UNIVERSITY OF COLORADO BOULDER
DEPARTMENT OF APPLIED MATHEMATICS

Steepest Descent vs the Conjugate Gradient Method: An Adventure in Linear System Solvers

Lauren Marsh

Serena DiLeonardo

University of Colorado **Boulder**

Spring 2021

Abstract:

In this paper, solvers for linear systems will be investigated. The main goals of this project are to:

1. Derive conjugate gradient method
2. Compare conjugate gradient (CG) vs steepest descent for a collection of problems for symmetric matrices
3. Describe and compare the performance of two alternatives to the CG, specifically Biconjugate Gradient (BiCG) and Generalized Minimal Residual Method (GMRES)
4. Test numerical examples to explore cases where these methods are most and least efficient, as well as the cases where they fail

These goals are approached with various techniques from numerical analysis. Our comparisons will delve into the number of iterations and time required to solve a linear system, as well as their sensitivity to changes in input. Through this analysis, we can come to conclusions as to which methods perform best for different types of linear systems.

1. Introduction and Background

The method of conjugate gradients was developed by Magnus Hestenes and Eduard Stiefel, published in 1952 as an iterative method for solving sparse systems of linear equations [1]. This computationally powerful iterative method is able to solve an N-dimensional symmetric positive definite system of linear equations in exactly N steps, and is used across fields of optimization, machine learning, and image processing [2]. The derivation however is not immensely straightforward, as numerous approaches with varying notation can be found in different research papers and textbooks. All derivations do however rely on the same crux – the orthogonality of the residuals and conjugacy of the search directions [3].

1.1. Krylov Subspaces

Krylov subspace methods consist of algorithms which solve problems such as linear systems, eigenvalue problems, and matrix equations. Generally, Krylov subspaces consist of the iterative schemes whose mth iterate x_m satisfies:

$$x_m \in x_0 + K_m(A, r_0)$$

for $k = 1, 2, 3, ...$ and $K_m(A, r_0) = span\{r_0, Ar_0, ..., A^{m-1}r_0\}$ denotes the mth Krylov subspace generated by matrix A and residual r_0 [4]. Krylov subspaces have a few interesting properties:

$$K_r(A, b), AK_r(A, b) \in K_{r+1}(A, b)$$

Krylov subspaces are used in algorithms for approximating solutions to high-dimensional linear algebra problems largely due to their property that the matrix A is not needed explicitly, just the ability to operate on A and a given vector v [4]. Thus, avoiding matrix-matrix operations, these iterative methods are able to multiply the matrix by vectors and work with the resulting vectors.

2. Derivation of the Conjugate Gradient Method

The method of conjugate gradients (CG) is used to solve a system of linear equations with the general form:

$$Ax = b$$

where A is a symmetric $n \times n$ matrix such that $A^T = A$, and x (unknown) and b (known) are $n \times 1$ column vectors.

2.1. Orthogonality and conjugate vectors

Let's take a moment to review what it means for two vectors to be conjugate. By definition, vectors v and w are conjugate with respect to matrix A if:

$$v^T A w = 0$$

Note that since A is assumed to be symmetric and positive definite, this is equivalent to the inner product with respect to A:

$$< v, w >_A = < Av, w > = < v, A^T w > = < v, Aw >$$

Thus v and w are mutually conjugate with respect to A if and only if they are orthogonal with respect to this inner product. Note that vectors being conjugate with respect to A may also be called $A-$orthogonal [5]. Since CG is actually a special case of the method of Conjugate Directions, we will first explore this a bit more in depth.

2.2. The Method of Conjugate Directions

A conjugate direction method generates conjugate directions applied to a positive definite matrix. Simply, modifications are made on steepest descent such that directions are conjugate to previous search directions, thereby in a way keeping track of previous directions. The $Q-$conjugacy of search directions and the error term is equivalent to minimizing $||e||_Q$ along the search direction, where $e_i = e_0 + \sum_{j=0}^{i-1} \alpha_j d_j$. The next step minimizes $||e||_Q$ along the next search direction, while remaining $Q-$orthogonal to all previous search directions. The residual r_i indicates how far our approximation is from the correct value of b, i.e. $r_i = b - Qx_i$, and can be interpreted as the direction of steepest descent. Since $r_i = -Qe_i$, the residual remains orthogonal to all old search directions as well, i.e. a new linearly independent search direction is produced at each step unless the residual is zero, which would imply the problem is already solved. [1, 2]

Consider the following set of $Q-$orthogonal direction vectors:

$$\{d_0, d_1, ..., d_{n-1}\}$$

where Q is a positive definite matrix size $n \times n$. This set is linearly independent, since Q is positive definite implying that $d_i^T Q d_i > 0$. Note that this is true since for some α_i,

$$\alpha_0 d_0 + ... + \alpha_k d_k = 0 \implies \alpha_i = 0$$

The overarching goal is to solve the quadratic minimization problem

$$\min_x \frac{1}{2} x^T Q x - b^T x$$

for x^* where $Qx^* = b$. This solution can be written as a linear combination of the set $\{d_0, d_1, ..., d_{n-1}\}$ as:

$$x^* = \alpha_0 d_0 + \alpha_1 d_1 + ... + \alpha_{n-1} d_{n-1}$$

We can compute the terms $\{\alpha_0, \alpha_1, ..., \alpha_{n-1}\}$ with the following relation:

$$\alpha_i = \frac{d_i^T Q x^*}{d_i^T Q d_i} = \frac{d_i^T b}{d_i^T Q d_i}$$

Note that α_i is expressed in terms of the known vector b, rather than the unknown vector solution x^*. Then, the solution x^* can be expressed as:

$$x^* = \sum_{i=0}^{n-1} \frac{d_i^T b}{d_i^T Q d_i} d_i$$

The conjugate direction method makes use of these properties, and applies an iterative technique to find a unique solution x^* to $Qx = b$. The residual r_i indicates how far our approximation is from the correct value of b, i.e. $r_i = b - Qx_i$, and can be interpreted as the direction of steepest descent. Taking a set of nonzero $Q-$orthogonal vectors $\{d_0, d_1, ..., d_{n-1}\}$, we can construct an iterative scheme:

$$x_{k+1} = x_k + \alpha_k d_k$$

where $\alpha_k = -\frac{g_k^T d_k}{d_k^T Q d_k}$ so that $g_k = Qx_k - b$. This scheme will converge to the solution of $Qx = b$, x^*, in exactly n steps $\implies x_n = x^*$. [1, 2]

2.3. The Method of Conjugate Gradients

At each step of CG, a new search direction is chosen such that it is $Q-$orthogonal to all previous search directions, as is the case for conjugate direction methods. The addition is that the direction is updated at each step. In this procedure, each new residual is orthogonal to all previous residuals and search directions, and each new search direction is constructed from the residual to be $Q-$orthogonal to all previous residuals and search directions.

