# FCM 2 HW 4

Ryan Bausback

March 2021

## 1 Executive Summary

Our goal is to implement the Steepest Descent (SD) and Conjugate Gradient (CG) methods for iteratively solving systems of equations involving Symmetric Positive Definite (SPD) matrices. We then analyzed the convergence of the methods in order to validate previously presented theorems. Additionally, we also compared SD and CG with and without a variety of preconditioners in terms of relative "true" solution error and relative residual. Overall, we found that our convergence results agreed with theory and that CG gave better performance for the majority of preconditioners when compared with SD on banded SPD matrices.

## 2 Statement of Problem

While the focus of the LU factorization to solve linear systems was to get as good a solution as possible in a single step, iterative methods allow you to gradually convergence to the "true" solution over many steps, potentially allowing your solution to more accurate. We will implement two different types of iterative methods: Steepest Descent (SD) and Conjugate Gradient (CG). The major difference between the two is that while SD takes the next optimization direction to exactly be the current residual, CG updates the current optimization direction with the residual such that it is orthogonal to all previous directions, not just the last one. This allows it to theoretically converge to the solution faster, something that we intend to validate.

We will also empirically confirm that using different preconditioners to turn the system into $\tilde{A}\tilde{x} = \tilde{b}$ actually allows the systems to be solved more quickly than without for both methods.[1] Four different preconditioners will be investigated: the Jacobi (diagonal) preconditioner, the Symmetric Gauss-Seidel preconditioner, the block preconditioner, and the tridiagonal preconditioner.

Additionally, we will also analyze the theoretical convergence rate bounds, as well as other convergence theorems, for both SD and CG, and verify their relationship to the eignevalues of the test problem. This will be done by taking $A = Q\Lambda Q^T$ where $\Lambda$ is the diagonal matrix containing the eigenvalues of A (which are all positive since A is symmetric positive definite).[1]

## 3 Description of Mathematics

### 3.1 Steepest Descent

The foundation of Steepest Descent (SD) is solving a minimization problem over the cost function Q(x), where the unique minimizer of Q(x), $x^*$, is the solution to the symmetric positive definite (SPD) system $Ax = b$. Since A is SPD, the inner product that it induces defines vector norm, $v^T Av = ||v||_A^2$, and we can therefore take the cost function Q(x) to be the linear least squares defined by this norm, i.e. $Q(x) = ||x - A^{-1}b||_A^2$. This can be further simplified to give $Q(x) = \frac{1}{2}x^T Ax - b^T x$.

Since this minimization is to be done iteratively, we can start at any position in the space, for instance $x_0$, and then update that position according to a short linear recurrence given by $x_{k+1} = x_k + \alpha p_k$. Here $p_k$ is the new direction that we would like to move in and $\alpha$ how far in that direction we will move. Since $x^*$ is the minimizer, we will eventually have to travel "down" towards it on the surface Q(x), and as its name implies, for steepest descent, it is required that we travel in the direction of most rapid descent from our initial starting point.[1] This is equivalent to taking the negative gradient of Q(x), which, using the above cost function for Q(x), gives $-\nabla Q = b - Ax = r$, where r is defined to be the residual. The residual r is thus by definition Euclidean orthogonal to the previous direction $p_k$.

To find the optimal stepsize, $\alpha$, we can plug the recurrence into the cost function and take the derivative such that we attain the minimum for that particular direction. This gives us $Q(\alpha) = 1/2 * (x_k + \alpha p_k)A(x_k + \alpha p_k) - b^T(x_k + \alpha p_k) = 1/2 * \alpha^2 Ap_k - \alpha r_k^T p_k - b^T x_k + 1/2 * x_k^T Ax_k$. Taking the derivative in terms of $\alpha$ and setting it equal to 0 gives $Q(\alpha) = ap_k^T Ap_k - r^T p_k = 0$ so $\alpha = \frac{p_k^T r_k}{p_k^T Ap_k} = \frac{r_k^T r_k}{r_k^T Ar_k}$ since we chose $p_k = r_k$ above.

### 3.2 Conjugate Gradient

For conjugate gradient (CG) however, we would like the next direction in which we move to be pairwise orthogonal under the A-norm to not only the current direction vector, but also all previous direction vectors as well within the space, such that $B_k = span[p_0, ..., p_k]$ where $B_k$ is a subspace. The same cost function defining the residual will be used, $Q(x) = ||x - x^*||_A^2$ but instead of minimizing it in a single direction, we now need to minimize Q(x) with respect to all previous directions. However, we also know that the residual, $r_k$, is Euclidean orthogonal to $p_j$ for all j=0,...,k-2, as the recurrence for $r_k = r_{k-1} + \alpha_{k-1}Ap_{k-1}$, if we select $p_0 = r_0$ as in Theorem 8.1. [2] That then allows us to develop a short recurrence for $p_k$, similar to $x_k$, such that $p_k = r_k + \beta_{k-1}p_{k-1}$ where $\beta_k = \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}}$. This $\beta_k$ comes from rearranging the $r_k$ recurrence $\alpha_{k-1}Ap_{k-1} = r_k - r_{k-1}$, using the A-orthogonality to get $\mu_k = \frac{p_{k-1}^T Ar_k}{p_{k-1}^T Ap_{k-1}}$, and then substituting $r_{k-1}^T Ap_{k-1} = \frac{r_k^T r_k}{-\alpha_{k-1}}$, where the new $\alpha_{k-1} = \frac{r_{k-1}^T r_{k-1}}{p_{k-1}^T Ap_{k-1}}$. This $\alpha$ is the same as for with SD but recognizing that under the assumptions of CG, $r_{k-1}^T r_{k-1} = p_{k-1}^T r_{k-1}$ due to orthogonality.

