Matlab Software for Iterative Methods and Algorithms to Solve a Linear System

PROF. D. A. GISMALLA

Abstract— **The term "iterative method" refers to a wide range of techniques which use successive approximations to obtain more accurate solutions .In this paper an attempt to solve systems of linear equations of the form AX=b, where A is a known square and positive definite matrix. We dealt with two iterative methods namely stationary(Jacobi, Gauss-Seidel, SOR) and non-stationary (Conjugate Gradient, Preconditioned Conjugate Gradient).To achieve the required solutions more quickly we shall demonstrate algorithms for each of these methods .Then using Matlab language these algorithms are transformed and then used as iterative methods for solving these linear systems of linear equations. Moreover we compare the results and outputs of the various methods of solutions of a numerical example. The result of this paper the using of non-stationary methods is more accurate than stationary methods. These methods are recommended for similar situations which are arise in many important settings such as finite differences, finite element methods for solving partial differential equations and structural and circuit analysis .**

***Index Terms*—Matlab language, iterative method, Jacobi.**

I. *STATIONARY ITERATIVE METHODS*

The stationary methods we deal with are Jacobi iteration method, Gauss-Seidel iteration method and SOR iteration method.

A. Jacobi iteration method

The Jacobi method is a method in linear algebra for determining the solutions of square systems of linear equations. It is one of the stationary iterative methods where the number of iterations is equal to the number of variables. Usually the Jacobi method is based on solving for every variable x_i of the vector of variables $X^T=(x_1,x_2,\ldots,x_n)$, locally with respect to the other variables .One iteration of the method corresponds to solve every variable once .The resulting method is easy to understand and implement, but the convergence with respect to the iteration parameter k is slow.

B. Description of Jacobi's method:-

Consider a square system of n linear equations in n variables

$$AX=b \qquad (1.1)$$

where the coefficients matrix known A is

$$A=(a_{i,j}) \quad i,j=1(1)n$$

The column matrix of unknown variables to be determined X is $X^t=(x_1,x_2,\ldots,x_n)$

and the column matrix of known constants b is

$$b^t=(b_1,b_2,\ldots,b_n)$$

The system of linear equations can be rewritten

$$(D+R)X=b \qquad (1.2)$$

where A=D+R , D $=(a_{i,i})$ $i=1(1)n$ is the diagonal matrix D of A and $R=L+U$, where L and U are strictly lower and strictly Upper matrix of A

Therefore, if the inverse D^{-1} exists and Eqn.(1.2) can be written as

$$X=D^{-1}(b-RX) \qquad (1.3)$$

The Jacobi method is an iterative technique based on solving the left hand side of this expression for X using a previous value for X on the right hand side .Hence , Eqn.(1.3) can be rewritten in the following iterative form after k iterations as

$$X^{(k)}=D^{-1}(b-RX^{(k-1)}) \text{ ,k=1,2,}\ldots \qquad (1.4)$$

Rewriting Eqn.(1.4) in matrix form . We get by equating corresponding entries on both sides

$$x_i^{(k)}=\frac{1}{a_{ii}}\left(b_i-\sum_{\substack{j=1\\ j\neq i}}^{n}a_{ij}\,x_j^{(k-1)}\right),$$

i=1,…,n ,k=1,2,…. (1.5)

We observe that Eqn. (1.4) can be rewritten in the form

$$X^{(k)}=TX^{(k-1)}+C \qquad k=1,2,\ldots\ldots, \qquad (1.6)$$

where $T=D^{-1}(-L-U)$ and $C=D^{-1}b$

or equivalently as in the matrix form of the Jacobi iterative method

$X^{(k)}=D^{-1}(-L-U)X^{(k-1)}+D^{-1}b$ k=1,2,….

where $T=D^{-1}(-L-U)$ and $C=D^{-1}b$

In general the stopping criterion of an iterative method is to iterate until

$$\frac{\|X^{(k)}-X^{(k-1)}\|}{\|X^{(k)}\|}<\epsilon$$

for some prescribed tolerance $\epsilon>0$ .For this purpose, any convenient norm can be used and usually the L_∞ norm , i.e.