For some starting point x_0, we set $d_0 = -g_0 = b - Qx_0$. The iterative procedure is similar to that of conjugate directions:

$$x_{k+1} = x_k + \alpha_k d_k$$

where $\alpha_k = -\frac{g_k^T d_k}{d_k^T Q d_k}$ and $g_k = Qx_k - b$, as before. The following step is where CG differs:

$$g_{k+1} = -g_{k+1} + \beta_k d_k \qquad d_{k+1} = -g_{k+1} + \beta_k d_k$$

which implies $\beta_k = \frac{g_{k+1}^T Q d_k}{d_k^T Q d_k}$. Iterations continue until a desired size of residual error is attained. At each step, CG requires one matrix-vector product and two inner products. The rate of convergence for the CG algorithm depends on the eigenvalue distribution, i.e. if A has tightly clustered eigenvalues which are bounded away from zero, then the literature states that CG will converge quickly [1].

3. Alternative Linear Solvers

3.1. Steepest Descent

The basic idea of steepest descent (SD) is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point (i.e. the direction of steepest descent) until a predefined residual value is reached. As a refresher, error is a vector that indicates how far we are from the solution x, while the residual indicates how far we are from the correct value of b. Thus steepest descent works by choosing point after point that is closer to the final solution using the direction of steepest descent (i.e. the negative of the gradient) [5]. The general implementation is as follows:

1. Pick a starting point x_0
2. Move in the direction of steepest descent
3. Calculate the gradient
4. Set the residual as the negative gradient i.e. steepest downward slope
5. Move in the opposite direction
6. Continue until a specified tolerance is met

In order to find the direction of steepest descent, this is approached as a minimization problem. For the equation:

$$f(x) = \frac{1}{2}x^T Ax - bx + c$$

we have the gradient

$$\nabla f(x) = \frac{1}{2}(A + A^T)x - b = Ax - b$$

by symmetry of A. Thus the negative gradient of the next iterate is set as the residual:

$$-\nabla f(x_{i+1}) = b - Ax_{i+1} = r_{i+1}$$

with the ith error $e_i = x_i - x$ and residual $r_i = b - Ax_i = -Ae_i$. The next iterate is given by:

$$x_{i+1} = x_i + \frac{r_i^T r_i}{r_i^T Ar_i} r_i$$

which ensures orthogonality to the previous iterate [5]. As mentioned, where CG differs primarily is here - CG uses *all* previous directions to build the next direction, making the next search direction A-orthogonal (i.e. conjugate) to all previous directions.

3.2. Biconjugate Gradient

The previous methods (SD and CG) can only be used for SPD matrices. Ideally, there would also exist a method that worked similarly but could be used on non-symmetric systems. This is where biconjugate gradient (BiCG) comes in!

The way BiCG works is by generating *two* CG-like sequences of vectors, one based on a system with the original coefficient matrix A, and one on A^T. Instead of orthogonalizing each sequence, they are made mutually orthogonal, i.e. bi-orthogonal. The price of replacing the original orthogonal sequence of residuals by two mutually orthogonal sequences is that the process no longer providing a minimization. This can lead to extremely erratic convergence behavior, as the minimization of residuals is not guaranteed [6].

In other words, BiCG simultaneously solves $Ax = b$ and $A^T x = b$. This provides two update sequences of residuals:

$$r_i = r_{i-1} - \alpha_i A p_i$$

$$\hat{r}_i = \hat{r}_{i-1} - \alpha_i A^T \hat{p}_i$$

and two search directions:

$$p_i = r_{i-1} + \beta_{i-1} p_{i-1}$$

$$\hat{p}_i = \hat{r}_{i-1} + \beta_{i-1} \hat{p}_{i-1}$$

with

$$\alpha_i = \frac{\hat{r}_{i-1}^T r_{i-1}}{\hat{p}_i^T}, \beta_i = \frac{\hat{r}_i^T r_i}{\hat{r}_{i-1}^T r_{i-1}}$$

If it is the case where matrix A is symmetric, then $r_i = \hat{r}_i$, $p_i = \hat{p}_i$, and CG would produce the same result at half the computational cost.

3.3. Generalized Minimum Residual Method

As with BiCG, the generalized minimum residual method (GMRES) can be used for matrices that are not SPD. It also happens to be that, like BiCG is a generalized method of CG, GMRES is a generalized method of MINRES, i.e. the minimum residual method. MINRES is used to solve linear systems for symmetric but not necessarily positive definite matrices. Both MINRES and GMRES work by minimizing the 2-norm of the residual by solving a least squares problem, where GMRES is for non-symmetric systems and MINRES uses a 3-term recurrence relation similar to CG [5]. In other words, the 2-norm of the residual $r_m = b - Ax_m$ at step m satisfies:

$$||r_m||_2 \leq ||b - Az||_2 \forall z \in x_0 + K_m(A, r_0)$$

Both of these methods are projection methods, like all Krylov subspace methods. The original problem is projected onto a [Krylov] subspace of lower dimension in which the solution of the residual minimization problem can be easily found by solving either a linear system or a least squares problem, and the least squares problem can be efficiently solved via QR factorization. For numerical stability reasons, it is best to project onto a subspace if an orthonormal basis for the subspace is known. In order to construct an orthonormal basis for the Krylov subspace, the Gram-Schmidt process is efficiently implemented. For MINRES, this implementation is referred to as the Lanczos process, and for GMRES it is referred to the Arnoldi process [5]. Note that if $A = A^*$, then the upper Hessenberg matrix $H_m = H_m^* = T_m$ is a tridiagonal matrix and the Arnoldi process becomes the Lanczos process (with cheaper storage). Successive residuals of GMRES are still orthogonal, however the complexity and required storage space increases linearly with each iteration. A common fix to this storage issue is to incorporate restarts into the algorithm, i.e. after $m < n$ steps taking that result and setting it as the initial guess of the algorithm, and starting over. This is commonly referred to as GMRES(m).

