Abstract

The solving of linear systems is one of the most important mathematical activities in science and engineering. The solution of most physical problems usually ends up as the solution of $Ax = b$. Examples include problems in Computational Engineering (FVM, FEM and FDM), Optimization, Economics and Medicine. There are numerous methods employed to solve such systems and many of them tend to be domain-specific. While methods such as Gaussian Elimination can be designed to be generic (working for dense, sparse, positive definite, indefinite etc. class of matrices), their algorithmic complexity is $O(N^3)$. Many iterative methods exist which can solve the system in $O(N^2)$. One such class of methods are the Krylov Subspace (KS) Methods. They have been traditionally used for solving sparse linear systems of the form $Ax = b$ where $A \in \mathbb{R}^{N \times N}$ and $b \in \mathbb{R}^N$. Methods such as CG, MINRES, QMR and their variants such as BiCG, BiCGSTAB and GMRES have been studied for their convergence and parallel implementation. All of the KS methods involve $O(N^2)$ Matrix-Vector multiplications which can be easily implemented using optimized hardware specific BLAS Level 2. Further, these methods have been developed as replacement for the Gaussian Elimination with Partial Pivoting since they can exploit the sparsity better than the elimination which causes fill-ins. Algorithms exist which prevent these fill-ins but their algorithmic complexity exceeds that of KS methods (which is $mO(N^2)$ where m is the number of iterations to convergence). In the thesis, an attempt is made to study the viability of using KS methods for solving dense systems. For any given system without any known structure, the current mode solving these systems is using Gaussian Elimination with Partial Pivoting. This algorithm is $O(N^3)$. By exploiting the recent advances in parallel computing technology and through the use of efficient preconditioners, it might be possible for KS methods to be faster than Gaussian Elimination beyond a given matrix size. The problem is analyzed through 2 independent perspectives viz. parallelization of algorithms and numerical experiments on algorithms. The former attempts to study the behaviour of the algorithms with increased available shared memory compute power and the latter attempts to correlate the time and convergence characteristics of 5 popular KS algorithms with the conditioning of the linear system.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction to Matrix Computation and Properties

1.1.1 Dot Product

A dot product of 2 vectors a and b ($a, b \in \mathbb{R}^n$) is defined as $\sum_{i=1}^{n} a_i b_i$. It is the sum of the product of the corresponding elements of 2 vectors of the same length.

$$a.b = a^T b \tag{1.1}$$

For orthogonal vectors, $a.b = 0$

1.1.2 Matrix Vector Multiplication

For A $\in \mathbb{R}^{l \times m}$ and x $\in \mathbb{R}^m$,

$$b = Ax = \sum_{j=1}^{n} x_j a_j \tag{1.2}$$

A matrix vector multiplication can be thought of as an operation constructing a linear combination of the rows of a matrix with coefficients equal to the corresponding elements of the vector. a_j denotes the j^{th} column of A

1.1.3 Matrix Matrix Multiplication

If A $\in \mathbb{R}^{l \times m}$ and C $\in \mathbb{R}^{m \times n}$, B = AC is defined with entries of B given by

$$b_{ij} = \sum_{k=1}^{m} a_{ik} c_{kj} \tag{1.3}$$

1.1.4 Symmetry

A matrix is called symmetric if it is the transpose of itself.

$$A^T = A \tag{1.4}$$

Tranposing is the operation of interchanging the rows and columns of a matrix.
If A $\in \mathbb{R}^{l \times m}$ then $A^T \in \mathbb{R}^{m \times l}$

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad\qquad A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

1.1.5 Range

Range of a matrix A $\in \mathbb{R}^{l \times m}$ is defined as the set of vectors that can be expressed as Ax for any x $\in \mathbb{R}^m$. It can also be thought of as the set of all vectors which can be expressed as a linear combination of the columns of A.

1.1.6 Rank

Rank of a matrix A $\in \mathbb{R}^{l \times m}$ is defined as the dimension of the space spanned by the rows and columns of A. Some important properties in relation to ranks.

- Row rank is always equal to column rank. In case of a rectangular matrix, the maximum rank of the matrix is the lesser of the 2 dimensions. Mathematically, For A $\in \mathbb{R}^{l \times m}$, max(rank(A)) = min$(l, m)$. Graphically what this states is that even if one has a billion 2 dimensional vectors, we cannot use them to express a 3 dimensional vector.
- If A is square and rank(A) $\neq 0$, det$(A) \neq 0$. The reverse is also true.
- A rank 0 matrix would imply that atleast one of its columns is a linear combination of the others. This column does not *add anything new* to the existing information in the matrix.

$$x + y = 2$$
$$2x + y = 3$$

Has a unique solution (x,y) = (1,1). This is because every equation *adds to the existing knowledge*

Consider now,

$$x + y = 2$$
$$2x + 2y = 4$$

This system can have infinitely many solutions. The rank of the matrix formed by the coefficients of unknowns is 1 (which is less than its minimum dimension).

1.1.7 Inverse

For A $\in \mathbb{R}^{m \times m}$, A matrix which reconstructs x in Ax=b is called the inverse of A. It is denoted by A^{-1}.

$$Ax = b$$
$$A^{-1}Ax = A^{-1}b$$
$$I_m x = A^{-1}b$$
$$x = A^{-1}b$$

where,
I_m is Identity Matrix of dimension m such that
$i_{ab} = 1$ if a = b
$i_{ab} = 0$ if a $\neq$ b
It is a diagonal matrix of all entries equal to unity.

1.1.8 Positive Definite

A matrix A is Positive Definite if
For $A \in \mathbb{R}^{l \times m}$

$$x^T Ax > 0 \qquad \forall x \in \mathbb{R}^m (x \neq 0) \tag{1.5}$$

1.1.9 Norm

Norms are one of the most important concepts of linear algebra. They are essentially yardsticks to measure vectors. It enables us to quantitatively decide on convergence and approx-

imations in iterative methods. There are 3 properties a norm must possess :

$$\begin{aligned} ||x|| &\geq 0 \\ ||x|| = 0 \qquad & iff\ x = 0 \\ ||x + y|| &\leq ||x|| + ||y|| \\ ||\alpha x|| &= |\alpha|||x|| \end{aligned}$$

In general,

$$||x||_p = (\sum_{i=1}^{n} |x|^p)^{1/p} \qquad (1 \leq p < \infty) \tag{1.6}$$

Most famous norm is the Euclidean 2 norm which is essentially the square root of the dot product of a vector with itself.

$$||x||_2 = \sqrt{(\sum_{i=1}^{n} |x|^2)} \tag{1.7}$$

1.1.10 Sparsity

Sparsity is one of the most important concepts of Linear Algebra. Knowing whether a matrix is sparse or not (often called Dense) completely changes the set of algorithms needed to tackle the problem at hand. Sparse matrices are matrices with very few nonzero elements. There is no threshold for calling a matrix sparse or dense and is more or less a choice left to the user. Sparse matrices can be found in many problems and especially the ones involving differential equations. Finite Difference Methods (FDM) and Finite Element Methods (FEM) often end up as problems of sparse systems. Because there are very few nonzero elements, very little energy is spent in operations such as matrix-vector, matrix-matrix operations or even in smartly designed exact methods. Although one can use methods used for dense systems to solve sparse systems, they will be sub-optimal. Many packages such as PARDISO are able to solve sparse systems much better than an optimized version of LAPACK will be able to. It is important to note that although a matrix maybe sparse, it can still be illconditioned making it susceptible to floating point problems and also sometimes, difficult to solve. Also, for a dense matrix, there is no choice but to store N^2 data entries in either row-major or column-major format. For sparse systems, data is stored much differently to enable both easier data access and calculation. It is quite possible to have all matrix computations and storage to be of the order of the nonzero elements of a matrix.

1.1.11 Eigenvectors and Eigenvalues

Eigenvalues and Eigenvectors are the fundamental parameters on which the convergence of any iterative method depends upon. For A $\in \mathbb{R}^{m \times m}$,

$$Av = \lambda v \tag{1.8}$$

λ is the eigenvalue whereas v is the eigenvector. Other than the mathematical importance of eigenvalues, it is necessary to understand the geometrical significance of this property.

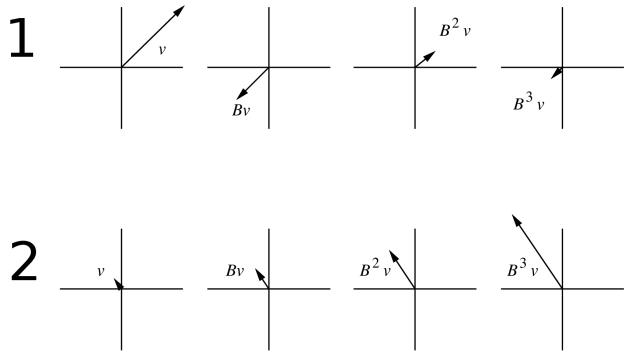

Figure 1.1: Eigenvector Scaling

The maximum of all eigenvectors is called the spectral radius

$$\rho(A) = \max |\lambda_i| \tag{1.9}$$

When a matrix is multiplied by its eigenvector, the eigenvector is *scaled* or in other words, the magnitude of the eigenvector is changed. As can be seen from the Equation (1.8) and Figure 1.1, if v is the eigenvector then Av still points in the direction of v but with a magnitude of λv.
Part 1 of Fig 1.1 shows the effect of eigenvector multiplication on its matrix when spectral radius is less than 1 and Part 2 when spectral radius is greater than 1.
Singular Values = $\sigma(A)$ = Eigenvalues of $A^T A$

1.1.12 Conditioning

Quoting Strang [28],

> *An error and a blunder are very different things. An error is a small mistake, probably unavoidable even by a perfect mathematician or a perfect computer. A blunder is much more serious, and larger by at least an order of magnitude. When the computer rounds off a number after 16 bits, that is an error, But when a*

problem is so excruciatingly sensitive that this roundoff error completely changes the solution, then almost certainly someone has committed a blunder.

Condition number = cond(A)

$$= \kappa(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)} \tag{1.10}$$

Let us consider our usual equation : Ax = b

$$\begin{aligned} Ax &= b \\ A(x + \delta x) &= b + \delta b \\ Ax + A\delta x &= b + \delta b \\ A\delta x &= \delta b \\ \delta x &= A^{-1}\delta b \end{aligned}$$

Thus, for a large change in x due to a small change in b, A^{-1} must be large. This is when A has either singular values (σ) spread across a large spectrum (thus causing $\kappa(A)$ to be large) or when A is near singular. The two conditions more often than not, do relate to each other.

An example (adapted from Strang [28])for it follows : Consider 2 matrices M and N.

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 1.0001 \end{bmatrix} \qquad N = \begin{bmatrix} 0.0001 & 1 \\ 1 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{1.11}$$

Solution :

$$x_m = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad\qquad x_n = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Now, if b were to be changed slightly,

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 1.0001 \end{bmatrix} \qquad N = \begin{bmatrix} 0.0001 & 1 \\ 1 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 1.1 \\ 1 \end{bmatrix} \tag{1.12}$$

Solution :

$$x_m = \begin{bmatrix} 1001.1 \\ -1000.0 \end{bmatrix} \qquad\qquad x_n = \begin{bmatrix} -0.1 \\ 1.1 \end{bmatrix}$$

Why the huge difference in the deviations in x after modifying b?

The answer lies in the condition number of the 2 matrices. The condition number of N is approximately 2.618 which is relatively small. The condition number of M is 40000 ! Thus,

a small perturbation in b can cause an immense change in value of the answer. Let us pause for a moment and understand the importance of this parameter.
If A is the matrix under consideration for a Finite Element problem which is modeled for a physical problem such as heat transfer; If during measurement, the observer made an error of 0.1 (assuming for simplicity that we have the same matrix), his error would be amplified by a factor of 1000 ! While most matrices we deal with in physical models are not ill-conditioned, it is not necessary that they will be well-conditioned either.
The most well-conditioned matrix would be the Identity Matrix. Its condition number is always unity.

1.2 Methods for Solving Linear Equations

Linear Algebra has evolved with the computing technology. The methods for solving linear systems can broadly be categorized into 2 types : Exact Methods and Iterative Methods. Any successful method essentially needs to find a vector x which satisfies $Ax = b$ for a known b. Needless to say, A must be non-singular and for our consideration, square. This thesis deals with methods which *solve* $Ax = b$ uniquely and not with approximations such as least squares.

Exact methods are those methods which attempt to use algorithms to yield an *exact* answer for the system in a finite number of steps. There are a few key points in this statement. Firstly, although these methods are exact methods, they too involve iterative or recursive computation. Further, the term *exact* is a misnomer when used in the context of computational methods since they are prone to floating point problems. These methods are O(N^3) in the time and have been traditionally used.
On the other hand are Iterative Methods[17, 24] which evolved in 1952 after the advent of the conjugate gradient method[1]. These attempt to solve the same systems in much less time than O(N^3) but do so approximately. Computationally speaking, such methods have a tolerance for the error and the iterations continue either till one reaches the maximum number of iterations set by the user or till convergence(to prevent stagnation of the method). The term convergence in this case would mean when the residue drops below the set tolerance.

1.2.1 Decompositions

Here are the three important factorizations, A = LU and A = QR and A = $U\Sigma V^T$:

1. Elimination reduces A to U by row operations using multipliers in L: A = LU = lower triangular times upper triangular. When solving linear systems, the user has no

[1]Although CG was introduced in 1952, it was treated as an exact method till it was revived later as an approximate solver

control over $\kappa(A)$, but the user also doesn't want to make the condition worse by a bad algorithm. Since elimination is the most frequently used algorithm in scientific computing, a lot of effort has been concentrated on doing it right. This algorithm is so important that the benchmarking of supercomputers takes place on the basis of how well it performs LU for Ax=b[8].

Often we reorder the rows of A, the main point is that small pivots are dangerous. To find the numbers that multiply rows, we divide by the pivots. Small pivots mean large multipliers in L. Then L (and probably U) are more ill-conditioned than A. This will increase the odds of taking a good system and ruining it with a bad algorithm. The simplest cure is to exchange rows by P, bringing the largest possible entry up into the pivot. Details of these and other methods can be found in any Numerical Linear Algebra textbook such as Trefethen and Bau [29].

2. Orthogonalization changes the columns of A to orthonormal columns in Q: A = QR = orthonormal columns times upper triangular. We are given an m by n matrix A with linearly independent columns $a_i, \ldots, a_n$. Its rank is n. Those n columns are a basis for the column space of A, but not necessarily a good basis. All computations are improved by switching from the a_i to orthonormal vectors $q_i, \ldots, q_n$. There are two important ways to go from A to Q : Gram-Schmidt and Householder's Reflections.

3. Singular Value Decomposition decomposes A as (rotation)(stretch)(rotation): A = $U\Sigma V^T$ = orthonormal columns X singular values X orthonormal rows. Since diagonalization involves eigenvalues, the matrices from A = QR will not be successful. Most square matrices A are diagonalized by their eigenvectors $x_1, \ldots, x_n$. If x is a combination $c_1x_1 + \ldots + c_nx_n$, then A multiplies each x_i by λ_i or $Ax = S\Lambda S^{-1}x$. Usually, the eigenvector matrix S is not orthogonal. Eigenvectors only meet at right angles when A is special (for example symmetric). If we want to diagonalize an ordinary A by orthogonal matrices, we need two different Q's. They are generally called U and V, so A = $U\Sigma V^T$

1.2.2 Exact Methods

Gauss Elimination

We assume that the reader is familiar with Gauss Elimination method and directly proceed to its analysis. Under what circumstances could the process break down? Something must go wrong in the singular case, and something might go wrong in the nonsingular case. The answer is: With a full set of n pivots, there is only one solution. The system is non singular, and it is solved by forward elimination and back-substitution. This is no different from

computing a full LU decomposition and then solving for x. But, in Gaussian Elimination, if a zero appears in a pivot position, elimination has to stopeither temporarily or permanently. The system might or might not be singular.
If the first coefficient is zero, in the upper left corner, the elimination from the other equations will be impossible. The same is true at every intermediate stage. Notice that a zero can appear in a pivot position, even if the original coefficient in that place was not zero. Roughly speaking, we do not know whether a zero will appear until we try, by actually going through the elimination process.
In many cases this problem can be cured, and elimination can proceed. Such a system still counts as nonsingular; it is only the algorithm that needs repair. In other cases a breakdown is unavoidable. Those incurable systems are singular, they have no solution or else infinitely many, and a full set of pivots cannot be found.

The asymptotic analysis of this method is equally important.
Ignoring the right-hand sides of the equations, the operations are of two kinds. We divide by the pivot to find out what multiple (say) of the pivot equation is to be subtracted. When we do this subtraction, we continually meet a multiply-subtract combination; the terms in the pivot equation are multiplied and then subtracted from another equation. Suppose we call each division, and each multiplication-subtraction, one operation. In column 1, it takes n operations for every zero we achieveone to find the multiple , and the other to find the new entries along the row. There are n - 1 rows underneath the first one, so the first stage of elimination needs n(n - 1) = n^2 - n operations. (Another approach to n^2 - n is this: All n^2 entries need to be changed, except the n in the first row.) Later stages are faster because the equations are shorter.
When the elimination is down to k equations, only k^2 - k operations are needed to clear out the column below the pivotby the same reasoning that applied to the first stage, when k equaled n. Altogether, the total number of operations is the sum of k^2 - k over all values of k from 1 to n:
Those are standard formulas for the sums of the first n numbers and the first n squares. Substituting n = 1 and n = 2 and n = 100 into the formula $\frac{(n^3-n)}{3}$, forward elimination can take no steps or two steps or about a third of a million steps:
If the size is doubled, and few of the coefficients are zero, the cost is multiplied by 8. Back-substitution is considerably faster. The last unknown is found in only one operation (a division by the last pivot). The second to last unknown requires two operations, and so on. Then the total for back-substitution is 1 + 2 + ... + n.
Forward elimination also acts on the right-hand side (subtracting the same multiples as on the left to maintain correct equations). This starts with n 1 subtractions of the first equation. Altogether the right-hand side is responsible for n2 operationsmuch less than the $\frac{n^3}{3}$ on the

left. The total for forward and backwark solving is of order n could not be solved with much fewer than $\frac{n^3}{3}$ multiplications.

There now exists a method that requires only C nlog_2 7 multiplications. It depends on a simple fact: Two combinations of two vectors in two-dimensional space would seem to take 8 multiplications, but they can be done in 7. That lowered the exponent from $log_2 8$, which is 3, to $log_2 7 \approx 2.8$. This discovery produced tremendous activity to find the smallest possible power of n. The exponent finally fell (at IBM) below 2.376. Fortunately for elimination, the constant C is so large and the coding is so awkward that the new method is largely (or entirely) of theoretical interest. The newest problem is the cost with many processors in parallel.

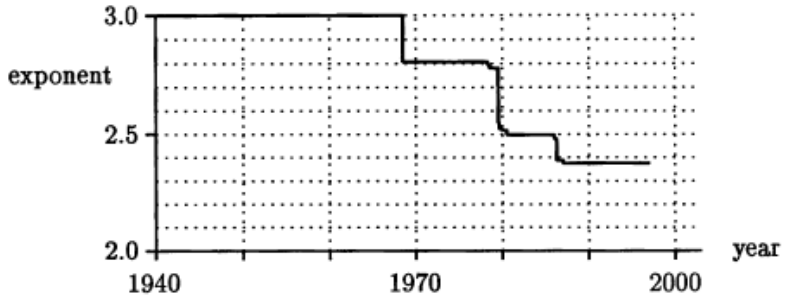

Figure 1.2: Reduction of Matrix Multiplication Exponent in History

Historical Bottleneck

The following table[29] gives a thumbnail history of matrix computations over the years:

1950: m = 20 (Wilkinson)

1965: m = 200 (Forsythe and Moler)

1980: m = 2000 (UNPACK)

1995: m = 20000 (LAPACK)

These numbers represent a rough approximation to what dimensions might have been considered "very large" for a dense, direct matrix computation at the indicated dates. In the mid-1960s, a matrix of dimension in the hundreds was large, stretching the limits of what could be calculated on available machines in a reasonable amount of time. Evidently, in the course of forty-five years, the dimensions of tractable matrix problems have increased by a factor of 10^3. This progress is impressive, but it pales beside the progress achieved by computer hardware in the same perioda speedup by a factor of 10^9, from flops to gigaflops. In the fact that 10^9 is the cube of 10^3, we see played out in history the $O(N^3)$ bottleneck of direct matrix algorithms. To put it another way, if matrix problems could be solved in $O(N^2)$ instead of $O(N^3)$ operations, some of the matrices being treated today might be 10

to 100 times larger. This is the aim, achieved for some matrices but not others, of matrix iterative methods.

A Perspective into $O(N^3)$

Let us, for the sake of perspective, attempt to analyse a sample matrix computation.

$$\begin{aligned} Let \qquad n &= 10^6, \\ Steps &= \frac{n^3}{3} \\ &= \frac{(10^6)^3}{3} \\ &= \frac{10^{18}}{3} \\ &\approx 3 \times 10^{17} \end{aligned}$$

Assuming a modest 100 GFlop machine,

$$\begin{aligned} T_{Required} &= \frac{3 \times 10^{17}}{10^{11}} \\ &= 3 \times 10^6 \quad seconds \\ &= \frac{3 \times 10^6}{86400} \quad days \\ &= 34.722 \quad days \end{aligned}$$

If a high end laptop were to start computing the solution for the system today, one would have to wait more than a month to get an answer whilst the laptop runs the computational 24 $\times$ 7. This is assuming that the processor can effectively compute 1 floating point operation per clock. While the theoretical maximum for processors today is 4 or above, usually, about 2 is possible. Accounting for the effect of instructions per second, the total time for execution will be divided by that amount.

1.2.3 Iterative Methods

Why Iterate?

Gaussian Elimination tends to be a stable and scalable mode of solving linear system. However, there are many reasons why iterative methods cannot be ignored. They are enumerated below:

- To solve a system of linear equations Ax = b when coefficient matrix A is large and sparse (i.e., contains many zero entries).
- If the order n of the matrix is so large that you cannot afford to spend about N^3 operations to solve the system by Gaussian elimination.
- No direct access to the matrix.

Basic Idea of Iterative Methods

To nd the solution x of the linear system $Ax = b$, all iterative methods require an initial guess $x_0 \in \mathbb{R}_n$ that approximate the true solution. So, a good guess x_0 begins the iterative method which can be taken by solving a much easier and similar problem.

Once we have x_0, we use it to generate a new guess x_1 which is used to guess x_2 and so on. Moreover, we attempt to improve this guess by reducing the error with a convenient and cheap approximation.

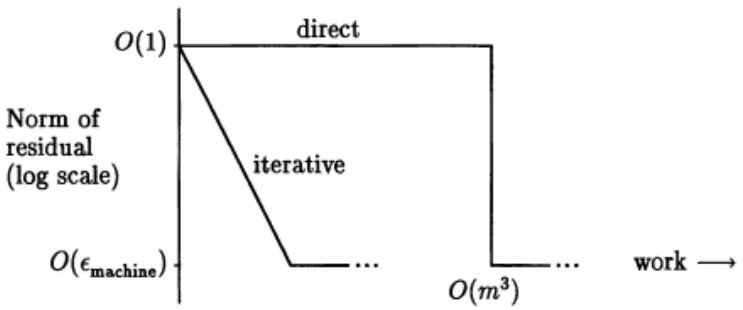

Figure 1.3: Mode of Solution Achievement

Let K be $n \times n$ be a simpler matrix which is similar to matrix A of the equation $Ax = b$. Then in step i + 1 : we get a relation as follows:

$$Kx_{i+1} = Kx_i + bAx_i \tag{1.13}$$

Let,

$$
\begin{aligned}
A \in \mathbb{R}^{n \times n} \quad and \quad \det A \neq 0 \\
x, b \in \mathbb{R}^n \\
b = Ax \\
b - Ax = 0 \\
b - Ax = Kx - Kx \\
b - Ax = Kx_{i+1} - Kx_i (Assumption) \\
Kx_{i+1} = Kx_i + b - Ax
\end{aligned}
$$

From above equation , we solve the new approximation x_{i+1} for the solution x of Ax = b. Depending upon the iterative method chosen, the value and structure of K changes.

Chapter 2

Literature Survey

2.1 Stationary Iterative Methods

In this and the following 2 chapters describe a summary of the common methods used for solving linear systems is presented. It serves as an introduction to the algorithms and their computational behaviour; please refer the book by Barrett et al. [3] for more details.

2.1.1 Jacobi Method

The Jacobi method is based on solving for every variable locally with respect to the other variables; one iteration of the method corresponds to solving for every variable once. The resulting method is easy to understand and implement, but convergence is slow.