## 3.3 Preconditioning

However, sometimes the system $Ax = b$ that we would like to solve is not well conditioned. In this case, a different but related problem with a smaller condition number can be solved, $\tilde{A}\tilde{x} = \tilde{b}$, that will also allow the system to be solved faster. In this case, $\tilde{A}\tilde{x} = \tilde{b}$ is related to $Ax = b$ by $\tilde{A} = L^{-1}AL^{-T}$, $\tilde{x} = L^T x$, and $\tilde{b} = L^{-1}b$ where $LL^T = M$ defines the symmetric positive definite preconditioner. Hence, for steepest descent, we can define $\tilde{r} = \tilde{b} - \tilde{A}\tilde{x} = L^{-1}b - L^{-1}AL^{-T}L^T x = L^{-1}(b - Ax) = L^{-1}r$, $\tilde{v} = \tilde{A}\tilde{r} = L^{-1}AL^{-T}L^{-1}r = L^{-1}AM^{-1}r = L^{-1}Az$ where $z = M^{-1}r$, and $\tilde{\alpha} = \frac{\tilde{r}^T\tilde{r}}{\tilde{r}^T\tilde{v}} = \frac{r^T L^{-T} L^{-T} r}{r^T L^{-T} L^{-T} Az}$ due to the symmetry of M, and hence $\alpha = \frac{r^T M^{-T} r}{r^T M^{-T} Az} = \frac{z^T r}{z^T Az}$ on each step.[3] This means that any two iterations for $x_i$ are related by $\tilde{x}_{k+1} = \tilde{x}_k + \tilde{r}_k \alpha_k$ which is identical to $x_{k+1} = x_k + z_k \alpha_k$, and therefore applying steepest descent to the altered problem is the same as solving the system $Mz = r$ on every step using the original problem. This can be shown similarly for CG where the derivations for r is the same. Hence since $\tilde{r}_0 = L^{-1}r_0$ and $p_0 = z_0$ then $\tilde{p}_0 = L^T p_0$. For $\alpha = \frac{\tilde{r}^T\tilde{r}}{\tilde{p}^T\tilde{v}}$ is also the same on step 1 given the definition of $\tilde{p}_0$. We need also that $\tilde{r}_k^T \tilde{r}_k = r_k^T L^{-T} L^{-1} r_k = r_k^T M^{-1} r_k = r_k^T z_k$ to define $\beta$ and the other definitions of the recurrences follow.

# 4 Algorithm and Implementation

For the steepest descent and conjugate gradient algorithms, the implementations are based on the codes in set 7 and set 8 using preconditioning.[1][2] The residual was calculated in a separate method for repeated re-use, and then passed to separate methods to solve the preconditioning equation Mz=r. Since four different preconditioners are to be tested, different methods were created to solve each type and called based on a integer that denotes the correct method using multiple if-elif-else statements. This initial z was set equal to p for conjugate gradient. The main part of the algorithm is a while loop that iterates on the relative residual, $||r||/||b||$, until it is less than a specified stopping condition. Since the algorithm is to almost entirely be run in single precision, this was taken to be $1x10^{-8}$ or approximately machine epsilon or 0 for single precision.

The next step in SD and CG is computation of the matrix-vector product, $w = Az_k$ or $v = Ap_k$ respectively. This was done differently depending on the structure of A, in two different methods that were differentiated by a boolean indicating A's structure. For the case when A was only restricted to being SPD, due to symmetry, only the lower half of A, including the diagonal, was stored. Therefore, the multiplying vector needed to be multiplied by the matrix normally up to and including the main diagonal left to right in the structure, and then swapping the column and row indicators for the upper half of the matrix. Thus the upper factor of the matrix is not actually stored and the transpose never computed for efficiency.

For the case when A was further restricted to be banded to k=0, ..., 6 depending, A was stored as a 2d vector array of size (k+1)n or O(n), with each column as a diagonal starting with the main diagonal and moving outward. The computation is then split into two steps: computing across the rows right to left (iterating from k to 0) and computing down the diagonals from the row under the current row to the edge. Because the data structure for A is no longer square, two different indexers needed to be iterated simultaneously, with one being restricted as less than n and the other less than k+1. This results in O(n) computations since there are at most n computations in a outer loop iteration and minimum k, less then $O(n^2)$.

Since one of the preconditioners is a tridiagonal matrix, a tridiagonal SPD solver was implemented. An independent method computes the Cholesky factorization for the matrix, then the solving method solves $Ly = b$ and $L^T x = y$, which results in $3Cn + O(1)$ or $O(n)$ computations over both methods. The Cholesky factorization method is only applied a single time at the beginning of the CG or SD routine, further reducing the computations. An additional solver that computes the solution to the system defined by $L$ or $L^T$ was also implemented in $O(n^2)$ since no structure is assumed for $L$. $L^T$ system was solved in place without transposing, which requires flipping the column and row indexers and decreasing from n-1 to 0.

Additional vector product routines for computation during the short linear recurrences in both SD and CG were also implemented in $O(n)$, and $2n + O(1)$ and $3n + O(1)$ were required to actually compute the recurrences for each iteration k.

## 5 Experimental Design and Results

### 5.1 Correctness

In order to verify the correctness of the the implementation, several systems were solved by first creating random X vectors and random A matrices of different sizes, using the matrix vector product routines to compute the b, and then passing A and b to the SD and CG routines to see if the X that is output is the same as the random initial X.

The matrices used to test correctness were: a 3x3 matrix of all integers with a bound of 0.001, a 15x15 random, diagonally dominant matrix with a bound of 0.001 for CG and $10^{-8}$ for SD, and two 30x30 random, diagonally dominant matrices with bounds of 0.001 and $10^{-8}$ for each. As Figure 1 illustrates, as the residual decreases, the error between the true solution to the system in double precision and the computed solution in single precision also decrease, indicating that both algorithms are actually approaching the true solution to the system as expected. We can also see that the error results for both are approximately linear under the log scaling of the y-axis, with some minor fluctuations. These fluctuations are likely to be a function of the roundoff error in single precision. We can also see than the number of iterations the CG takes to reach the same residual threshold as SD is significantly less, as theory predicts.

