Iterative Methods: GMRES and CG

Emma Hayes

May 11, 2025

1 Iterative Methods

Solving linear systems $Ax = b$ is a very common problem in numerical linear algebra, but many direct methods are costly. A lot of the A matrices we work with in practice have patterns that direct methods do not exploit. For instance, we can often represent these matrices sparsely, allowing us to work on a smaller input. Iterative methods allow us to exploit the sparsity of matrices in practice.

The iterative methods I will be discussing are Krylov subspace methods. The idea of a Krylov subspace method comes from an intuitive argument. A natural first guess at a solution to $Ax = b$ is some $x_1 \in span\{b\}$, then we would have to take Ab and our next guess would be some $x_2 \in span\{b, Ab\}$. We continue on in this way, constructing a Krylov subspace $x + k \in span\{b, Ab, A^2b, \ldots, A^{k-1}b\}$

2 Conjugate Gradient (CG)

Let $A \in \mathbb{R}^{m\times m}$ be a symmetric, nonsingular, positive definite matrix and $b \in \mathbb{R}^m$ a vector. Then, let $\mathcal{K}_n$ denote the Krylov subspace $span\{b, Ab, \ldots, A^{n-1}b\}$. Our goal is to solve the system $Ax = b$, and we can denote the exact solution as $x_* = A^{-1}b$.

We can define the A-norm, $\|x\|_A = \sqrt{x^T Ax}$. Then we will concern ourselves with the error at step n $e_n = x_* - x_n$. The conjugate gradient method can be described as a system of recurrence formulas that generates the unique sequence of iterates $\{x_n \in \mathcal{K}_n\}$ with the property that at step n, $\|e_n\|_A$ is minimized. The algorithm, as written in [1] is as follows,

Algorithm 1 Conjugate Gradient Method

x_0 = initial guess
$r_0 = b - Ax_0$
$s_0 = r_0$
for $k = 0, 1, 2, \ldots$ **do**
    $\alpha_k = r_k^T r_k / s_k^T A s_k$
    $x_{k+1} = x_k + \alpha_k s_k$
    $r_{k+1} = r_k - \alpha_k A s_k$
    $\beta_{k+1} = r_{k+1}^T r_{k+1} / r_k^T r_k$
    $s_{k+1} = r_{k+1} + \beta_{k+1} s_k$
end for

From this algorithm, we can deduce that the residuals are orthogonal, and the search directions A-conjugate, written formally as

Theorem 2.1 *Let A be a symmetric positive definite matrix, and apply the CG algorithm to the system $Ax = b$. As long as the iteration has not yet converged, the algorithm proceeds without divisions by zero, and we have the following identities of subspaces:*

$$\begin{aligned} \mathcal{K}_n &= \langle x_1, x_2, \ldots, x_n \rangle = \langle p_0, p_1, \ldots, p_{n-1} \rangle \\ &= \langle r_0, r_1, \ldots, r_{n-1} \rangle = \langle b, Ab, \ldots, A^{n-1}b \rangle \end{aligned} \tag{1}$$

Moreover, the residuals are orthogonal

$$r_n^T r_j = 0 \quad (j < n) \tag{2}$$

and the search directions are A-conjugate,

$$p_n^T A p_j = 0 \quad (j < n) \tag{3}$$

From this, we can also confirm that the conjugate gradient algorithm minimizes the error $\|e_n\|_A$ at each step.

Theorem 2.2 *Let A be a symmetric positive definite matrix, and apply the CG algorithm to the system $Ax = b$. If the iteration has not already converged, then x_n is the unique $\mathcal{K}_n$ that minimizes $\|e_n\|_A$. The convergence is monotonic*

$$\|e_n\|_A \leq \|e_{n-1}\|_A , \tag{4}$$

and $e_n = 0$ is achieved for some $n \leq m$.