$\frac{\|X^{(k)}-X^{(k-1)}\|_\infty}{\|X^{(k)}\|_\infty}<\epsilon, X^{(k)}\neq 0.$

This means that if $X^{(k-1)}$ is an approximation to $X^{(k)}$,then the absolute error is $\|X^{(k)}-X^{(k-1)}\|_\infty$, and the relative error is $\frac{\|X^{(k)}-X^{(k-1)}\|_\infty}{\|X^{(k)}\|_\infty}$, provided that $X^{(k)}\neq 0$. This implies that a vector X^* can approximate X to t significant digits if t is the largest non negative integer for which $\frac{\|X-X^*\|_\infty}{\|X\|_\infty}<5*10^{-t}$

C. The Matlab Program for JACOBI

The Matlab Program for Jacobi's Method with it's Command Window is shown in Fig.(1.1)

D. Gauss-Seidel iteration 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

Manuscript received February 01, 2014.

PROF. D. A. GISMALLA, Dept. of Mathematics , College of Arts and Sciences, Ranyah Branch,TAIF University ,Ranyah(Zip Code :21975) , TAIF, KINDOM OF SAUDI ARAIBA, Moblie No. 00966 551593472 .

method will converge faster than the Jacobi method, though still relatively slowly.

Now if We consider again the system in Eqn.(1.1)

$$AX=b \qquad (1.1)$$

and if We decompose A into a lower triangular component L_* and a strictly upper triangular component U. Moreover if D is the diagonal component of A and L is the strictly lower component of A then

$$A= L_*+U$$

Where

$$L_* =L+D$$

Therefore the given system of linear equations in Eqn.(1.1) can be rewritten as:

$$(L_* +U)X=b$$

Using this we get

$$L_*X=b-UX \qquad (1.7)$$

The Gauss-Seidel method is an iterative technique that solves the left hand side of this expression for X, using previous values for X on the right hand side. Using Eqn.(1.7)iteratively ,it can be written as:

$$X^{(k)}= L_*^{-1}(b-UX^{(k-1)}),\ k=1,2,\ldots . \qquad (1.8)$$

provided that L_*^{-1} exists. Substituting for L_* from above we get

$$X^{(k)} = (D + L)^{-1}(b)-(D+L)^{-1}UX^{(k-1)},$$
$$k=1,2,\ldots, \qquad (1.9)$$

where $T = -((D + L)^{-1})U$ and $C=(D + L)^{-1}b$

Therefore Gauss-Seidel technique has the form

$$X^{(k)}=TX^{(k-1)}+C,\ k=1,2,\ldots. \qquad (1.10)$$

However ,by taking advantage of the triangular forms of D,L,U and as in Jacobi's method ,the elements of $X^{(k)}$ can be computed sequentially by forward substitution using the following form of Eqn. (1.5) above:

$$x_i^{(k)}=\frac{1}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}\, x_j^{(k-1)} - \sum_{j<i} a_{ij}\, x_j^{(k-1)}\right) \qquad (1.11)$$

$$i=1,2,\ldots,n \quad , \quad k=1,2,3,\ldots\ldots$$

The computation of $x_i^{(k)}$ uses only the previous elements of $X^{(k-1)}$ that have already been computed in advanced to iteration k.This means that ,unlike the Jacobi method, only one storage vector is required as elements can be over rewritten as they are computed ,which can considered as an advantageous for large problems .The computation for each element cannot be done in parallel . Furthermore, the values at each iteration are dependent on the order of the original equations.

E. The Matlab Program for Gauss-Seidel iteration method

The Matlab Program for **Gauss-Seidel** Method with it's Command Window is shown in the Fig.(1.2)

F. SOR iteration method (Successive Over Relaxation)

SOR method is devised by applying an extrapolation w to the Gauss-Seidel method .This extrapolation takes the form of a weighted average between the previous iteration and the computed Gauss-Seidel iteration successively.

