

4.1 Algorithm 1: Gradient Descent

```
function x = Gradient_Descent(A, b, tol, alpha, MaxIter)
x = zeros(length(b),1);
r = b;
k = 0;
tol = tol*norm(f);
    while norm(r) ≥ tol && k < MaxIter
        r = b- A*x;
        x = x + alpha*r;
        k = k+1;
    end
end
```

This is easy to understand but it is not very good. To get an idea of how not efficient this code is, when $N = 100$ and $\alpha = 10^{-5}$, it takes a full 60 seconds for the code to compute an answer that has a relative error of 0.005%. Simply running $A \setminus f$ to compute the answer takes 0.07 milliseconds for comparison. Toying with α and other parameters can speed up the code or make it more accurate. If we were stuck with just *some* convex function it can be hard to do better than this, but we know more about our problem than just convexity.

We can make an improvement by having α_k instead of just α. What if choose the α_k, so that the update from $\mathbf{x}_{k+1}$ to $\mathbf{x}_k$ makes $\varphi(\mathbf{x}_{k+1})$ as small as possible. This is the steepest gradient descent algorithm. In equations we want that

$$\alpha_k = \arg\min_{\alpha \in \mathbb{R}} \varphi(\mathbf{x}_k + \alpha \mathbf{r}_k). \tag{25}$$

It is actually not that hard to find the minimum of this function.

$$\alpha_k \mathbf{r}_k = \mathrm{Proj}^A_{\mathbf{r}_k}(\mathbf{x}_* - \mathbf{x}_k) = \frac{\langle \mathbf{r}_k, \mathbf{x}_* - \mathbf{x}_k \rangle_A}{\langle \mathbf{r}_k, \mathbf{r}_k \rangle_A} \mathbf{r}_k = \frac{\langle \mathbf{r}_k, \mathbf{r}_k \rangle}{\langle \mathbf{r}_k, \mathbf{r}_k \rangle_A} \mathbf{r}_k. \tag{26}$$

By using the A-projection we can choose α_k so that $\mathbf{x}_k + \alpha_k \mathbf{r}_k$ is the closest point (measured in the A-norm) to $\mathbf{x}_*$ on the line spanned by $\{\mathbf{x}_k + \alpha \mathbf{r}_k\}$ as α ranges over $\mathbb{R}$. This will also minimize the value of φ along the line $x_k + \alpha \mathbf{r}_k$. From this we get an improved gradient descent algorithm.

4.2 Algorithm 2: Steepest Gradient Descent

```
function x = Steepest_Gradient_Descent(A, b, tol, MaxIter)
x = zeros(length(b),1);
r = b;
k = 0;
tol = tol*norm(b);
    while norm(r) ≥ tol && k < MaxIter
        r = b- A*x;
        alpha = (r'*r)/(r'*(A*r));
        x = x + alpha*r;
        k = k+1;
    end
end
```

This is a vast improvement from regular gradient descent; this program does the same job ($N = 100$) in only 23ms. Not bad, but $A \setminus \mathbf{b}$ solves the problem in 0.05ms so we can go a lot faster. When we try $N = 1,000$ steepest gradient descent starts to sputter though, it takes 7s to find the inverse, but $A \setminus \mathbf{b}$ still only takes 0.06ms. Our algorithm is not guaranteed to terminate after a fixed point, if A is even somewhat ill-conditioned, then the steepest gradient method could still zig-zag through the level curves for a quite a while before coming close to a solution. As we will see later, A is in fact ill conditioned for large values of N.

We introduce the first CG method as a modification of the steepest gradient descent algorithm. The crux of the idea is to use an A-orthogonal set of vectors $\mathbf{p}_k$, instead of the residuals, to update our guesses. At each step of the process we will use $\mathbf{r}_k$ to construct a new vector $\mathbf{p}_k$ which is conjugate to $S_{k-1} := \mathrm{span}\{\mathbf{p}_0, \ldots \mathbf{p}_{k-1}\}$ and then use $\mathbf{p}_k$ as the direction to move from $\mathbf{x}_k$ to $\mathbf{x}_{k+1}$.

4.3 Algorithm 3.0: CG Method (sketch)

1. Initialize with $\mathbf{x}_0 = 0$ and $\mathbf{r}_0 = \mathbf{p}_0 = \mathbf{b}$.
2. Repeat the following steps until $\|\mathbf{r}_k\|$ is small.
3. Select a step size α_k.
4. Update our guess $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$.
5. Compute the residual $\mathbf{r}_{k+1} = \mathbf{b} - A\mathbf{x}_{k+1}$,
6. and the new conjugate direction $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} - \mathrm{Proj}^A_{S_k} \mathbf{r}_{k+1}$.