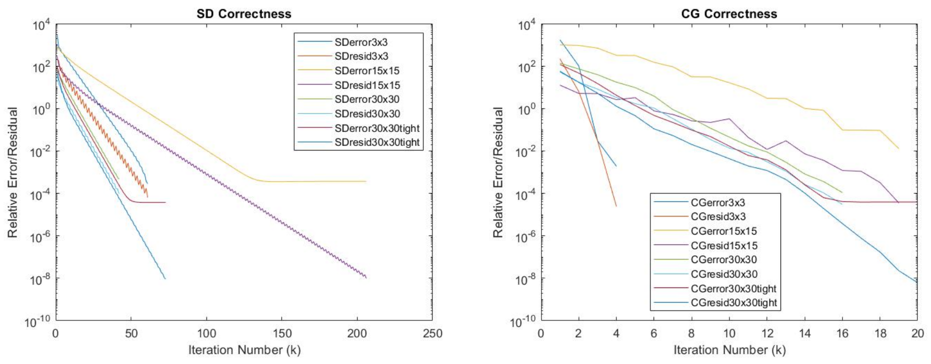

Figure 1: Correctness testing for SD and CG based on different sized systems

## 5.2 Influence of Spectrum

Using similar arguments and manipulations as for section 3.3 on preconditioning, we know that $Ax = b$ and $\Lambda\tilde{x} = \tilde{b}$ have the same error under the 2-norm, $||e||_2 = ||\tilde{e}||_2$, where $\tilde{x} = Q^T x$ and $\tilde{b} = Q^T b$, meaning $A = Q\Lambda Q^T$.[3] As $\Lambda$ is a diagonal matrix containing the eigenvalues of A, we realize that the convergence behavior (i.e. the error) is only related to the eigenvalues of A. Thus we can solve the easier system $\Lambda x = b$ in order to test the convergence properties of SD and CG. The matrix $\Lambda$ was thus generated to have random positive eigenvalues in the range of [1,100] plus the value of the index in which the eigenvalue was stored. Thus any matrix A that can be decomposed as described above with the specified $\Lambda$ will have identical convergence behavior and is therefore automatically being tested for.

The first convergence property that we would like to verify is that the convergence rate for steepest descent is:

$$||e^{(k+1)}||_A \leq \frac{\kappa(A)_2 - 1}{\kappa(A)_2 + 1}||e^k||_A$$

where $\kappa(A)_2 = \frac{\lambda_{max}}{\lambda_{min}}$ and $||e^k||_A$ is the error $x_k - x^*$ with respect to the A-norm. [1] Therefore, if we subtract the right-hand-side from the left-hand-side, we should expect the result to be positive.

As Figure 2 shows, this remains true only up to a certain point, as indicating by the blue line not extending out as far as the relative error and relative residual lines in the plots, since Matlab cannot plot a value on the log scaled

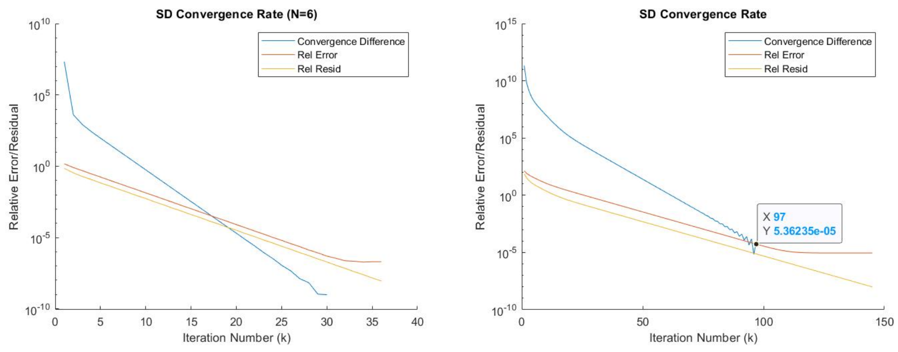

Figure 2: Difference between RHS and LHS in SD convergence rate theorem for 2 representative examples

y-axis if it is negative. However, if we look at the histograms for 100 randomly generated systems with n=6 and n=20 for the number of iterations, the relative errors, and the relative residuals at the first instance where the convergence relationship becomes negative, we can see that this happens at around machine epsilon for single precision. This is seen in Figures 3 and 4. This is indicating that the system has already converged to the true solution in single precision and therefore, any negative values in the difference are a result of the finite precision representation of the problem on the machine. Therefore, the A-norm of the errors do in fact remain bounded by the condition number ratio times the previous A-norm error, empirically proving the convergence theorem.

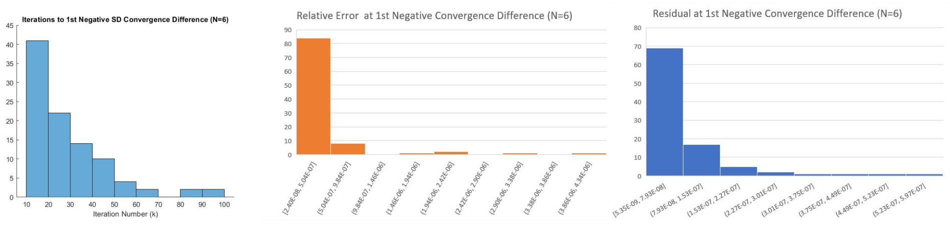

Figure 3: Iteration Number, Relative Error, and Relative Residual Values at the 1st Instance of Negative Convergence Rate Difference for 100 samples, n=6

For CG, the convergence rate is bounded by:

$$||e^{(k)}||_A \leq 2\alpha^k ||e^{(0)}||_A$$

$$\alpha = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}$$

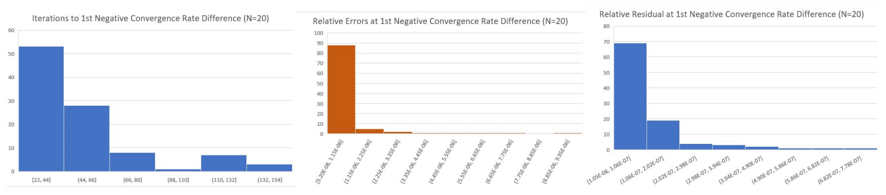

Figure 4: Iteration Number, Relative Error, and Relative Residual Values at the 1st Instance of Negative Convergence Rate Difference for 100 samples, n=20

in the A norm and:

$$||e^{(k)}||_2 \leq 2\sqrt{\kappa}\alpha^k||e^{(0)}||_2$$

$$\alpha = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$$

in the 2-norm, where $\kappa$ is the condition number of A defined in terms of the eigenvalues identical to SD.[2] Therefore, as with SD, if the RHS is subtracted from the LHS for each of the convergence rate bounds, we should expect the difference to be positive.

Additionally, we also know by Theorem 8.5, "if A has m distinct eigenvalues, then CG converges in at most m steps."