4. Numerical Examples

In order to analyze and compare the performance of Conjugate Gradient to other methods, we developed a problem set that demonstrated both the strengths and weaknesses of the method. Each problem involved solving the system $Ax = b$ where the initial guess x_0 was set to be the zero vector and b was set as an appropriately sized vector of ones. These were kept consistent across examples in order to reduce the variance across examples. The tolerance used for the methods was set as 1×10^{-6}, corresponding to the norm of the residual, and the maximum number of iterations ($Nmax$) was set at 100. While A was different in each test case, it always maintained the property of being SPD.

4.1. Ill-conditioned Matrices (Hilbert Systems)

To analyze the worst case behavior regarding numerical instability, we used a system where A was a Hilbert matrix. Hilbert matrices are symmetric and positive definite, with elements defined by $H_{ij} = \frac{1}{i+j-1}$ for $i, j = 1, 2, 3, ..., n$. They are also canonical examples of ill-conditioned matrices, meaning that they have large condition numbers and are nearly singular. We chose to experiment with Hilbert matrices for these reasons as well as for their common use as test cases for numerical methods, as they are a classic example for demonstrating round-off error difficulties.

Figure 1: Norm of the residual per iteration for solving a 5×5 Hilbert system

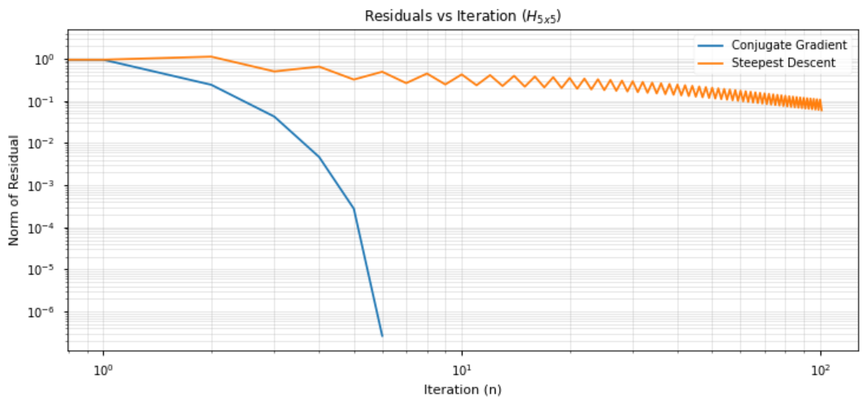
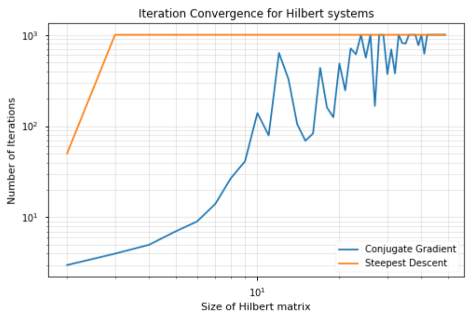
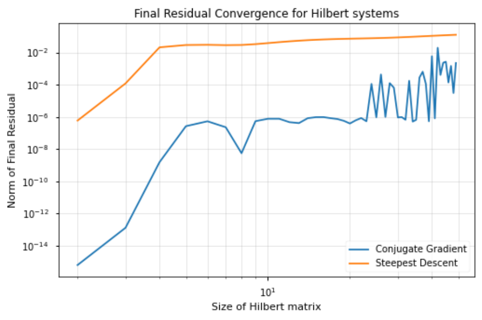


Figure 2: Convergence Analysis for ill-conditioned systems (Hilbert systems)



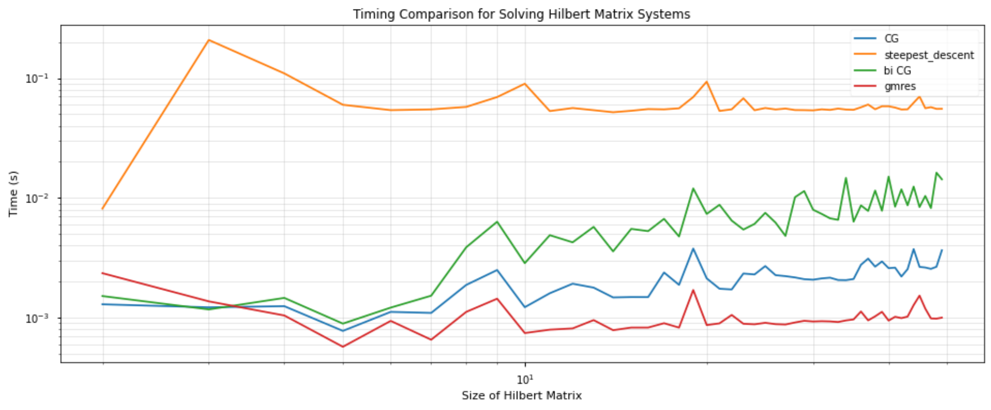
(a) Total number of iterations required to solve $m \times m$ Hilbert system



(b) Norm of the final residual after solving $m \times m$ Hilbert system

The behavior expected from this test was that steepest descent would converge at a slower rate than CG. This is demonstrated in the plots from both **Figure 1** and **Figure 2**. As expected, for every metric we measured CG significantly outperformed steepest descent. In **Figure 2b**, steepest descent reaches the maximum number of iterations even for small Hilbert matrices. CG continues to converge to the tolerance, but **Figure 2a** shows that CG begins to break down and show erratic behavior for Hilbert matrices over 10×10.