We will simplify our notation going forward a little bit by saying $\Pi^k = \mathrm{Proj}^A_{S^k}$. For this algorithm there is only a description and no accompanying code for this sketch because of two glaring issues, picking the best α_k and computing $\Pi^k \mathbf{r}_{k+1}$. For the α_k, when we are using the $\mathbf{p}_k$ as our directions instead of the residuals we have a better option than simply being greedy with the objective function. Recall (18), since we forced the $\mathbf{p}_k$ to be mutually conjugate we can take advantage of this and select the α_i so that our algorithm must terminate in n steps. From equations (19) and (20) we see that

$$\alpha_k = \frac{\langle \mathbf{b}, \mathbf{p}_k \rangle}{\langle \mathbf{p}_k, \mathbf{p}_k \rangle_A} \tag{27}$$

The other problem is calculating $\Pi^{k-1}\mathbf{r}_k$. From (17) we have one way to compute the projection.

$$\Pi^{k-1}\mathbf{r}_k = \frac{\langle \mathbf{r}_k, \mathbf{p}_{k-1}\rangle_A}{\langle \mathbf{p}_{k-1}, \mathbf{p}_{k-1}\rangle_A}\mathbf{p}_{k-1} + \Pi^{k-2}\mathbf{r}_k = \sum_{j=1}^{k-1} \frac{\langle \mathbf{r}_k, \mathbf{p}_j\rangle_A}{\langle \mathbf{p}_j, \mathbf{p}_j\rangle_A}\mathbf{p}_j. \tag{28}$$

If we try to put this into our code it will be wildly inefficient and waste not only time but also space. More insight into the linear algebra of our setup reveals that we actually do not need to compute all of those dot products in order to find the projection.

Lemma 4.1. *The vector $\mathbf{r}_k$ is A-orthogonal to S_{k-2}, that is $\Pi^{k-2}\mathbf{r}_k = 0$ (the cases where $k = 0, 1$ are exceptional, but also trivial).*

Proof. This proof is a little unintuitive. First we show that $\mathbf{r}_k \perp S_{k-1}$, then from there we can prove that $\mathbf{r}_k$ is A-orthogonal to S_{k-2}. At each step our algorithm chooses the α_{k-1} so that $\mathbf{x}_k$ is the closest point (in the A-norm) to $\mathbf{x}_*$ in the space S^{k-1}. In equations this means that,

$$\mathbf{x}_k = \Pi^{k-1}\mathbf{x}_*. \tag{29}$$

We can rewrite this to get the following,

$$\mathbf{x}_* - \mathbf{x}_k = (I - \Pi^{k-1})\mathbf{x}_* \implies (\mathbf{x}_* - \mathbf{x}_k) \perp_A S_{k-1} \tag{30}$$

$$\implies A(\mathbf{x}_* - \mathbf{x}_k) \perp S_{k-1} \implies \mathbf{r}_k \perp S_{k-1}. \tag{31}$$

Remember that $(I - \Pi_S)$ is a projection onto $S^{\perp}$. Since $\mathbf{r}_0 = \mathbf{p}_0 = \mathbf{b}$ and $\mathbf{r}_k \perp S_{k-1}$ it means that, by induction, $\mathbf{r}_k \in S^k$ for all k. This is neat, in particular we have that the $\mathbf{r}_k$ and $\mathbf{p}_k$ will span the same subspaces. There is a recursive way to write the resolvent

$$\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k = \mathbf{b} - A\mathbf{x}_{k-1} + A\mathbf{x}_{k-1} - A\mathbf{x}_k \tag{32}$$

$$\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_{k-1}A\mathbf{p}_{k-1}. \tag{33}$$

Suppose that $0 \le i \le k-2$, from the previous equation we see that

$$A\mathbf{p}_i \in \operatorname{span}\{\mathbf{r}_i, \mathbf{r}_{i+1}\} \subset S_{i+1} \subset S_{k-1}. \tag{34}$$

From the fact that $\mathbf{r}_k \perp S_{k-1}$, we can reach our conclusion, for all i with $0 \le i \in k-2$ we have,

$$\mathbf{r}_k \perp S_{k-1} \implies \mathbf{r}_k \perp A\mathbf{p}_i \implies \mathbf{r}_k \perp_A \mathbf{p}_i \implies \mathbf{r}_k \perp_A S_{k-2}. \tag{35}$$

□

From this lemma we can find a more efficient expression for $\mathbf{p}_{k+1}$.