If w>0 is a constant, the system of linear equations in Eqn.(1.1) can be written as

$$(D+wL)X=wb-[wU+(w-1)D]\ X \qquad (1.12)$$

Using this Eqn.(1.12) the SOR iteration method is given by

$$X^{(k)} = (D + wL)^{-1}[(1 - w)D - wU]X^{(k-1)} + w(D + wL)^{-1}b \quad k=1,2,3\ldots \quad (1.13)$$

provided $(D + wL)^{-1}$ exists .

```
function jacobi2(A,b,tol,x,N)
%jacobi(A, b, tol,X,N) solve iteratively a system of linear equations %whereby
%A is the coefficient matrix, and b is the right-hand side column vector
%tol : relative residual error tolerance for break condition
%X : Nx1 start vector (the initial guess)
%N is the maximum number of iterations
%The method implemented is the Jacobi iterative
%The starting vector is the null vector, but can be adjusted to one's needs
%The iterative form is based on the Jacobi transition/iteration matrix
%Tj = inv(D)*(L+U) and the constant vector cj = inv(D)*b
%The output is the solution vector X
%splitting matrix A into the three matrices L, U and D
  D = diag(diag(A));
  L = tril(-A,-1);
  U = triu(-A,1);
%transition matrix and constant vector used for iterations
Tj = inv(D)*(L+U);
cj = inv(D)*b;
k = 1;
while k <= N
x(:,k+1) = Tj*x(:,k) + cj;
B= norm(x(:,k+1)-x(:,k));
if B<tol
disp('The procedure was successful')
disp('Condition ||x^(k+1) - x^(k)|| <tol was met after k iterations')
disp(k); disp('x = ');disp(x(:,k+1));
break
end
k = k+1;
end
if B>tol || k > N
disp('Maximum number of iterations reached without satisfying condition:');disp('||x^(k+1) - x^(k)||
<tol'); disp(tol);
disp('Please, examine the sequence of iterates')
disp('In case you observe convergence, then increase the maximum number of iterations')
disp( x);
end
```

```
>> A=[ 0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1;
       1 -1 60 0 -2;  1 1 0 8 4 ;
        0 -1 -2 4 700] ;
>> x=[0;0;0;0;0];
>>  b=[1;2;3;4;5] ;
>> tol=0.00005 ;
>> N=91 ;
>> jacobi2(A,b,tol,x,N)
 The procedure was successful
Condition ||x^(k+1) - x^(k)|| <tol was
 met after k iterations
   91
x =
   7.8597
   0.4229
  -0.0736
  -0.5406
   0.0106
```

Fig.(1.1) The JACOBI Matlab Program with its Command Window for RUNNING it.

```
function gauss_seidel(A, b,tol,x, N)
 %Gauss_Seidel(A, b,tol,X, N) solve iteratively a system of linear %equationswhereby
 %A is the coefficient matrix, and b is the right-hand side column
%vector
%tol : relative residual error tolerance for break condition
%X : Nx1 start vector (the initial guess)
%N is the maximum number of iterations
%The method implemented is the Gauss-Seidel iterative
%The starting vector is the null vector, but can be adjusted to one's needs
%The iterative form is based on the Gauss-Seidel transition/iteration matrix
%Tg = inv(D-L)*U and the constant vector cg = inv(D-L)*b
%The output is the solution vector x
%splitting matrix A into the three matrices L, U and D

D = diag(diag(A));
L = tril(-A,-1);
U = triu(-A,1);

%transition matrix and constant vector used for iterations
 Tg = inv(D-L)*U;
cg = inv(D-L)*b;
k = 1;
while k <= N
x(:,k+1) = Tg*x(:,k) + cg;
B=norm(x(:,k+1)-x(:,k));
I f  B<tol
disp('The procedure was successful')
disp('Condition ||x^(k+1) - x^(k)|| <tol
                was met after k iterations')
 disp(k); disp('x = ');disp(x(:,k+1));
 break
 end
  k = k+1;
 end
if B >tol || k > N
disp('Maximum number of iterations reached without satisfying condition:')
disp('||x^(k+1) - x^(k)|| <tol'); disp(tol);
disp('Please, examine the sequence of iterates')
disp('In case you observe convergence, then increase the maximum number of iterations')
disp(x);
end
%  &&&&& IN THE COMMAND WINDOW TYPE TO RUN THE PROGRAM  &&&&&&&&
%  A=[0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60 0 -2; 1 1 0 8 4 ; 0 -1 -2 4 700]
%  N=20
%  b=[1;2;3;4;5]
%  x=[0;0;0;0;0]
%  tol=0.00005
%  gauss_seidel(A, b,tol,x, N)
```

```
>> A = [0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60 0 -2;
        1 1 0 8 4 ; 0 -1 -2 4 700];
>> x= [0;0;0;0;0];
>> b= [1;2;3;4;5];
>> N=32;
>> tol=0.00005 ;
>>  gauss_seidel(A, b,tol,x, N)
    The procedure was successful
    Condition ||x^(k+1) - x^(k)|| <tol
    was met after k iterations  31
    x =
       7.8596
       0.4229
      -0.0736
      -0.5406
       0.0106
```