Two representative examples for systems of sizes n=6 and n=20 were generated. The eigenvalues for the first system were (10, 3, 3, 5, 7, 14) meaning that A has 5 distinct eigenvalues while the eigenvalues for the second system were (1, 10, 8, 13, 14, 12, 9, 13, 12, 11, 19, 21, 21, 16, 23, 17, 24, 19, 27, 24) meaning that A has 15 distinct eigenvalues. Therefore, we should expect CG to converge in 5 and 15 steps at most respectively.

As Figure 5 shows, the absolute errors for both examples remain well below their prospective thresholds for all iterations on both norms. Additionally, we can see that CG took exactly 15 steps to converge for n=20, as predicted. For n=6, by closely examining the turquoise line, it is apparent that the residual is at the order of $10^{-8}$ at 5 iterations, which is machine epsilon or approximately 0 for single precision. However, because the stopping threshold was set at $10^{-8}$, the algorithm continues for an additional step to get a residual below $10^{-8}$. However, because it at the level of machine epsilon after 5 steps, we can say CG converged correctly in this case as well.

As the left histograms in Figures 6 and 7 show, the case where CG takes an additional iteration in order to convergence is indeed a numerical fluke, denoted by the difference between the number of iterations to reach the threshold and the number of distinct eigenvalues in $\Lambda$ being zero for most cases. As n increased to 20, more iterations where required for some outliers (as the negative values indicated). This can be attributed to the finite precision computations in single

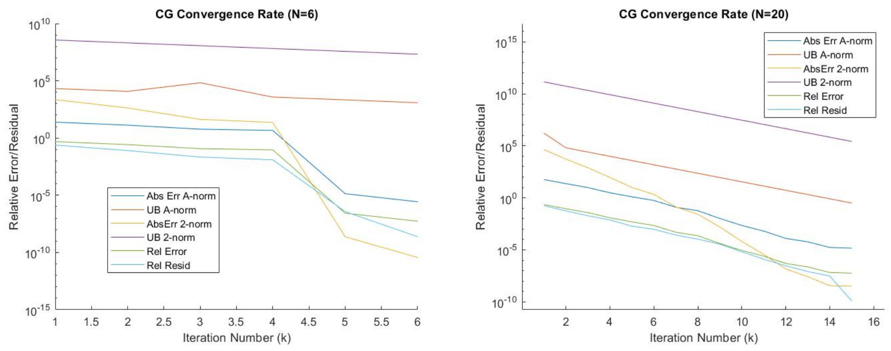

Figure 5: CG convergence rate theorem plots for 2 representative examples

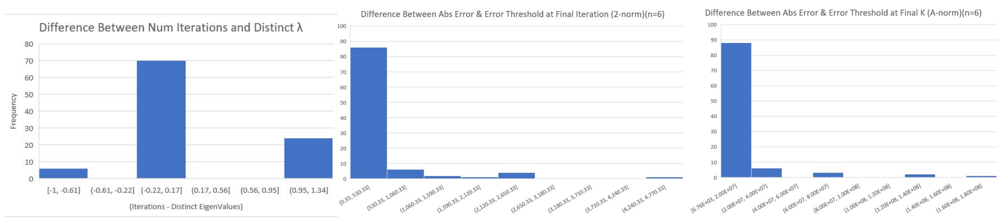

Figure 6: Difference between Iteration Number and Number of Distinct Eigenvalues, and Difference between Absolute Error and Error Threshold under 2-norm and A-norm for 100 samples, n=6

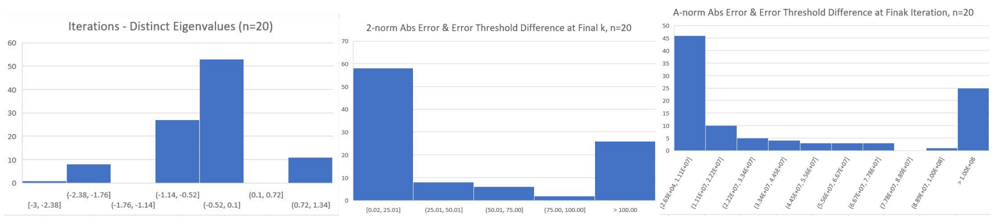

Figure 7: Difference between Iteration Number and Number of Distinct Eigenvalues, and Difference between Absolute Error and Error Threshold under 2-norm and A-norm for 100 samples, n=20

precision accuracy perturbing the values enough to require additional iterations, since the stopping threshold for the relative residual was set very low ($10^{-8}$ as before). However, because the solution in the majority of cases took exactly the same steps as distinct eigenvalues to convergence or even less, this is strong empirical evidence of Theorem 8.5.

Additionally, we also see that for all samples for both n=6 and n=20 that the absolute error at the final step remained well below the error threshold described the convergence rate theorems. This is shown by all of the values for both histograms being positive, since the difference between the threshold and the absolute error was taken for each sample. This also confirms the predictions of the convergence rate theorems.

## 5.3 Influence of Preconditioners

As discussed in section 3.3, preconditioning transforms the problem in one that can be solved more quickly and reliably by reducing the condition number, $\kappa(A)$. Four different preconditioners were investigated given a banded symmetric positive definite matrix A: the diagonal preconditioner, the SGS preconditioner (Symmetric Gauss-Seidel), the block preconditioner, and the tridiagonal preconditioner.

### 5.3.1 Tridiagonal Preconditioner

The first preconditioner to be investigated was the tridiagonal preconditioner. The tridiagonal preconditioner is defined as the sub, main, and super diagonals of the matrix A, so therefore k or the number of bands in A must be greater than 1 since otherwise A=M. To begin with, A was generated as a random, strictly positive, diagonally dominant, symmetric banded matrix using the columns-as-diagonal storage architecture as in section 4. Diagonal dominance by both rows and columns is the same as diagonal dominance by either due to symmetry, and this was ensured by multiplying the value of the main diagonal by 2*n*(a random int between 1 and 100). Then the cholesky factorization of A was computed in order to ensure A was actually positive-definite, since the eigenvalues of A were not explicitly controlled during generation. If A is not positive definite, then the cholesky factorization will fail as the value under the square root will be negative. If this was the case, the matrix was thrown out and a new matrix was generated.