Choose an initial guess $x^{(0)}$ to the solution x.
for $k = 1, 2, \ldots$
  for $i = 1, 2, \ldots, n$
    $\bar{x}_i = 0$
    for $j = 1, 2, \ldots, i-1, i+1, \ldots, n$
      $\bar{x}_i = \bar{x}_i + a_{i,j} x_j^{(k-1)}$
    end
    $\bar{x}_i = (b_i - \bar{x}_i)/a_{i,i}$
  end
  $x^{(k)} = \bar{x}$
  check convergence; continue if necessary
end

Figure 2.1: Detailed Jacobi Algorithm

2.1.2 Gauss-Seidel method

The Gauss-Seidel method is like the Jacobi method, except that it uses updated values as soon as they are available. In general, if the Jacobi method converges, the Gauss-Seidel method will converge faster than the Jacobi method, though still relatively slowly.

Choose an initial guess $x^{(0)}$ to the solution x.
for $k = 1, 2, \ldots$
  for $i = 1, 2, \ldots, n$
    $\sigma = 0$
    for $j = 1, 2, \ldots, i-1$
      $\sigma = \sigma + a_{i,j} x_j^{(k)}$
    end
    for $j = i+1, \ldots, n$
      $\sigma = \sigma + a_{i,j} x_j^{(k-1)}$
    end
    $x_i^{(k)} = (b_i - \sigma)/a_{i,i}$
  end
  check convergence; continue if necessary
end

Figure 2.2: Detailed Gauss-Seidel Algorithm

2.1.3 Successive Over Relaxation method (SOR method)

Successive Overrelaxation (SOR) can be derived from the Gauss-Seidel method by introducing an extrapolation parameter ω. For the optimal choice of ω, SOR may converge faster than Gauss-Seidel by upto an order of magnitude.

2.1.4 Symmetric SOR method (SSOR method)

Symmetric Successive Overrelaxation (SSOR) has no advantage over SOR as a standalone iterative method; however, it is useful as a preconditioner for nonstationary methods.

2.2 Non-Stationary Iterative Methods

- Conjugate Gradient method(CG method)
  The conjugate gradient method derives its name from the fact that it generates a sequence of conjugate (or orthogonal) vectors of the residuals of the iterates. They are also the gradients of a quadratic functional, the minimization of which is equivalent to solving the linear system. CG is an extremely effective method when the coefficient

Choose an initial guess $x^{(0)}$ to the solution x.
for $k = 1, 2, \ldots$
  for $i = 1, 2, \ldots, n$
    $\sigma = 0$
    for $j = 1, 2, \ldots, i-1$
      $\sigma = \sigma + a_{i,j} x_j^{(k)}$
    end
    for $j = i+1, \ldots, n$
      $\sigma = \sigma + a_{i,j} x_j^{(k-1)}$
    end
    $\sigma = (b_i - \sigma)/a_{i,i}$
    $x_i^{(k)} = x_i^{(k-1)} + \omega(\sigma - x_i^{(k-1)})$
  end
  check convergence; continue if necessary
end

Figure 2.3: Detailed SOR Algorithm

matrix is symmetric positive definite, since storage for only a limited number of vectors is required owing to the 3 term recurrences.

- General minimal Residual method(GMRES method)
  The Generalized Minimal Residual method computes a sequence of orthogonal vectors (like MINRES), and combines these through a least-squares solve and update. However, unlike MINRES (and CG) it requires storing the whole sequence, so that a large amount of storage is needed. For this reason, restarted versions of this method are used. In restarted versions, computation and storage costs are limited by specifying a fixed number of vectors to be generated. This number is termed as the restart number. This method is useful for general nonsymmetric matrices.

- Minimal Residual (MINRES method) and Symmetric LQ Method (SYMMLQ method)
  These methods are computational alternatives for CG for coefficient matrices that are symmetric but possibly indefinite. SYMMLQ will generate the same solution iterates as CG if the coefficient matrix is symmetric positive definite.

- Biconjugate Gradient method (BiCG method)
  The Biconjugate Gradient method generates 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, or bi-orthogonal. This method, like CG, uses limited storage. It is useful when the matrix is nonsymmetric and nonsingular; however, convergence may be irregular, and there is a possibility that the method will break down. BiCG requires a multiplication with the coefficient matrix and with its transpose at each iteration.

Choose an initial guess $x^{(0)}$ to the solution x.
Let $x^{(\frac{1}{2})} = x^{(0)}$.
for $k = 1, 2, \ldots$
  for $i = 1, 2, \ldots, n$
    $\sigma = 0$
    for $j = 1, 2, \ldots, i-1$
      $\sigma = \sigma + a_{i,j} x_j^{(k-\frac{1}{2})}$
    end
    for $j = i+1, \ldots, n$
      $\sigma = \sigma + a_{i,j} x_j^{(k-1)}$
    end
    $\sigma = (b_i - \sigma)/a_{i,i}$
    $x_i^{(k-\frac{1}{2})} = x_i^{(k-1)} + \omega(\sigma - x_i^{(k-1)})$
  end
  for $i = n, n-1, \ldots, 1$
    $\sigma = 0$
    for $j = 1, 2, \ldots, i-1$
      $\sigma = \sigma + a_{i,j} x_j^{(k-\frac{1}{2})}$
    end
    for $j = i+1, \ldots, n$
      $\sigma = \sigma + a_{i,j} x_j^{(k)}$
    end
    $x_i^{(k)} = x_i^{(k-\frac{1}{2})} + \omega(\sigma - x_i^{(k-\frac{1}{2})})$
  end
  check convergence; continue if necessary
end

Figure 2.4: Detailed SSOR Algorithm

- Biconjugate Gradient Stabilized (Bi-CGSTAB method)
  The Biconjugate Gradient Stabilized method is a variant of BiCG, like CGS, but using different updates for the A^T sequence in order to obtain smoother convergence than CGS.

- Conjugate Gradient Squared (CGS method)
  The Conjugate Gradient Squared method is a variant of BiCG that applies the updating operations for the A-sequence and the A^T sequences both to the same vectors. Ideally, this would double the convergence rate, but in practice convergence may be much more irregular than for BiCG. A practical advantage is that the method does not need the multiplications with the transpose of the coefficient matrix.

- Quasi-Minimal Residual (QMR method)
  The Quasi-Minimal Residual method applies a least-squares solve and update to the BiCG residuals, thereby smoothing out the irregular convergence behavior of BiCG. Also, QMR largely avoids the breakdown that can occur in BiCG. On the other hand,

it does not effect a true minimization of either the error or the residual, and while it converges smoothly, it does not essentially improve on the BiCG.

- Conjugate Gradients on the Normal Equations (CGNE and CGNR methods)
  These methods are based on the application of the CG method to one of two forms of the normal equations for Ax = b. CGNE solves the system $AA^Ty = b$ for y and then computes the solution $x = A^Ty$. CGNR solves $A^TAx = \tilde{b}$ for the solution vector x where $\tilde{b} = A^Tb$. When the coefficient matrix A is nonsymmetric and nonsingular, b the normal equations matrices AA^T and A^TA will be symmetric and positive definite, and hence CG can be applied. The convergence may be slow, since the spectrum of the normal equations matrices will be less favorable than the spectrum of A.

- Chebyshev Iteration
  The Chebyshev Iteration recursively determines polynomials with coefficients chosen to minimize the norm of the residual in a min-max sense. The coefficient matrix must be positive definite and knowledge of the extremal eigenvalues is required. This method has the advantage of requiring no inner products.

Matrix Type	$Ax = b$	$Ax = \lambda x$
$A \in SPD$	CG CGNE CGNR	Lanczos
$A \notin SPD$	GMRES BiCG Bi-CGSTAB MINRES SYMMLQ CGS QMR	Bi-Lanczos Arnoldi

2.2.1 Introduction to Krylov Subspaces

- The sequence of the form
  $v, Av, A^2v, A^3v, \ldots$ is called Krylov sequence.

- The Matrix of the form $K_k(A, v) = [v, Av, A^2v, A^3v, ..., A^{k-1}v]$ is called the k^{th} Krylov Matrix associated with A and v.

- The corresponding subspace $K_k(A, v) = span(v, Av, A^2v, A^3v, ..., A^{k1}v)$ is called the k^{th} Krylov Space associated with A and v.

Lemma Let A $\in \mathbb{R}^{n\times n}$ and v $\in \mathbb{R}^n$ with v $\neq 0$ then,

$$
\begin{aligned}
K_k(A,v) &\subset K_{k+1}(A,v) \\
AK_k(A,v) &\subset K_{k+1}(A,v) \\
If \neq 0, K_k(A,v) &= K_k(A,v) = K_k(A,v) \\
K_k(A,v) &= K(A-cI,v) \qquad \forall c \\
K_k(W^{-1}AW, W^{-1}v) &= W^{-1}K_k(A,v)
\end{aligned}
$$

Theorem The space $K_k(A,v)$ can be written in the form

K_k (A, v) = p(A)(v) : p is a polynomial of degree at most k - 1

Theorem The Krylov sequence $v, Av, A^2v, A^3v, \ldots$ terminates at l if l is the smallest integer s.t.

$$
\begin{aligned}
K_{l+i}(A,v) &= K_l(A,v) \\
dim[K_{l+i}(A,v)] &= dim[K_l(A,v)]
\end{aligned}
$$

2.2.2 Basic Idea of Krylov Subspace Methods

- In Krylov subspace methods, we keep all computed approximants x_i and combine them for a better solution. Dene the Krylov Subspace as

  $K_n(A, r_0) = span(r_0, Ar_0, A^2r_0, A^3r_0, ..., A^{n-1}r_0)$
  where,
  $r_0 = b - Ax_0$

- If the vectors $r_0, Ar_0, ..., A^{n-1}r_0$, spanning the Krylov subspace $K_n(A, r_0)$, are linearly independent, we note that

  $dim(K_n(A, r_0)) = n$

  As soon as $A^n r_0 \in K_n(A, r_0)$, we obtain that

  $K_{n+i}(A, r_0) = K_n(A, r_0) \qquad \forall i \in N$

and we get a linear combination

$0 = c_0 r_0 + c_1 A r_0 + ... + c_{n-1} A^{n-1} r_0 + c_n A^n r_0$

where at least $c_n = 0$ and $c_0 = 0$, otherwise the multiplication by A^{-1} would contradict the assumption that the vectors $r_0, Ar_0, ..., A^{n1} r_0$ are linearly independent. Indeed as $c_0 = 0$ we get;

$$A^{-1} r_0 = \frac{-1}{c_0} \sum_{i+1}^{n} c_i A^{i-1} r_0 \in K_n(A, r_0) \tag{2.1}$$

The smallest index n with

$n = dim[K_n(A, r_0)] = dim[K_{n+1}(A, r_0)]$

is called the grade of A with respect to r_0 and denoted by $\bar{r}(A, r_0)$

- The above argument also shows that there is no smaller Krylov subspace which contains $A^{-1} r_0$; therefore

  $\bar{r}(A, r_0) = min(n : A^{-1} r_0 \in K_n(A, r_0))$

- The degree of the minimum polynomial of A is an upper bound for $\bar{r}(A, r_0)$

- The basic idea of a Krylov solver is to construct a sequence of approximations getting closer to the exact solution x^* , such that

  $x_n \in x_0 + K_n(A, r_0)$

  where x_0 is the initial approximation,

  $r_i = b - A x_i$

  is the i_{th} residual and $e_i = x_i - x^*$ denotes the i_{th} error. After $\bar{r}$ (A, r_0) iterations the exact solution is contained in the current affine Krylov subspace, i.e.

$$\begin{aligned} x^* &\in x_0 + K_r(A, r_0) \\ x_0 + A^{-1} r_0 &= x_0 + A^{-1} \\ (Ax^* - Ax_0) &= x^* \end{aligned}$$

2.2.3 History of Krylov Subspace Methods

One of the most powerful tools for solving large and sparse systems of linear algebraic equations is a class of iterative methods called Krylov subspace methods [19, 30].Their significant

advantages like low memory requirements and good approximation properties make them very popular, and they are widely used in applications throughout science and engineering. The use of the Krylov subspaces in iterative methods for linear systems is even counted among the Top 10 algorithmic ideas of the 20th century. Convergence analysis of these methods is not only of a great theoretical importance but it can also help to answer practically relevant questions about improving the performance of these methods. As we show, the question about the convergence behavior leads to complicated nonlinear problems. Despite intense research efforts, these problems are not well understood in some cases.

- Since the early 1800s, researchers have been attracted towards iteration methods to approximate the solutions of large linear systems.
- The first iterative method for system of linear equations is due to Karl Friedrich Gauss. His least square method led him to a system of equations which is called Block wise Gauss-seidel method. The others who played an important role in the development of iterative methods, are:
  - Carl Gustav Jacobi
  - David Young
  - Richard Varga
  - Cornelius Lanczos
  - Walter Arnoldi
- From the beginning, scientists and researchers used to have both positive and negative feelings about the iterative methods because they are better in comparison of direct methods but the so-called stationary iterative methods like Jacobi, Gauss Seidel and SOR methods in general converge too slow to the solution of practical use. In the mid 1950s, the observation in Ewald Bodewigs textbook was that iteration methods were not useful, except when A approaches a diagonal matrix.
- Around the early 1950s the idea of Krylov subspace iteration was established by Cornelius Lanczos and Walter Arnoldi. Lanczos method was

  based on two mutually orthogonal vector sequences and his motivation came from eigenvalue problems. In that context, the most prominent feature of the method is that it reduces the original matrix to tridiagonal form. Lanczos later applied his method to solve linear systems, in particular the symmetric ones.
- In 1952, Magnus Hestenes and Eduard Stiefel presented a paper which is about the classical description of the conjugate gradient method for solving linear systems. Although

error-reduction properties are proved and experiments showing premature convergence are reported in this paper, the conjugate gradient method is presented here as a direct method, rather than an iterative method.

- This Hestenes/Stiefel method is closely related to a reduction of the Lanczos method to symmetric matrices, reducing the two mutually orthogonal sequences to one orthogonal sequence, but there is an important algorithmic dierence. Whereas Lanczos used three-term recurrences, the method by Hestenes and Stiefel uses coupled two-term recurrences. By combining the two two-term recurrences (eliminating the search directions) the Lanczos method is obtained.

- The Conjugate gradient method did not receive much recognition in its first 20 years because of the lack of exactness. Later researchers realized that this method is more fruitful to consider as truly iterative method. John Reid (in 1972) was the first one who pointed out in this direction. Moreover, it is now an important and cheaper iterative method for the symmetric case.

- Similarly, for the nonsymmetric case, GMRES method was proposed in 1986 by Youcef Saad and Martin Schultz. This method is based on Arnoldi algorithm and it is comparatively expensive. In this method, one has to store a full orthogonal basis for the Krylov subspace for nonsymmetric matrix A, which means the more iterations, the more basis vectors one must store. For many practical problems, GMRES can take hundreds of iterations, which makes a full GMRES unfeasable. This led to a search for cheaper near-optimal methods.

- Vance Faber and Thomas Manteuffels famous result showed that generally there is no possibility of constructing optimal solutions in the Krylov subspace for nonsymmetric matrix A by short recurrences, as in the conjugate gradients method,

- The generalization of conjugate gradients for nonsymmetric systems, i.e. Bi-CG, often displays an irregular convergence behavior, including a possible breakdown. Roland Freund and Noel Nachtigal gave an elegant remedy for both phenomena in their QMR method. BiCG and QMR have the disadvantage that they require an operation with A^T per iteration step. This additional operation does not lead to a further residual reduction.

- In the mid 1980s, Peter Sonneveld recognized that one can use the A^T operation for a further residual reduction through a minor modication to the Bi-CG scheme, almost without additional computational costs which is called CGS method. This method was often faster but signicantly more irregular.

- In 1992, Henk A. van der Vorst, showed that Bi-CG could be made faster and smoother, at almost no additional cost, with minimal residual steps.i.e. Bi-CGSTAB algorithm.
- Another class of acceleration methods that has been developed since around 1980, are the multigrid or multilevel methods. These methods apply to grid-oriented problems, and the idea is to work with coarse and ne grids.

2.3 GMRES

2.3.1 Arnoldi Method

Before we can discuss the GMRES algorithm, we first need to understand the dynamics of the Arnoldi method. The following are its steps :

1. Take a vector v $\in \mathbb{R}^m$ and orthonormalize it as

$$v_1 = \frac{v}{||v||}$$

2. For the kth step,

$$\begin{aligned}
\tilde{v}_{k+1} &= Av_k - \sum_{j=1}^{k} v_k h_{jk} \\
v_{k+1} &= \frac{\tilde{v}_{k+1}}{h_{k+1,k}} \\
h_{k+1,k} &= ||\tilde{v}_{k+1}||_2 \\
Av_k &= \sum_{j=1}^{k+1} v_j h_{jk} \\
AV_m &= V_{m+1} H_{m+1,m} \\
H_{m+1,m} &= \begin{bmatrix}
h_{11} & h_{12} & \dots & h_{1,m-1} & h_{1,m} \\
h_{21} & h_{22} & \dots & h_{2,m-1} & h_{2,m} \\
0 & h_{32} & \dots & h_{3,m-1} & h_{3,m} \\
\vdots & & \ddots & & \vdots \\
0 & \dots & \dots & h_{m,m-1} & h_{m,m} \\
0 & 0 & \dots & 0 & h_{m+1,m}
\end{bmatrix}
\end{aligned}$$

which is upper-hessenberg matrix.

2.3.2 GMRES : Theory

GMRES method is designed to solve nonsymmetric linear systems.This method is based on modified Gram-Schmidt procedure and was proposed in 1986 by Yousef Saad and M.Schultz.The Generalized Minimum Residual (GMRES) method is designed to solve nonsymmetric linear systems. The most popular form of GMRES is based on the modified Gram-Schmidt procedure, and uses restarts to control storage requirements. If no restarts are used, GMRES (like any orthogonalizing Krylov-subspace method) will converge in no more than n steps (assuming exact arithmetic). Of course this is of no practical value when n is large; moreover, the storage and computational requirements in the absence of restarts are prohibitive. Indeed, the crucial element for successful application of GMRES(m) revolves around the decision of when to restart; that is, the choice of m. Unfortunately, there exist examples for which the method stagnates and convergence takes place only at the nth step. For such systems, any choice of m less than n fails to converge.

Saad and Schultz have proven several useful results. In particular, they show that if the coefficient matrix A is real and nearly positive definite, then a reasonable value for m may be selected. Implications of the choice of m are discussed below.

$$
\begin{aligned}
x_m &= x_0 + V_m y_m \quad s.t. \\
y_m &\in \mathbb{R}^m \\
||b - Ax_m|| &= ||r_m|| \\
&= ||b - A(x_0 + V_m y_m)|| \\
&= ||b - Ax_0 - AV_m y_m|| \\
&= ||r_0 - V_m H_{m+1,m} y_m|| \\
&= ||\beta v_1 - V_m H_{m+1,m} y_m|| \\
&= ||V_{m+1}(\beta e_1 - V_m H_{m+1,m} y_m)|| \\
&= ||\beta e_1 - V_m H_{m+1,m} y_m||
\end{aligned}
$$

2.3.3 GMRES : Algorithm

$$
\begin{aligned}
r_0 = b - Ax_0 \qquad \beta = r_0 \qquad V_1 := r(0)\beta \\
H_{m+1,m} = h_{ij_{(1\leq i\leq m+1),(1\leq j\leq m)}} \qquad H_{m+1,m} = 0 \\
w_j := Av_j \\
h_{ij} := w_j^T v_i \\
w_j := w_j - h_{ij}v_i \\
h_{j+1,j} := ||w_j||_2 \\
v_{j+1} = \frac{w_j}{h^{j+1,j}} \\
Compute \qquad \beta e_1 - V_{m+1}H_{m+1,m}y_m \\
x_m = x_0 + V_m y_m
\end{aligned}
$$

The major drawback to GMRES is that the amount of work and storage required per iteration rises linearly with the iteration count. Unless one is fortunate enough to obtain extremely fast convergence, the cost will rapidly become prohibitive. The usual way to overcome this limitation is by restarting the iteration. After a chosen number (m) of iterations, the accumulated data are cleared and the intermediate results are used as the initial data for the next m iterations. This procedure is repeated until convergence is achieved. The difficulty is in choosing an appropriate value for m. If m is too small, GMRES(m) may be slow to converge, or fail to converge entirely. A value of m that is larger than necessary involves excessive work (and uses more storage). Unfortunately, there are no definite rules governing the choice of m-choosing when to restart is a matter of experience.

2.4 Conjugate Gradients

CG is the most popular iterative method for solving large systems of linear equations. CG is effective for systems of the form Ax = b where x is an unknown vector, b is a known vector, and A is a known, square, symmetric, positive-definite (or positive-indefinite) matrix.[26] Iterative methods like CG are suited for use with sparse matrices. If matrix is dense, the generally accepted course of action is probably to factor and solve the equation by backsubstitution. The time spent factoring a dense is roughly equivalent to the time spent solving the system iteratively; and once is factored, the system can be backsolved quickly for multiple values of b. Having said that, there are many reasons why iterative methods may be chosen instead of direct methods. These have been enumerated in the later sections.

$x^{(0)}$ is an initial guess
for $j = 1, 2, ...$
  Solve r from $Mr = b - Ax^{(0)}$
  $v^{(1)} = r/\|r\|_2$
  $s := \|r\|_2 e_1$
  for $i = 1, 2, ..., m$
    Solve w from $Mw = Av^{(i)}$
    for $k = 1, ..., i$
      $h_{k,i} = (w, v^{(k)})$
      $w = w - h_{k,i}v^{(k)}$
    end
    $h_{i+1,i} = \|w\|_2$
    $v^{(i+1)} = w/h_{i+1,i}$
    apply $J_1, ..., J_{i-1}$ on $(h_{1,i}, ..., h_{i+1,i})$
    construct J_i, acting on ith and $(i+1)$st component
    of $h_{.,i}$, such that $(i+1)$st component of $J_i h_{.,i}$ is 0
    $s := J_i s$
    if $s(i+1)$ is small enough then (UPDATE($\tilde{x}, i$) and quit)
  end
  UPDATE($\tilde{x}, m$)
end

In this scheme UPDATE($\tilde{x}, i$)
replaces the following computations:

Compute y as the solution of $Hy = \tilde{s}$, in which
the upper $i \times i$ triangular part of H has $h_{i,j}$ as
its elements (in least squares sense if H is singular),
$\tilde{s}$ represents the first i components of s
$\tilde{x} = x^{(0)} + y_1 v^{(1)} + y_2 v^{(2)} + ... + y_i v^{(i)}$
$s^{(i+1)} = \|b - A\tilde{x}\|_2$
if $\tilde{x}$ is an accurate enough approximation then quit
else $x^{(0)} = \tilde{x}$

Figure 2.5: Detailed GMRES Algorithm

2.4.1 Quadratic Form

The quadratic form of a matrix is given by :

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{2.2}$$

where is A is the matrix, x and b are vectors, and c is a scalar constant. If A is symmetric and positive-definite,

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \frac{\partial}{\partial x_3} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix} \tag{2.3}$$

If A is symmetric,

$$f'(x) = Ax - b \tag{2.4}$$

For minima condition,

$$f'(x) = 0 \tag{2.5}$$

$$Ax = b \tag{2.6}$$

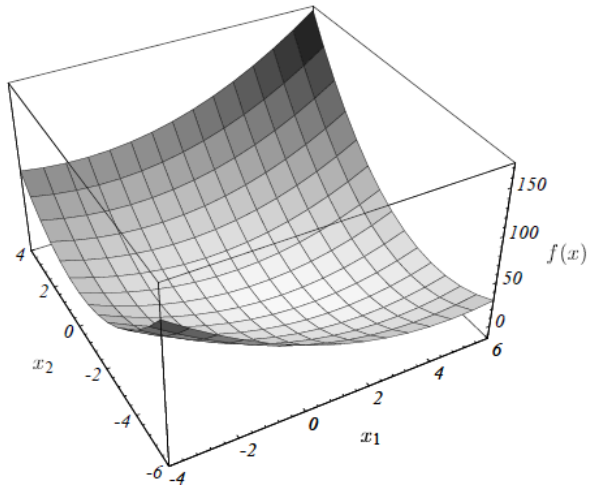

Figure 2.6: Quadratic Form Graph

2.4.2 Steepest Descent

Before any exposition is made about the steepest descent problem or conjugate gradients, a few key definitions must be introduced.
error in i^{th} iteration $= e_i = x_i - x$
residue in i^{th} iteration $= r_i = b - Ax_i$
It can be seen that $r_i = -Ae_i$
For steepest descent, the algorithm used it simply to start with an initial guess and move along a certain direction till we reach a point that minimizes the norm of the residual. There are few questions that we need to answer before we can write out the algorithm. As mentioned above, the i^{th} residue is the difference between the actual value of RHS and the value obtained using the i^{th} approximation of x. Also, $r_i = -f'(x_i)$. From this equation, we can form a generalization : The direction of steepest descent is same as the residue and this answers our question. Now that we have decided the direction to move in to find the value of x_{i+1}; now we must decide how far we must move.

$$x_{i+1} = x_i + \alpha \times r_i \tag{2.7}$$
$$r_{i+1}{}^T ri = 0 \tag{2.8}$$
$$(b - Ax_{i+1})^T r_i = 0 \tag{2.9}$$
$$(b - A(x_i + \alpha r_i)^T r_i = 0 \tag{2.10}$$
$$(b - Ax_i)^T r_i - \alpha (Ar_i)^T r_i = 0 \tag{2.11}$$
$$(b - Ax_i)^T r_i = \alpha (Ar_i)^T r_i \tag{2.12}$$
$$r_i^T r_i = \alpha r_i^T (Ar_i) \tag{2.13}$$
$$\alpha = \frac{r_i^T r_i}{r_i^T (Ar_i)} \tag{2.14}$$

Thus, the algorithm is :

$$r_i = b - Ax_i$$
$$\alpha = \frac{r_i^T r_i}{r_i^T (Ar_i)}$$
$$x_{i+1} = x_i + \alpha \times r_i$$

2.4.3 Convergence of Steepest Descent

It can be shown that the convergence of Steepest Descent is given by

$$||e_i||_A \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^i ||e_0||_A \tag{2.15}$$

where, e_i is the error in the i^{th} iteration
$||.||_A$ is the A norm of the vector
This formula provides with a proof for the intuition that condition number directly affects the error. Higher condition numbers necessarily mean worse convergence.