Fig.(1.2) The Gauss-Seidel Matlab Program with its Command Window for RUNNING it.

```
function SOR(A, b,tol,x, N)
%sor(A, b, tol,x,N) solve iteratively a system of linear equations %whereby
%A is the coefficient matrix, and b is the right-hand side column vector
%tol : relative residual error tolerance for break condition
%X : Nx1 start vector (the initial guess)
%N is the maximum number of iterations
%The method implemented is that of Successive Over Relaxation
%The starting vector is the null vector, but can be adjusted to one's needs
%The iterative form is based on the SOR. transition/iteration matrix
%Tw = inv(D-w*L)*((1-w)*D+w*U) and the constant vector
% cw = w*inv(D-w*L)*b
%The optimal parameter w is calculated using the spectral radius of the
%Jacobi  transition matrix Tj = inv(D)*(L+U)
%The output is the solution vector X

%splitting matrix A into the three matrices L, U and D
D = diag(diag(A));
L = tril(-A,-1);
U = triu(-A,1);

Tj = inv(D)*(L+U); %Jacobi iteration matrix
rho_Tj = max(abs(eig(Tj)));     %spectral radius of Tj
w = 1.25;%overrelaxation parameter
disp('w ='');disp(w);
Tw = inv(D-w*L)*((1-w)*D+w*U); %SOR iteration matrix
cw = w*inv(D-w*L)*b; %constant vector needed for iterations
k = 1;
while k <= N
x(:,k+1) = Tw*x(:,k) + cw;
B=norm(x(:,k+1)-x(:,k));
if  B<tol
disp('The procedure was successful')
disp('Condition ||x^(k+1) - x^(k)|| <tol was met after k iterations')
disp(k); disp('x = ');disp(x(:,k+1));
break
end
k = k+1;
end
if B >tol || k >=N
disp('Maximum number of iterations reached without satisfying condition:')
disp('||x^(k+1) - x^(k)|| <tol'); disp(tol);
disp('Please, examine the sequence of iterates')
disp('In case you observe convergence, then increase the maximum number of iterations'); disp(x);
end
% &&&&& IN THE COMMAND WINDOW TYPE TO RUN THE PROGRAM  &&&&&&
% A=[0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60 0 -2; 1 1 0 8 4 ; 0 -1 -2 4 700];
% N=30;
% b=[1;2;3;4;5]; % x=[0;0;0;0;0];
% tol=0.00005 ;  %  SOR(A, b,tol,x, N)
```

```
>> A=[0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60
      0 -2;  1 1 0 8 4 ; 0 -1 -2 4 700];
>> N=32;
>> tol=0.00005 ;
>> x=[0;0;0;0;0];
>> b=[1;2;3;4;5];
>> SOR(A, b,tol,x, N)
w =  1.2500

The procedure was successful
Condition ||x^(k+1) - x^(k)|| <tol
was met after k iterations   15
x =
   7.8597
   0.4229
  -0.0736
  -0.5406
   0.0106
```

Fig.(1.3) The SOR Matlab Program with its Command Window for RUNNING it.

Now , let $T_w = (D + wL)^{-1}[(1 - w)D - wU]$
and $C_w = w(D + wL)^{-1}b$
Then the SOR technique has the iterative form

$$X^{(k)} = T_w X^{(k-1)} + C_w \quad \text{for a constant w> 0, k=1,2,…,} \qquad (1.14)$$

By Eqn. (1.14) it is obvious that the method of SOR

is an iterative technique that solves the left hand side of this expression for $X^{(k)}$ when the previous values for $X^{(k-1)}$ on the right hand side are computed .Analytically ,this may be written as:

$$X^{(k)} = (D + wL)^{-1}\left(wb - [wU + (w - 1)D]X^{(k-1)}\right), \quad k=1,2,3\ldots \qquad (1.15)$$

By taking advantage of the triangular form of (D+ wL) it can proved that $(D + wL)^{-1} = D^{-1}$.
Using this, the elements of $X^{(k)}$ can be computed sequentially using forward substitution as in Eqn.(1.3) in section 1.1and Eqn.(1.10) in section 1.2 above , i.e.