Figure 3: Run-time analysis for solving $m \times m$ Hilbert matrix systems (ill-conditioned)



The next metric we investigated was run-time. Steepest descent was still expected to run the slowest, with biCG running twice as long as CG and GMRES running the fastest for general systems. This behavior is reflected in the timing results for the four methods in solving Hilbert systems of various sizes. Steepest descent timing reflects a relatively constant rate in this case due to the fact that the algorithm was reaching maximum iterations every time, meaning it was running the same number of iterations for each increasing size of the Hilbert system.

4.2. Moderately Ill-conditioned Systems (Pascal Systems)

Following the Hilbert testing for CG, we performed tests for Pascal systems, which are good examples of moderately ill-conditioned matrices where the binomial coefficients make up its elements. We aimed to investigate whether CG broke down at the same places that it did for Hilbert systems, and whether steepest descent would perform any better.

Figure 4: Norm of the residual per iteration for solving a 5×5 Pascal system

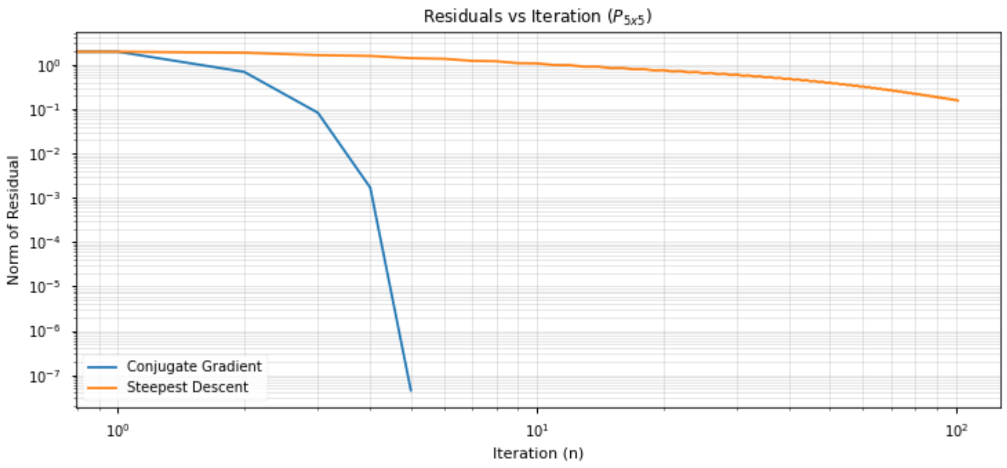
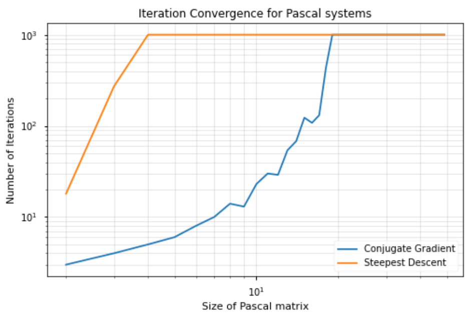
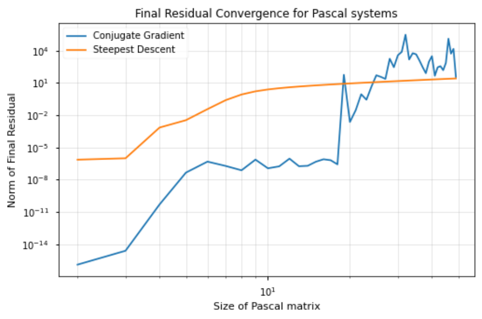


Figure 5: Convergence Analysis for moderately ill-conditioned systems (Pascal systems)



(a) Total number of iterations required to solve $m \times m$ Pascal system



(b) Norm of the final residual after solving $m \times m$ Pascal system

The behavior for steepest descent vs CG in solving Pascal systems seemed to match the behavior seen when solving Hilbert systems. Through analyzing the norm of the final residual, steepest descent appears to perform better than CG for larger Pascal systems. However, this behavior can be a result of steepest descent not truly converging to tolerance, but rather reaching the maximum iterations.

4.3. Sparse Matrix Systems

Previous literature and research suggested that CG was used commonly in situations with large sparse matrix systems. Using a matrix from the *SuiteSparse* Matrix Library [7], we were able to test on larger and larger subsets of the test matrix we chose, which all happened to be SPD. This was due to the nature of the construction of the matrix. This allowed us to test the claim of CG's superior performance in solving large spare linear systems when compared with other methods.

Figure 6: Norm of the residual per iteration for solving a single 5×5 Sparse matrix system.