I will reiterate the proof in [1]: We know $x_n \in \mathcal{K}_n$ from theorem 2.1, but let's consider an arbitrary point $x = x_n - \Delta x \in \mathcal{K}_n$ with error $e = x_* - x = e_n + \Delta x$. We can see

$$\begin{aligned} \|e\|_A^2 &= (e_n + \Delta x)^T A (e_n + \Delta x) \\ &= e_n^T A e_n + (\Delta x)^T A (\Delta x) + 2 e_n^T A (\Delta x) \end{aligned}$$

The last term in this equation is $2r_n^T(\Delta x)$, which is an inner product of r_n with a vector in $\mathcal{K}_n$, so by Theorem 2.1 we know this is 0. Then, we have

$$\|e\|_A^2 = e_n^T A e_n + (\Delta x)^T A (\Delta x)$$

Since A is positive definite, $(\Delta x)^T A (\Delta x) \geq 0$ and is equal to 0 only when $\Delta x = 0$, ie when $x_n = x$. Therefore, $\|e\|_A^2$ is minimized when $x_n = x$, as desired. The monotonicity property (4) is a consequence of the inclusion $\mathcal{K}_n \subseteq \mathcal{K}_{n+1}$, and since $\mathcal{K}_n \subset \mathbb{R}^m$ of dimension n, as long as it hasn't converged it must in at most m steps.

2.1 Preconditioning

We can try to enhance the efficiency of the CG algorithm through preconditioning, which involves adding an additional M operator to our formulation and solving an equivalent problem.

The original conjugate gradient problem can be though of as minimizing $\frac{1}{2}x^TAx - x^Tb$, but with preconditioning it can be thought of as minimizing $\frac{1}{2}(M^{1/2}y)^TAM^{1/2}y - (M^{1/2}y)^Tb$ ([2], [3]). Choosing the right preconditioner depends on the tradeoff between convergence rate and cost per iteration. There are a plethora of preconditioners to choose from, some of the commonly used types are:

- **Jacobi**: M is a diagonal matrix with entries equal to those from A
- **Block Diagonal or Block Jacobi**: Partition the indices $1, \ldots, n$ into disjoint sets. If i and j are in the same subset, then $M(i,j) = A(i,j)$. If not, we set $M(i,j) = 0$. How we partition the sets is a choice: along lines or planes in two or three dimensional grids, respectively, or grouping together physical variables that correspond to a common node.
- **SSOR**: Using a matrix splitting of the form $A = L + D + L^T$, we can take $M = (D + L)D^{-1}(D + L)^{-1}$. If we introduce the SSOR relaxation parameter ω

$$M(\omega) = \frac{1}{2-\omega}\left(\frac{1}{\omega}D + L\right)\left(\frac{1}{\omega}D\right)^{-1}\left(\frac{1}{\omega}D + L\right)^T$$

  With the optimal choice of ω, the SSOR preconditioner is capable of reducing the condition number to $\text{cond}(M^{-1}A) = \mathbb{O}(\sqrt{\text{cond}(A)})$.
- **Incomplete Factorization**: Ideally, we would solve the linear system using the Cholesky factorization $A = LL^T$, but that may not be possible. Instead, we can use an approximate factorization $A \approx \hat{L}\hat{L}^T$, then we can use $M = \hat{L}\hat{L}^T$ as a preconditioner.
- **Polynomial**: M^{-1} is taken to be a polynomial in A that approximates A^{-1}.
- **Approximate Inverse**: M^{-1} is determined by using an optimization algorithm to minimize the residual $\|I - AM^{-1}\|$ or $\|I - M^{-1}A\|$ in some norm, with M^{-1} restricted to have a prescribed pattern of nonzero entries.

Once a preconditioner M is chosen, we can incorporate it into our algorithm without many changes, as we can see in Algorithm 2 from [4].