$$x_i^{(k)} = (1 - w)x_i^{(k-1)} + \frac{w}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij} x_j^{k-1} - \sum_{j<i} a_{ij} x_j^{k-1}\right), \quad i=1,2,\ldots,n,\ k=0,1,\ldots. \qquad (1.16)$$

The choice of the relaxation factor w is not necessarily easy and depends upon the properties of the coefficient matrix .For positive –definite matrices it can be proved that 0<w<2will lead to convergence, but we are generally interested in faster convergence rather than just convergence.

G. The Matlab Program for SOR iteration method

The Matlab Program for **SOR** Method with it it's Command Window is shown in Fig.(1.3)

II. THE CONJUGATE GRADIENT METHOD

It is used to solve the system of linear equations

$$\mathbf{Ax} = \mathbf{b} \qquad (2.1)$$

for the vector **x** where the known *n*-by-*n* matrix **A** is symmetric (i.e. $\mathbf{A}^T = \mathbf{A}$), positive definite (i.e. $\mathbf{x}^T\mathbf{Ax} > 0$ for all non-zero vectors **x** in $\mathbf{R}^n$), and real, and **b** is known as well. We denote the unique solution of this system by $\mathbf{x}_*$.

A. The conjugate gradient method as a direct method

We say that two non-zero vectors *u* and *v* are conjugate (with respect to *A*) if

$$\mathbf{u}^T\mathbf{Av} = 0. \qquad (2.2)$$

Since **A** is symmetric and positive definite, the left-hand side defines an inner product

$$(U,V)_A := (AU,V) = (U,A^TV) = (U,AV) = U^TAV \quad (2.3)$$

Two vectors are conjugate if they are orthogonal with respect to this inner product. Being conjugate is a symmetric relation: if **u** is conjugate to **v**, then **v** is conjugate to **u**. (Note: This notion of *conjugate* is not related to the notion of complex conjugate.)
Suppose that $\{\mathbf{P_k}\}$ is a sequence of *n* mutually conjugate directions. Then the $\{\mathbf{P_k}\}$ form a basis of $\mathbf{R}^n$, so we can expand the solution $\mathbf{x}_*$ of **Ax** = **b** in this basis:

$$X_* = \sum_{i=1}^{n} \alpha_i P_i \qquad (2.4)$$

and we see that

$$b = AX_* = \sum_{i=1}^{n} \alpha_i A P_i \qquad (2.5)$$

The coefficients are given by

$$P_K^T b = P_K^T AX_* = \sum_{i=1}^{n} \alpha_i P_K^T A P_i = \alpha_K P_K^T A P_K \quad (2.6)$$

(because $\forall i \neq K, P_i, P_K$ are mutually conjugate)

$$\alpha_K = \frac{P_K^T b}{P_K^T A P_K} = \frac{(P_K,\ b)}{(P_K,\ P_K)_A} = \frac{(P_K,\ b)}{\|P_K\|_A^2} \qquad (2.7)$$

This result is perhaps most transparent by considering the inner product defined above.
This gives the following method for solving the equation **Ax** = **b**: find a sequence of *n* conjugate directions, and then compute the coefficients α_k.

B. The conjugate gradient method as an iterative method

The direct algorithm was then modified to obtain an algorithm which only requires storage of the last two residue

vectors and the last search direction, and only one matrix vector multiplication.

The algorithm is detailed below for solving **Ax** = **b** where **A** is a real, symmetric, positive-definite matrix. The input vector $\mathbf{x}_0$ can be an approximate initial solution or **0**.

$r_0 := b - Ax_0$
$P_0 := r_0$
$K := 0$
repeat
$\alpha_k := \frac{r_K^T r_K}{P_k^T A P_k}$
$x_{K+1} := x_K + \alpha_K P_k$
$r_{K+1} := r_K - \alpha_k A P_k$
if r_{K+1} is sufficiently small then exit loop
$\beta_k := \frac{r_{k+1}^T r_{K+1}}{r_K^T r_K}$
$P_{k+1} := r_{k+1} + \beta_k P_k$
$k := k + 1$
end repeat
The result is x_{K+1}

This is the most commonly used algorithm. The same formula for β_K is also used in the Fletcher–Reeves nonlinear conjugate gradient method.