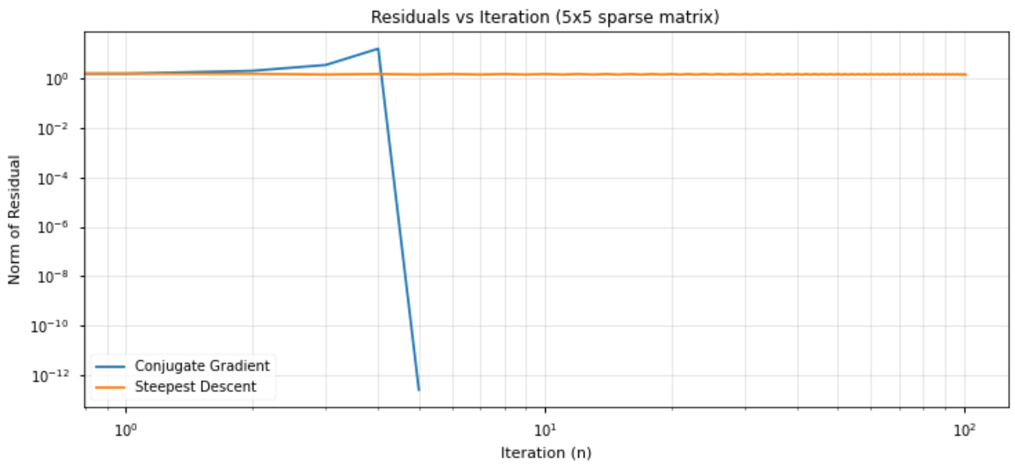
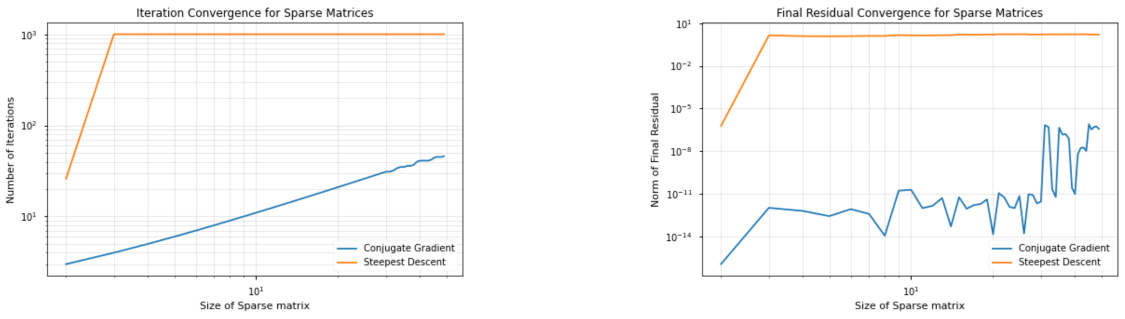


Figure 7: Convergence Analysis for Sparse Matrix Systems

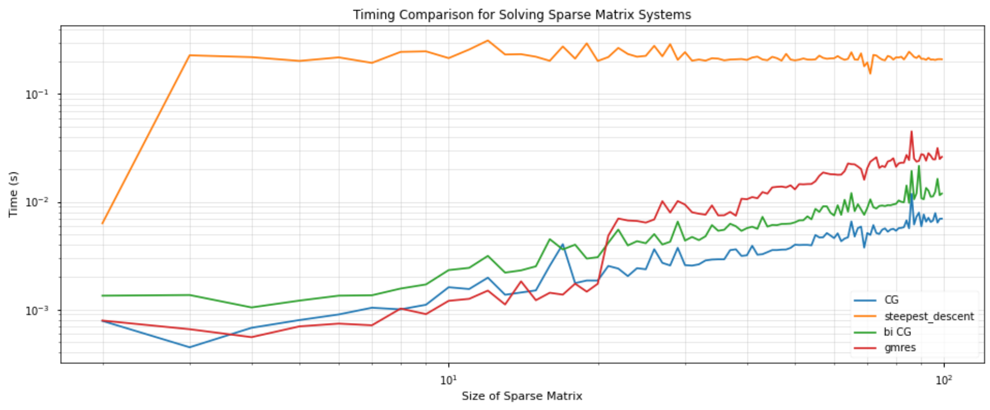


(a) Total number of iterations required to solve $m \times m$ Sparse matrix system.

(b) Norm of the final residual after solving $m \times m$ Sparse matrix system.

It was surprising that steepest descent continued to perform poorly even with the sparse matrix system. In Figures **6** and **7** it appears that the performance of steepest descent did not improve much from the results for ill-conditioned systems, even though working with sparse matrices was expected to improve the performance for any method. Looking just at CG, there is significant improvement in the performance compared to the Hilbert and Pascal systems.

Figure 8: Run-time analysis for solving 10 × 10 Sparse Matrix Systems.



The run-time analysis for all four methods in solving sparse systems showed an interesting result. Steepest descent continued being the slowest, and biCG went consistently twice as slow as CG for each size, which was exactly what we anticipated after reading the literature. However, there was a point after the sparse matrix reached 10 × 10 where GMRES started to perform worse than both CG (and biCG). This supported the claim we found in previous research arguing that CG should be chosen over other methods for large sparse matrix systems when considering run-time.

4.4. Matrix Systems with Clustered Eigenvalues

The final optimized situation for the CG method that we investigated was for solving matrix systems with clustered eigenvalues. To test this case, we created random SPD matrices with a method in which we were able to predetermine the eigenvalues.

The method for creating the random SPD system with predetermined eigenvalues involved starting with a diagonal matrix that contained the desired eigenvalues on the diagonal. Let this matrix be called D_λ with eigenvalues $\lambda_1, \lambda_2, ..., \lambda_m$. The random element of the final matrix is introduced by creating a random orthogonal matrix Q. The final matrix A is constructed as:

$$A = Q^T D_\lambda Q \tag{1}$$

Because Q is an othogonal matrix, A has the same eigenvalues of D_λ. The random orthogonal matrix is created using a built-in library, scipy.stats.ortho_group. [8]

Figure 9: Norm of the residual per iteration for solving a single 10×10 matrix system with clustered eigenvalues