3 Generalised Minimal Residual Method (GMRES)

Let $A \in \mathbb{C}^{m\times m}$ be a square, nonsingular matrix and $b \in \mathbb{C}^m$ a vector. Then, let $\mathcal{K}_n$ denote the Krylov subspace $span\{b, Ab, \ldots, A^{n-1}b\}$. Our goal is to solve the system $Ax = b$, and we can denote the exact solution as $x_* = A^{-1}b$. The general idea of GMRES is that on iteration n we approximate x_* by the vector $x_n \in \mathcal{K}_n$ which minimizes the norm of the residual $r_n = b - Ax_n$. Basically, we determine x_n by solving a least squares problem.

Algorithm 2 Preconditioned Conjugate Gradient

$x_0 =$ initial guess
$r_0 = b - Ax_0$
$s_0 = M^{-1}r_0$
for $k = 0, 1, 2, \ldots$ **do**
    $\alpha_k = r_k^T M^{-1} r_k / s_k^T A s_k$
    $x_{k+1} = x_k + \alpha_k s_k$
    $r_{k+1} = r_k - \alpha A s_k$
    $\beta_{k+1} = r_{k+1}^T M^{-1} r_{k+1} / r_k^T M^{-1} r_k$
    $s_{k+1} = M^{-1} r_{k+1} + \beta_{k+1} s_k$
end for

3.1 Arnoldi Iteration

The Arnoldi iteration is the analogue of Gram-Schmidt for similarity transformations to Hessenberg form rather than QR factorization. Since we are focusing on iterative methods, computing the full reduction to Hessenberg form ($A = QHQ^*$) is unlikely, so we will focus on the first n columns of $AQ = HQ$ instead. This would mean we have $Q_n \in \mathbb{C}^{m \times n}$, and some $\tilde{H}_n$, the $(n+1) \times n$ upper-left section of H, which is also a Hessenberg matrix. So we have $AQ_n = Q_{n+1}\tilde{H}_n$, which looks like:

$$\begin{bmatrix} & & \\ & A & \\ & & \end{bmatrix} \begin{bmatrix} q_1 & \cdots & q_n \\ & & \end{bmatrix} = \begin{bmatrix} q_1 & \cdots & q_{n+1} \\ & & \end{bmatrix} \begin{bmatrix} h_{11} & \cdots & h_{1n} \\ & \ddots & \\ & & h_{n+1,n} \end{bmatrix}$$

So, the nth column of this is written $Aq_n = h_{1n}q_1 + \cdots + h_{nn}q_n + h_{n+1,n}q_{n+1}$. We can succinctly write the Arnoldi iteration algorithm as in [1].

Algorithm 3 Arnoldi Iteration

$b =$ arbitrary
$q_1 = b / \|b\|$
for $n = 1, 2, 3, \ldots$ **do**
    $v = Aq_n$
    for $j = 1$ to n **do**
        $h_{jn} = q_j^* v$
        $v = v - h_{jn} q_j$
    end for
    $h_{n+1,n} = \|v\|$
    $q_{n+1} = v / h_{n+1,n}$
end for

3.2 The GMRES Algorithm

We use the Arnoldi iteration to construct a sequence of Krylov matrices Q_n whose columns span successive Krylov subspaces $\mathcal{K}_n$, so we can write $x_n = Q_n y$. Now, the least squares problem to

solve is

$$\|AQ_n y - b\| = \text{minimum} \tag{5}$$

But we know from our Arnoldi iteration, that we have $AQ_n = Q_{n+1}\tilde{H}_n$, so our least squares problem will become

$$\left\|Q_{n+1}\tilde{H}_n y - b\right\| = \text{minimum} \tag{6}$$

Now, since both vectors in the norm are in the column space of Q_{n+1}, we multiply on the left by Q^*_{n+1}

$$\left\|\tilde{H}_n y - Q^*_{n+1} b\right\| = \text{minimum} \tag{7}$$

Since we constructed Krylov subspaces, we know $Q^*_{n+1} b = \|b\| e_1$ and we have the final form of our least squares problem:

$$\left\|\tilde{H}_n y - \|b\| e_1\right\| = \text{minimum} \tag{8}$$

Now, with an intuitive understanding of the GMRES algorithm we can write it in Algorithm 4, which is an amalgamation of the algorithms written in [5], [6], and [1].