The preconditioning matrix M was generated from A using the same efficient diagonals-as-columns architecture as in section 4, meaning that $2n-1$ storage is required for M due to symmetry. As discussed in section 4, at the beginning of either SD or CG, the cholesky facorization of the preconditioner M was computed, and then tridiagonal-specific forward/backward solver was applied at every step.

As Figure 8 clearly shows, on a 20x20 banded SPD linear system, tridiagonal preconditioning has a significant impact on reducing the number of iterations to reach a certain residual threshold (in this case $10^{-8}$). Matrices of size 20x20 were chosen in this case because if something is wrong with the code, it will be easier visible since the system is so large. For SD, the number of iterations to reach $10^{-8}$ dropped from 300 iterations to just 11! For CG, the number of iterations was cut by more than half from 20 (which is consistent with previous experimentation as there are 20 eigenvalues) to just six. This seems to confirm the fact that preconditioning lowers the condition number such that

$\kappa(\tilde{A}) < \kappa(A)$, as the condition number is defined by the max eigenvalue over the minimum eigenvalue. Since we know from part 1 that CG requires approximately the same number of iterations as distinct eigenvalues, since CG took 6 iterations in this case, that means preconditioning changed the eigenvalues such that there are now a minimum of six distinct ones, which severely limits opportunities for the maximum eigenvalue to be greater than the minimum since most eigenvalues are all the same.

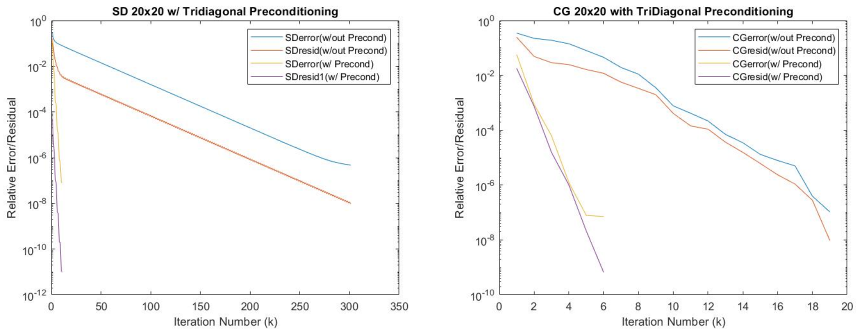

Figure 8: Representative Examples of CG and SD with and without Tridiagonal Preconditioning

However, because the stopping point of the method is determined by the relative residual, which is inherently connected to the relative error, preconditioning in this case did not lower the overall error achieved or the residual. However, because preconditioning took fewer iterations, the rate at which the residual reaches the threshold has increased dramatically, as well as the rate at which error is reduced. Since the plots above are plotted against the log of the errors and residuals in to show greater detail at the level of $10^{-8}$, the linear trend of the error and residuals of CG and SD are easily visible, with some numerical fluctuations.

Additionally, 100 matrices of size 20x20 with random numbers of bands ranging from 2 to the max of 6 were generated, and the errors, residuals, and number of iterations at the stopping point were tracked. As Figures 9 - 12 illustrate, the number of iterations was significantly less across all the 100 samples for Tridiagonal preconditioning versus without preconditioning for both SD and CG. As seen in the representative example, the error and relative residual remain fairly constant however, since the relative residual is the stopping criterion for the algorithms. For SD, many of the systems took well over 200 iterations to get within $10^{-8}$ without preconditioning, while with it, only 1 system took close to 200. A greater portion took less than 40 iterations with preconditioning compared to without it also. The same pattern was true for CG as well, were the max number of iterations with preconditioning was 9, while without it, all systems took between 14 and 24 iterations to solve. This is consistent with the results from part 1 as well.

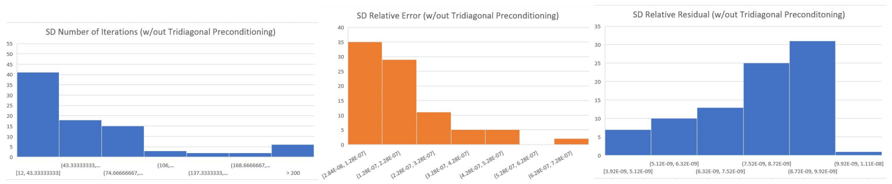

Figure 9: SD 20x20 Matrix WITHOUT Tridiagonal Preconditioning

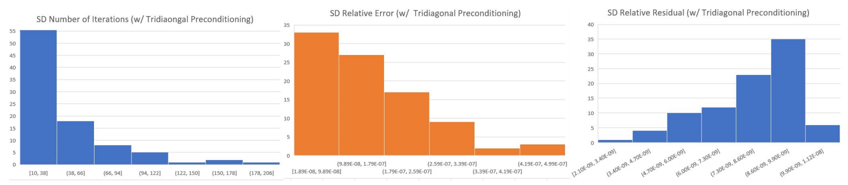

Figure 10: SD 20x20 Matrix WITH Tridiagonal Preconditioning

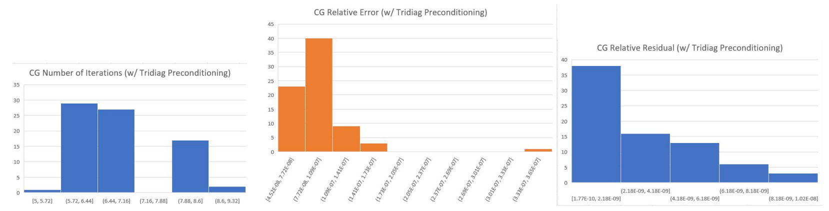

Figure 11: CG 20x20 Matrix WITH Tridiagonal Preconditioning

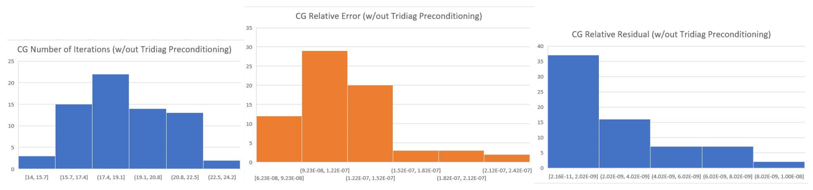

Figure 12: CG 20x20 Matrix WITHOUT Tridiagonal Preconditioning

It is also important to note that for a greater portion of the 100 samples generated failed to produce any result for SD and CG without preconditioning than with preconditioning. This means that the histograms only represent the portion of the generated samples that were able to be succcessfully solved. Since the cholesky factorizations in all cases were successfully computed, this is not a failure of generating matrices there were not actually SPD. Rather, more likely that the conditioning of the problem was extremely poor as the system was randomly generated, as would be indicated by a huge condition number, which led to complete numerical instability. This also could be connected to the limitations of single precision when representing numbers greater than $10^8$ which may have caused overflow, resulting in "NAN". Therefore, since more of the systems with preconditioning where able to be successfully computed, it provides further empirical evidence that tridiagonal preconditioning reduces the condition number and therefore increases the numerical stability.