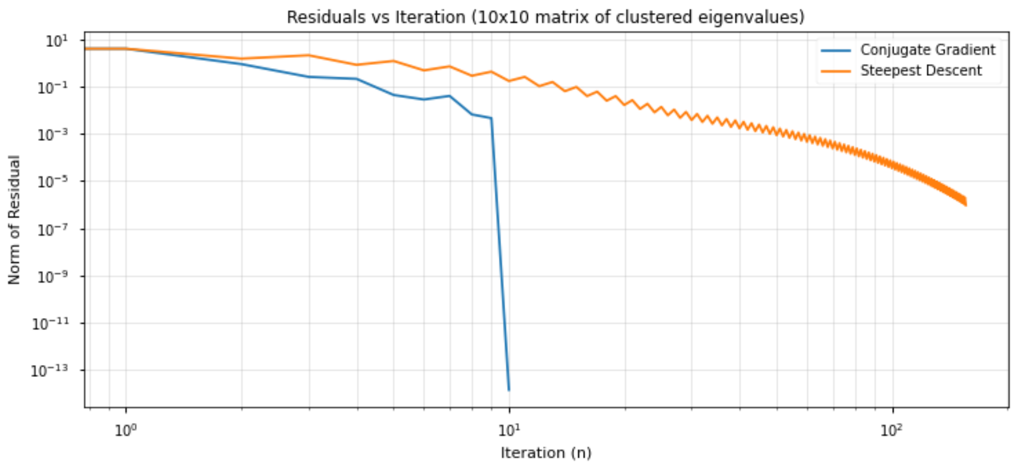
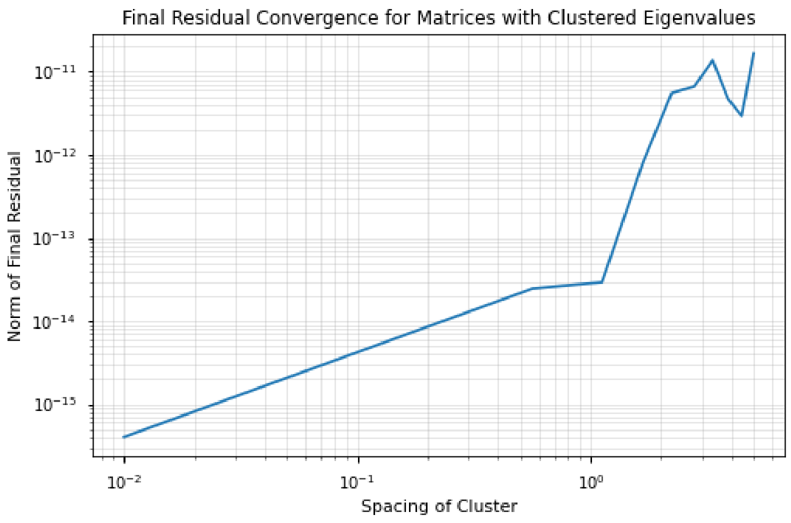


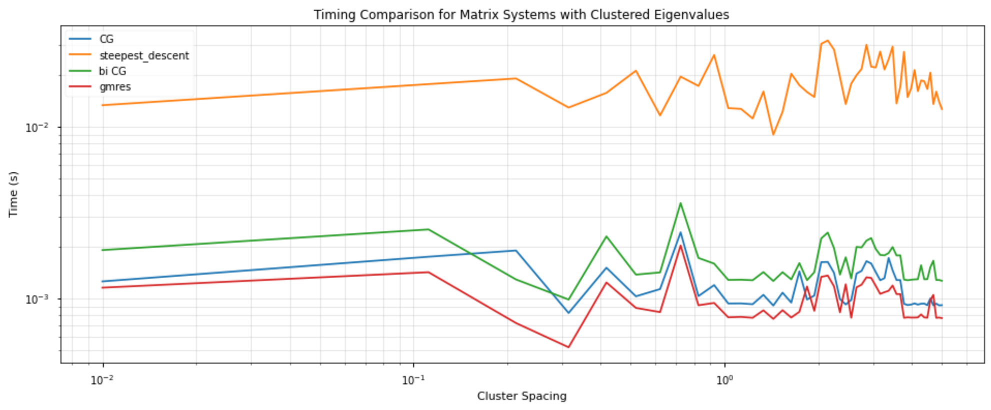
Figure 10: Norm of the final residual of CG after solving a 10×10 matrix system with varying degrees of eigenvalue clustering



In this plot, a value more to the left corresponds to stronger clustering of eigenvalues in our matrix A. Extending the maximum iteration bound for steepest descent allowed it to converge to tolerance with this system. Clustered eigenvalues improved performance for both CG and steepest descent in the 10×10 system in **Figure 9** compared with the Hilbert, Pascal, and Sparse system performances. However, when testing varying degrees of cluster spacing, there was negligible change in the performance for steepest descent. Therefore we concluded that this relative improvement in performance was most likely due to the construction of A rather than its property of clustered eigenvalues.

CG's behavior in **Figure 10** shows that the closer the eigenvalue cluster is, the smaller the norm of the final residual, for cluster spacing less than 1. As the spacing gets larger, the matrix is no longer considered as having the property of clustered eigenvalues and its behavior reflects the results of finding the solution to any arbitrary 10×10 SPD matrix.

Figure 11: Run-time analysis for solving 10×10 matrix systems with varying degrees of eigenvalue clustering



The timing results show that although CG does return a smaller final residual, it's timing is not improved drastically enough for it to be faster than GMRES.

4.5. Sensitivity Analysis

An important investigation was testing CG for sensitivity to variations in input. This testing was performed using the previous test matrices and adding fixed perturbations, then looking at the final residual results.

Figure 12: Plot demonstrating the sensitivity of CG to systematic variations in input, where each matrix entry is perturbed by the same amount

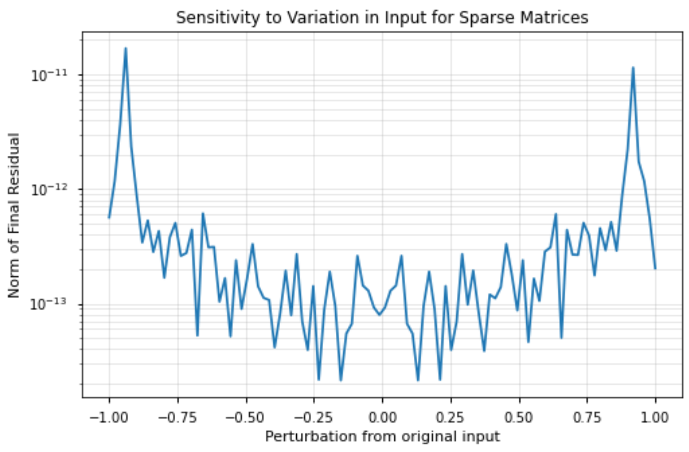


Figure 12 shows the results of testing CG by introducing a simulated systematic error to the matrix A. The norm of the final residual didn't vary drastically in response to perturbations. This shows that CG is not sensitive to variations in input if symmetry is maintained. However, further experiments where we added perturbations to single entries in the matrix which altered the symmetric property did not affect its ability to converge to a solution for larger (10×10) matrices, though larger perturbations resulted in larger residual errors. For small matrices, CG was also not extremely sensitive to perturbations affecting its property of symmetry. This shows that CG, in general, will not break down in response to error in input. Therefore, special care should be taken in using the algorithm, and an understanding of the input and any output error should be accounted for independently.