Algorithm 4 GMRES

$r_0 = b - Ax_0$
$q_1 = r_0 / \|r_0\|$
for $j = 1, \ldots, m$ **do**
    $q_{j+1} = A * q_j$
    for $i = 1$ to j **do**
        $h_{ij} = q_i^T q_{j+1}$
        $q_{j+1} = q_{j+1} - h_{ij} * q_i$
    end for
    $q_{j+1} = q_{j+1} / \left\|q_{j+1}\right\|$ (essentially this has just been Arnoldi)
    find y to minimize $\left\|h_{j+2,j+1} y - \|b\| e_1\right\|$
    r = $\left\|h_{j+2,j+1} y - \|b\| e_1\right\|$
    if $r <$ TOL **then**
        return $q_{j+1} y + x_0$
    end if
end for
return $q_{j+1} y + x_0$

All that remains is to choose how to perform the least squares, which I will discuss in section 4 as a part of my implementation.

4 Implementation

I implemented both the standard Conjugate Gradient method and GMRES in Matlab. The full code was submitted with this document, but the major functions appear in this section.

In order to see that my implementations are performing as expected, I first chose an example problem to test them on. I chose the 2D Laplace Problem, because I imagine that one of the real-world application of these methods is solving partial differential equations like the Laplace problem.

4.1 2D Laplace Problem

We can define the 2D Laplace Problem

$$\Delta u = u_{xx} + u_{yy}$$

Then using centered finite differences, we see

$$u_{xx} = \frac{u(x-h,y) - 2u(x,y) + u(x+h,y)}{h^2} + O(h^2)$$

$$u_{yy} = \frac{u(x,y-h) - 2u(x,y) + u(x,y+h)}{h^2} + O(h^2)$$

$$\Delta u = \frac{u(x-h,y) + u(x,y-h) - 4u(x,y) + u(x+h,y) + u(x,y+h)}{h^2} + O(h^2)$$

Then, we can define a mesh

$$x_i = ih \text{ for } i = 1, \ldots, m+2 \quad y_j = jh \text{ for } j = 1, \ldots, n+2$$

and our finite difference matrix for the Laplacian becomes

$$D_{ij} = \frac{u(x_{i-1},y) + u(x,y_{i-1}) - 4u(x_i,y_i) + u(x_{i+1},y_j) + u(x_i,y_{i+1})}{h^2}$$

Then, we can code this (inefficiently) in Matlab in the following way

```
function [A] = build_A(n)
    h = 1/(n-1);
    de = 1/h^2;

    n2 = n-2;

    d1 = ones(1, n2*n2)*4*de;
    d2 = ones(1,n2*n2 - 1)*-1*de;
    d3 = ones(1,n2*(n2-1))*-1*de;

    for i=1:(n2-1)
        d2((i)*n2) = 0;
    end

    A1 = diag(d1,0);
    A2 = diag(d2, 1);
    A3 = diag(d2, -1);
    A4 = diag(d3, n2);
    A5 = diag(d3, -n2);

    A = A1 + A2 + A3 + A4 + A5;
end
```

This now allows us to make matrix problems of the form $AU = f$ for A the differentiation matrix for the Laplacian.