2.4.4 Conjugate Gradient

As is evident from the figure, the steepest descent algorithm takes a step in the right direction, turns, proceeds in an orthogonal direction and returns back to the original one. Even without any mathematical analysis, one can see that we can do better. Ideally, the best algorithm would be the one which takes 1 step per direction. In other words, if it selects a direction, it does a line search, finds the optimal point, proceeds to it and never proceeds on this direction again. This is exactly what conjugate gradients does. Now,

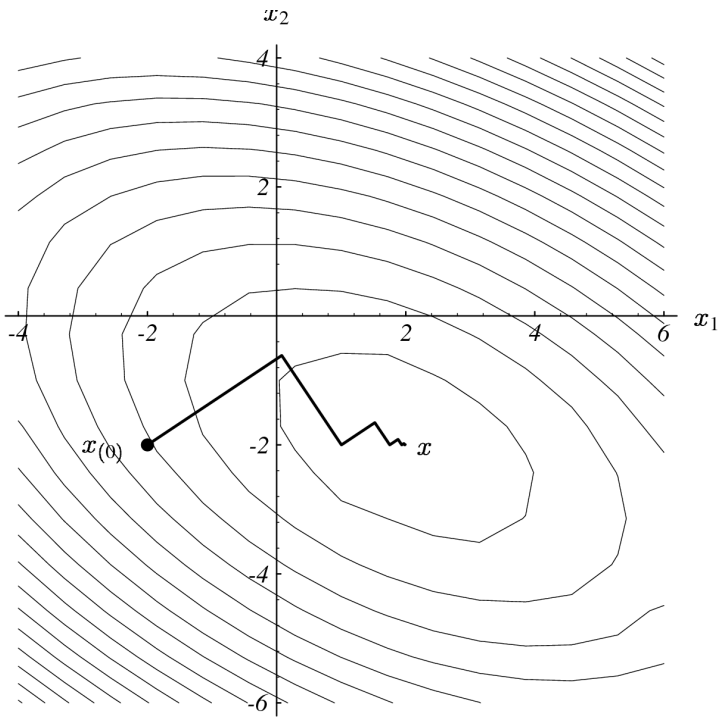

Figure 2.7: Convergence of Steepest Descent for [2,-2]

$$
\begin{aligned}
x_{i+1} &= x_i + \alpha \times d_i \\
d_i^T e_{i+1} &= 0 \\
d_i^T (e_i + \alpha d_i) &= 0 \\
\alpha &= -\frac{d_i^T e_i}{d_i^T d_i}
\end{aligned}
$$

This exercise might seem futile since α cannot be calculated till e is known and if e was known, there was no need to do anything ! However, this derivation leads to a solution. The trick is to make the 2 subsequent directions A orthogonal instead of just orthogonal.
That is to say,

$$d_i^T A d_j = 0 \tag{2.16}$$

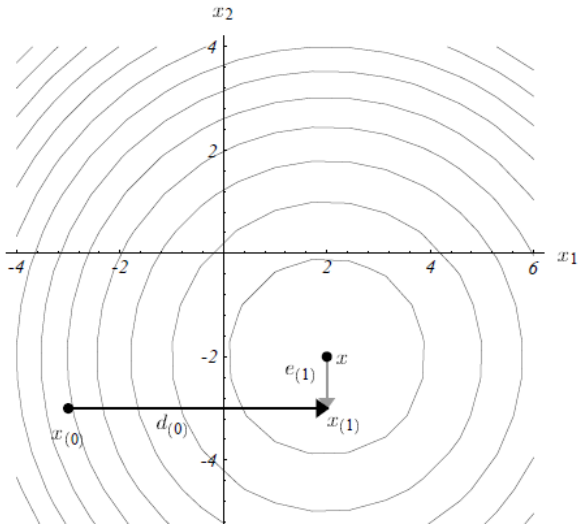

Figure 2.8: Intuitive Idea of CG

Proceeding similar to the case of Steepest Descent,
We can derive the algorithm for CG :

$$
\begin{aligned}
d_0 &= r_0 = b - Ax_0 \\
\alpha &= \frac{r_i^T r_i}{d_i^T (Ad_i)} \\
x_{i+1} &= x_i + \alpha \times d_i \\
r_{i+1} &= r_i - \alpha A d_i \\
\beta_{i+1} &= \frac{r_{i+1}^T r_{i+1}}{r_i^T (r_i)} \\
d_{i+1} &= r_{i+1} + \beta_{i+1} d_i
\end{aligned}
$$

2.5 CGNE and CGNR

The conjugate gradient method can be applied on the normal equations. The CGNE and CGNR methods are variants of this approach that are the simplest methods for nonsymmetric or indefinite systems. Since other methods for such systems are in general rather more complicated than the conjugate gradient method, transforming the system to a symmetric definite one and then applying the conjugate gradient method is attractive for its coding simplicity.

If a system of linear equations Ax=b has a nonsymmetric, possibly indefinite (but nonsingular) coefficient matrix, one obvious attempt at a solution is to apply the conjugate gradient

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = {r^{(i-1)}}^T z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/{p^{(i)}}^T q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence; continue if necessary
end

Figure 2.9: Detailed CG Algorithm

method to a related symmetric positive definite system $A^T Ax = A^T b$. While this approach is easy to understand and code, the convergence speed of the conjugate gradient method now depends on the square of the condition number of the original coefficient matrix. Thus the rate of convergence of the conjugate gradient procedure on the normal equations may be slow.

Several proposals have been made to improve the numerical stability of this method. The best known is by Paige and Saunders (1982) and is based upon applying the Lanczos method to the auxiliary system

A clever execution of this scheme delivers the factors L and U of the LU decomposition of the tridiagonal matrix that would have been computed by carrying out the Lanczos procedure with $A^T A$.

Another means for improving the numerical stability of this normal equations approach is suggested by Björck and Elfving (1979). The observation that the matrix $A^T A$ is used in the construction of the iteration coefficients through an inner product like $(pA^T A, Ap)$ leads to the suggestion that such an inner product be replaced by (Ap, Ap).

2.6 BiCG

The Conjugate Gradient method is not suitable for nonsymmetric systems because the residual vectors cannot be made orthogonal with short recurrences. The GMRES method retains orthogonality of the residuals by using long recurrences, at the cost of a larger storage demand. The BiConjugate Gradient method takes another approach, replacing the orthogonal sequence of residuals by two mutually orthogonal sequences, at the price of no longer pro-

viding a minimization. The update relations for residuals in the Conjugate Gradient method are augmented in the BiConjugate Gradient method by relations that are similar but based on A^T instead of A. Thus we update two sequences of residuals

$$
\begin{aligned}
r_i &= r_{i-1} - \alpha_i A p_i \\
\tilde{r}_i &= \tilde{r}_{i-1} - \alpha_i A \tilde{p}_i \\
p_i &= r_{i1} + \beta_{i-1} p_{i-1} \\
\tilde{p}_i &= \tilde{r}_{i1} + \beta_{i-1} \tilde{p}_{i-1} \\
\alpha_i &= \frac{\tilde{r}_{i-1}^T r_{i-1}}{\tilde{p}_i^T A p_i} \\
\beta &= \frac{\tilde{r}_i^T r_i}{\tilde{r}_{i-1}^T r_{i-1}} \\
\tilde{r}_i^T r_j = \tilde{p}_i A p_j &= 0 \quad for \quad i \neq j.
\end{aligned}
$$

Few theoretical results are known about the convergence of BiCG. For symmetric positive definite systems the method delivers the same results as CG, but at twice the cost per iteration. For nonsymmetric matrices it has been shown that in phases of the process where there is significant reduction of the norm of the residual, the method is more or less comparable to full GMRES (in terms of numbers of iterations). In practice this is often confirmed, but it is also observed that the convergence behavior may be quite irregular, and the method. Breakdown may occur due to failed LU factorization or when z & r turn out to be orthogonal. Sometimes, breakdown or near-breakdown situations can be satisfactorily avoided by a restart at the iteration step immediately before the (near-) breakdown step. Another possibility is to switch to a more robust (but possibly more expensive) method, like GMRES.In a parallel environment, the two matrix-vector products can theoretically be performed simultaneously; however, in a distributed-memory environment, there will be extra communication costs associated with one of the two matrix-vector products, depending upon the storage scheme for A. A duplicate copy of the matrix will alleviate this problem, at the cost of doubling the storage requirements for the matrix. Care must also be exercised in choosing the preconditioner, since similar problems arise during the two solves involving the preconditioning matrix. It is difficult to make a fair comparison between GMRES and BiCG. GMRES really minimizes a residual, but at the cost of increasing work for keeping all residuals orthogonal and increasing demands for memory space. BiCG does not minimize a residual, but often its accuracy is comparable to GMRES, at the cost of twice the amount of matrix vector products per iteration step. However, the generation of the basis vectors is relatively cheap and the memory requirements are modest. Several variants of BiCG have been proposed that increase the effectiveness of this class of methods in certain circumstances.

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$.
Choose $\tilde{r}^{(0)}$ (for example, $\tilde{r}^{(0)} = r^{(0)}$).
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  solve $M^T\tilde{z}^{(i-1)} = \tilde{r}^{(i-1)}$
  $\rho_{i-1} = {z^{(i-1)}}^T \tilde{r}^{(i-1)}$
  if $\rho_{i-1} = 0$, **method fails**
  if $i = 1$
    $p^{(i)} = z^{(i-1)}$
    $\tilde{p}^{(i)} = \tilde{z}^{(i-1)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)}$
    $\tilde{p}^{(i)} = \tilde{z}^{(i-1)} + \beta_{i-1}\tilde{p}^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\tilde{q}^{(i)} = A^T\tilde{p}^{(i)}$
  $\alpha_i = \rho_{i-1}/{\tilde{p}^{(i)}}^T q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  $\tilde{r}^{(i)} = \tilde{r}^{(i-1)} - \alpha_i \tilde{q}^{(i)}$
  check convergence; continue if necessary
end

Figure 2.10: Detailed BiCG Algorithm

2.7 QMR

The BiConjugate Gradient method often displays rather irregular convergence behavior. Moreover, the implicit LU decomposition of the reduced tridiagonal system may not exist, resulting in breakdown of the algorithm. A related algorithm, the Quasi-Minimal Residual method of Freund and Nachtigal [11] attempts to overcome these problems. The main idea behind this algorithm is to solve the reduced tridiagonal system in a least squares sense, similar to the approach followed in GMRES. Since the constructed basis for the Krylov subspace is bi-orthogonal, rather than orthogonal as in GMRES, the obtained solution is viewed as a quasi-minimal residual solution, which explains the name. Additionally, QMR uses look-ahead techniques to avoid breakdowns in the underlying Lanczos process, which makes it more robust than BiCG. The convergence behavior of QMR is typically much smoother than for BiCG. Freund and Nachtigal present quite general error bounds which show that QMR may be expected to converge about as fast as GMRES. From a relation between the residuals in BiCG and QMR one may deduce that at phases in the iteration process where BiCG makes significant progress, QMR has arrived at about the same approximation for x. On the other hand, when BiCG makes no progress at all, QMR may still show slow convergence. The look-ahead steps in the QMR method prevent breakdown in all cases but the so-called incurable breakdown, where no number of look-ahead steps would yield a next iterate.

2.8 Computational Aspect of Algorithms

Efficient solution of a linear system is largely a function of the proper choice of iterative method. However, to obtain good performance, consideration must also be given to the computational kernels of the method and how efficiently they can be executed on the target architecture. This point is of particular importance on parallel architectures.

Iterative methods are very different from direct methods in this respect. The performance of direct methods, both for dense and sparse systems, is largely that of the factorization of the matrix. This operation is absent in iterative methods (although preconditioners may require a setup phase), and with it, iterative methods lack dense matrix suboperations. Since such operations can be executed at very high efficiency on most current computer architectures, we expect a lower flop rate for iterative than for direct methods. Furthermore, the basic operations in iterative methods often use indirect addressing, depending on the data structure. Such operations also have a relatively low efficiency of execution. However, this lower efficiency of execution does not imply anything about the total solution time for a given system. Furthermore, iterative methods are usually simpler to implement than direct methods, and since no full factorization has to be stored, they can handle much larger sytems than direct methods. Also, one of the most important attributes regarding the iterative methods is that the matrix need not be provided in explicit form; a routine which computes a matrix-vector multiplication given a known vector will suffice.

In this section we summarize for each method There are 2 criteria when evaluating iterative methods for a given problem:

- **Matrix properties:** Not every method will work on every problem type, so knowledge of matrix properties is the main criterion for selecting an iterative method.
- **Computational Kernels:** Different methods involve different kernels, and depending on the problem or target computer architecture this may rule out certain methods or might guarantee a good performance for some others.

1. Jacobi Method
    - This method is extremely easy to use and implement, but unless the matrix is strongly diagonally dominant, this method is probably best only considered as an introduction to iterative methods or as a preconditioner in a nonstationary method.
    - Owing to only local dependence, it is fairly trivial to parallelize.
2. Gauss-Seidel Method

- It is typically faster convergence than Jacobi, but in general not competitive with the nonstationary methods.
- It is applicable to strictly diagonally dominant, or symmetric positive definite matrices.
- Parallelization properties depend on structure of the coefficient matrix. Different orderings of the unknowns have different degrees of parallelism; multicolor orderings may give almost full parallelism.
- This is a special case of the SOR method, obtained by choosing $\omega = 1$.

3. Successive Over-Relaxation (SOR)

   - This method accelerates convergence of Gauss-Seidel ($\omega > 1$, over-relaxation); may yield conver gence when Gauss-Seidel fails ($0 < \omega < 1$, under-relaxation).
   - Its speed of convergence depends critically on ω; the optimal value for ω may be estimated from the spectral radius of the Jacobi iteration matrix under certain conditions.
   - Its parallelization properties are the same as those of the Gauss-Seidel method.

4. Conjugate Gradient (CG)

   - It is applicable to symmetric positive definite systems.
   - Its speed of convergence depends on the condition number; if extremal eigenvalues are well-separated then superlinear convergence behavior can result. As proved in Shewchuk [26],

$$||(e_i)||_A \leq 2. \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^i ||e_0||_A$$

   - Inner products act as synchronization points in a parallel environment.
   - The parallel properties are largely independent of the coefficient matrix, but depend strongly on the structure the preconditioner.

5. Generalized Minimal Residual (GMRES)

   - It is applicable to general matrices i.e. nonsymmetric, indefinite.
   - GMRES leads to the smallest residual for a fixed number of iteration steps, but these steps become increasingly expensive because of linear storage requirement.
   - In order to limit the increasing storage requirments and work per iteration step, restart ing is necessary. The optimal value the restart parameter has been a topic of great interest.

- GMRES requires only matrix-vector products with the coefficient matrix.
- The number of inner products grows linearly with the iteration number, up to the restart point. In an implementation based on a simple Gram-Schmidt process the inner products are independent, so together they imply only one synchronization point. A more stable implementation based on modified Gram-Schmidt orthogonalization has one synchronization point per inner product.

6. Biconjugate Gradient (BiCG)

   - Similar to GMRES, it is also applicable to general matrices.
   - It requires matrix-vector products with the coefficient matrix and its transpose. This automatically dis qualifies the method for cases where the matrix is only given implicitly as an operator, since usually no corresponding transpose operator is available in such cases.
   - Its parallelization properties are similar to those for CG; the two matrix vector products (as well as the preconditioning steps) are independent, so they can be done in parallel, or their communication stages can be packaged.

7. Quasi-Minimal Residual (QMR)

   - It is applicable to general matrices.
   - It was designed to avoid the irregular convergence behavior of BiCG, it avoids one of the two breakdown situations of BiCG.
   - If BiCG makes significant progress in one iteration step, then QMR delivers about the same result at the same step. But when BiCG temporarily stagnates or diverges, QMR may still further reduce the residual, albeit very slowly.
   - Computational costs per iteration are similar to BiCG, but slightly higher. The method requires the transpose matrix-vector product. QMR is disqualified in case of implicit matrix description without the presence of transpose operator as well.
   - Its parallelization properties are as for BiCG.

8. Conjugate Gradient Squared (CGS)

   - It is applicable to general matrices.
   - While it converges (or diverges) typically about twice as fast as BiCG, its convergence behavior is often quite irregular, which may lead to a loss of accuracy in the updated residual.

- One of its biggest disadvantage is that it tends to diverge if the starting guess is close to the solution.
- Computational costs per iteration are similar to BiCG, but the method doesnt require the transpose matrix.
- Unlike BiCG, the two matrix-vector products are not independent, so the number of synchronization points in a parallel environment is larger. This increases the wait time per iteration for concurrent threads.

9. Biconjugate Gradient Stabilized (Bi-CGSTAB)

    - It is applicable to general matrices.
    - Its computational costs per iteration are similar to BiCG and CGS, but the method doesnt require the transpose matrix.
    - An alternative for CGS that avoids the irregular convergence patterns of CGS while maintaining about the same speed of convergence; as a result less loss of accuracy is observed in the updated residual.

10. Chebyshev Iteration

    - It is applicable to general matrices.
    - This method requires some explicit knowledge of the spectrum (or field of values); in the symmetric case the iteration parameters are easily obtained from the two extremal eigenvalues, which can be estimated either directly from the matrix, or from applying a few iterations of the Conjugate Gradient Method.
    - The computational structure is similar to that of CG, but there are no synchronization points.
    - The Adaptive Chebyshev method can be used in combination with methods as CG or GMRES, to continue the iteration once suitable bounds on the spectrum have been obtained from these methods.

Selecting the *best* method for a given class of problems is largely a matter of trial and error. It also depends on how much storage one has available (GMRES), on the availability of A^T (BiCG and QMR), and on how expensive the matrix vector products are in comparison to DAXPYs and inner products. If these matrix vector products are relatively expensive, and if sufficient storage is available then it may be attractive to use GMRES and delay restarting as much as possible. However, this is not a foolproof way of ensuring convergence, in the subsequent chapters, the results of numerical experiments conducted on these methods is described.

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
$\tilde{v}^{(1)} = r^{(0)}$; **solve** $M_1 y = \tilde{v}^{(1)}$; $\rho_1 = \|y\|_2$
Choose $\tilde{w}^{(1)}$, for example $\tilde{w}^{(1)} = r^{(0)}$
solve $M_2^t z = \tilde{w}^{(1)}$; $\xi_1 = \|z\|_2$
$\gamma_0 = 1; \eta_0 = -1$
for $i = 1, 2, \ldots$
  if $\rho_i = 0$ or $\xi_i = 0$ **method fails**
  $v^{(i)} = \tilde{v}^{(i)}/\rho_i$; $y = y/\rho_i$
  $w^{(i)} = \tilde{w}^{(i)}/\xi_i$; $z = z/\xi_i$
  $\delta_i = z^T y$; **if** $\delta_i = 0$ **method fails**
  solve $M_2 \tilde{y} = y$
  solve $M_1^T \tilde{z} = z$
  if $i = 1$
    $p^{(1)} = \tilde{y}$; $q^{(1)} = \tilde{z}$
    else
    $p^{(i)} = \tilde{y} - (\xi_i \delta_i / \epsilon_{i-1}) p^{(i-1)}$
    $q^{(i)} = \tilde{z} - (\rho_i \delta_i / \epsilon_{i-1}) q^{(i-1)}$
  endif
  $\tilde{p} = A p^{(i)}$
  $\epsilon_i = {q^{(i)}}^T \tilde{p}$; **if** $\epsilon_i = 0$ **method fails**
  $\beta_i = \epsilon_i / \delta_i$; **if** $\beta_i = 0$ **method fails**
  $\tilde{v}^{(i+1)} = \tilde{p} - \beta_i v^{(i)}$
  solve $M_1 y = \tilde{v}^{(i+1)}$
  $\rho_{i+1} = \|y\|_2$
  $\tilde{w}^{(i+1)} = A^T q^{(i)} - \beta_i w^{(i)}$
  solve $M_2^T z = \tilde{w}^{(i+1)}$
  $\xi_{i+1} = \|z\|_2$
  $\theta_i = \rho_{i+1} / (\gamma_{i-1} |\beta_i|)$; $\gamma_i = 1/\sqrt{1 + \theta_i^2}$; **if** $\gamma_i = 0$ **method fails**
  $\eta_i = -\eta_{i-1} \rho_i \gamma_i^2 / (\beta_i \gamma_{i-1}^2)$
  if $i = 1$
    $d^{(1)} = \eta_1 p^{(1)}$; $s^{(1)} = \eta_1 \tilde{p}$
  else
    $d^{(i)} = \eta_i p^{(i)} + (\theta_{i-1} \gamma_i)^2 d^{(i-1)}$
    $s^{(i)} = \eta_i \tilde{p} + (\theta_{i-1} \gamma_i)^2 s^{(i-1)}$
  endif
  $x^{(i)} = x^{(i-1)} + d^{(i)}$
  $r^{(i)} = r^{(i-1)} - s^{(i)}$
  check convergence; continue if necessary
end

Figure 2.11: Detailed QMR Algorithm

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
Choose $\tilde{r}$ (for example, $\tilde{r} = r^{(0)}$)
for $i = 1, 2, \ldots$
  $\rho_{i-1} = \tilde{r}^T r^{(i-1)}$
  if $\rho_{i-1} = 0$ **method fails**
  if $i = 1$
    $u^{(1)} = r^{(0)}$
    $p^{(1)} = u^{(1)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $u^{(i)} = r^{(i-1)} + \beta_{i-1} q^{(i-1)}$
    $p^{(i)} = u^{(i)} + \beta_{i-1}(q^{(i-1)} + \beta_{i-1} p^{(i-1)})$
  endif
  solve $M\hat{p} = p^{(i)}$
  $\hat{v} = A\hat{p}$
  $\alpha_i = \rho_{i-1}/\tilde{r}^T \hat{v}$
  $q^{(i)} = u^{(i)} - \alpha_i \hat{v}$
  solve $M\hat{u} = u^{(i)} + q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i \hat{u}$
  $\hat{q} = A\hat{u}$
  $r^{(i)} = r^{(i-1)} - \alpha_i \hat{q}$
  check convergence; continue if necessary
end

Figure 2.12: Detailed CGS Algorithm

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
Choose $\tilde{r}$ (for example, $\tilde{r} = r^{(0)}$)
for $i = 1, 2, \ldots$
  $\rho_{i-1} = \tilde{r}^T r^{(i-1)}$
  if $\rho_{i-1} = 0$ **method fails**
  if $i = 1$
    $p^{(i)} = r^{(i-1)}$
  else
    $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
    $p^{(i)} = r^{(i-1)} + \beta_{i-1}(p^{(i-1)} - \omega_{i-1} v^{(i-1)})$
  endif
  solve $M\hat{p} = p^{(i)}$
  $v^{(i)} = A\hat{p}$
  $\alpha_i = \rho_{i-1}/\tilde{r}^T v^{(i)}$
  $s = r^{(i-1)} - \alpha_i v^{(i)}$
  check norm of s; if small enough: set $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p}$ and stop
  solve $M\hat{s} = s$
  $t = A\hat{s}$
  $\omega_i = t^T s / t^T t$
  $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p} + \omega_i \hat{s}$
  $r^{(i)} = s - \omega_i t$
  check convergence; continue if necessary
  for continuation it is necessary that $\omega_i \neq 0$
end

Figure 2.13: Detailed BiCGSTAB Algorithm

Chapter 3

Parallelization of Krylov Subspace Methods

Experiments were conducted for testing the scalability of 2 popular Krylov Subspace algorithms : CG and QMR. The parallelization of the algorithms was achieved through:

- Intel i3 processor with 2 real cores and 2 virtual cores through hyperthreading running at 2.13GHz.
- 4GB RAM $\approx$ 8 GBps memory-processor bandwidth.

The author wishes to emphasize that the values quoted are real or actual values as opposed to ideal values. The Intel documentation states that the memory-processor bandwidth is 17.1 GB/s [6] however from implementation and thumb rules, only about 50 percent is possible in actuality.