### 5.3.2 Diagonal Preconditioner

The next preconditioner to be investigated is the diagonal or Jacobi preconditioner, which is defined by the elements of the matrix A on the main diagonal. The matrix A was again defined as a random diagonally dominant SPD matrix in the same fashion as for the Tridiagonal Preconditioner, except with the restriction for the number of bands to be greater than 1 lifted. This time, matrices of size 8x8 were generated, as this is larger than the max possible number of bands for the matrix of 6. The matrix M was generated as a 2d vector, with row-length 1 in order to fit the parameters of the functions, with the main diagonal as the only column. The preconditioning system was solved in $O(n)$ by just dividing the elements of r by the corresponding indexes in M for every iteration of SD and CG.

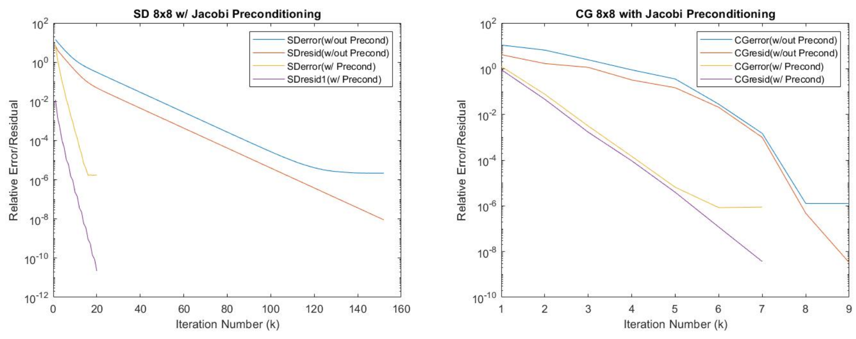

Figure 13: Representative Examples of CG and SD with and without Jacobi Preconditioning

Once again, by examining Figure 13, we see the tremendous impact that preconditioning has on reducing the number of iterations for steepest descent. The number of iterations dropped from 152 to 20, which is a major difference. For CG, there was a reduction in the number of iterations as well, but less pronounced that with the 20x20 system with tridiagonal preconditioning. In both cases for CG, the number of iterations closely matched the size of the matrix or the number of eigenvalues, continuing to prove the convergence theorem in part 1.

By examining the 100 generated samples as in Figure 14 - 17, we see similar

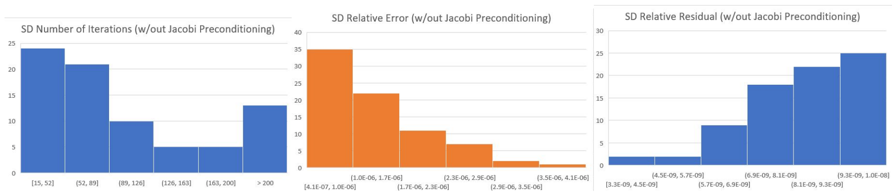

Figure 14: SD 8x8 Matrix WITHOUT Jacobi Preconditioning

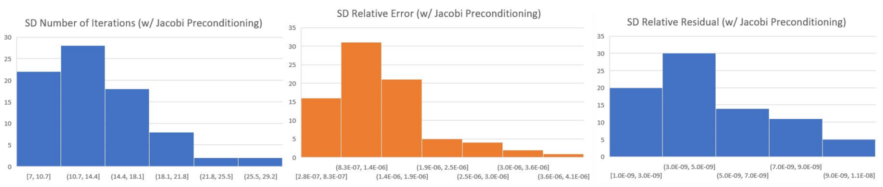

Figure 15: SD 8x8 Matrix WITH Jacobi Preconditioning

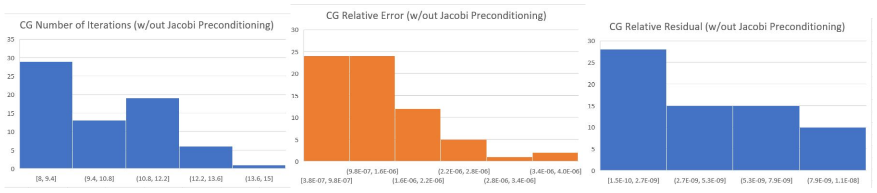

Figure 16: CG 8x8 Matrix WITHOUT Jacobi Preconditioning

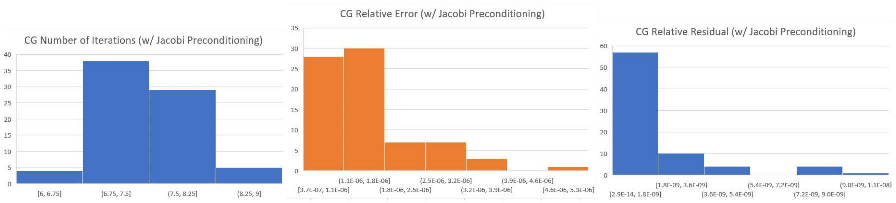

Figure 17: CG 8x8 Matrix WITH Jacobi Preconditioning

trends to tridiagonal preconditioning as well, where the relative error and relative residual are not generally smaller, but which reach the threshold quicker due to the fewer iterations. In fact, for without preconditioning on SD, many of the examples require over 200 iterations to solve but with preconditioning, no system takes over 30. The same is true for CG, where the number of iterations is at times a few iterations higher than the size of the matrix, even up to 14, indicating a very numerically ill-conditioned problem. However, for Jacobi Preconditioning on CG, no system takes more than 9 iterations, which is within a tolerance since the threshold for the residual is set so low.