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k = \mathbf{r}_{k+1} - \frac{\langle \mathbf{r}_{k+1}, \mathbf{p}_k\rangle_A}{\langle \mathbf{p}_k, \mathbf{p}_k\rangle_A}\mathbf{p}_k \tag{36}$$

Now we are ready to give the first implementation of our algorithm.

4.4 Algorithm 3.1: CG Method

```
function x = CG1(A, b, tol)
x = zeros(length(b),1);
p = b;
r = b;
tol = tol*norm(b);
k = 1;
    while norm(r) ≥ tol && k < length(b)+1
        Ap = A*p;
        p2 = p'*Ap;
        alpha = (b'*p)/p2;
        x = x + alpha*p;
        r = r - alpha*Ap;
        beta = - (r'*Ap)/p2;
        p = r + beta*p;
        k = k+1;
    end
end
```

This one solves the $N = 1,000$ problem in 7.3ms, a thousand times faster than steepest gradient descent, but nowhere near as fast as $A \setminus \mathbf{b}$. It still does not scale as efficiently either, for $N = 10,000$, CG takes 1s but $A \setminus \mathbf{b}$ only takes 0.2ms. By elaborating on the concepts presented in the proof of Lemma 4.1 it is possible to improve this implementation of CG a little more. We can come up with alternative expressions for α_k and β_k, these do not speed up the process very much, but these expressions are more numerically stable.

Proposition 4.1.

$$\alpha_k = \frac{\langle \mathbf{b}, \mathbf{p}_k \rangle}{\langle \mathbf{p}_k, \mathbf{p}_k \rangle_A} = \frac{\langle \mathbf{r}_k, \mathbf{r}_k \rangle}{\langle \mathbf{p}_k, \mathbf{p}_k \rangle_A} \tag{37}$$

Proof. We can use the A-orthogonality of the $\mathbf{p}_k$ and the fact that $\mathbf{r}_k = r_0 - \sum_{i=0}^{k-1} \alpha_i A\mathbf{p}_i$ to get the following

$$\langle \mathbf{x}_*, \mathbf{p}_k \rangle_A = \langle \mathbf{r}_0, \mathbf{p}_k \rangle = \langle \mathbf{r}_k, \mathbf{p}_k \rangle \tag{38}$$

Then we use the fact that $\mathbf{r}_k \perp S_{k-1}$ and the formula $\mathbf{p}_k = \mathbf{r}_k + \sum_{i=0}^{k-1} \beta_i \mathbf{p}_i$ to get

$$\langle \mathbf{r}_k, \mathbf{p}_k \rangle = \langle \mathbf{r}_k, \mathbf{r}_k \rangle. \tag{39}$$

Since $\mathbf{r}_0 = \mathbf{b}$ we have the desired result. □

Proposition 4.2.

$$\beta_k = -\frac{\langle \mathbf{r}_{k+1}, \mathbf{p}_k \rangle_A}{\langle \mathbf{p}_k, \mathbf{p}_k \rangle_A} = \frac{\langle \mathbf{r}_{k+1}, \mathbf{r}_{k+1} \rangle}{\langle \mathbf{r}_k, \mathbf{r}_k \rangle} \tag{40}$$

Proof. We can use the recursive form of the residual and the orthogonality of the $\mathbf{r}_k$ to get that,

$$\langle \mathbf{r}_{k+1}, \mathbf{r}_{k+1} \rangle = \langle \mathbf{r}_{k+1}, \mathbf{r}_k \rangle - \alpha_k \langle \mathbf{r}_{k+1}, A\mathbf{p}_k \rangle = -\alpha_k \langle \mathbf{r}_{k+1}, A\mathbf{p}_k \rangle. \tag{41}$$

Now we apply Proposition 4.1,

$$\beta_k = -\frac{\langle \mathbf{r}_{k+1}, \mathbf{p}_k \rangle_A}{\langle \mathbf{p}_k, \mathbf{p}_k \rangle_A} = \frac{\langle \mathbf{r}_{k+1}, \mathbf{r}_{k+1} \rangle}{\alpha_k} \frac{1}{\langle \mathbf{p}_k, \mathbf{p}_k \rangle_A} = \frac{\langle \mathbf{r}_{k+1}, \mathbf{r}_{k+1} \rangle}{\langle \mathbf{r}_k, \mathbf{r}_k \rangle} \tag{42}$$