The results of the algorithms discussed below are for the given configuration:

- Use of vendor BLAS subroutines through Intel Math Kernel Library (MKL)[5] for matrix-vector multiplication, norm and dot product calculations. These were linked to DGEMV, DNRM2 and DDOT respectively.
- Tuning of DAXPY, DSCAL and DCOPY subroutines for the multicore system.

Among the various implementation of BLAS available viz. NETLIB, Intel MKL, GoTo[13] and ATLAS[33, 34, 35, 32, 25], Intel MKL exhibited superior performance both in serial and multithreaded mode. BLAS is able to achieve much better performance as compared to intrinsic subroutines of Fortran by using block algorithms which optimize the subroutines taking into account the memory hierarchy, architecture and other aspects of the software-hardware interaction. Libraries such as ATLAS (Automatically Tuned Linear Algebra Software), as its name suggests, tunes the subroutines for a particular machine. Because BLAS forms the

basis for all efficient computation in the linear algebra domain, most hardware vendors distribute optimized versions of BLAS to map effectively on their hardware. Examples include Intel's MKL and AMD's ACML (Math Core Library).

3.1 Scalability Curves

Scientific computing codes tend to be limited in their scalability either due to the memory bandwidth or the total flops available. In the former case, once the memory flow reaches the peak value, no further increase in scalability can be made possible. In the latter case, the codes can be scaled by increasing the number of processors available to the system. Krylov subspace methods tend to be memory throttled whereas direct solvers such as LU and Gaussian Elimination with Partial Pivoting tend to be flop throttled. Only by improving the memory bandwidth can the Krylov Subspace methods profitably make use of the compute power. Consider the case of a 1E5 $\times$ 1E5 dimension linear system, the total number of elements are dominated by the matrix of size 1E10. In double precision, this would take 8E10 bytes for storage which is roughly 80 GB worth of data. Assuming a modest 500 iterations for convergence and a 8 GBps memory bus, the memory transfer would take 5000 seconds in total. (The cache effects are ignored since the size of the matrix greatly exceeds the size of the cache. Also, any amount of locality will not be able to help much since the reusability of the data per iteration is limited). Whereas, for the flop calculation, the algorithmic complexity is $mO(N^2)$ which is approximately $500 \times 2 \times 1E10$ (Assuming the leading constant to be 2). Which is a total of 1E13 operations. With a 2.13 GHz processor with 2 operations per cycle, that amounts to a total time of roughly 2400 seconds. This clearly proves that the KS methods are memory bound. The major problem with respect to the KS methods is the need to move the data from the main memory to the processor proportional to the number of iterations. In direct methods, since there is no such movement, the methods tend to be flop bound as stated above. While the calculations performed above are approximate, the author believes that they do provide a bird's eye view of the underlying problems in mapping the algorithm to the hardware.

Experiments were conducted to test the scalability of CG and QMR on multicore systems. The figure below shows the variation of the speedup with dimension of the methods discussed above. The codes were based on NETLIB's implementation and were manually tuned for the aforementioned system.

There are 3 levels of BLAS. Among them, BLAS Level 1 consists of operations defined on one vector or a pair of vectors, BLAS Level 2 is defined on a matrix and a vector while the BLAS Level 3 consits of matrix-matrix operations along with those concerning the solution of linear systems. Multithreaded implementations are available for BLAS Level 3. Although multithreaded implementations are also available for BLAS 1 and 2, the communication costs

across threads tends to be higher than the operation cost and thus are usually not efficient. Experiments conducted have suggested that vendor implementations of certain BLAS Levels 1 and 2 are weaker in performance than manually tuned OpenMP implemenatations. In certain cases, intrinsic sequential operations performed better than multithreaded vendor optimized BLAS for even large problem sizes.

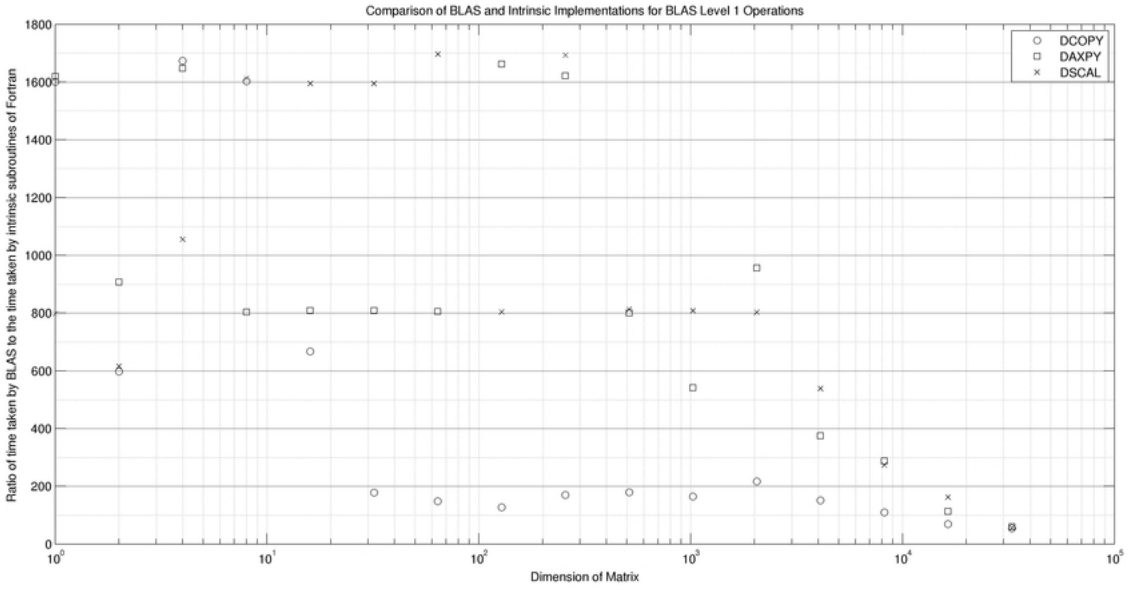

Figure 3.1: BLAS Level 1 Operations' Comparison

The figure above summarizes the results of the experiment. For matrix sizes ranging from the order of 10^0 to 10^5, a comparison was made between Intel MKL's DAXPY, DCOPY, DSCAL subroutines with Fortran's intrinsic subroutines as shown in the table below. The ratio of the time taken by Intel MKL's implementation to Fortran's implementation was plotted against the dimension of the matrix.

DAXPY(n,alpha,x,1,y,1)	y = alpha*x + y
DSCAL(b,alpha,a,1)	a = alpha*a
DCOPY(n,a,1,b,1)	b = a

For all matrices tested for, Fortran's intrinsic implementation performed much better than Intel MKL's BLAS Level 1 subroutines.

CG consists of 1 matrix vector multiplications and 2 dot products. On the other hand, QMR requires 2 matrix-vector multiplications and 5 dot products. These operations were linked to corresponding BLAS subroutines while the rest were implemented using Fortran's intrinsic functions.

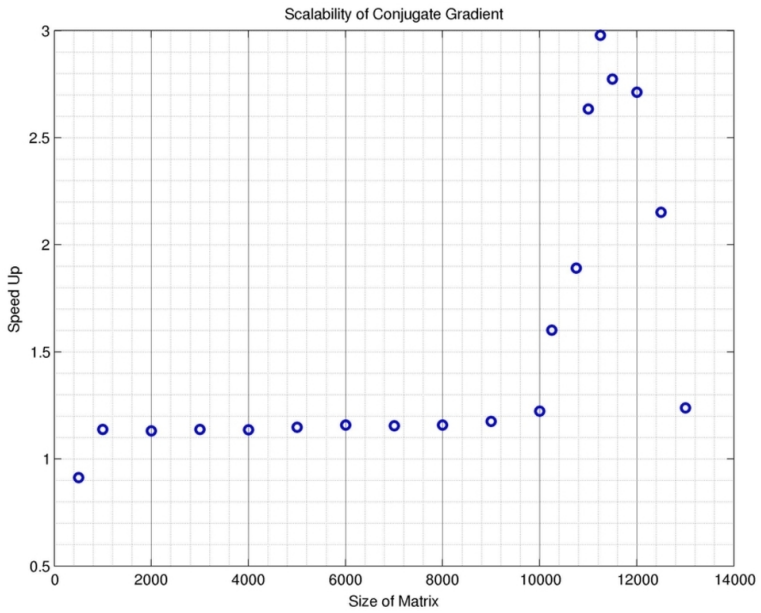

Figure 3.2: Scalability curve of CG

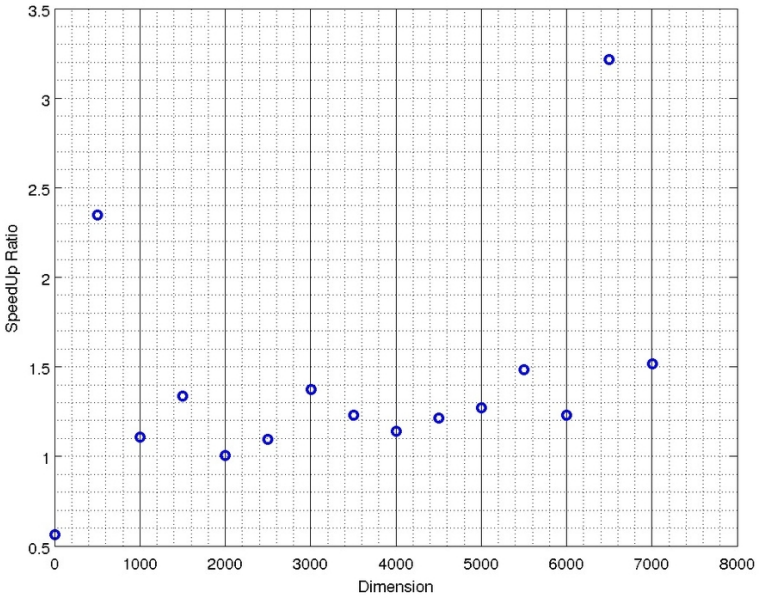

Figure 3.3: Scalability curve of QMR

The figures indicate the scalability of CG and QMR for a system with 2 real and 2 virtual cores. However, similar implementations based on Intel Xeon processor with 12 real and 12 virtual cores gave identical results. The speedup for such systems was limited to a maximum of 3.

Gaussian Elimination, however, scaled very well on all systems and yielded a speedup roughly equal to the number of real cores of a system. Using the LAPACK [2] framework, unstructured matrices are solved with the DGESV subroutine. The following figure shows

the behaviour of DGESV with matrix size for the Intel i3 system.

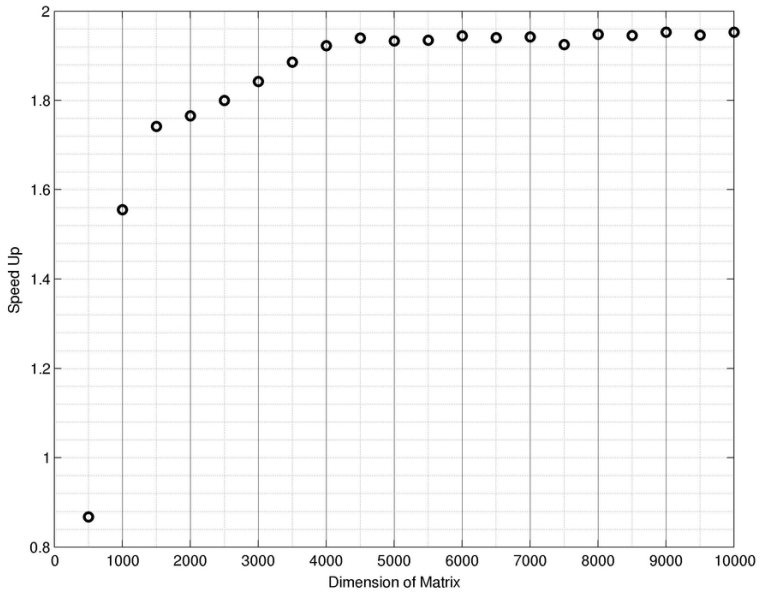

Figure 3.4: Scalability curve of Gaussian Elimination

3.2 Comments on Programming Languages

In the context of Scientific Computing, 2 programming languages are important : Fortran and C. Traditionally, Fortran has been viewed as a "Mathematician's Language" while C as a system language. A large portion of the scientific codes were written in Fortran including BLAS and LAPACK. However, for modern computational linear algebra, the difference in performance between C and Fortran tends to be insignificant. All libraries relevant to scientific computing are usually available in both C and Fortran and if not, wrappers around them can be created to allow for cross-language programming. The compilers for C codes are not very different in conversion to machine code than the compilers for Fortran. This was verified from experiments where the same subroutines and similar machine codes caused bottlenecks in the performance of CG and QMR. Both C and Fortran had bottlenecks for the matrix-vector multiplication (DGEMV) in CG and QMR & the greatest cumulative time was spent on double-double multiplication (MULPD) for the assembly conversion of both languages.

Although the primary language for the experiments conducted as a part of this thesis has been Fortran, implementations in C have yielded similar performance and scalability. Having said that, the author wishes to emphasize that there are several parameters for these languages beyond execution time. They are enumerated below:

- Extendibility: Owing to the backward compatability of C++ with C, codes written in C can be easily extended to C++. C++ allows for richer data structures and the ability

to code in an object oriented environment. Owing to this, among others, it allows for function and operator overloading. In the presence of single-double mixed programming, such as with mixed iterative refinement, it is possible to overload SGEMV and DGEMV with a single GEMV function which, although does not aid the performance, will lead to a cleaner looking code for the end user.

- Data Structures: Fortran excels at array processing. If the problem can be described in terms of simple data structures and in particular arrays, Fortran is well adapted. C/C++ is better suited for complex and highly dynamic data structures.

- Parallel programming: Both MPI and OpenMP work just fine with both C and Fortran. However, if real control of threads is desired, to have a fully dynamic shared-memory computation, working in Fortran will lead to problems while in C, the user will have access to the standard Pthreads. As a thumbrule, most computations that rely on access to the operating system, e.g. threads, processes, file system etc are better served with C.

- Different hardware architectures: The whole CUDA architecture is built around kernels in C[21]. While the Portland Group now has a CUDA-capable fortran compiler too, but it is commercial, and it is not released by NVIDIA as an SDK. Same goes for OpenCL, for which most projects only support a few basic calls. With the introduction of newer hardware (including GPGPU) built aroud the C, converting codes written for multicore systems to those for the emerging hardware is easier than converting the same codes from Fortran.

The author wishes to emphasize the importance of higher level languages such as Python in the scientific community. With the addition of C and Fortran wrappers around Python either explicitly or in the form of mathematical libraries such as SciPy[18] and NumPy[22], it is possible to use high level programming paradigms of Python with the speed, efficiency, performance and tuning ability of C and Fortran. With the introduction of new IPython shell[23], it is possible for the researcchers in the scientific computing communities to have implementations of complicated systems in concise form. Coupled with all these benefits, Python also provides an extensive support for non-numerical libraries making it easy to construct codes for applications spanning medicine, engineering science and economics.

3.2.1 A comparison of OpenMP and Pthreads

Explanation is due regarding the reason for the choice of OpenMP as the API for shared memory parallelization. A few important points regarding the same as summarized below:

- Pthreads and OpenMP[4] represent two totally different multiprocessing paradigms. Pthreads is a low-level API for working with threads. Thus, it is possible to have fine-grained control over thread management, mutexes, and so on.

  On the other hand, OpenMP is much higher level, is more portable and doesn't limit the programmer to use C. It also scales easier than Pthreads. One specific example of this is OpenMP's work-sharing constructs, which allow for division of work across multiple threads with relative ease.

- OpenMP is task-based whereas Pthreads is thread based. It means that OpenMP will allocate the same number of threads as number of cores. Further, OpenMP provides reduction features as they are needed when computing partial results in threads and combining them. This is possible in a single line of code while it is fairly non-trivial in Pthreads.

Considering the ease of converting a serial code into a parallel one, the high level interface and the ability to have requisite control over the threads, OpenMP was chosen as opposed to Pthreads.

Chapter 4

Numerical Experiments on Krylov Subspace Methods

Numerical experiments conducted on matrices are now described. 5 popular Krylov iterative methods viz. GMRES, QMR, CGS and BiCGSTAB were compared. CG was also compared when the matrices were positive definite. The comparison was based on 2 factors:

- Residue characteristics
- Time for completion

The time for completion was also compared against Gaussian Elimination with Partial Pivoting. This was implemented through an explicit LU factoring with a forward-backward solve. Further, comparison was also made with the \ (backslash) operator was also compared. This operator, which is also called the "mldivide" operation, which attempts to search for a suitable algorithm before attempting a direct solve. The Cholesky factorization was also compared when the matrices were positive definite. This was implemented with an explicit creation of the resultant triangular matrix and a subsequent forward-backward solve. The work of Cullum [7] on the comparison of GMRES and QMR was based on normality of matrices, the work of this thesis is based on the eigenvalue spectrum of the non-normal matrices.

Matrices were generated by passing their eigenvalues as inputs. Unless otherwise stated, the matrices generated are unique. The method for generating them has been discussed in the following paragraph. A few important points vital for reproducibility of the results are enumerated below:

- The matrices are generated using the algorithm below:

Algorithm for Generating Matrices

To get a generate a matrix with condition number c ,

Let D be a diagonal matrix whose diagonal consists of the desired eigenvalues of the matrix.

Using a similarity transformation, via Householder transformations, a matrix is formed as : $A := (I - tuu^T)D(I - tuu^T)$, where $t = 2/u^T u$.

To form this matrix with $O(n^2)$ operations,

1. $v := Du$, where $u \in \mathbb{R}^N, u_i = 1.0 \; \forall \; i$
2. $s := \dfrac{t^2 u^T v}{2}$
3. $w := tv - su$
4. $A = D - uw^T - wu^T$.

- The matrices are generated uniquely by passing the values of the desired eigenvalues.
- The matrices are necessarily symmetric. Thus, passing a positive eigenvalues generates symmetric positive definite matrices.
- For GMRES, the implementation was restart based with a restart value at 20.
- The algorithms are implemented using OpenMP threading for shared memory system.
- The time graphs represent the multithreaded elapsed time.
- The thread allocation is set to dynamic. The number of threads to be used is decided dynamically at runtime by the system rather than the user.
- All systems were solved without the use of preconditioners for a single RHS.
- The tolerance criterion was set to tolerance less than 1E-6.
- While the matrices described are of size 5000×5000, tests run on matrices with sizes 10000×10000 have exhibited consistent results.

4.1 Symmetric Positive Definite Matrices

4.1.1 Linear Distribution of Eigenvalues

Matrices were generated for eigenvalues with a linear distribution from 1 to 2000 (Figure 4.1) and from 1 to 10000 (Figure 4.2). While all iterative methods converged within the tolerance limits defined, GMRES expressed a rather peculiar convergence. In spite of a well conditioned matrix, GMRES took significantly longer than other iterative methods to converge.

This behaviour gets exemplified when the condition number of the matrix and hence, concomitantly, the size of the matrix, increases. The time for completion of the algorithms was comparable to the time taken by direct solvers.

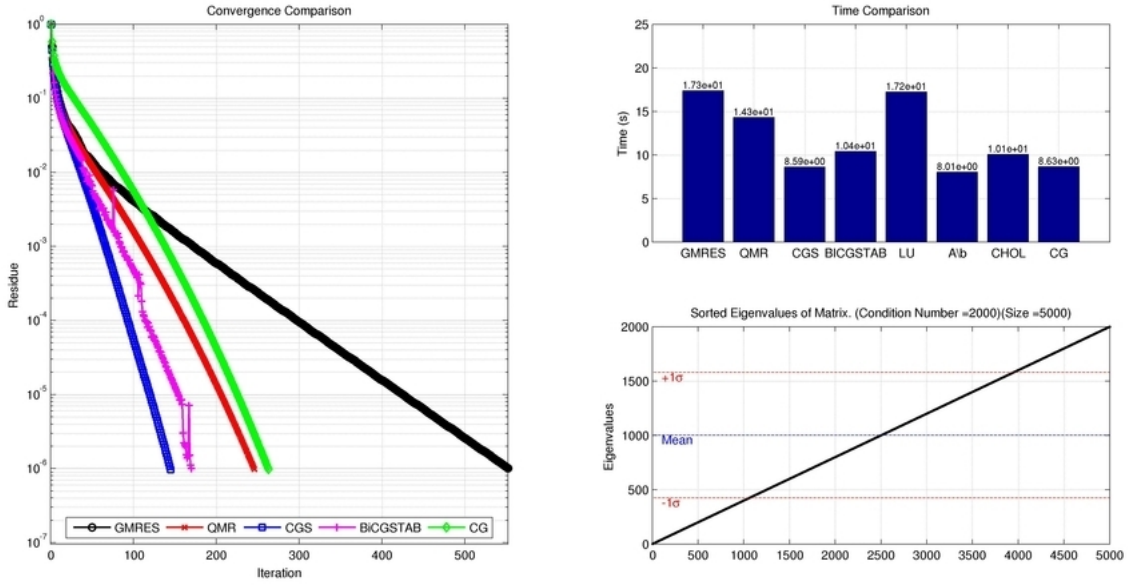

Figure 4.1: Linear Distribution from 1 to 2000

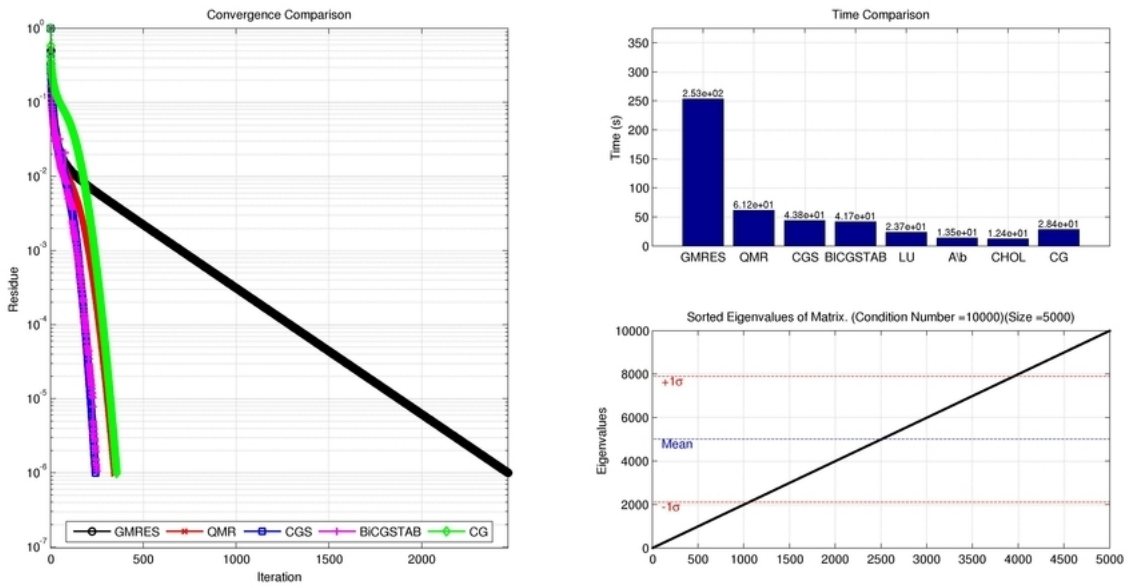

Figure 4.2: Linear Distribution from 1 to 10000

An investigation was due regarding the cause of GMRES's convergence behaviour for such matrices. Consequently, continuing the experiment by with eigenvalues distributed between 1 and 10000, matrices were generated with the same minimum and maximum eigenvalues (and thus, same condition number) but instead of having a uniform linear distribution, the eigenvalues were distributed non-uniformly. A ratio R is defined as the number of eigenvalues linearly spaced between [1,2000] and [8000,10000].

As can be seen from Figures 4.3 4.4 and 4.5 GMRES did not deviate from its peculiar convergence behaviour for any of these cases.

Thus, GMRES proves to be a poor algorithm in the presence of eigenvalue outliers and linearity. Figure 4.2 especially points out the poor convergence of GMRES in spite of having an excellent spectral structure.