C. The code in Matlab for the Conjugate

Iterative Algorithm in Fiq.(1.4) & Fiq.(1.5)

Alternatively another Matlab Code for Conjugate Gradient Algorithm is in Fiq.(1.5)

D. Convergence properties of the conjugate gradient method

The conjugate gradient method can theoretically be viewed as a direct method, as it produces the exact solution after a finite number of iterations, which is not larger than the size of the matrix, in the absence of round-off error. However, the conjugate gradient method is unstable with respect to even small perturbations, e.g., most directions are not in practice conjugate, and the exact solution is never obtained. Fortunately, the conjugate gradient method can be used as an iterative method as it provides monotonically improving approximations x_k to the exact solution, which may reach

the required tolerance after a relatively small (compared to the problem size) number of iterations. The improvement is typically linear and its speed is determined by the condition number $K(A)$ of the system matrix A: the larger is $K(A)$, the

slower the improvement.

If $K(A)$ is large, preconditioning is used to

replace the original system
$\mathbf{Ax - b = 0}$ with $M^{-1}(\mathbf{Ax - b}) = \mathbf{0}$ so that

$K(M^{-1}A)$ gets smaller than $K(A)$, see below

E. The preconditioned conjugate gradient method

Preconditioning is necessary to ensure fast convergence of the conjugate gradient method. The preconditioned conjugate gradient method takes the following form:

$r_0 := b - Ax_0$
$z_0 := M^{-1} r_0$
$P_0 := z_0$
$K := 0$
repeat
$\alpha_K := \frac{r_k^T z_k}{P_k^T A P_k}$
$x_{K+1} := x_K + \alpha_k P_k$
$r_{K+1} := r_K - \alpha_k A P_k$
if $\mathbf{r}_{k+1}$ is sufficiently small **then** exit loop **end if**
$z_{k+1} := M^{-1} r_{k+1}$
$\beta_k := \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
$P_{k+1} := z_{k+1} + \beta_k P_k$
$k := k + 1$
end repeat
The result is x_{k+1}

The above formulation is equivalent to applying the conjugate gradient method without preconditioning to the system

$$\mathbf{E^{-1}A(E^{-1})^T\hat{x} = E^{-1}b}$$

where $EE^T = M$ and $\hat{x} = E^T x$

The preconditioned matrix **M** has to be symmetric positive-definite and fixed, i.e., cannot change from iteration to iteration. If any of these assumptions on the preconditioned is violated, the behavior of the preconditioned conjugate gradient method may become unpredictable.

An example of a commonly used preconditioned is the incomplete Cholesky factorization.

F. The code in Matlab for the Preconditioned Conjugate Gradient Algorithm in Fig(1.6)

*G. **The flexible preconditioned conjugate gradient method***

In numerically challenging applications sophisticated preconditioners are used, which may lead to variable preconditioning, changing between iterations. Even if the preconditioned matrix is symmetric positive-definite on every iteration, the fact that it may change makes the arguments above invalid, and in practical tests leads to a significant slowdown of the convergence of the algorithm presented above. Using the **Polak–Ribière** formula