### 5.3.3 Symmetric Gauss-Seidel (SGS) Preconditioner

The Symmetric Gauss-Seidel (SGS) preconditioner is defined by $M = (D - L)D^{-1/2}D^{-1/2}(D - L^T) = CC^T$ by Cholesky factorization when $A = D - L - L^T$ and $D = diag(A)$. Therefore, due to symmetry, we only need to store one of the halves of the preconditioner, which can be formed directly from A without having to apply a separate Cholesky factorizaiton routine. This factor of the preconditioner was passed to SD and CG algorithms respectively and then the forward and backward solves $Cy = r$ and $C^Tz = y$ were applied on each iteration, as C is lower triangular.

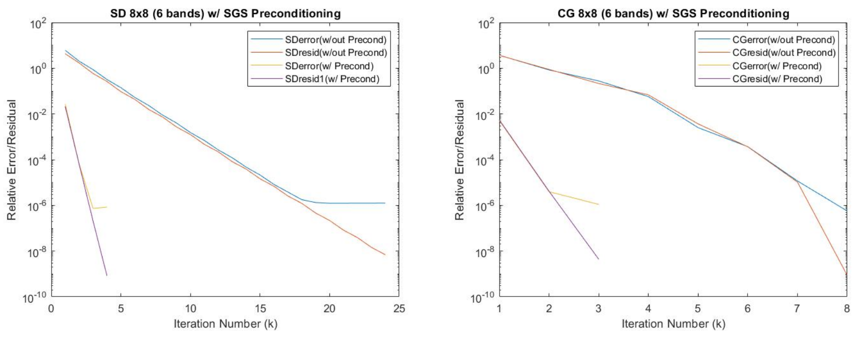

Figure 18: Representative Examples of CG and SD with and without SGS Preconditioning

As for the diagonal preconditioning, an 8x8 random diagonally dominant SPD system was generated with a banded structure of 6 bands. As Figure 18 shows, there was a similar reduction in iterations to reach the single precision machine epsilon threshold, with similar ending error and residual in both cases, as expected. The reduction in number of iterations this time was dramatic, reducing the number of iterations by more than half for CG and by an eighth for SD. The linear-under-log trends are again generally seen in the error and residual away from machine epsilon for float.

Additionally, in order to determine whether the performance of the preconditioner is affected by how the matrix was generated, the 20x20 tridiagonal Toeplitz matrix with 2 on the main diagonal and -1 on the sub and super diagonals was tested. Since the structure of the matrix was known before testing, the eigenvalues were computed in Matlab, with the maximum being 3.9777 and the minimum being 0.0223. This results in a condition number of 178, meaning we should expect to lose 2 decimal places from the end of the precision, or about $10^{-6}$ since the precision of float is $10^{-8}$.

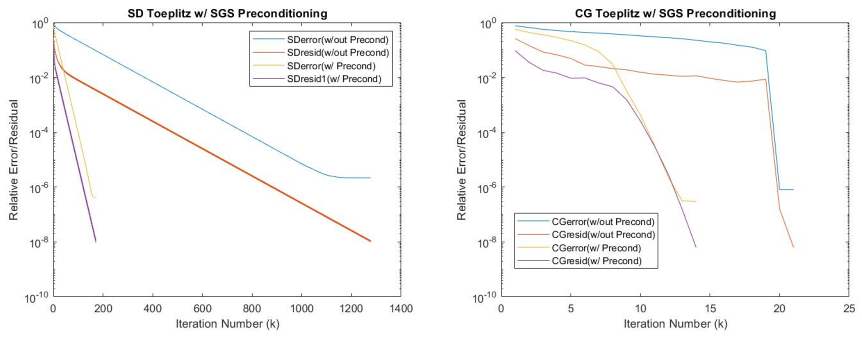

Figure 19: CG and SD on Tridiagonal Toeplitz with and without SGS Preconditioning

As Figure 19 illustrates, the prediction of accuracy from the order of magnitude of the condition number is approximately correct, as the error achieved for all runs with and without preconditioning is around $10^{-6}$. The impact of preconditioning on the number of iterations is once again easily visible, cutting SD from over 1000 iterations to just under 200. CG additionally dropped from 21 iterations, within the margin of error predicted by convergence theorems, to 14, a reduction of almost half. The rate of convergence is approximately linear for SD, as expected, but for CG, the error and residual remain very steady before dropping off a cliff right near the number of iterations equal to the size of the matrix. This behavior is decidedly nonlinear. However, we can see by comparing the plots with and without preconditioning that the same behavior is apparent in both. This seems to indicate that this convergence behavior is a combination of conjugate gradient itself and the matrix, and not necessarily from the impact of the matrix generation technique on the preconditioner.

### 5.3.4 Block Diagonal Preconditioner

The final preconditioner to be investigated is the block diagonal preconditioner, which takes the 2x2 blocks of A along the main diagonal in order to form the preconditioner. The efficient storage scheme implemented for this preconditioner involves saving the lower half (due to symmetry) of the 2x2 block as 3 indices in a row, for each of the n/2 blocks, resulting in 3/2n or $O(n)$ storage. The same methodology for the tridiagonal preconditioner was implemented, where the Cholesky factorization of each of the 2x2 blocks was formed and saved identically on the first step, and then the forward and backward solves applied on subsequent iterations.

The test matrices were again generated as 8x8 random, banded, diagonally dominant SPD, with a random number of bands ranging from 1 to 6. This was checked using an initial Cholesky factorization as described above. As Figure 20 shows, the same behavior of reducing the number of iterations to reach a certain level of relative error and residual was once again observed for both SD and CG. The convergence behavior of Steepest Descent remained approximately linear under the log but conjugate gradient showed the same bowed outward behavior that was most salient in the example involving Toeplitz matrices.