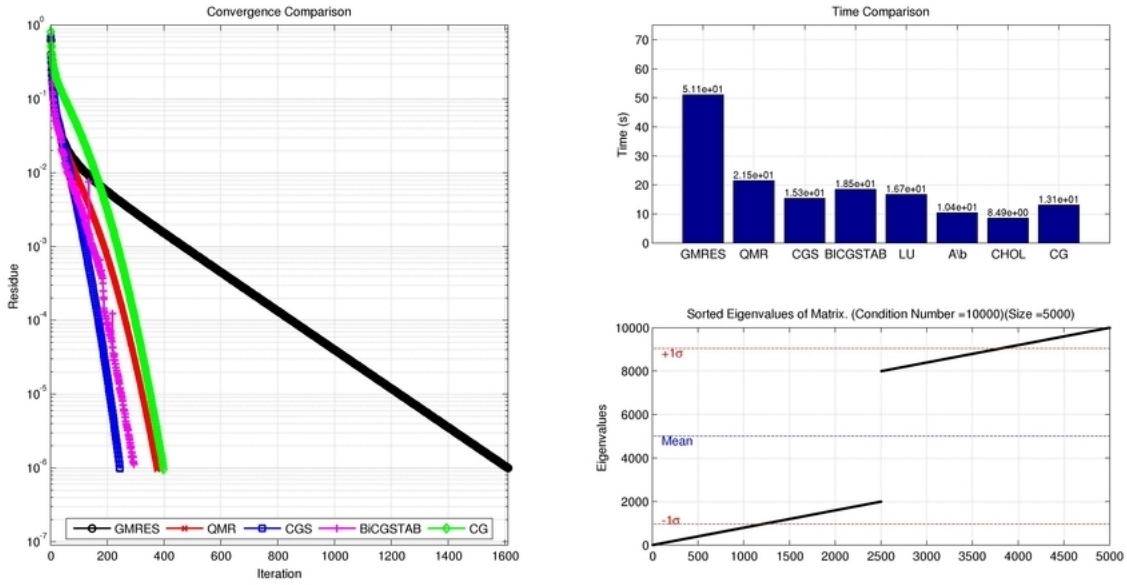

Figure 4.3: GMRES Problem with Linear Distribution. Ratio = 1:1

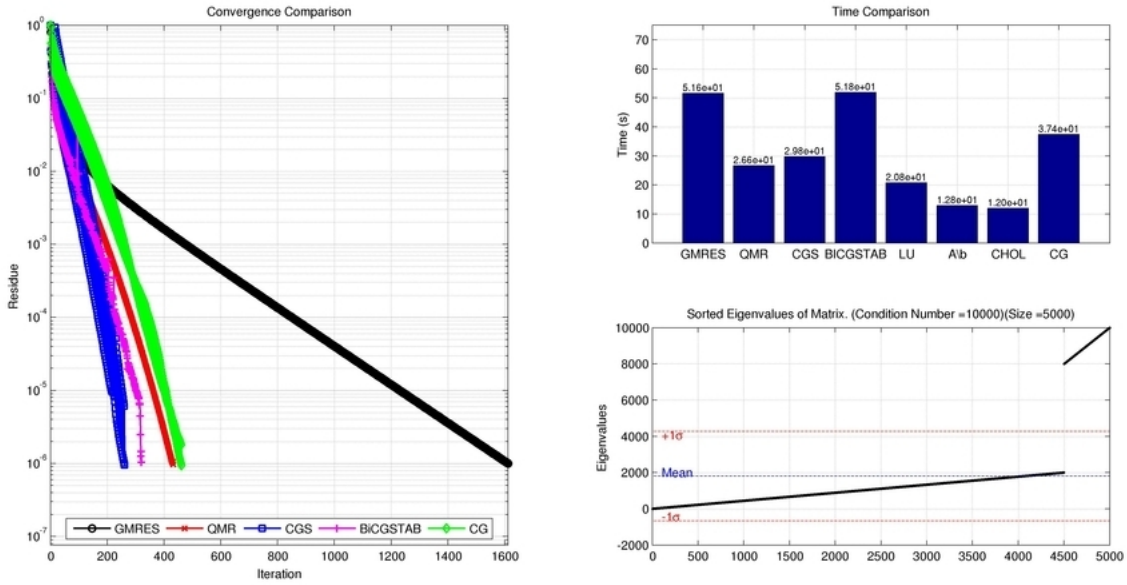

Figure 4.4: GMRES Problem with Linear Distribution. Ratio = 9:1

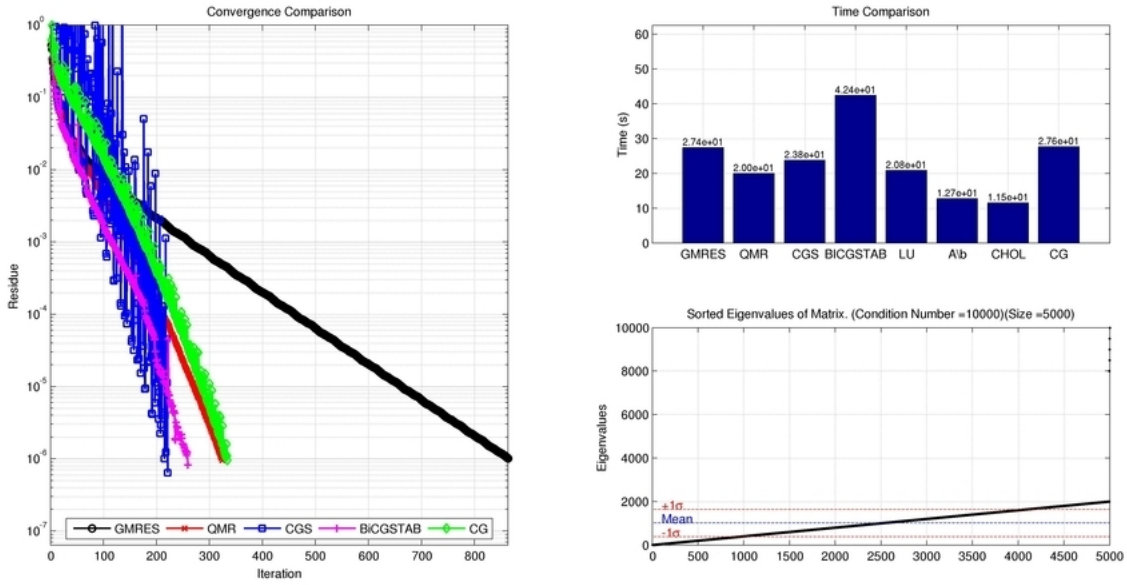

Figure 4.5: GMRES Problem with Linear Distribution. Ratio ≈ 999:1

To verify that the above results were caused due to linearity and outliers instead of any effects due to conditioning of the matrix, an experiment was conducted with the same minimum and maximum eigenvalue but rather than a linear structure, the eigenvalues were clustered around the minimum and maximum values (Figure 4.6). GMRES, in this case, converged as well as the other iterative algorithms both in residue reduction rate and time for completion.

The time taken by iterative methods was significantly lower than the time taken for direct solves.

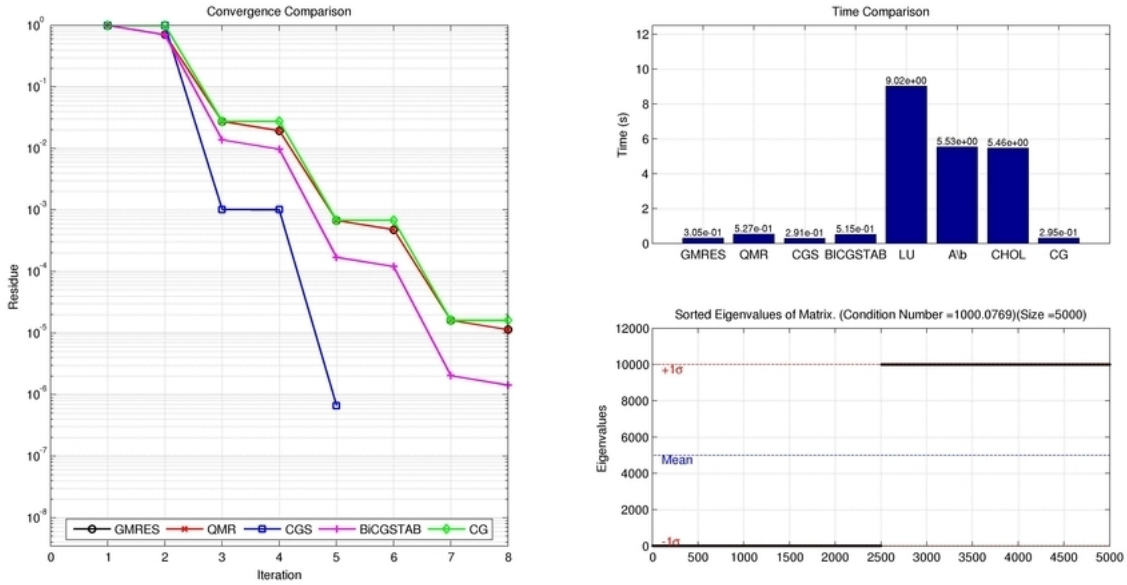

Figure 4.6: 2 clusters at Centres 10 and 10000

As stated above, the implementation of GMRES was a restarted version of it with a restart value of 20. Experiments were conducted to check if the restart value had any effect on this peculiar behaviour of GMRES. 4 experiments were conducted for restart values of 10, 30, 100 and 500 keeping the same spectrum as in the experiment depicted by Figure 4.2.

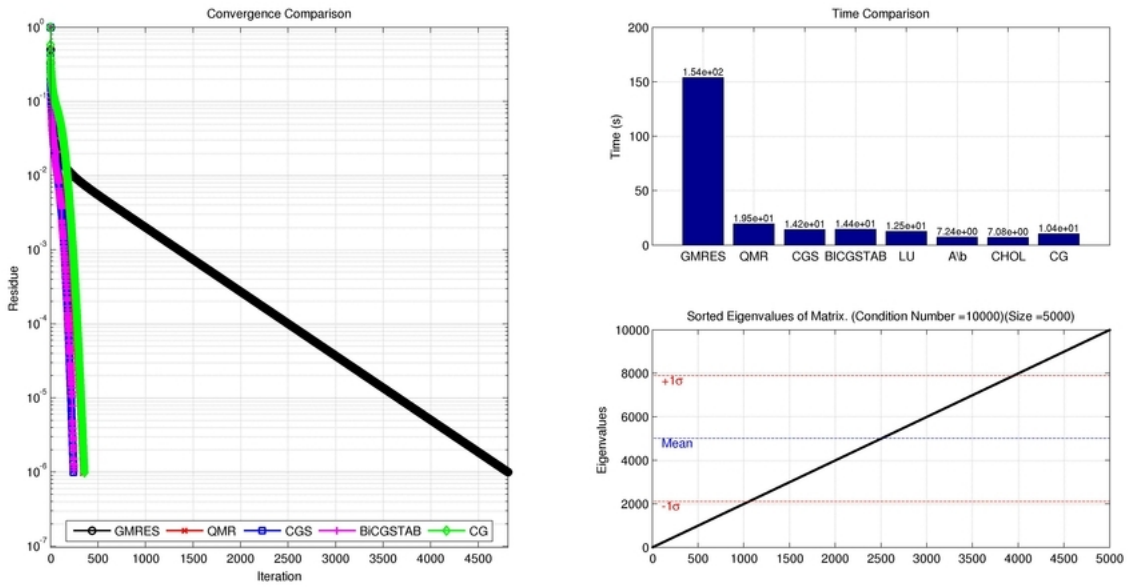

Figure 4.7: GMRES with Restart = 10

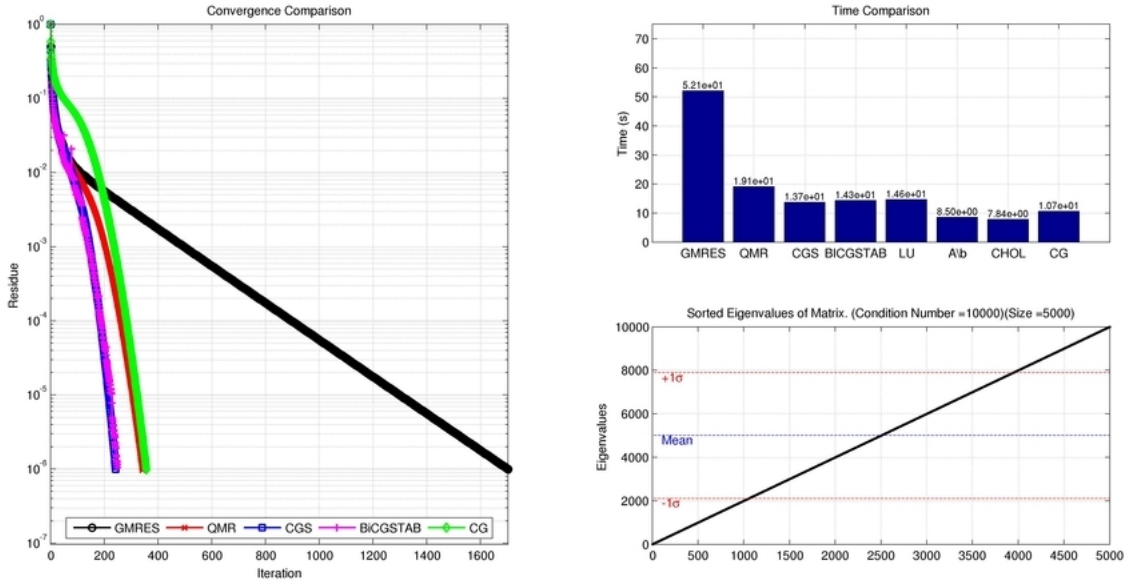

Figure 4.8: GMRES with Restart = 30

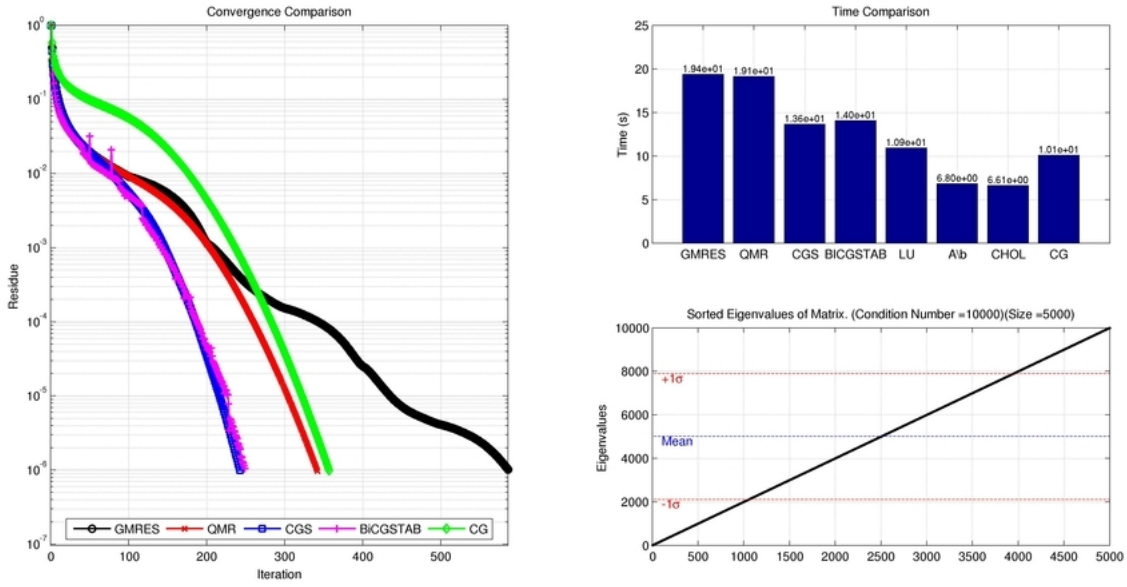

Figure 4.9: GMRES with Restart = 100

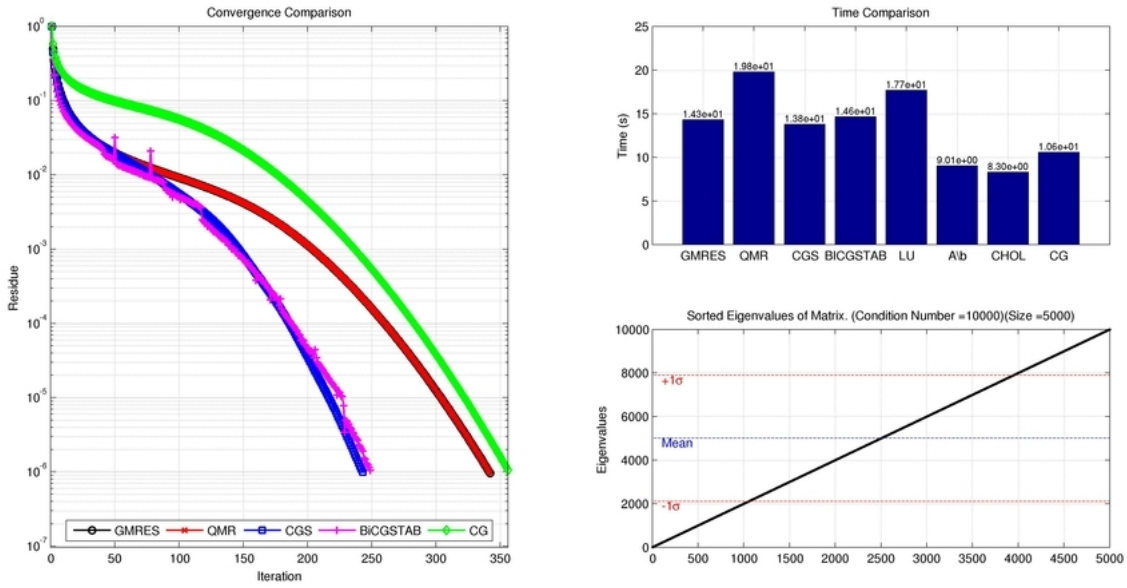

Figure 4.10: GMRES with Restart = 500

The number of iterations required as also the time required for GMRES to converge reduced with an increase in restart in the above example. However, literature in this field has showed that the convergence of GMRES is fairly non-trivial. The work of Embree [9] showed that the a high restart number might, in some cases, take more iterations than cases with low restart numbers. Thus, increasing the restart value need not always aid the user when performance improvements are desired. Further, the work of Greenbaum and Strakos [15] showed that any non-increasing curve can approximate the convergence curve of GMRES.

4.1.2 Linear Distribution of Eigenvalues with Outliers

In section 4.1.1, experiments were concentrated on linear eigenvalue distribution which yielded results regarding the convergence of GMRES. In the presence of linearities or outliers, GMRES converged poorly while the other algorithms did converge within an acceptable time and convergence frame.

Further, experiments with eigenvalue outliers (Figure 4.5) suggested that the other algorithms might have convergence instability as well. The figure showed the oscillations with other iterative methods in their residue characteristics. To verify this, experiments were conducted with eigenvalue outliers for 4 cases.

- Linear distribution of eigenvalues from 1 to 100 with an outlier at 1000. The eigenvalues were uniformly distributed from 1 to 100. An additional eigenvalue was present with a value of 1000. (Figure 4.11)

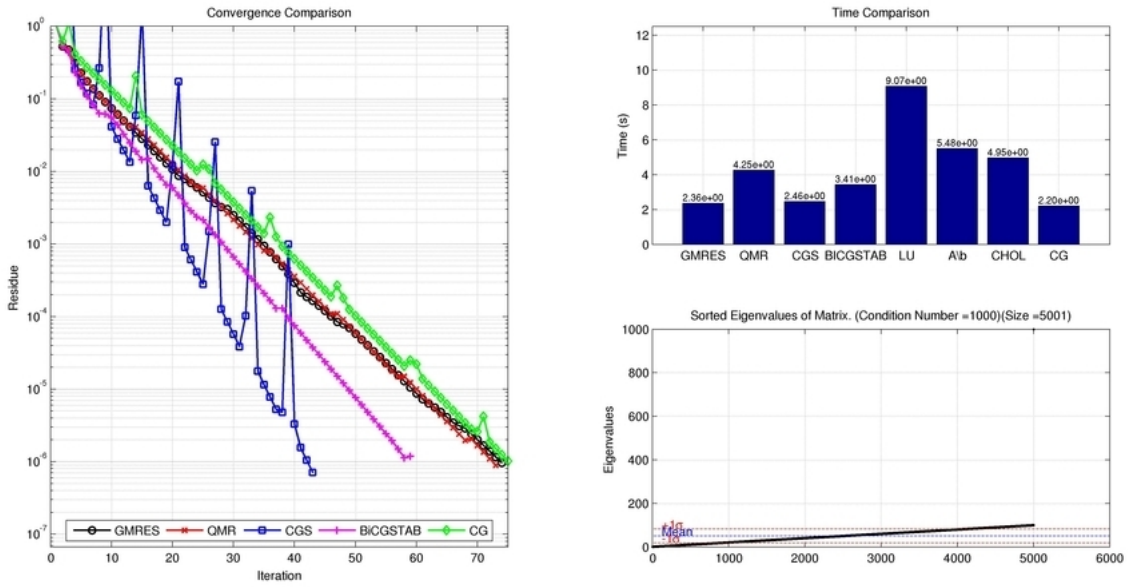

Figure 4.11: Linear Distribution from 1 to 100 with an outlier at 1000

- Linear distribution of eigenvalues from 1 to 100 with an outlier at 10000. The eigenvalues were uniformly distributed from 1 to 100. An additional eigenvalue was present with a value of 10000. (Figure 4.12)

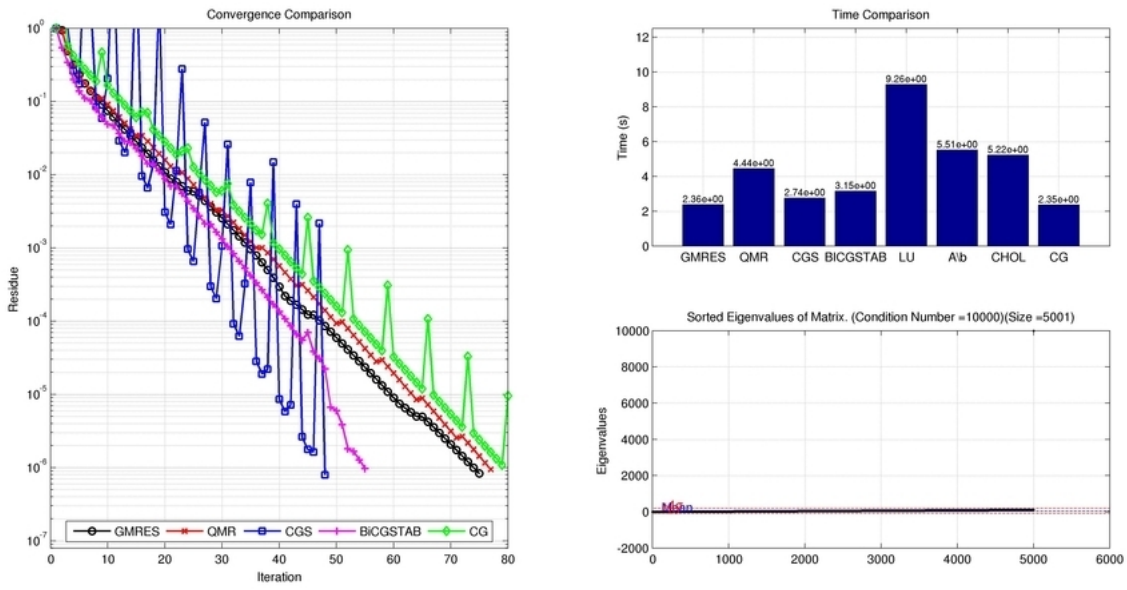

Figure 4.12: Linear Distribution from 1 to 100 with an outlier at 10000

- Linear distribution of eigenvalues from 1 to 1000 with an outlier at 10000. The eigenvalues were uniformly distributed from 1 to 1000. An additional eigenvalue was present with a value of 10000.(Figure 4.13)

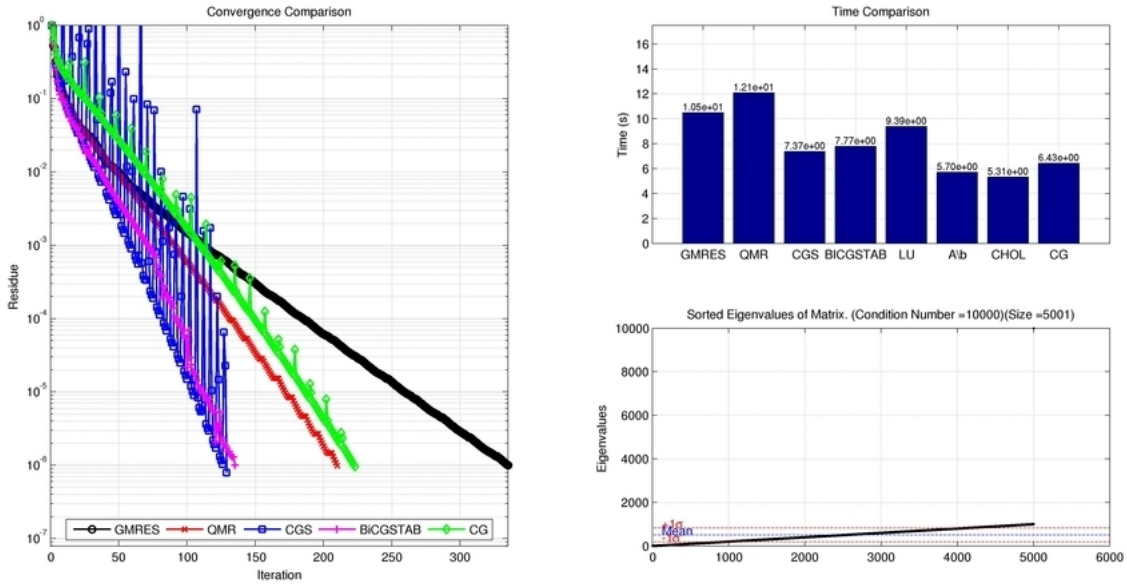

Figure 4.13: Linear Distribution from 1 to 1000 with an outlier at 10000

- Linear distribution of eigenvalues from 1 to 1000 with an outlier at 100000. The eigenvalues were uniformly distributed from 1 to 1000. An additional eigenvalue was present with a value of 100000. (Figure 4.14)

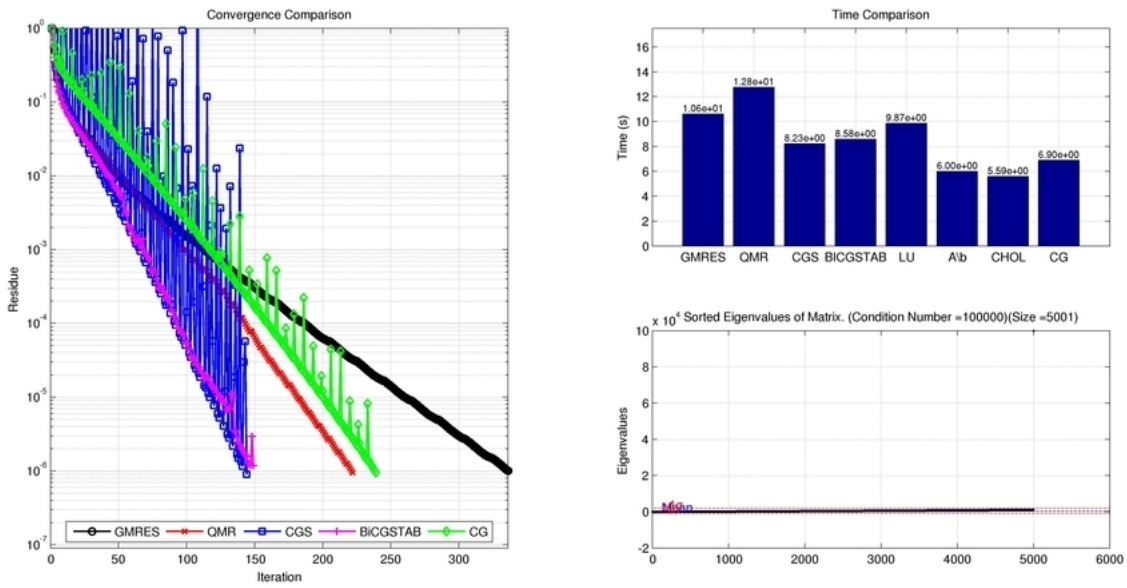

Figure 4.14: Linear Distribution from 1 to 1000 with an outlier at 100000

The CG-family of methods (CG, BiCGSTAB and CGS) exhibited oscillations in the convergence which increased in frequency with increase in the magnitude of the outlier. However, in spite of the oscillations, the time required for the completion of CG was comparable to those of direct solvers.

