PROJECT A13

ENUME

Mazen Ibrahim

295924

Table of Contents

Introduction

In this chapter I will discuss briefly how my project is structured and what is it about. So this project was about investigating numerical analysis in the world of linear algebra and I was tasked to explore certain algorithms and methods to solve system of linear equations and compare them and find their pros and cons and explain why they behaved in a certain way. The way the report is structed is as follows. In theory I discuss the methods and how they work then in my analysis I provide the results I got from my code the I have written in MatLab which brings me to the last part of the report which is the appendix in that last chapter I will be providing all the functions I have used throughout the project to get my results used in analysis.

Machine Epsilon

Theory

Machine Epsilon is defined as the smallest number such that 1+macheps>1. It is the difference between 1 and the next nearest number representable as a machine number.

In IEEE 754 representation we either have 32 bit or 64 bit in both of them the very first bit on left side is always reserved for the sign if it is positive number it will be 0 and if it's negative number it will be represented as 1. In 32 bit we then have 8 bits reserved for shifted exponent followed by 24 bits but since first bit is always 1 we can omit it so it will be 23 bits for mantissa. In 64 bit we have 11 bits reserved for shifted exponent and 52 bit for mantissa since again first bit is omitted.

Now to relate both IEEE 754 floating point representation and machine epsilon, if there are t bit used for magnitude of mantissa then machine epsilon is 2^{-t}. Looking at IEEE 754 double precision representation we have magnitude of mantissa is 52 so our machine epsilon should be 2^{-52}.

Now the reason why machine epsilon is important is because it's upper bound on the absolute relative true error in representing a number, meaning our absolute relative true error will be always be lower than the value of machine epsilon.

Analysis

```
macheps = 1;                    %assigning value of 1 to macheps
while 1+(macheps) > 1           %running a loop until macheps reaches 0
    macheps = macheps/2;        %keep dividing macheps by 2
end                             %end loop when we reach smallest precision
macheps = macheps * 2;          % multiply it by 2 to find macheps
```

Executing the code above in Matlab we get our mecheps = 2.2204e-16 and when I compared that result to the built in function in Matlab **eps** I got the same answer which means my code works correctly. Also knowing my laptop uses double precision 2^{-52} confirms those results.

Gauss-Elimination (Partial Pivoting)

Theory

Gauss-Elimination is an algorithm for solving a system of linear of equations and it is also considered to be a finite method, meaning the solution is obtained after a finite number of elementary arithmetical operations defined by the method and the dimension of the problem.

When we are using Gauss-Elimination we can think of two ways to implement it to solve our system, which is either without pivoting or with pivoting. For without pivot method to succeed all the submatrices for a given matrix $A \in R^{n\times n}$ should be all non-singular and for $a_{kk}^{(k)} \neq 0$ for k = 1,….,n and matrix A has a decomposition A = LU where L is lower triangular with 1's on diagonal and U is upper triangular with nonzero diagonal elements but if for some k = 1…,n $a_{kk}^{(k)} = 0$ then the method without pivoting fails so we revert to pivoting and in order for pivoting method to succeed for the same matrix our submatrices should be non-singular and $a_{kk}^{(k)} \neq 0$ for k = 1,…,n but if for some reason after applying the algorithm we still end up with $a_{kk}^{(k)} = 0$ then we use the method fails due to the fact we can't obtain a unique solution and because either there isn't any or there is infinite number of solutions. Regardless the system provided, it is always better to implement the algorithm with pivoting at every step to get smaller numerical errors.

The algorithm for Gauss-Elimination partial pivoting is as follows:

1. We select the entry with largest absolute value from the column of the matrix that is currently considered is pivot element. In other words from $a_{jk}^{(k)}$ (k ≤ j ≤ n) elements we choose our value according to this $\left|a_{ik}^{(k)}\right| = \max_j \left\{\left|a_{kk}^{(k)}\right|, \left|a_{k+1,k}^{(k)}\right|, \dots, \left|a_{nk}^{(k)}\right|\right\}$
2. If $a_{ik}^{(k)} = 0$ then we stop and don't carry on with the process if all elements for $a_{jk}^{(k)}$, j = k,k+1,….,n of the k-th column were zero then A is singular but if $a_{ik}^{(k)} \neq 0$ then we interchange pivot row i and k-th row to get the largest entry on the main diagonal each time.
3. For i = k +1,k+2,…n , let $a_{i,k} = a_{i,k} \div a_{k,k}$ and for j = k+1,k+2,….n+1, let $a_{i,j} = a_{i,j} - a_{i,k}* a_{k,j}$. then our upper triangular matrix is obtained
4. Last step is back substitution the vector solutions to Ax = b will be obtained using the following formula : for i = n,n-1,…,1 we have $x_i = \frac{\left(a_{i,n+1} - \sum_{j=i+1}^{n} a_{i,j}x_j\right)}{a_{ii}}$