4.2 Conjugate Gradient

The translation from Algorithm 1 is a near one-to-one, and I implemented the algorithm as follows:

```
function [x,iters] = conjugate_gradient(A, b, x_init, TOL)
    iters = 0;
    r = b - A*x_init; %initial residual
    d = r; %initial direction to step in

    x = x_init;

    while (dot(r',r) > TOL)
        iters = iters+1;
        r_old = r;
        w = A*d;
        alpha = dot(r',r) / dot(d', w); %step length

        x = x + alpha*d; %new solution

        r = r - (alpha * w); %new residual

        beta = dot(r',r) / dot(r_old', r_old); %improvement this step

        d = r + (beta * d); %new search direction
    end
end
```

To test our example problem, we can choose our right hand side vector, b, to be a vector of ones and our initial guess, x_0, to be zero. I chose to examine a few different sizes of A, namely $n = 10, 32, 128$. I would have liked to go further, but when I tried to use 256, Matlab threw an error because I was attempting to make my A matrix 64516×64516, which would take too much storage in its current implementation.

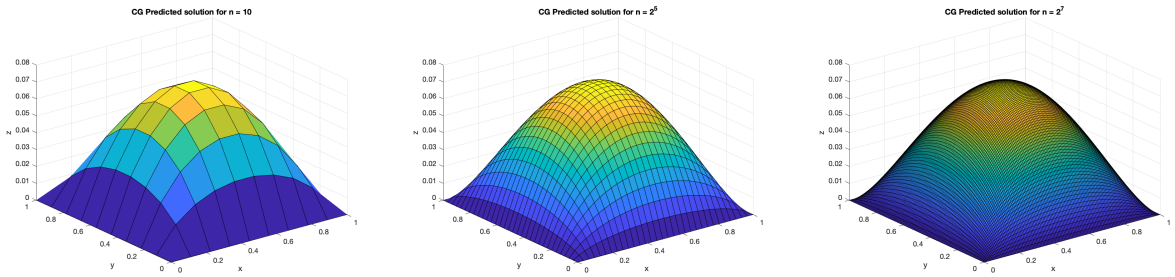

Figure 1: For $n = 10$, converged after 9 iterations and 0.046 seconds. For $n = 32$, converged after 38 iterations and 0.02 seconds. For $n = 128$, converged after 172 iterations and 9.204 seconds.

4.3 GMRES

Translating Algorithm 4 to Matlab was more complicated than for CG. Before I could translate the algorithm to Matlab code, I had to chose what method to perform my minimization. At first, I thought to use Matlab's built in least squares function, but upon further reading, [6] makes mention of using Givens rotations instead. So, I implemented the Arnoldi iteration, Givens rotations, and ultimately the GMRES algorithm as follows:

```
%iterate iters time or until the residual is below TOL
function [h,q] = arnoldi(A,Q,k)
    q = A*Q(:,k);
    for i = 1:k
        h(i) = q' * Q(:,i);
        q = q - h(i) * Q(:,i);
    end
    h(k+1) = norm(q);
    q = q/h(k+1);
end

function [h,cs,sn] = Givens_rotation(h, cs, sn, j)
    for i = 1:j-1
        gamma = cs(i)*h(i) + sn(i)*h(i + 1);
        h(i+1) = -sn(i)*h(i) + cs(i)*h(i+1);
        h(i) = gamma;
    end
    delta = sqrt(h(j)^2 + h(j+1)^2);
    sn = h(j+1) / delta;
    cs = h(j) / delta;

    h(j) = cs*h(j) + sn*h(j+1);
    h(j+1) = 0;
end

function [x] = my_gmres(A, b, x, iters, TOL)
  sn = zeros(iters, 1);
  cs = zeros(iters, 1);
  e1 = zeros(iters+1, 1);
  e1(1) = 1;

  r = b - A * x;
  beta = norm(r);
  Q(:,1) = r / beta;

  g = norm(r) * e1;
  for k = 1:iters
    % Arnoldi Iteration
    [H(1:k+1, k), Q(:, k+1)] = arnoldi(A, Q, k);

    % Givens Rotation
    [H(1:k+1, k), cs(k), sn(k)] = Givens_rotation(H(1:k+1,k), cs, sn, k);

    g(k+1) = -sn(k)*g(k);
    g(k) = cs(k)*g(k);

    if (norm(g) < TOL)
      break;
    end
  end
  % if we aren't below the tolerance, return what we have
  y = H(1:k, 1:k) \ g(1:k);
  x = x + Q(:, 1:k) * y;
end
```