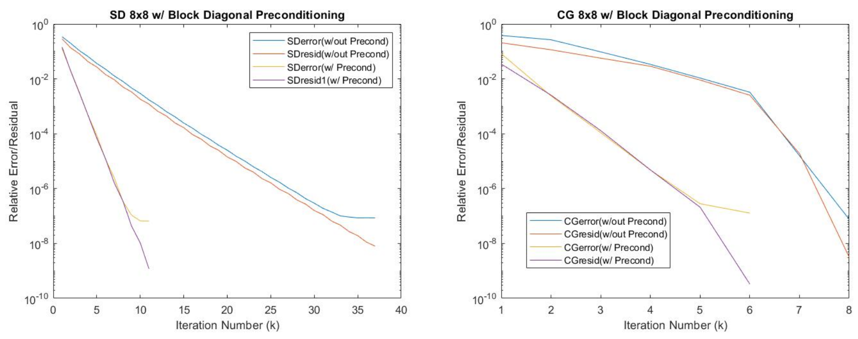

Figure 20: Representative Examples of CG and SD with and without Block Diagonal Preconditioning

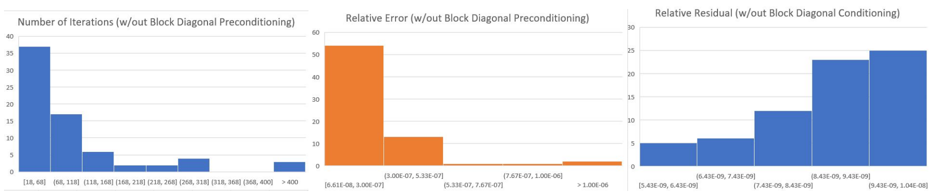

Figure 21: SD 8x8 Matrix WITHOUT Block Diagonal Preconditioning

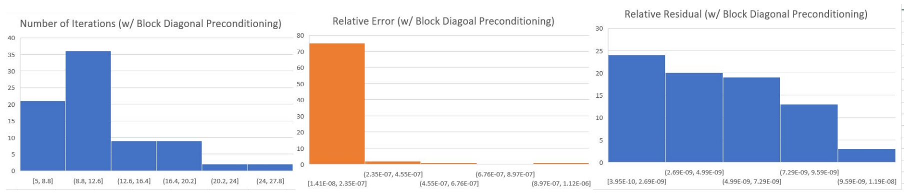

Figure 22: SD 8x8 Matrix WITH Block Diagonal Preconditioning

As before, 100 8x8 random diagonally dominant banded SPD matrices were generated and the number of iterations, relative error, and relative residual were tracked at the final iteration of both the CG and SD algorithms (Figures 21 - 24). The same trend that was observed in the representative examples is also seen here for the samples that computed, were the number of iterations on

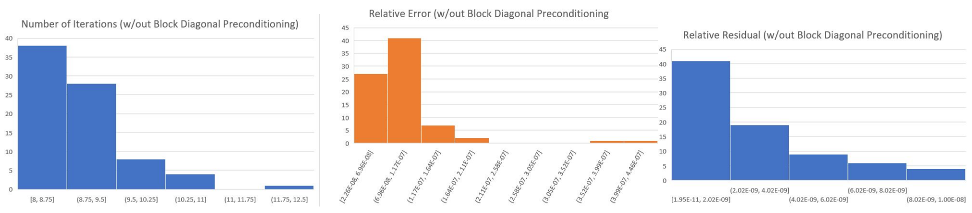

Figure 23: CG 8x8 Matrix WITHOUT Block Diagonal Preconditioning

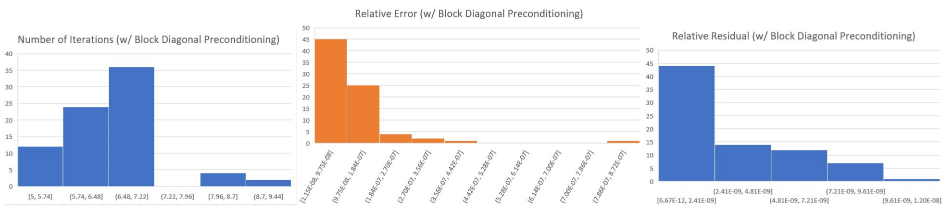

Figure 24: CG 8x8 Matrix WITH Block Diagonal Preconditioning

average was decreased by a significant amount using the preconditioner, while the relative error and relative residual remained fairly constant as the relative residual is the stopping threshold. In the most dramatic example, SD dropped in number of iterations from over 2000 in an outlier case to a max of 27 iterations with the preconditioning, while one average dropping from around 50 to around 10 iterations. For CG, eight or more itearations was reduced to 6 on average, which is not as dramatic a decrease but a decrease nonetheless.

## 6 Conclusion

In conclusion, the steepest descent and conjugate gradient algorithms were able to be successfully implemented in order to solve linear systems of equations of SPD matrices of various types and sizes. The convergence rates for both CG and SD were able to validated successfully, as well as that the number of iterations to solve with CG is the same as the number of distinct eigenvalues in the system. This was further reinforced later during the testing of preconditioners, when all the generated systems took within a margin of error of the number of iterations corresponding to the size of the matrix or less to converge.

The influence of four different preconditioners was also investigated. Similar performance was found for all of them using the same SPD matrix generation technique, with the major differences coming from the change in algorithms from CG to SD, and from using a different matrix generation technique. The number of iterations to reach a particular relative residual threshold was significantly reduced when using the preconditioners compared to without. No change in the relative error was observed, which is to be expected since the relative error is intimately related to the relative residual.

Running the algorithms in single precision resulted in the relative error getting stuck at around $10^{-7}$, even when the relative residual would continue to decrease. This make sense because machine epsilon for single precision is approximately $10^{-8}$, and given that the relative residual is calculated by subtraction and then division, this introduces opportunities for cancellation that perturb the true error, resulting in a number that is off by a decimal place. Therefore, this error floor is function of running the algorithm on the machine and not a bug in the algorithm itself.

Overall, conjugate gradient was able to achieve significantly faster convergence than steepest descent both with and without preconditioning, validating the theory.

## 7 Program Files

Both conjugate gradient and steepest descent were coded in C++ in a single large .cpp file on the local machine and compiled using "g++". The histograms were created by outputting a .txt file and then importing that into Excel, while the plot were creating in the same way but using Matlab.

## References

[1] www.math.fsu.edu/ gallivan/courses/FCM2new/Locked/Sets/set7.pdf

[2] www.math.fsu.edu/ gallivan/courses/FCM2new/Locked/Sets/set8.pdf

[3] www.math.fsu.edu/ gallivan/courses/FCM2new/Locked/Solutions/solhw3.pdf