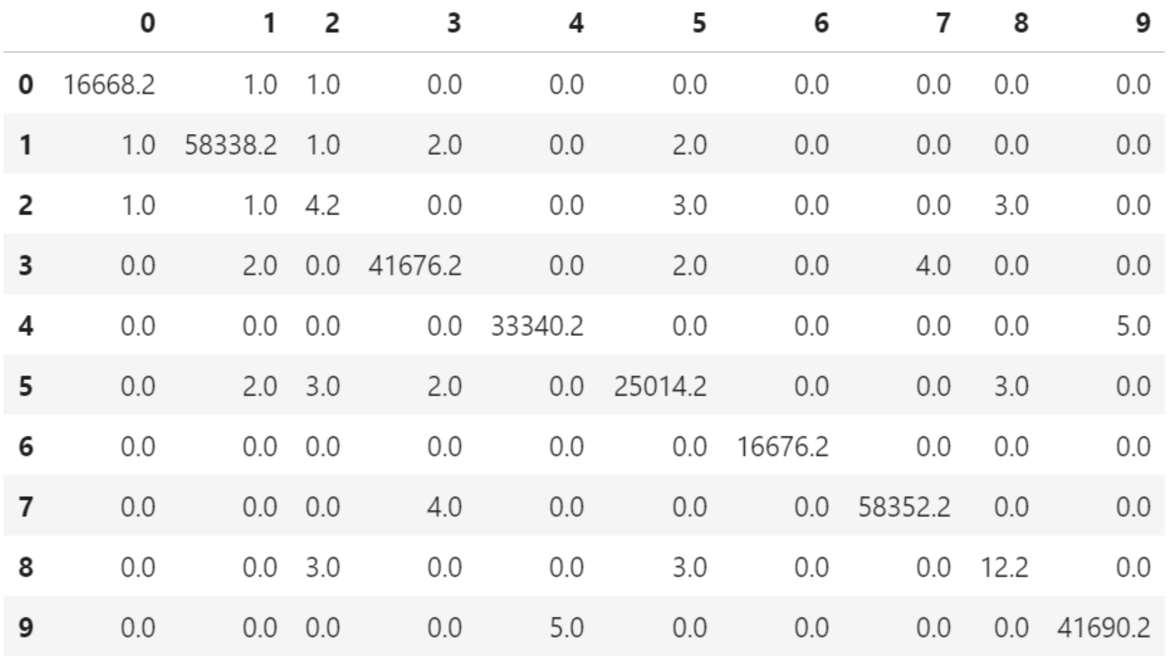
5. Discussion and Conclusion

In practice, these methods are used with preconditioning. This consists of replacement of the original system $Ax = b$ with an equivalent system $\hat{A}\hat{x} = \hat{b}$ for which our desired algorithm would experience better convergence [4]. It is likely that the preconditioner then would involve $\hat{A}$ being well-conditioned or having clustered eigenvalues. Note that only SPD preconditioners could be used with CG and MINRES. It is also worth noting that the application of preconditioning can lead to varying convergence behavior depending on which side the preconditioner is applied to. Specifically, right preconditioning preserves the 2-norm of the residual which is minimized, while left preconditioning minimizes the the preconditioned residual [4].

This project served to investigate linear system solvers including steepest descent, conjugate gradient, biconjugate gradient, and the generalized minimum residual method. Multiple test cases were developed to understand the optimal use case for CG. The methods were tested for ill-conditioned systems (Hilbert systems), moderately ill-conditioned systems (Pascal systems), sparse matrix systems, and matrix systems with clustered eigenvalues. In each case, CG performed better and faster than steepest descent, where steepest descent often struggled to converge within tolerance with the given maximum iteration bound. In all but one case, GMRES worked best with respect to run-time performance. In the case of large sparse matrix systems, CG and biCG's run-time performance was better than GMRES. In general, CG showed an ability to converge within tolerance even when input or symmetry was affected. We conclude that from these results, CG would be the best method for solving large sparse linear systems and GMRES would be the preferred method for other general systems.

Appendices

A. Data Table



	0	1	2	3	4	5	6	7	8	9
0	16668.2	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	1.0	58338.2	1.0	2.0	0.0	2.0	0.0	0.0	0.0	0.0
2	1.0	1.0	4.2	0.0	0.0	3.0	0.0	0.0	3.0	0.0
3	0.0	2.0	0.0	41676.2	0.0	2.0	0.0	4.0	0.0	0.0
4	0.0	0.0	0.0	0.0	33340.2	0.0	0.0	0.0	0.0	5.0
5	0.0	2.0	3.0	2.0	0.0	25014.2	0.0	0.0	3.0	0.0
6	0.0	0.0	0.0	0.0	0.0	0.0	16676.2	0.0	0.0	0.0
7	0.0	0.0	0.0	4.0	0.0	0.0	0.0	58352.2	0.0	0.0
8	0.0	0.0	3.0	0.0	0.0	3.0	0.0	0.0	12.2	0.0
9	0.0	0.0	0.0	0.0	5.0	0.0	0.0	0.0	0.0	41690.2

This matrix is a 10×10 subset from the "barrier Hessian from convex QP (CUTEr)" matrix found on the *SuiteSparse* Matrix Library [7]. The original matrix is $50,000 \times 50,000$ with $349,968$ nonzero values, equating to a relative matrix density of 1.399872×10^{-7}. It is real and SPD.