GMRES and QMR, on the other hand, had a approximate monotonic convergence in such circumstances. In all test cases enumerated, both these methods never exhibited the high frequency oscillations like CG, BiCGSTAB and CGS. This is true for all matrix sizes.

QMR was especially stable concerning its residue characteristics; among all test cases, QMR never exhibited oscillations of any nature during convergence. Even in the case of ill-conditioned matrices, QMR stagnated but did not oscillate.

4.1.3 Clusters of Eigenvalues

Experiments were conducted to check the behaviour of the iterative methods in the presence of matrices with eigenvalues arranged as clusters. 3 experiments were conducted. The clusters was generated by generating random eigenvalues required for the algorithm centered around a certain number with a maximum distance of spread corresponding to the radius. In the test cases enumerated below,
The centre corresponds to the mean of the eigenvalues located in the cluster.
The radius corresponds to the upper limit of the Euclidean distance of the eigenvalue from the centre.

- 1 cluster with centre 7 and radius 2 (Figure 4.15)

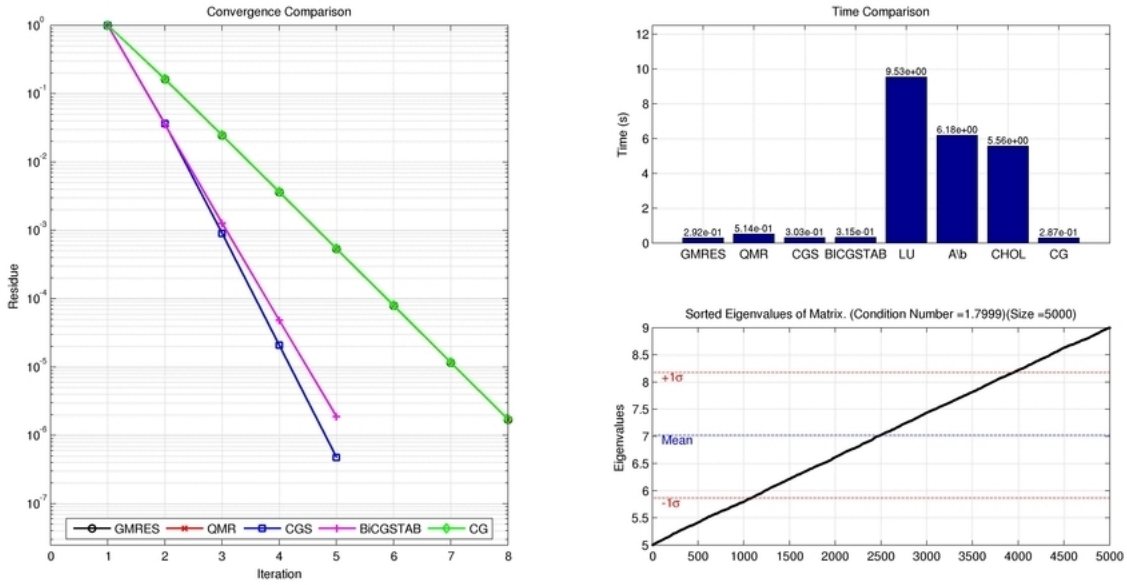

Figure 4.15: 1 Cluster with Centre 7 and Radius 4

- 2 clusters with centres at 40 and 70 and radius 10 (Figure 4.16)

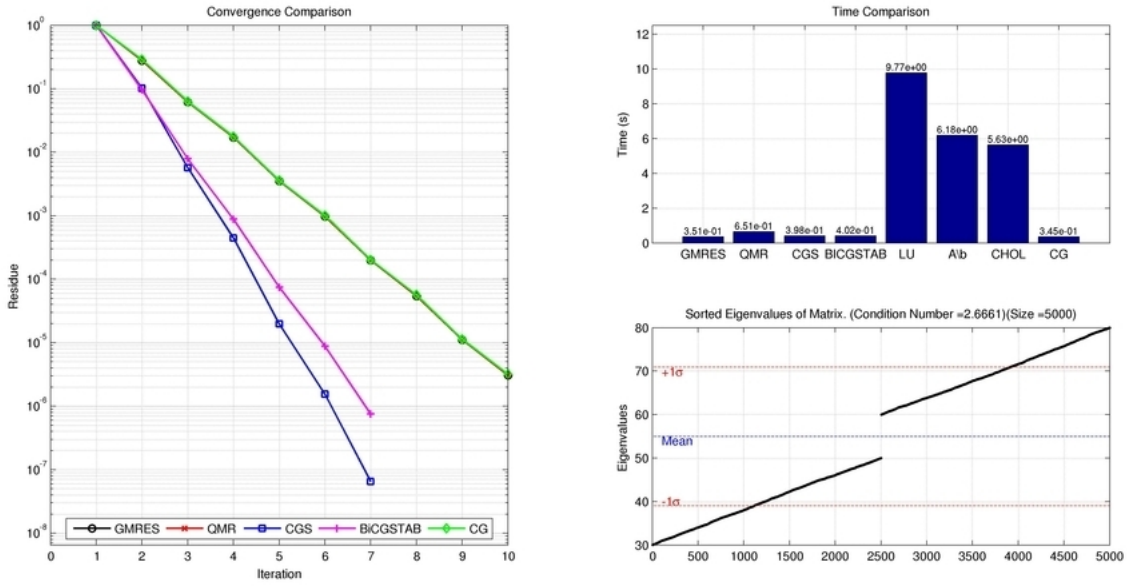

Figure 4.16: 2 Clusters with Centres at 40 and 70 and Radius 20

- 3 clusters with centres at 15, 55 and 95 and radius 5 (Figure 4.17)

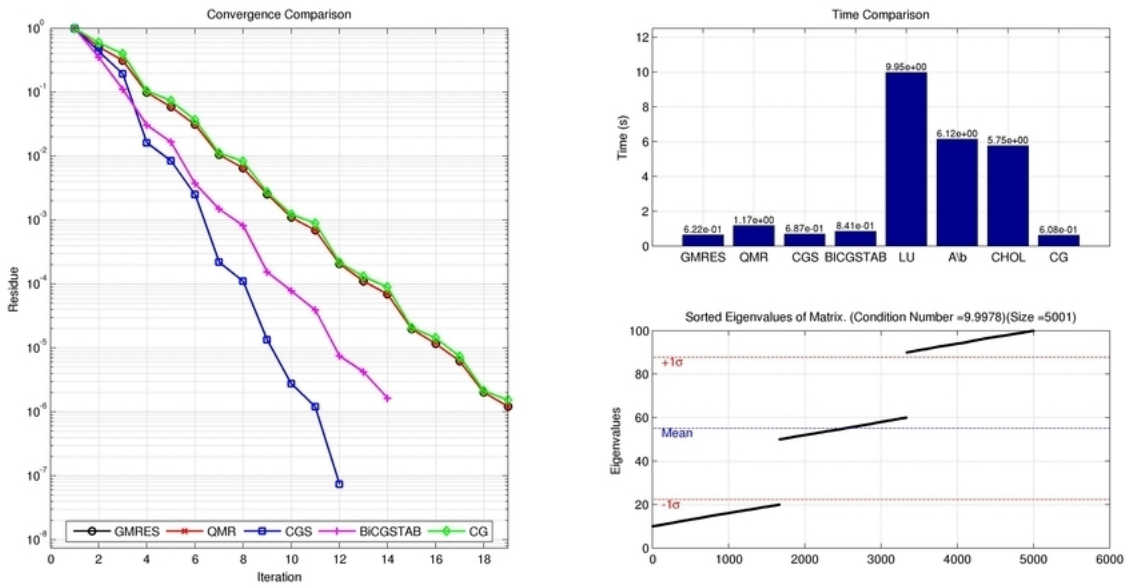

Figure 4.17: 3 Clusters with Centres at 15, 55 and 95 and Radius 10

A few conclusions can be drawn from the results of the above experiments:

- The presence of clusters reduces the number of iterations required for convergence for all methods.
- The time taken for completion by iterative methods is much lesser than the time taken by direct solvers.
- Higher number of clusters introduced oscillations in the convergence behaviour of the CG-family of methods.

4.1.4 Random Distribution of Eigenvalues

Experiments were conducted to test the algorithms for matrices with randomly distributed eigenvalues about a mean with a set standard deviation. The eigenvalues were normally generated.

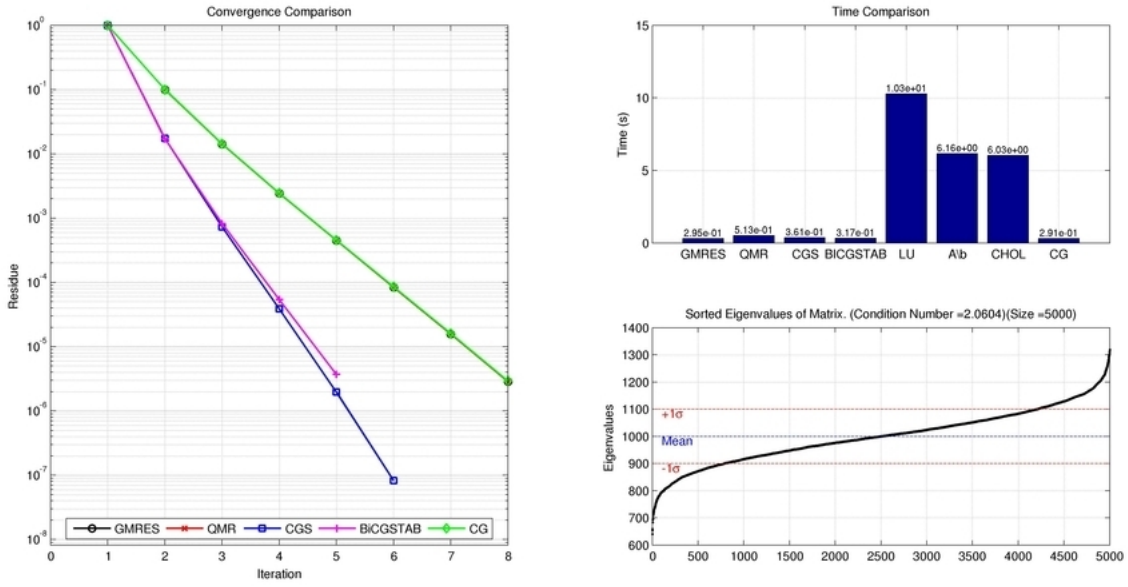

Figure 4.18: Randomly distributed eigenvalues with Standard Deviation of 100 and mean 1000

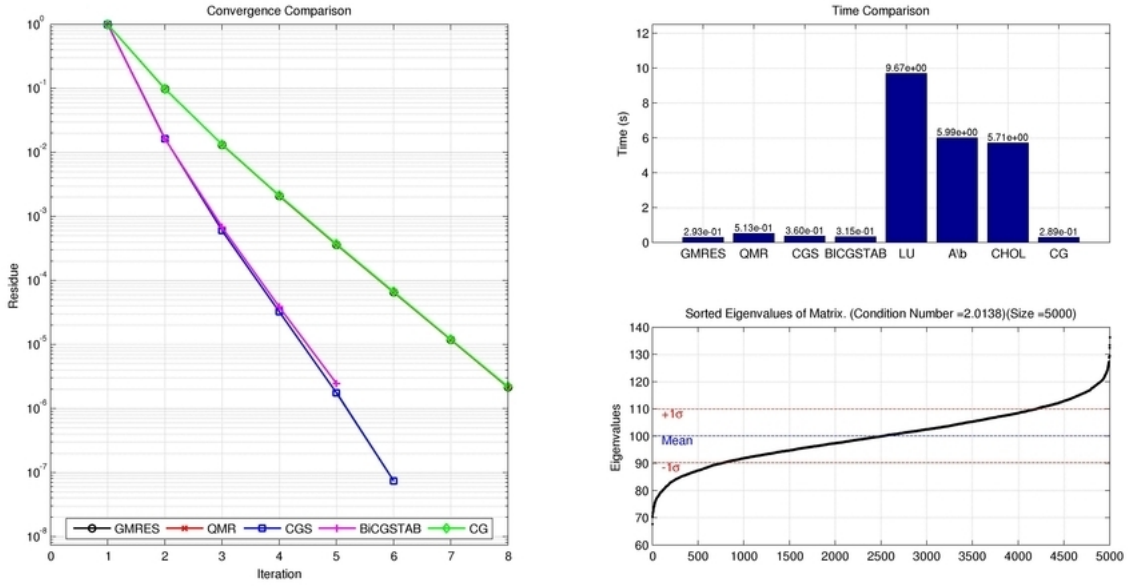

Figure 4.19: Randomly distributed eigenvalues with Standard Deviation of 10 and Mean 100

The behaviour of the methods to matrices with randomly distributed eigenvalues is similar to that of matrices with clusters of eigenvalues (Section 4.1.3). The convergence was monotonic, quick and has no oscillations. The amount of time taken for completion was much lesser than that of direct solvers.

4.1.5 Presence of Eigenvalues Clustered Around the Origin

In this experiment, an attempt was made to study the effect of the presence of small valued eigenvalues in the matrix spectrum. 2 experiments were conducted: one with the minimum eigenvalue as 0.1 and another as 0.001. To make sure that no other parameters affected the convergence comparison, the size of the matrix and the condition number was kept the same. Thus, the first case had eigenvalues uniformly distributed between [0.1, 1000] while the second had eigenvalues uniformly distributed between [0.001, 10].

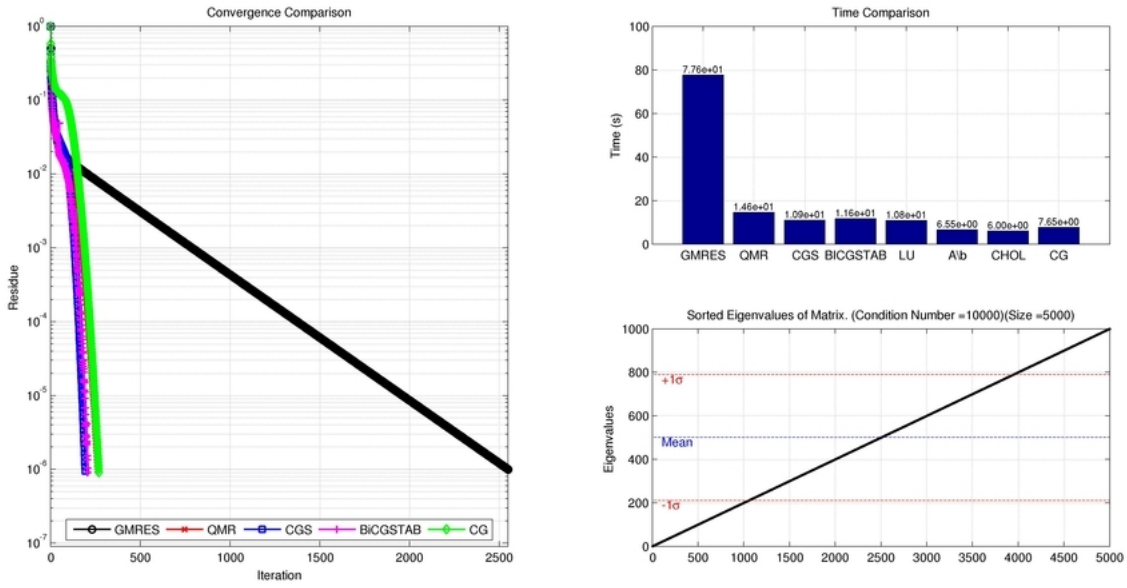

Figure 4.20: min(eig(A)) = 0.1

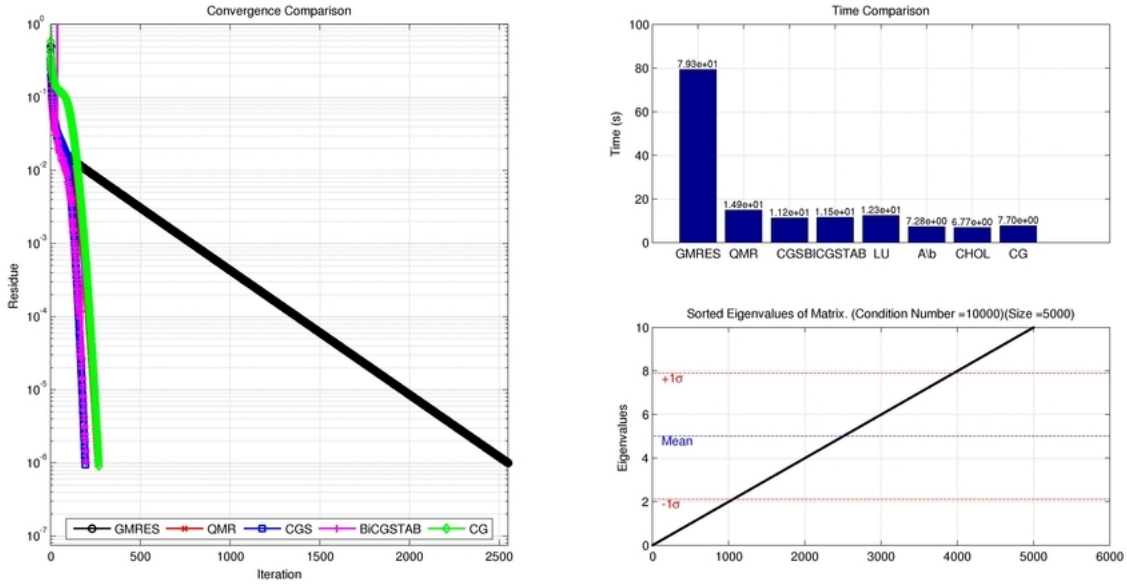

Figure 4.21: min(eig(A)) = 0.01

The behaviour of such matrices is similar to case of linear distribution of eigenvalues. The convergence of GMRES is abnormal owing to the linearity of the distribution of the eigenvalues. The rest of the iterative methods converged within an acceptable time frame with normal reduction in residue in both cases.

The presence of small eigenvalues did not disrupt the functioning of the iterative methods. The CG-family of methods and QMR converged in a time frame much lesser than the direct solvers.

Results of experiments conducted for even smaller values of eigenvalue were consistent with the results discussed above.

4.2 Symmetric Indefinite Matrices

In the previous chapter, the algorithms were studied on test cases of positive definite matrices. In the current chapter, experiments are conducted along similar lines for indefinite matrices.

4.2.1 Linear and Symmetric Distribution of Eigenvalues

Experiments were conducted for indefinite matrices such that the eigenvalues were linearly distributed away from the origin (in both positive and negative direction) to maintain good conditioning. The eigenvalue distribution is symmetric. The author wishes to emphasize the 2 different notions of symmetry : one concerning the symmetry of the matrix and the other about the symmetry of the eigenvalues about the X axis. The matrices are necessarily symmetric. Whenever the word "unsymmetric" is mentioned, it is in context to the distribution of the eigenvalues about the X axis. To study the effect of this distribution, 3 experiments were conducted:

- Linear (Symmetric) from 5 to 10. (Figure 4.22)

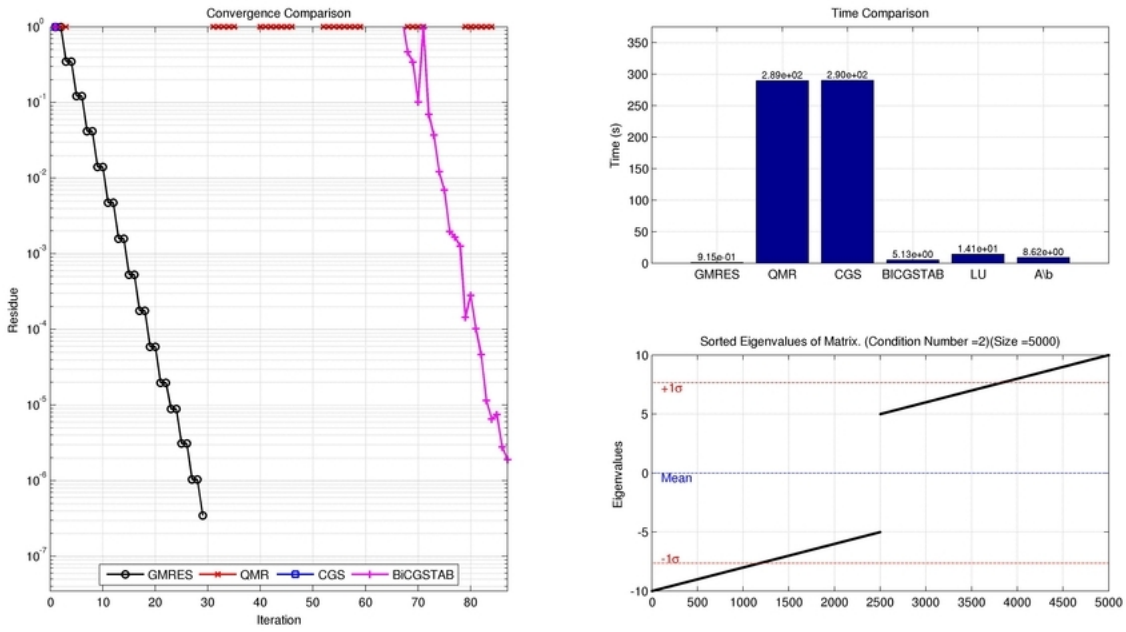

Figure 4.22: Linear (Symmetric) from 5 to 10.