□

4.5 Algorithm 3.2: CG Method

```
function x = CG2(A, b, tol)
x = zeros(length(b),1);
p = b;
r = b;
tol = tol*norm(b);
k = 1;
r2 = r'*r;
    while norm(r) ≥ tol && k < length(b)+1
        Ap = A*p;
        alpha = r2/(p'*Ap);
        x = x + alpha*p;
        r = r - alpha*Ap;
        r2old = r2;
        r2 = r'*r;
        beta = r2/r2old;
        p = r + beta*p;
        k = k+1;
    end
end
```

This version is basically the same speed, but for $N = 20,000$ both algorithms take about 3s to find an approximation. This version of CG outputs a much more accurate answer; the relative errors are 11% and 2.7×10^{-9}% respectively. This is the best method we will derive; but judging by the performance of Matlab it is clear that faster and more sophisticated methods are known.

5 Convergence, condition, and preconditioning

Definition 5.1 (Krylov Subspaces.)**.** *We define the k^{th} Krylov Subspace for A,*

$$\mathbb{V}_k = \text{span}\{\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, \ldots, A^k\mathbf{b}\}. \tag{43}$$

The Conjugate Gradient method is known as a Krylov subspace method because the k^{th} estimate, $\mathbf{x_k}$, will be the closest point to $\mathbf{x}_*$ in $\mathbb{V}_{k-1}$ when measuring distance using the A-norm. There are other well known Krylov subspace methods, notably GMRES (see [4]), but we will not cover them in this exposition.

Lemma 5.1.

$$S_k = \mathbb{V}_k \tag{44}$$

Proof. For $k = 0$ it is obvious since $\mathbf{p}_0 = \mathbf{r}_0 = \mathbf{b}$. We proceed by induction. Suppose it holds up to k, we can apply the recursive formula for the residual to get the desired result.

$$S_{k+1} = \mathbb{V}_k + \text{span}\{\mathbf{r}_{k+1}\} = \mathbb{V}_k + \text{span}\{\mathbf{r}_k, A\mathbf{p}_k\} = \mathbb{V}_k + \text{span}\{A\mathbf{p}_k\} = \mathbb{V}_k + A\mathbb{V}_k = \mathbb{V}_{k+1}. \tag{45}$$

□

Theorem 5.1. *Let $\mathbf{x}_k$ be the k^{th} estimate from the CG method, and let $\mathcal{P}_k$ be the set of real polynomials with degree at most k. Then we have that*

$$\|\mathbf{x}_* - \mathbf{x}_k\|_A = \inf_{\mathbf{v}\in\mathbb{V}_{k-1}} \|\mathbf{x}_* - \mathbf{v}\|_A. \tag{46}$$

$$\|\mathbf{x}_* - \mathbf{x}_k\|_A = \inf_{p\in\mathcal{P}_k, p(0)=1} \|p(A)\mathbf{x}_*\|_A \tag{47}$$

$$\leq \|\mathbf{x}_*\|_A \inf_{p\in\mathcal{P}_k, p(0)=1} \sup_{\lambda\in\sigma(A)} |p(\lambda)| \tag{48}$$

Proof. To prove (46) it simply follows from the fact that $\mathbf{x}_* - \mathbf{x} = (I - \Pi_{k-1})\mathbf{x}_*$; i.e. the (A)-projection is the closest point (in the A norm) in any subspace.

It should be clear that (48) follows immediately from (47), so all that is left is to prove (47).

Consider $\mathbf{v}$ from (46), since it is in the Krylov subspace we can write

$$\mathbf{v} = \sum_{i=0}^{k-1} c_i A^i\mathbf{b} = \sum_{i=0}^{k-1} c_i A^i(A\mathbf{x}_*) = \sum_{i=1}^{k} c_{i-1} A^i\mathbf{x}_* \tag{49}$$

Suppose $p_k(x) = 1 - \sum_{i=1}^{k} c_{i-1}x^i$. Then we have that

$$\mathbf{x}_* - \mathbf{v} = p_k(A)\mathbf{x}_*. \tag{50}$$

From (46), this gives the desired result. □

We should focus on (48), if we can estimate the infimum then we can get a much better idea of exactly how fast CG converges.