$$\beta_K := \frac{z_{K+1}^T (r_{K+1} - r_k)}{z_{K+1}^T r_K}$$

instead of the **Fletcher–Reeves** formula

$$\beta_K := \frac{z_{K+1}^T r_{K+1}}{z_{K+1}^T r_K}$$

may dramatically improve the convergence in this case. This version of the preconditioned conjugate gradient method can be called **flexible,** as it allows

```
function  [x, niter, flag] = CONJUGATE_GRAD(A, b, s, tol, maxiter)
%CG  Conjugate Gradients method
%Input parameters
%A : Symmetric, positive definite NxN matrix
%b : Right-hand side Nx1 column vector
%s : Nx1 start vector (the initial guess)
%tol : relative residual error tolerance for break condition
%maxiter : Maximum number of iterations to perform
%Output parameters
%x : Nx1 solution vector
%niter : Number of iterations performed
%flag : 1 if convergence criteria specified by TOL could
%not be fulfilled within (the specified maximum Successful)
%number of iterations, 0 otherwise
x= s;       % Set x0 to the start vector s
r = b - A*s;   % Compute first residuum
p = r;
rho = r'*r;
niter = 0;   % Init counter for number of iterations
flag = 0;   % Init break flag
%Compute norm of right-hand side to take relative residuum as
%break condition
normb = norm(b);
if normb<eps  % if the norm is very close to zero, take the
%absolute residuum instead as break condition
%norm(r) >tol  since the relative
warning(['norm(b) is very close to zero, taking absolute as a break condition']);
normb = 1;
end
while (norm(r)/normb>tol)   % Test break condition
a = A*p;
alpha = rho/(a'*p);
x = x + alpha*p;
r = r - alpha*a;
rho_new = r'*r;
p = r + rho_new/rho * p;
rho = rho_new;
niter = niter + 1;
% if max. number of iterations
if (niter == maxiter)
flag = 1;   % is reached, break
break
end
end
%  &&&&& IN THE COMMAND WINDOW TYPE TO RUN THE PROGRAM  &&&&&&
%  A=[0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60 0 -2; 1 1 0 8 4 ; 0 -1 -2 4 700];
%  maxiter=6;
%  b=[1;2;3;4;5];
%  s=[0;0;0;0;0];
%  tol=0.00005;
%  [x, niter, flag] = CONJUGATE_GRADIENT(A, b, s, tol, maxiter)
```

```
>> A=[0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60 0 -2; 1 1 0 8 4 ; 0 -1 -2 4 700];
>> maxiter=6;
>> b=[1;2;3;4;5];
>> s=[0;0;0;0;0];
>> tol=0.00005;
>> [x, niter, flag] = CONJUGATE_GRADIENT(A,  b, s, tol, maxiter)

  x =  7.8597  0.4229  -0.0736  -0.5406  0.0106
  niter  =  5
  flag  =  0
```

Fig.(1.4) The CONJUGATE_GRADIENT Matlab Program with its Command Window for RUNNING it & below it the output

```
function [iterationno , x] = conjgrad(A,b,x)
   r=b-A*x;
   p=r;
   rsold=r'*r;

   for i=1:10000000
      Ap=A*p;
      alpha=rsold/(p'*Ap);
      x=x+alpha*p;
      r=r-alpha*Ap;
      rsnew=r'*r;
      if sqrt(rsnew)<5e-10
          break;
      end
      p=r+rsnew/rsold*p;
      rsold=rsnew;
   end
   iterationno=i ;
% & IN THE COMMAND WINDOW TYPE TO RUN THE PROGRAM  &

% A=[0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60 0 -2; 1 1 0 8 4 ; 0 -1 -2 4 700];
% b=[1;2;3;4;5];
% x=[0;0;0;0;0];
% [iterationno , x] = conjgrad(A,b,x)
```

```
>> A=[0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60 0 -2; 1 1 0 8 4 ; 0 -1 -2 4 700];
>> b=[1;2;3;4;5];
>> x=[0;0;0;0;0];
>> [iterationno , x] = conjgrad(A,b,x)

      iterationno =6

      x=   7.8597
           0.4229
          -0.0736
          -0.5406
           0.0106
```

Fig.(1.5) The CONJUGATE_GRADIENT Matlab Program with its Command Window for RUNNING it & below it the output result

for variable preconditioning. The implementation of the flexible version requires storing an extra vector. For a fixed preconditioned, $z_{K+1}^T r_K = 0$ so both

formulas for β_K are equivalent in exact arithmetic, i.e.,

without the round-off error.

The mathematical explanation of the better convergence behavior of the method with the **Polak–Ribière** formula is that the method is **locally optimal** in this case, in particular, it converges not slower than the locally optimal steepest descent method.

III. Conclusion

When We ran each Matlab Code program and testified with a numerical example We observe that the non-stationary algorithms converge faster than the stationary algorithms.

The reader can look and compare at the output result programs in Figures denoted above ((these Figures are Fig.(1) , Fig(2) , ….,Fig(5) and Fig(6))) to ensure that non-stationary algorithms especially the Preconditioned Conjugate Algorithm approximates the solution accurately to five decimal places with the number of iterations N equals 5 which less compared to the other Algorithms tackled in this paper.

The paper also analysis the Convergence properties of the conjugate gradient method and shows there are two methods, the direct or the iterative conjugate.