- Linear (Symmetric) from 50 to 100. (Figure 4.23)

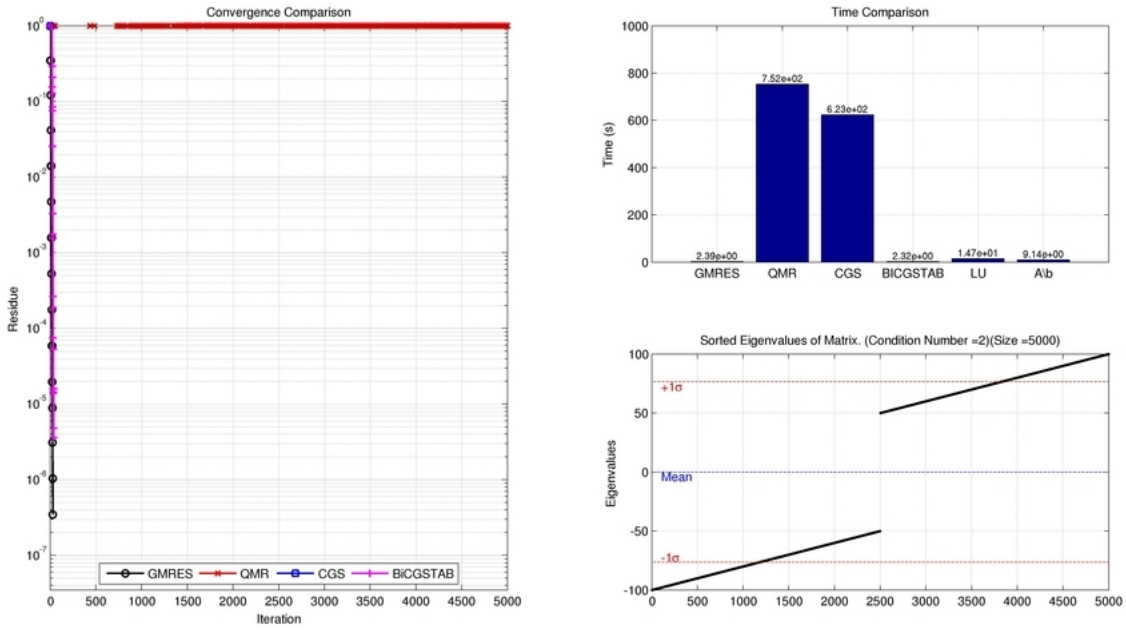

Figure 4.23: Linear (Symmetric) from 50 to 100.

- Linear (Symmetric) from 500 to 1000. (Figure 4.24)

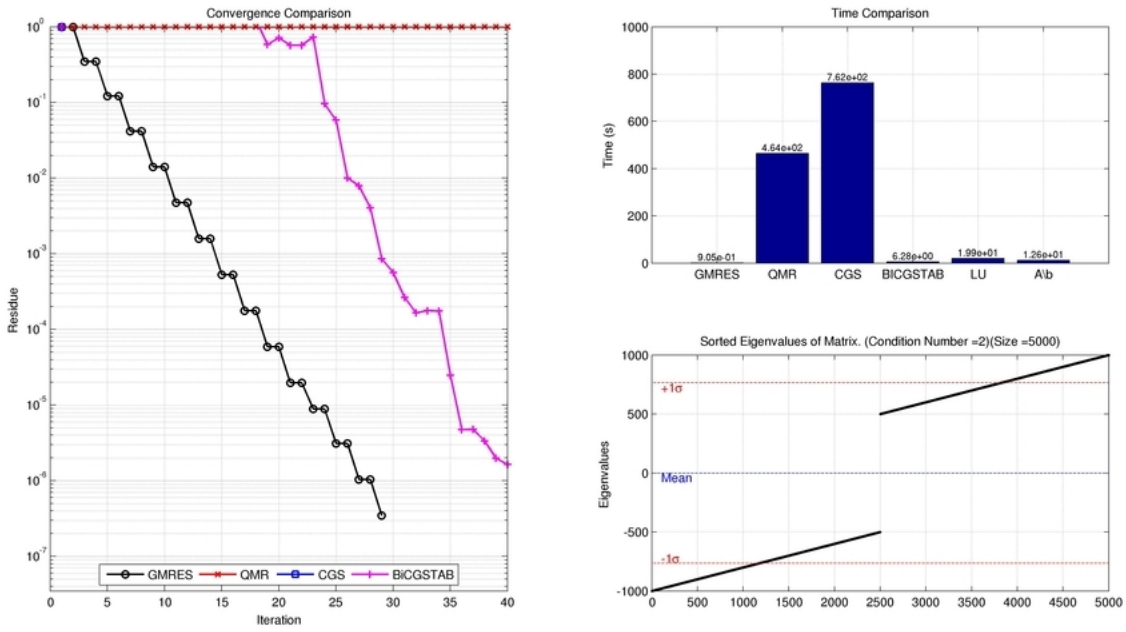

Figure 4.24: Linear (Symmetric) from 500 to 1000.

In all 3 cases, GMRES and BiCGSTAB converged while the rest failed to converge. GMRES failed to converge for positive definite matrices with a linear distribution of eigenvalues. However, GMRES was the only stable algorithm among the ones discussed which converged for indefinite matrices with a linear distribution of eigenvalues.

The time required for the completion of the process as well as the convergence rate in terms of residue were within acceptable limits. The amount of time spent for solving the system was much lesser than the direct solvers including the implicit Gaussian Elimination carried out by A\b.

4.2.2 Linear and Unsymmetric Distribution of Eigenvalues

Following the analysis of section 4.2.1, algorithms other than GMRES failed to converge for matrices with good conditioning and moderate sizes. This convergence failure could be alluded to the symmetry of the eigenvalues about the X axis. To verify this, 6 experiments were conducted with unsymmetric distribution of eigenvalues.

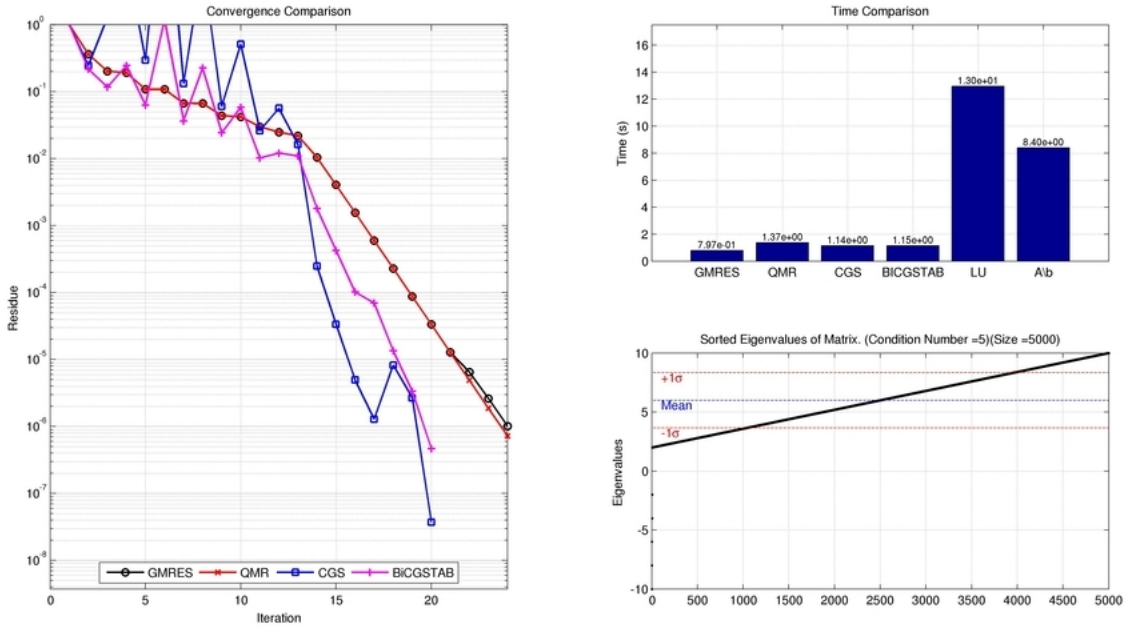

Figure 4.25: Eigenvalues distributed from [2,10] and [-2,-10] with the former having more weight.

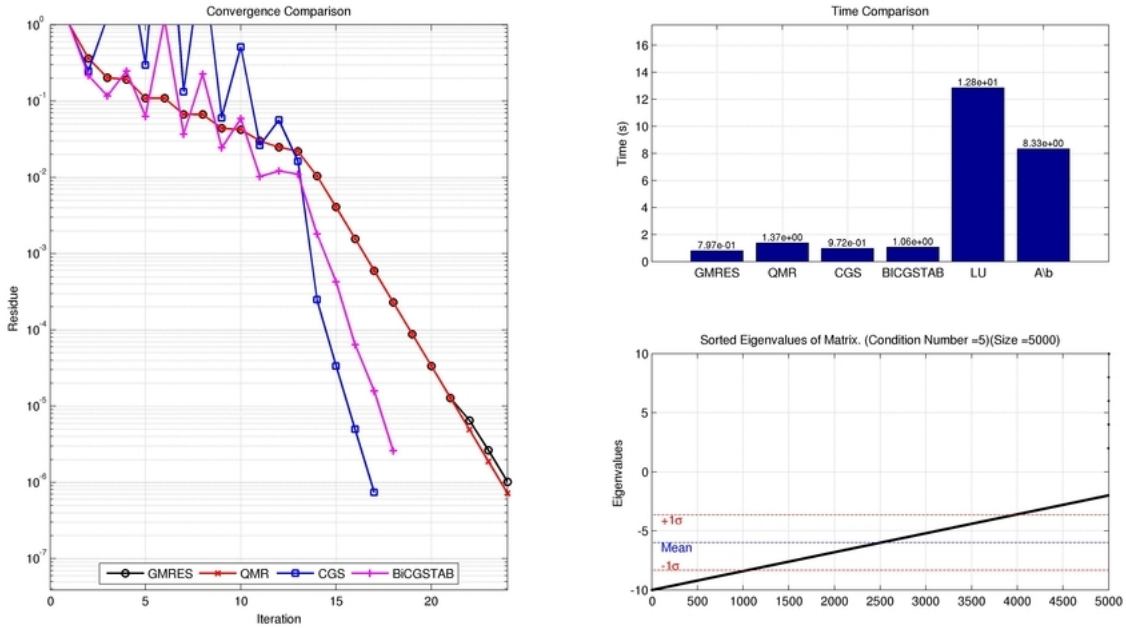

Figure 4.26: Eigenvalues distributed from [2,10] and [-2,-10] with the latter having more weight.

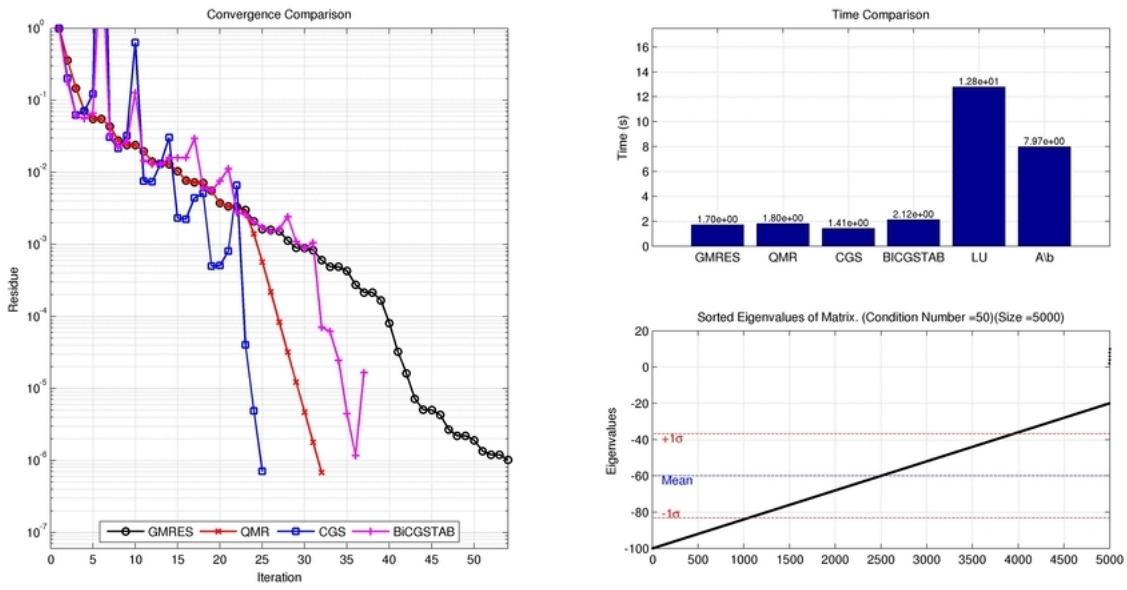

Figure 4.27: Eigenvalues distributed from [2,10] and [-20,-100] with the latter having more weight.

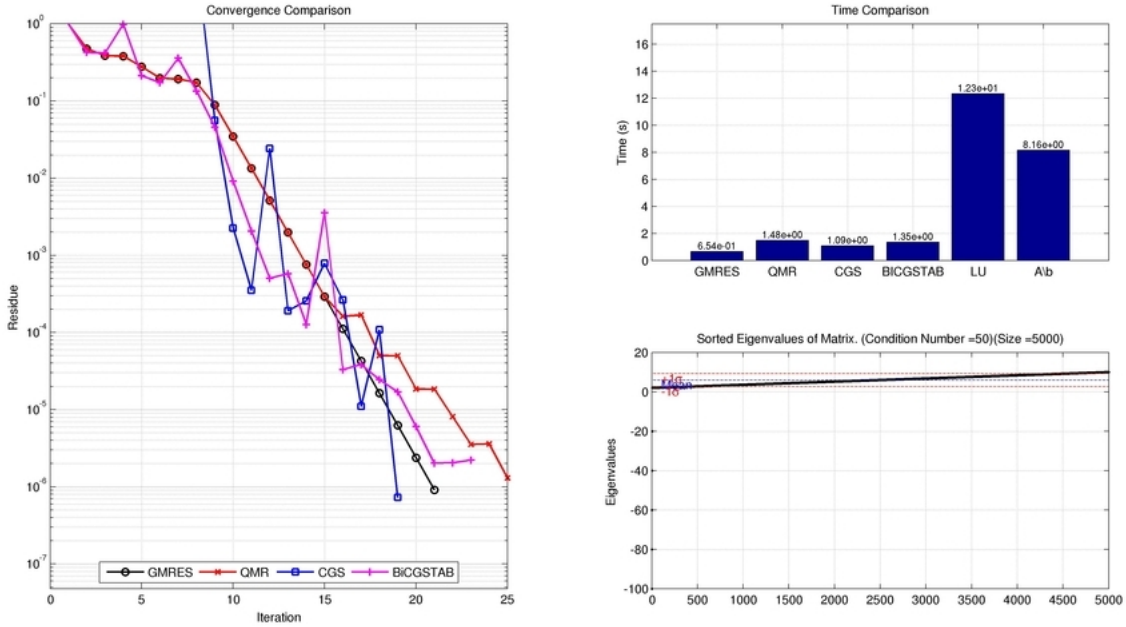

Figure 4.28: Eigenvalues distributed from [2,10] and [-20,-100] with the former having more weight.

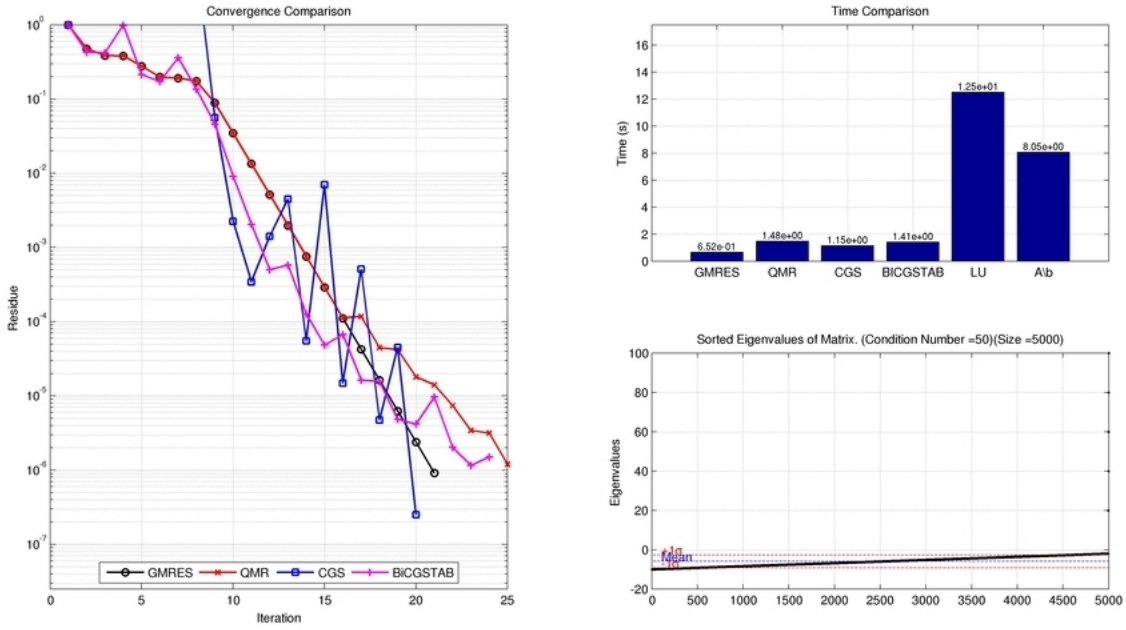

Figure 4.29: Eigenvalues distributed from [20,100] and [-2,-10] with the latter having more weight.

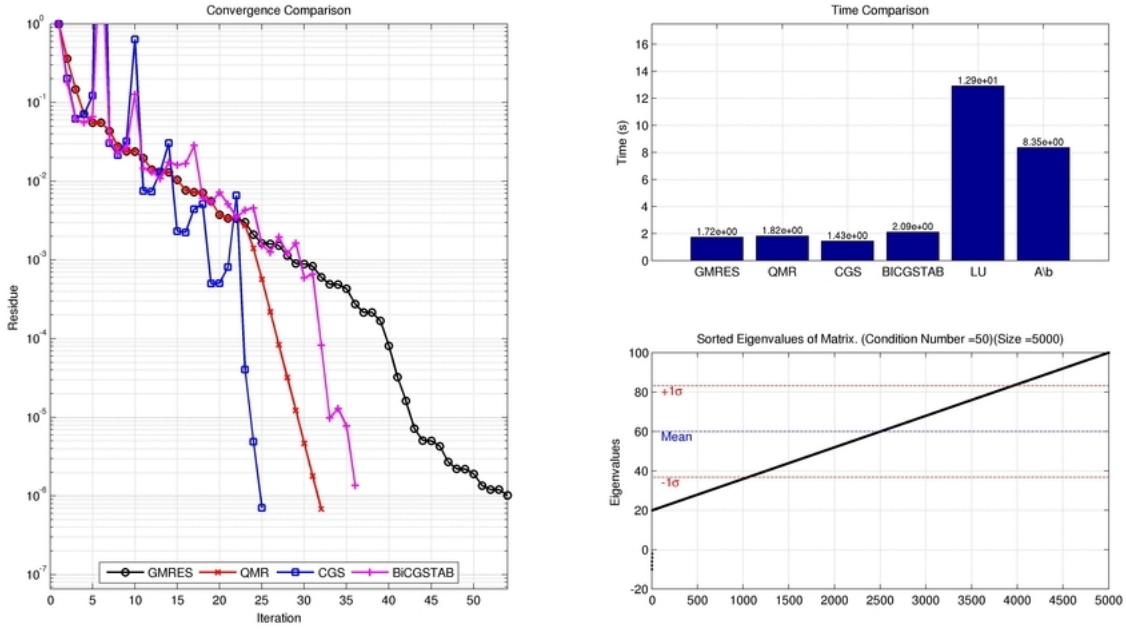

Figure 4.30: Eigenvalues distributed from [20,100] and [-2,-10] with the former having more weight.

A few important conclusions can be drawn from the above results:

- The loss of symmetry for the eigenvalues about the X axis caused all the methods to converge as opposed to only GMRES when the eigenvalues were symmetrically distributed.

- The time for convergence for all methods was much lower than the time required by direct solvers.
- In all the above test cases, GMRES exhibited stable and quick convergence. If CGS was to be excluded, GMRES was the fastest algorithm on all 6 tests conducted above. The exclusion of CGS is warranted on account of its unstable nature in the presence of outliers.

4.2.3 Clustered of Eigenvalues

Along the same lines as the experiments conducted for matrices with clusters of eigenvalues, experiments were conducted for symmetric indefinite matrices with clusters of eigenvalues about 3 points.

- 3 Clusters with Centres at ±10 and -2 with Radius 1 (Figure 4.31)

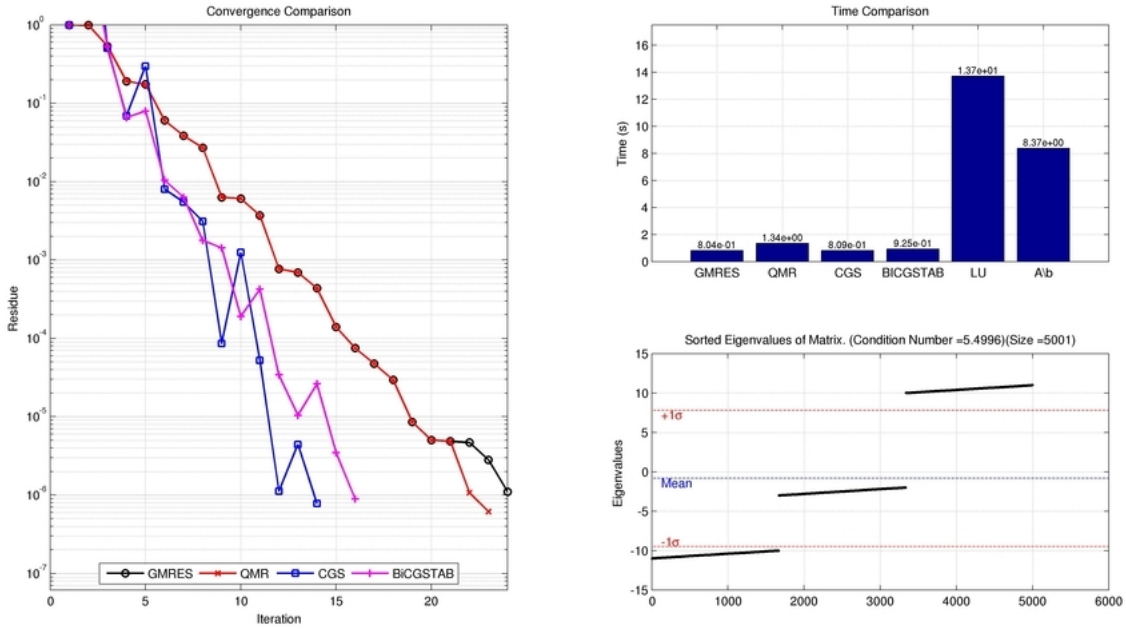

Figure 4.31: 3 Clusters with Centres at ±10 and -2 with Radius 1

- 3 Clusters with Centres at ±10 and 2 with Radius 1 (Figure 4.32)

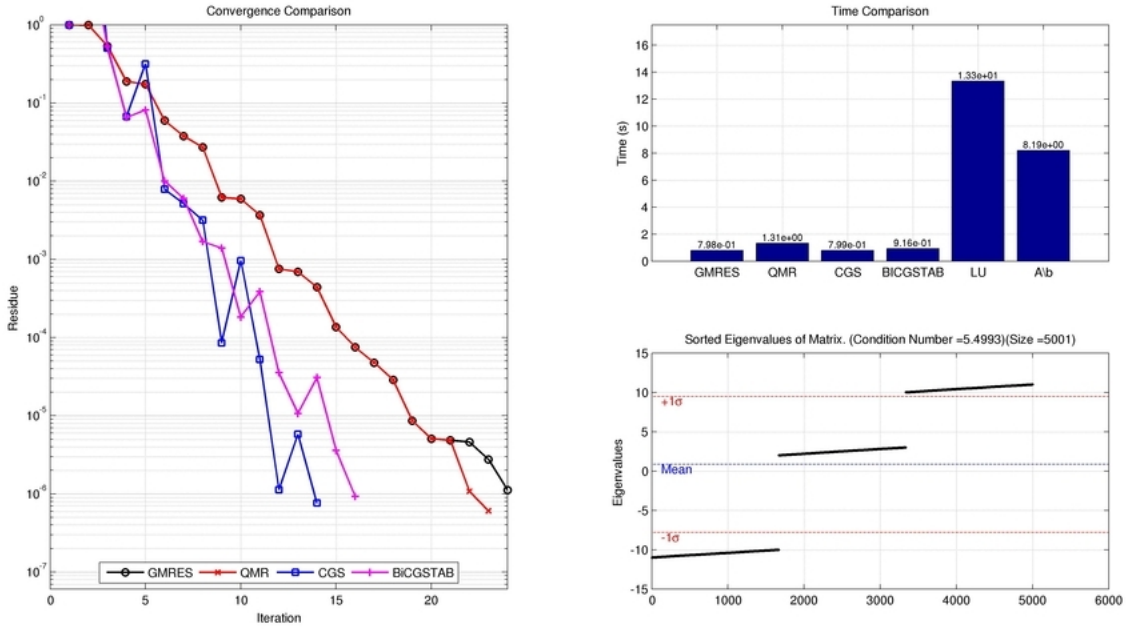

Figure 4.32: 3 Clusters with Centres at ±10 and 2 with Radius 1

- 2 Clusters with Centres at ±50 with Radius 20 (Figure 4.33)

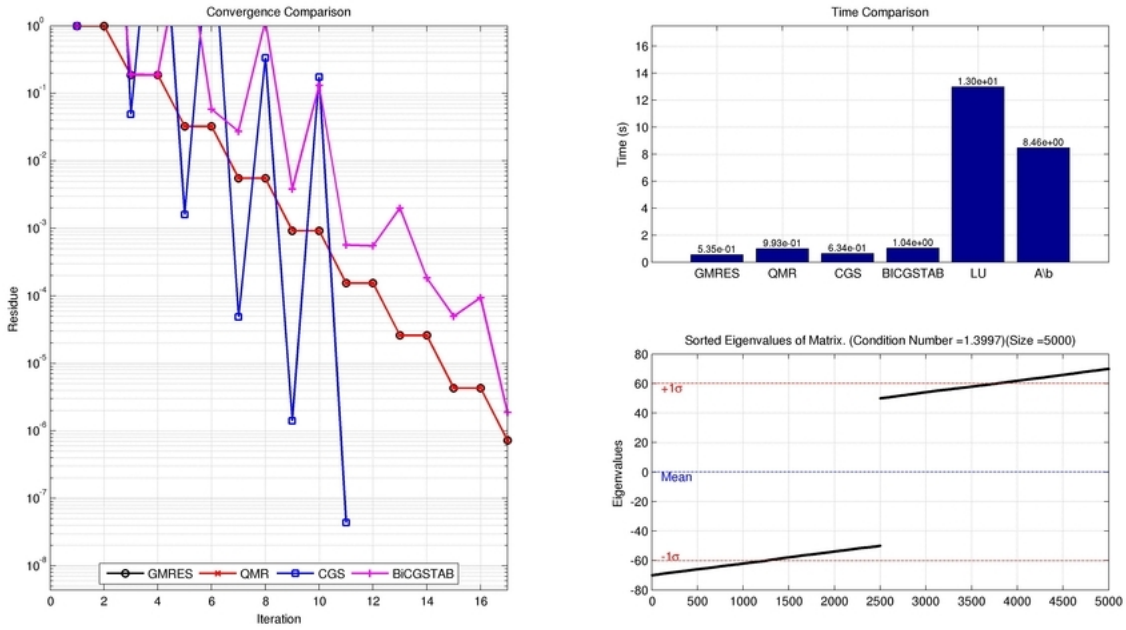

Figure 4.33: 2 Clusters with Centres at ±50 with Radius 20

These results yield 2 conclusions: Firstly, From the experiments with 3 cluster zones, (Figure 4.31 and Figure 4.31), one may conclude that for a given cluster which is a part of eigenvalue system with odd number of clusters, if the one of the clusters is swapped in sign keeping all other constant, the convergence of the methods is unaffected. Secondly, in

section 4.2.1, linearly distributed eigenvalues on both side of the axis were found to cause stagnation (or divergence) in these methods. However, as seen in (Figure 4.33), the methods converge fairly well if the eigenvalues were distributed as cluster instead of linearly about the same points. In all of the above cases, GMRES was again among the fastest algorithms both in terms of time for completion and residue convergence. Further, the time required for iterative methods was much lesser than the time required for direct solvers.