Definition 5.2 (Condition Number)**.**

$$\kappa = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}. \tag{51}$$

Theorem 5.2. *Let $\mathbf{x}_k$ be the k^{th} step of the CG method. Then,*

$$\|\mathbf{x}_* - \mathbf{x}_k\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|\mathbf{x}_*\|_A. \tag{52}$$

Proof. This follows from (48). We can estimate the infimum over $\mathcal{P}_k$ from above by simply selecting any particular polynomial. Consider the Chebyshev polynomials.

$$T_k(x) = \begin{cases} \cos(k\cos^{-1}(x)) & \text{if } |x| \leq 1 \\ \cosh(k\cosh^{-1}(x)) & \text{if } |x| \geq 1 \end{cases} \tag{53}$$

Using Euler's formula ($e^{i\theta} = \cos\theta + i\sin\theta$) it is not too difficult to show that the Chebyshev polynomials really are polynomials. We are going to have to rescale them. Let $b = \lambda_{\max}(A)$ and let $a = \lambda_{\min}(A)$. We can use a mapping

$$g : [a, b] \to [-1, 1] \text{ as } f(x) = \frac{b + a - 2x}{b - a}. \tag{54}$$

Then we select our polynomial to use in (48),

$$p_k(x) = \frac{T_k(g(x))}{T_k(g(0))}. \tag{55}$$

It is clear that $p_k(x)$ really is a polynomial of degree k and that $p_k(0) = 1$, so,

$$\|\mathbf{x}_* - \mathbf{x}_k\|_A \leq \|\mathbf{x}_*\|_A \sup_{\lambda\in\sigma(A)} |p_k(\lambda)| \leq \|\mathbf{x}_*\|_A \sup_{\lambda\in[a,b]} |p_k(\lambda)| \tag{56}$$

$$\leq \|\mathbf{x}_*\|_A \sup_{\lambda\in[a,b]} \frac{1}{|T_k(g(0))|} \text{ since } g(x) \in [-1, 1] \text{ for } x \in [a, b] \text{ and } |cos(x)| \leq 1. \tag{57}$$

$$= \frac{\|\mathbf{x}_*\|_A}{|T_k(g(0))|} = \|\mathbf{x}_*\|_A \left|T_k\left(\frac{b + a}{b - a}\right)\right|^{-1} \tag{58}$$

Now we have that $g(0) > 1$ so let's say that

$$\frac{b + a}{b - a} = \cosh\sigma \tag{59}$$

For some σ, using the fact that $\cosh(x) = \frac{1}{2}(e^x + e^{-x})$ and that $\kappa = b/a$, we can get that

$$e^{\sigma} = \frac{\sqrt{\kappa} + 1}{\sqrt{\kappa} - 1} \tag{60}$$

Therefore,

$$T_k\left(\frac{b + a}{b - a}\right) = \cosh(k\sigma) = \frac{e^{k\sigma} + e^{-k\sigma}}{2} \geq \frac{1}{2}e^{k\sigma} = \frac{1}{2}\left(\frac{\sqrt{\kappa} + 1}{\sqrt{\kappa} - 1}\right)^k. \tag{61}$$

Returning to the estimate for $\|\mathbf{x}_* - \mathbf{x}_k\|_A$ we get that,

$$\|\mathbf{x}_* - \mathbf{x}_k\|_A \leq \|\mathbf{x}_*\|_A \left|T_k\left(\frac{b + a}{b - a}\right)\right|^{-1} \leq 2\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k \|\mathbf{x}_*\|_A. \tag{62}$$

□

First we will note that picking the Chebyshev polynomials is not arbitrary, in fact it can be shown that they are the optimal choice for these kinds of problems; but we will not show that in these notes.

Second, it is important to understand the condition number conceptually since it is going to tell us the speed of convergence for CG. κ is a measure of how spread out the eigenvalues of an SPD matrix are, if they are identical then $A = cI$ (the inverse is trivial) then $\kappa = 1$, and if the eigenvalues are far from each other, so $\lambda_{\min}$ is close to zero and $\lambda_{\max}$ is far from zero, then $\kappa \gg 1$ and convergence will be slow.

It turns out that the matrix A from (5), really is ill conditioned. We are going to estimate $\kappa(A)$ from above by finding an upper bound on $\lambda_{\min}$ and a lower bound on $\lambda_{\max}$. To estimate $\lambda_{\min}$, we begin by writing A as a sum of two matrices. We have that,

$$A = A_0 + E = \begin{bmatrix} 1 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ 0 & -1 & 2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \tag{63}$$

Notice that the row sums of A_0 are all 0. Suppose (for simplicity) that A has side length N and let $\mathbf{u} = \frac{1}{\sqrt{N}}[1, 1, \dots, 1]^T$, so that $|\mathbf{u}| = 1$, then

$$0 < \lambda_{\min}(A) = \inf_{|\mathbf{v}|=1} |A\mathbf{v}| \leq |A\mathbf{u}| = |A_0\mathbf{u} + E\mathbf{u}| = \left|\mathbf{0} - \frac{1}{\sqrt{N}}[1, 0, \dots, 0, 1]\right| = \sqrt{\frac{2}{N}}. \tag{64}$$

The lower bound is simply because A is SPD. Now we are going to use a completely different idea to get the bound for $\lambda_{\max}$.