Further, better convergence behavior can be achieved when using **Polak–Ribière formula** which converges **locally optimal not slower** than the locally **optimal steepest descent methed.**

Furthermore, these Algorithms can are recommended for similar situations which are arise in many important settings such as finite differences, finite element methods for solving partial differential equations and structural and circuit analysis.

```
function  Preconditioned_Conjugate(A,b,c,tol,N)
[n,m]=size(A);
 invc=inv(c);
x=zeros(n,1);
r=b;
p=invc*b;
y=invc*r;
err(1)=norm(r);
k=0;
while norm(r) >tol&&k<N
   k=k+1;
   z=A*p;
   alpha=(y'*r)/(p'*z);
   x=x+alpha*p;
  mew=r-alpha*z;
   ynew=invc*mew;
    beta=(ynew'*mew)/(y'*r);
    p=ynew+beta*p;
    r=mew;
    y=ynew;
err(k)=norm(r);
end
disp('  x='); disp(x');
 disp('  err='); disp(err);
%  &&&&& IN THE COMMAND WINDOW TYPE  TO RUN THE PROGRAM
&&&&&&&&&
%  A=[0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60 0 -2; 1 1 0 8 4 ; 0 -1 -2 4 700];
%  N=5 ;
%  b=[1;2;3;4;5];
%  c=eye(5,5);
%  tol=0.00005 ;
%  Preconditioned_Conjugate(A,b,c,tol,N)
```

```
>> A=[0.2 0.1 1 1 0 ; 0.1 4 -1 1 -1; 1 -1 60 0 -2; 1 1 0 8 4 ; 0 -1 -2 4 700]
;
>> N=5 ;
>>  b=[1;2;3;4;5];
>> c=eye(5,5);
>> tol=0.00005 ;
>> Preconditioned_Conjugate(A,b,c,tol,N)

  x=   7.8597   0.4229   -0.0736   -0.5406   0.0106
  err=  7.5271   5.5600   0.7239   0.5572   0.0000
```

Fig.(1.6) The Preconditioned_Conjugate Matlab Program with Command Window for RUNNING it & below it the output result

Acknowledgment

First, I would like to thank the department of Mathematics, Tabuk University Saudi Arabia to allow me teach the Course of Mathematica 2010 that give me the KNOWLEDGE of what the symbolic languages are. Second, I would like to thank the department of Mathematics, Gezira University Sudan to teach the Course of Numerical Analysis applied with Matlab Language during the years 2011 & 2012 that give me

the KNOWLEDGE to differentiate between the symbolic languages and the processing languages

REFERENCES

[1] David G. lay ,linear algebra and it's applications, 2nd ed,AddisonWesleyPublishing, Company, (August 1999).

[2]Jonathan Richard Shewchuk, An Introduction to the ConjugateGradient Method, Without the Agonizing Pain, School of Computer Science Carnegie Mellon University ,(August 1994).

[3]KendeleA.Atkinson, Introduction to Numerical Analysi, 2nd ed, John Wilev and Sons,(1988).

[4]Luenberger,David G., inear and Non linear Programming, 2nd ed, Addison Wesley ,Reading MA,(1984).

[5]Magnus R. Hestenes and Eduard, Methods of Conjugate Gradients for Solving Linear, Systems, the National Bureau of Standards ,(1952).

[6]Michael Jay Holst B.S. ,Software for Solving Linear Systems with Conjugate Gradient Methods, Colorado State University,(1987).

[7]Ortega ,J.M. , Numerical Analysis ,a second course, Academic press, NewYork,(1972).

[8]R.BarrettM.Berry and T.Fchan and J.Demmel and J.Donato and V.Eijkhout and R.Pozo and C.Romine and H.Vander Vorst , Template for the Solution of Linear Systems Building Blocks for Iterative Methods, 2nd ,Siam,(1994).

[9]Richard L.Burden, J.DouglasFaire Numerical Analysis, 7th ed, wadsworth group, (2001).

[10] Ssacson,E. and H.B.Keller, Analysis of Numerical Methods, John Wiley&Sons,New York (1996)

PROF. D. A. GISMALLA, Dept. of Mathematics , College of Arts and Sciences, Ranyah Branch,TAIF University ,Ranyah(Zip Code :21975), TAIF, KINDOM OF SAUDI ARAIBA, Moblie No. 00966 551593472 .