4.2.4 Random Distribution of Eigenvalues

Experiments were conducted for the behaviour of iterative algorithms when the eigenvalues were distributed normally about the mean with a set standard deviation. 2 experiments were conducted and the mean was chosen to be 0. This allowed the eigenvalues to approach 0 and allow for poor conditioning of the matrix. The standard deviation was kept at 1 for the first experiment and 0.01 for the second.

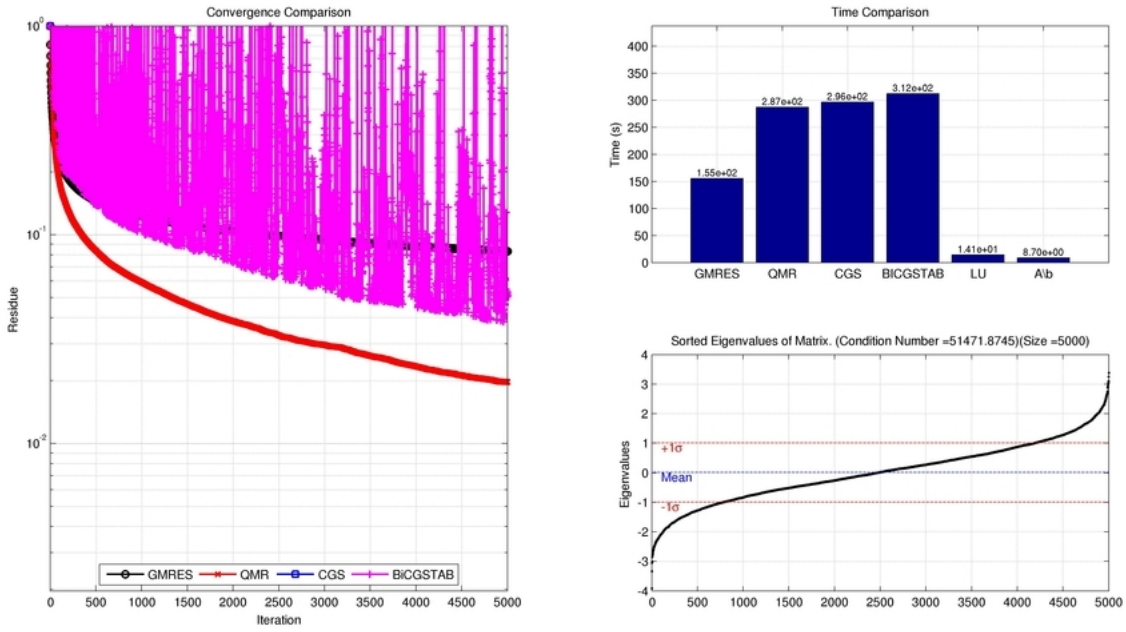

Figure 4.34: Mean = 0 , STD = 1

•

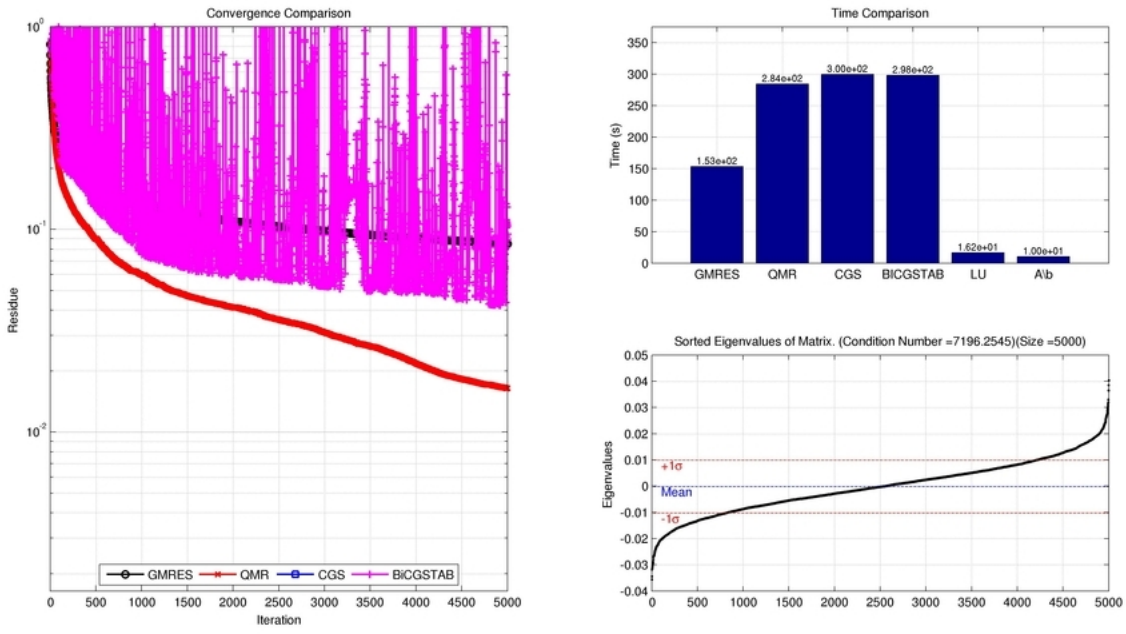

Figure 4.35: Mean = 0 , STD = 0.01

•

The following conclusions can be drawn from the experiment:

- While all algorithms failed to converge to the set tolerance, BiCGSTAB exhibited extreme instability with the above test cases.
- GMRES and QMR had relatively stable residue convergence but both failed to converge.
- Running GMRES with a higher restart value aligned the residue reduction curve with that of QMR, however, higher values of restart tested did not cause the convergence of the algorithm.

4.2.5 Presence of Eigenvalues Clustered Around the Origin

Similar to the previous section, experiments were conducted to test the algorithms for matrices with small valued eigenvalues.

- Minimum absolute eigenvalue of matrix = 0.1

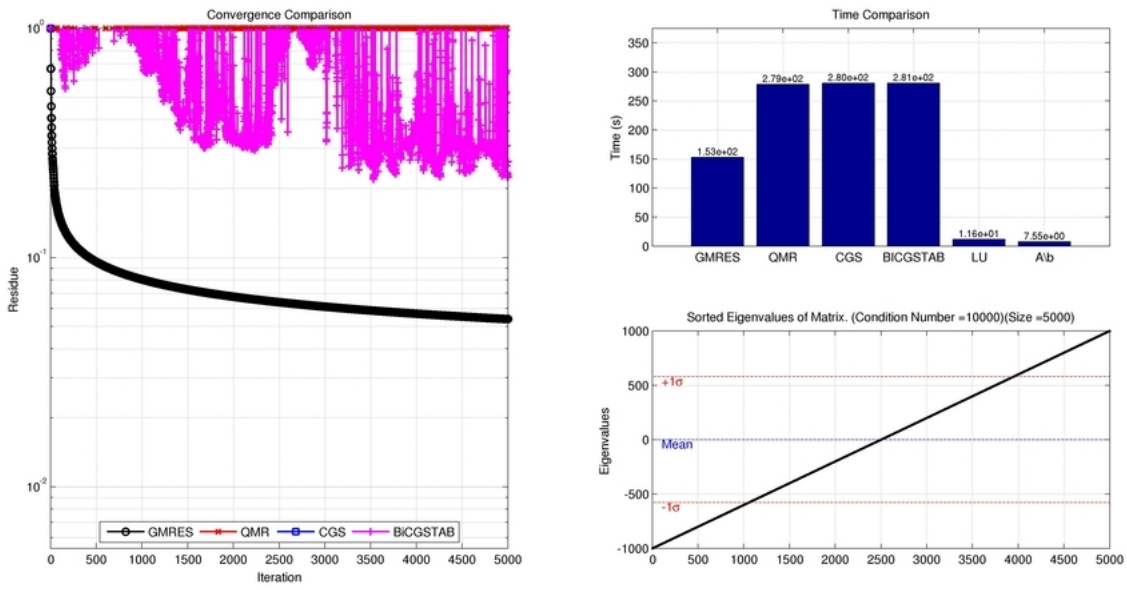

Figure 4.36: min(|eig(A)|) = 0.1

- Minimum absolute eigenvalue of matrix = 0.01

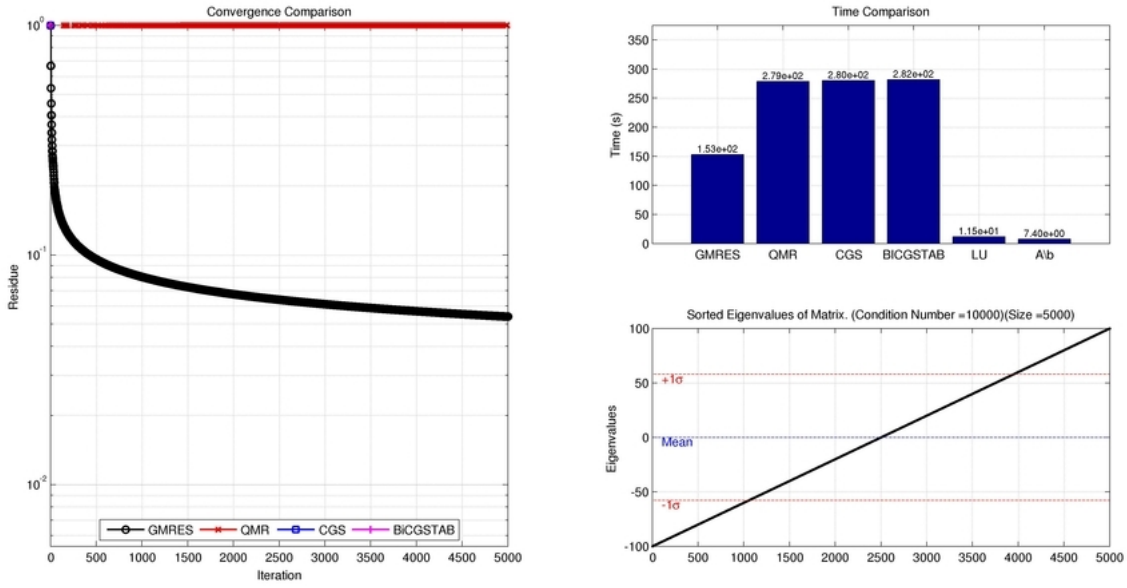

Figure 4.37: min(|eig(A)|) = 0.01

Although none of the algorithms converged, GMRES was the only algorithm to have a substantial decrease in the residue from the initial residue. Increasing the restart value for GMRES did not cause convergence. In this test case and the previous cases , BiCGSTAB and CGS were found to have residue higher than initial residues during the algorithm solving process. The instability of these algorithms make them a poor choice for a black box implementation of solvers for solving large dense linear systems.

Chapter 5

Conclusions of Numerical Experiments

A few conclusions drawn from the results of the experiments on both positive definite and indefinite systems are listed below.

- GMRES converges very poorly for matrices with linear distribution of eigenvalues in positive definite matrices in both time and residue characteristics. On the other hand, GMRES converges for linear distribution of eigenvalues in indefinite matrices. Results showed that GMRES was the only algorithm which converged for this set of matrices.

- QMR converges well for positive definite matrices of all structures tested. The convergence was within acceptable limits of both time and residue characteristics. However, QMR fails for indefinite matrices with linear distribution of eigenvalues.

- All methods converged well for matrices with clusters of eigenvalues. This result holds true for both positive definite and indefinite matrices.

- All methods fail for extremely ill-conditioned indefinite matrices. Ill conditioned can be defined as matrices with significant clustering of eigenvalues in the vicinity of the origin. Also, the degree of poor conditioning of a matrix influences the time and residue rate of convergence in case of iterative algorithms. In case of direct solvers, the execution time for an algorithm is roughly constant for a given problem size.

- The convergence rate and the time for execution of GMRES depends greatly on the restart value. A low restart value for a matrix with linear uniform distribution of eigenvalue will ensure slow convergence and high execution time. A high value in such case would allow for quicker execution and higher convergence rate.

- In cases of ill-conditioned matrices where all methods fail but QMR exhibits a better

performance, increasing the GMRES restart value aligns the residue plot of GMRES with that of QMR.

- For most test cases with a well defined conditioning of the matrix, the time taken by iterative methods was comparable to the time taken by direct solvers.
- Among all the experiments conducted, iterative methods were appreciably slower than direct methods only for indefinite systems with clustering of eigenvalues close to 0. In this case, iterative methods failed to converge. Even in the case of linear distribution of eigenvalues for indefinite systems, GMRES did converge in a time frame comparable to direct methods.
- In most cases discussed, iterative methods were faster than direct methods. This is especially true for matrices with clusters. The presence of outliers did influence the convergence of the CG family of methods, however QMR and GMRES were able to compete with direct solvers in terms of time.
- Although for positive definite systems, CG, CGS and BiCGSTAB are algorithms which guarantee a solution in N iterations where N is the size of the matrix, their convergence has been shown to be unstable. Even in spite of their poor residue characteristics, the time for execution in most well-conditioned matrices has been comparable to those of other methods. However, in the case of indefinite systems, BiCGSTAB and CGS failed to converge for a large number of test cases; even with well conditioned ones.

5.1 Choice of Algorithm

In the following subsections conclusions regarding the choice of algorithm have been rendered. The conclusions apply to the matrices of spectrum similar to the ones discussed in the thesis.

5.1.1 Positive Definite Systems

For positive definite systems, the default choice of algorithm is CG. Experiments conducted have suggested that CG is one of the fastest methods for solving such systems. BiCGSTAB and CGS are poor algorithms in the presence of eigenvalue outliers whereas GMRES (with a low restart value) was a poor algorithm for eigenvalues of linear distribution. QMR was consistently a good algorithm both in convergence characteristics and execution time. However, owing to the higher number of operations necessary per iteration and the approximately same number of iterations to convergence, QMR was generally slower than CG. While QMR applies for matrices of any structure, CG applies to matrices which are symmetric positive

definite. A comment is made regarding the choice of methods when the structure of the matrix is unknown.

5.1.2 Indefinite Systems

For indefinite systems, BiCGSTAB and CGS have exhibited oscillations and unstable convergence. Thus, the choices for the algorithm are QMR and GMRES. Among them, GMRES converged for larger number of test cases, had greater flexibility and had user-set memory requirements proportional to the restart value.

There is one important deficiency for each of GMRES and QMR: GMRES has linearly growing storage requirement and QMR requires transposed matrix-vector product. If the problem or the hardware constrains the use of any of these methods, then the choice is clear. For instance, QMR is the only choice for a memory deficient system while GMRES is the only choice when the matrix is provided through a matrix-vector routine (as opposed to providing the matrix explicitly) without a tranpose operator. However, in case neither of these deficiencies pose a problem, the choice of the algorithm is more of a *trial and error* answer.

In a randomly chosen test case, GMRES is not only more likely to converge but is also more likely to be faster than QMR. GMRES has also been studied[10] and improved for its scalability[1] and maps better than QMR to emerging architectures. In conclusion, while QMR is certainly a stable and fast algorithm for solving indefinite systems, the uncertainity of convergence and the limited scalability act as disadvantages for its choice. Thus, GMRES(with a high restart number) would a preferred algorithm for these type of matrices.

5.1.3 Unknown Spectrum

Without the knowledge of the spectrum of the matrix, attempting to solve the system with iterative methods would probably lead to poor performance. This can be caused because of a number of reasons including :

1. Poor spectrum of the matrix with too many eigenvalues close to 0. In such a case, it is possible that none of the algorithms converge.

2. Poor spectrum of the matrix with eigenvalues distributed in a fashion which increases the number of iterations required for convergence.

3. Poor choice of algorithm

4. Poor scalability of algorithm for concerned architecture.

However, with no knowledge of the spectrum, GMRES is an algorithm which, although not guaranteed, is probable to run better than all other iterative methods. The reasons are the same as stated above for the comparison between GMRES and CG. GMRES is scalable, converges for higher number of test cases and the user has control over an additional parameter, the restart number, which can greatly influence convergence.

In conclusion, for a matrix whose spectrum is not known a-priori, as far as possible, a direct solve should be attempted. However, in case this is not possible, GMRES should be chosen as the first preference followed by QMR. If both these methods fail and a preconditioner is not available, a direct solve is unavoidable.

5.2 Scope for Future Work

5.2.1 Preconditioners

Preconditioners are matrices that are multiplied to the original system to make it more *amenable*. By using intelligent preconditioners, it is possible to drastically reduce the time needed to solve systems. For instance, by providing a Full LU factorization to QMR, it converges in a single step with remarkable accuracy. This was rather trivial since by using a pair of backward and forward solvers, we could achieve an excellent accuracy as well. However, instead of a full factorization if we supplied incomplete versions of the factorizations, we would still be able to reduce the time well.
Preconditioners, more often than not, require us to closely understand the field in which the matrix occurs because preconditioners have a physical significance. A single matrix is able to bring the condition number of the system to a number that is much more manageable.

$$Ax = b \tag{5.1}$$

$$KAx = Kb \tag{5.2}$$

The best preconditioner would be the inverse of the system. But if this was accurately known, there was no real need to do any effort in getting the solution since direct multiplication of the inverse and b would yield the answer. For a matrix with unknown spectrum, application of unpreconditioned Krylov algorithms would probably cause stagnation or divergence. By using a good preconditioner, it is possible to reduce the original system to one of the cases where the Krylov algorithms performed better than direct solvers.

The design of preconditioners, currently, tends to be highly application specific. It is difficult to find a preconditioner which will work for all systems. Two candidates include : Partial LU factorization with thresholding and Multigrid. However, LU factorization with uniformly distributed elements is not likely to help since the the partial LU factorization with

thresholding is actually, the full LU factorization. Multigrid, by itself, is $O(n)$ and tends to be useful only for a certain class of linear systems.

$$A^{-1}Ax = A^{-1}b \tag{5.3}$$

$$x = A^{-1}b \tag{5.4}$$

5.2.2 GPU Parallelization

Recently, there has been an increase in the applications of graphic processors (GPUs) for scientific computing. Through CUDA, it is possible to program the GPU to compute single and double precision matrix calculations. There are significant advantages and disadvantages to using such systems but they will surely occupy an important position in the future. Many groups (such as Shi et al. [27]) have reported 100x speedup while working with GPUs for scientific computing. Much research has been devoted in trying to prove or disprove[31] its advantages but research isn't yet conclusive[16].

5.2.3 Hybrid Systems

Research is due in the field of hybrid systems that is using both CPU and GPU for computation. There are 3 significant directions through which the speed of solving a system can be reduce : Preconditioners, Parallelization and Algorithm Choice. GPUs have higher computation capability at single precision and CPUs have traditionally been used for everything. By using GPUs to precondition a matrix while CPUs iterate, it might be possible to reduce the execution time. This is still a field of active research. A good solution for iterative methods would be to use single precision GPUs to find a preconditioner for the linear system and supply it to the CPU which iterates to the solution. Another alternative is to have an approximate solution through the GPU which is refined by the CPU.

Bibliography

[1] Donald C. S. Allison, Maria Sosonkina, and Layne T. Watson. Scalability analysis of parallel gmres implementations.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8 (paperback).

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[4] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ISBN 0262533022, 9780262533027.

[5] Intel Corporation. Intel math kernel library for linux* os, 2009. URL `http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/fortran/lin/mkl/userguide.pdf`.

[6] Intel Corporation. Intel core i7-600, i5-500, i5-400 and i3-300 mobile processor series, January 2010. URL `http://download.intel.com/design/processor/datashts/322812.pdf`.

[7] Jane K. Cullum. Iterative methods for solving ax = b: Gmres/fom versus qmr/bicg. In *Advances in Computational Mathematics*, 1996.

[8] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[9] Mark Embree. The tortoise and the hare restart gmres. *SIAM Review*, 45:2003.

[10] Mark Embree. How descriptive are gmres convergence bounds? Technical report, Oxford University Computing Laboratory, 1999.

[11] Roland W. Freund and Nol M. Nachtigal. Qmr: a quasi-minimal residual method for non-hermitian linear systems, 1991.

[12] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press. ISBN 9780801854149.

[13] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008. ISSN 0098-3500. doi: 10.1145/1356052.1356053. URL `http://doi.acm.org/10.1145/1356052.1356053`.

[14] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics. ISBN 9780898713961.

[15] Anne Greenbaum and Zdenek Strakos. Any nonincreasing convergence curve is possible for gmres. *SIAM J. Matrix Anal. Appl*, 17:465–469, 1996.

[16] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate gpu vs. cpu performance without the answer. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS, Austin, TX, April 2011.

[17] Martin H. Gutknecht. A brief introduction to krylov space methods for solving linear systems.

[18] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`.

[19] Jörg Liesen and Petr Tichý. Convergence analysis of Krylov subspace methods. *GAMM-Mitt.*, 27(2):153–173, 2004.

[20] Noël M. Nachtigal, Satish C. Reddy, and Lloyd N. Trefethen. How fast are nonsymmetric matrix iterations. *SIAM J. Matrix Anal. Appl.*, 13(3):778–795, July 1992. ISSN 0895-4798. doi: 10.1137/0613049. URL `http://dx.doi.org/10.1137/0613049`.

[21] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[22] Travis E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007. URL `http://link.aip.org/link/?CSX/9/10/1`.

[23] Fernando Pérez and Brian E. Granger. Ipython: A system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007. URL `http://link.aip.org/link/?CSX/9/21/1`.

[24] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. ISBN 0898715342.

[25] See homepage for details. Atlas homepage. http://math-atlas.sourceforge.net/.

[26] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[27] Jin Shi, Yici Cai, Wenting Hou, Liwei Ma, Sheldon X.-D. Tan, Pei-Hsin Ho, and Xiaoyi Wang. Gpu friendly fast poisson solver for structured power grid network analysis. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 178–183, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-497-3. doi: 10.1145/1629911.1629961. URL `http://doi.acm.org/10.1145/1629911.1629961`.

[28] G. Strang. *Linear Algebra and Its Applications*. Thomson, Brooks/Cole, 2006. ISBN 9780030105678.

[29] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997. ISBN 0898713617.

[30] Henk A. van der Vorst. Krylov subspace iteration. *Computing in Science and Engg.*, 2(1):32–37, January 2000. ISSN 1521-9615. doi: 10.1109/5992.814655. URL `http://dx.doi.org/10.1109/5992.814655`.

[31] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In *HotPar'10: Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, page 13, Berkeley, CA, USA, 2010. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1863086.1863099`.

[32] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.**
URL: `http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`.

[33] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[34] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2): 101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[35] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).