We will think of our partition of $[0,1]$ as a graph. The matrix A will be the Graph Laplacian. Notice that each row and each column of A correspond to a particular node in the partition of $[0,1]$, the entries on the main diagonal are all 2's and this corresponds to the fact that each node (except for the endpoints which are different because of the boundary conditions) has two neighbors. Off of the main diagonal the entry $A_{ij} = -1$ if the i^{th} node is directly adjacent to the j^{th} node, and otherwise $A_{ij} = 0$. We are going to reorder the vertices, this will correspond to caluculating $P^{-1}AP$ for some permutation matrix P. It is well known that this permutation has no effect on the eigenvalues of A.

We will color the nodes in the partition in alternating red and blue.

The next step is to permute the rows and columns of A according to this coloring, we can represent the permutation easily in the form of a block matrix.

$$P^{-1}AP = \begin{bmatrix} 2I & Q^T \\ Q & 2I \end{bmatrix} \tag{65}$$

Here Q is some matrix that contains only 0's and -1's it represents the connections between blue and red nodes. On the main diagonal there is only $2I$ because no blue node is adjacent to a blue node and no red node is adjacent to a red node. Now we can use the Schur complement for block matrices, to discover something about the characteristic polynomial of A.

$$\det(\lambda I - A) = \det(P^{-1}(\lambda I - A)P) = \det(\lambda I - P^{-1}AP) = \det((\lambda - 2)^2 I - Q^T Q) \tag{66}$$

So $\det(A - \lambda I) = q((2-\lambda)^2)$ for some polynomial q. This means that if λ is eigenvalue for A, then $2 - \lambda$ is also an eigenvalue of A. From this and (64) we see that

$$2 - \sqrt{\frac{2}{N}} \leq \lambda_{\max} \leq 2. \tag{67}$$

Therefore

$$\kappa \geq \frac{2 - \sqrt{\frac{2}{N}}}{\sqrt{\frac{2}{N}}} = \sqrt{2N} - 1 > \sqrt{N}. \tag{68}$$

We can return to Theorem 5.2, notice that $\frac{\sqrt{x}-1}{\sqrt{x}+1}$ is a decreasing function. We get that

$$\|\mathbf{x}_* - \mathbf{x}_k\|_A \leq 2 \left(\frac{\sqrt[4]{N} - 1}{\sqrt[4]{N} + 1} \right)^k \|\mathbf{x}_*\|_A. \tag{69}$$

We have the fourth root of N, in the coefficient, so this tells us that CG will perform well for our matrices at least until $\sqrt[4]{N}$ starts to get large.

It is possible to make one more improvement on CG, the so called PCG method. This method is very interesting, unfortunately it is *too* interesting for me to cover it in detail. Briefly the idea is to find a different SPD matrix $B = R^T R$ and then look at modified problem

$$\begin{gathered} R^{-T}AR^{-1}\mathbf{y} = R^{-T}\mathbf{b} \\ \mathbf{y} = R\mathbf{x} \end{gathered} \tag{70}$$

This matrix B is called the preconditioner and we will want to select B so that

$$\kappa(R^{-T}AR^{-1}) \ll \kappa(A). \tag{71}$$

When we apply CG to the modified system we will be able to speed up the convergence. The problem of finding a good preconditioner quickly is well known and well known to be difficult. The Incomplete Cholesky factorization is a popular preconditioner, but in general how to select B is an open field of research.

References

[1] Long Chen. *Conjugate Gradient Methods*. 2016. URL: https://www.math.uci.edu/~chenlong/lectures.html.

[2] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, 1997. DOI: 10.1137/1.9781611970937. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970937. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611970937.

[3] Magnus R. Hestenes and Eduard Stiefel. "Methods of conjugate gradients for solving linear systems". In: *Journal of research of the National Bureau of Standards* 49 (1952), pp. 409–435.

[4] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics, 1995. DOI: 10.1137/1.9781611970944. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970944. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611970944.