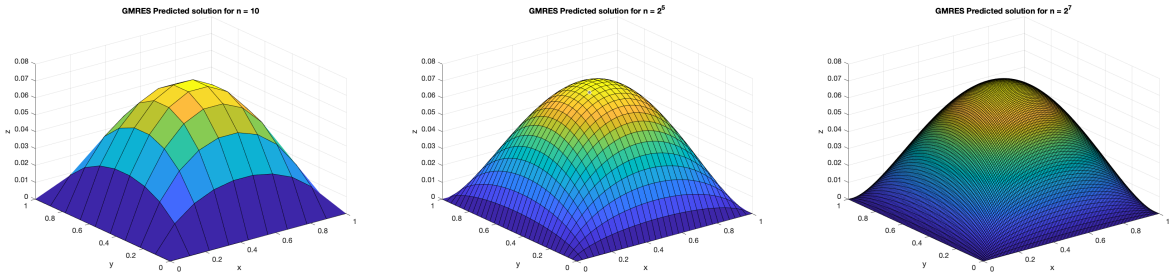

Figure 2: For all n's the algorithm maxed out at 500 iterations. For $n = 10$ this took 0.131 seconds; for $n = 32$, 0.396 seconds; and for $n = 128$, 29.972 seconds.

I performed the same three tests using the 2D Laplacian with this algorithm. Numerically, the results are extremely similar (identical up to some small ϵ) and using the eyeball norm, they seem identical.

5 Conclusion and Future Work

Iterative methods are a powerful tool in numerical linear algebra for solving systems with specific properties we can exploit. Krylov subspace methods are a particular family of iterative methods that are based around the formation of a Krylov space $\mathcal{K}_n = \langle b, Ab, \ldots, A^{n-1}b \rangle$. I discussed the mathematics behind and implemented two Krylov subspace methods, the Conjugate Gradient (CG) method, and the Generalised Minimal Residual (GMRES) Method.

I would like to explore why my GMRES algorithm was reaching its maximum iterations every run, even though the solutions seem to be correct (I numerically examined the solutions produced by both algorithms). I would also like to implement these using sparse matrices to be able to test with even higher values of n. Another avenue for further study is using preconditioners for the algorithms, and discussing the tradeoff between different preconditioner matrices.

References

[1] L. N. Trefethen, D. Bau, Numerical linear algebra, Society for Industrial and Applied Mathematics, 1997.

[2] G. H. Golub, D. P. OÄôLeary, Some history of the conjugate gradient and lanczos algorithms: 1948Äì1976, SIAM Review 31 (1) (1989) 50–102. arXiv:https://doi.org/10.1137/1031003, doi:10.1137/1031003.
URL https://doi.org/10.1137/1031003

[3] J. L. Nazareth, Conjugate gradient method, WIREs Computational Statistics 1 (3) (2009) 348–353. arXiv:https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.13, doi:https://doi.org/10.1002/wics.13.
URL https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.13

[4] M. T. Heath, Scientific Computing, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2018. arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611975581, doi:10.1137/1.9781611975581.
URL https://epubs.siam.org/doi/abs/10.1137/1.9781611975581

[5] Y. Saad, M. H. Schultz, Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing 7 (3) (1986) 856–869. arXiv:https://doi.org/10.1137/0907058, doi:10.1137/0907058.
URL https://doi.org/10.1137/0907058

[6] Q. Zou, Gmres algorithms over 35 years, Applied Mathematics and Computation 445 (2023) 127869. doi:https://doi.org/10.1016/j.amc.2023.127869.
URL https://www.sciencedirect.com/science/article/pii/S0096300323000383