B. Relevant Code

B.1. Packages Utilized

numpy, numpy.linalg, scipy.sparse, scipy.sparse.linalg, scipy.io, time, matplotlib.pyplot

B.2. Linear Solver Function

The following code details the linear solver function containing the steepest descent and conjugate gradient methods[9], as well as calls to the built in bicg and gmres algorithms from the scipy.sparse.linalg library.

```
def linear_solver(A, b, x0, solvername, tol=1e-6, Nmax=1000):
    # check if A is square
    if not is_square(A):
        print("A␣not␣nxn")
        return

    if solvername == "conjugate_gradient":
        """
        conjugate gradient method
        matrix needs to be: nxn
        """
        # check if A is symmetric positive definite
        if not is_sympd(A):
            print("Matrix␣needs␣to␣be␣symmetric␣positive␣definite.")
            return

        # initialize values
        r = b
        k = 0       # number of iterations
        x = np.zeros((A.shape[1], 1))
        x_steps = [x]
        y_steps = [1/2 * x.transpose() @ A @ x - x.transpose() @ b]
        err = [LA.norm(r)]

        while err[-1] > tol and k <= Nmax:
            if k == 0:
                p = r
            else:
                gamma = -(p.transpose() @ A @ r)/(p.transpose() @ A @ p)
                p = r + gamma * p
            alpha = (p.transpose() @ r) / (p.transpose() @ A @ p)
            x = x + alpha * p
            r = r - alpha * (A @ p)
            k = k+1
            x_steps.append(x)
            y_steps.append(1/2 * x.transpose() @ A @ x - x.transpose() @ b)
            err.append(LA.norm(r))

        if k == Nmax:
            print("Reached␣max␣iterations")
            return x, x_steps, y_steps, err
        return x, x_steps, y_steps, err

    elif solvername == "steepest_descent":
        """
        steepest descent method
        matrix needs to be: nxn
        """
        # check if A is symmetric positive definite
        if not is_sympd(A):
            print("Matrix␣needs␣to␣be␣symmetric␣positive␣definite.")
            return

        # initialize values
        x = x0
        x_steps = [x]
        y_steps = [1/2 * x.transpose() @ A @ x - x.transpose() @ b]
        r = b - A @ x
        k = 0  # number of iterations
        err = [LA.norm(r)]

        while err[-1] > tol and k <= Nmax:
            p = r
            q = A @ p
            alpha = (p.transpose() @ r) / (p.transpose() @ q)
            x = x + alpha * p
            r = r - alpha * q
            # update vars
            k = k + 1
            x_steps.append(x)
            y_steps.append(1/2 * x.transpose() @ A @ x - x.transpose() @ b)
            err.append(LA.norm(r))

        if k == Nmax:
            print("Reached␣max␣iterations")
            return x, x_steps, y_steps, err
        return x, x_steps, y_steps, err


    elif solvername == "biconjugate_gradient":
        # use built-in scipy function for biconjugate gradient
        #  matrix needs to be: nxn
        return bicg(A, b, x0, tol=tol, maxiter=Nmax)  # returns x and info

    elif solvername == "gmres":
        # use built-in scipy function for biconjugate gradient
        # matrix needs to be: nxn
        return gmres(A, b, x0, tol=tol, maxiter=Nmax)

    else:
        print("Please␣enter␣valid␣solver␣name.")
        return
```

B.3. Function for Creating Random Matrix Systems with Predetermined Eigenvalues

The function below details the creation of a random matrix with given eigenvalues contained in the vector lam. [8]

```
def eigcluster(lam):
    n = len(lam)
    Q = scipy.stats.ortho_group.rvs(n)
    return (Q.T @ np.diag(lam) @ Q)
```

Example of construction for 10 x 10 matrix with eigenvalues clustered away from the origin:

```
m = 10          # size of matrix
clus = 50       # large eigenval to create cluster around -- cluster center
spac = 0.25     # dist from cluster center
lam = np.concatenate((np.linspace(1, m, m-3), np.linspace(clus-spac, clus+spac, 3))

# define system
A = eigcluster(lam)
b = np.ones((m, 1))        # define b as vector of ones
x0 = np.zeros((m, 1))      # define initial guess (x0)
```

References

[1] Magnus Rudolph Hestenes, Eduard Stiefel, et al. *Methods of conjugate gradients for solving linear systems.* Vol. 49. 1. NBS Washington, DC, 1952.

[2] Muhammad Ali Raza Anjum. "One-Minute Derivation of The Conjugate Gradient Algorithm". In: *CoRR* abs/1608.08691 (2016). URL: `http://arxiv.org/abs/1608.08691`.

[3] T.A. Straeter. *On the Extension of the Davidon-Broyden Class of Rank One, Quasi-Newton Minimization Methods to an Infinite Dimensional Hilbert Space with Application to Optimal Control Problems.* North Carolina State University at Raleigh, 1971. URL: `https://books.google.com/books?id=1-bkGwAACAAJ`.

[4] Michele Benzi. *Preconditioning Techniques for Large Linear Systems Part II: Krylov Subspace Methods.*

[5] David Chin-Lung Fong and Michael A Saunders. "CG versus MINRES: An empirical comparison". In: *SQU Journal for Science* 17.1 (2012), pp. 44–62.

[6] Dianne P O'Leary. "The block conjugate gradient algorithm and related methods". In: *Linear algebra and its applications* 29 (1980), pp. 293–322.

[7] Timothy A. Davis and Yifan Hu. "The university of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software* 38.1 (2011), pp. 1–25. DOI: `10.1145/2049662.2049663`.

[8] Rick Wicklin on The DO Loop. *Generate a random matrix with specified eigenvalues.* Mar. 2012. URL: `https://blogs.sas.com/content/iml/2012/03/30/geneate-a-random-matrix-with-specified-eigenvalues.html`.

[9] Sophia Yang. *Descent method-Steepest descent and conjugate gradient in Python.* Dec. 2020. URL: `https://medium.com/dsc-msit/descent-method-steepest-descent-and-conjugate-gradient-in-python-85aa4c4aac7b`.