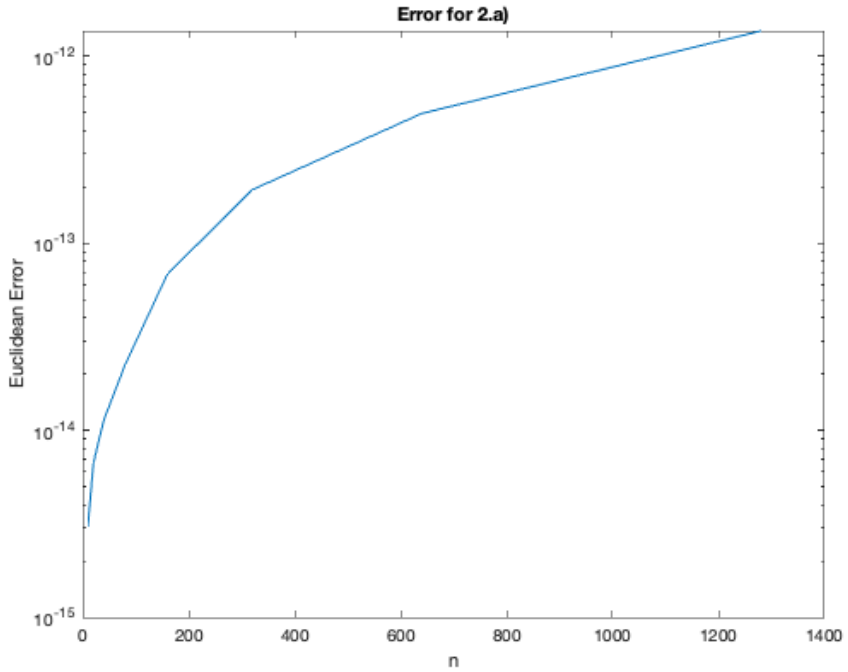
Now to calculate the Euclidian norm we are going to use this formula $\|x\|_2 := \sqrt{x_1^2 + \cdots + x_n^2}$ then to calculate the residuum we will take the value we got and use this

formula **r = Ax-b**. Calculating the Euclidian norm makes it easier for us to describe the size of matrix or vector then we can use this to consider whether we can bound the size product of a matrix and a vector and given we know both of their sizes it helps us decided whether they are compatible or not if they match then they are. These kind of bounds helps us in error analysis .

Analysis

After running the code written in Matlab which is found in the appendix of this report we generated the following results.

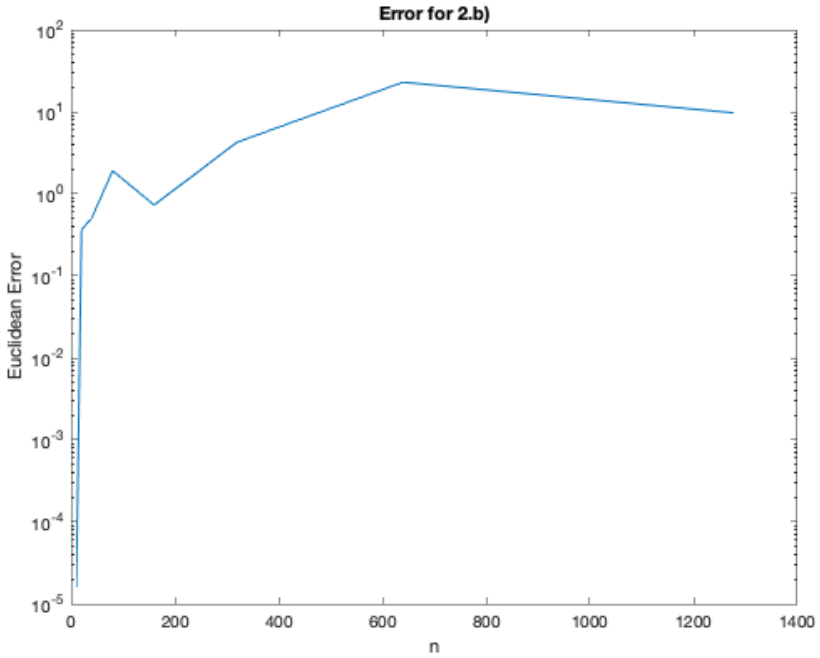
a)



As we can see as number of the equations increases the more Euclidean error increases as well due it to the fact the Gauss-Elimination has problems with rounding off.

Next I was tasked to print the solutions error for **n = 10** and note down the result then apply residual correction and compare both results to see if the error improved and after running my program my Euclidean error stayed the same as before residual and didn't improve much since the error is very small. The result was **3.0767e-15.**

b)



Unfortunately the method fails at **n = 20** because our matrix is ill-conditioned matrix which means it has a large condition number and to prove my assumption I calculated the condition numbers for Matrix 1from subtask 2a and Matrix 2 from subtask 2b for **n = 10** and I got the following results for Matrix 1 my condition number is **4.5503** while the condition number for Matrix 2 is **1.6027e+13** which is much higher than the condition number for well-conditioned Matrix 1. The way I calculated the condition number was as follows Cond(A) = norm(inv(A))*norm(A)

Again just like in previous task I was asked to print solutions error for **n = 10** and compare the results after and before the residual correction. There was slight improvement but the error was still high. The Euclidean error before residual correction is **1.6204e-05** and after the residual correction is **1.271e-05.** A printed values for x can be demonstrated during project presentation if asked.

Gauss-Seidel And Jacobi

Theory

In the last task we were discussing a finite method (direct method) to solve system of linear equations in this this task we will be dealing with iterative methods (indirect method) to solve the given system of linear equations. An iterative method is a mathematical procedure that uses an initial guess to generate a sequence of improving approximate solutions for a class of problems, in which the *n*-th approximation is derived from the previous ones. Iterative methods for solving systems of linear equations are used for problems of large dimension and with a sparse matrix A. Initial point is usually generated according to the following formula $x^{(i+1)} = Mx^{(i)} + w$ where A is a specific matrix and w is a vector. In order for iterative method to work properly we need to meet 2 conditions which are :

1) Convergence condition : sr (M) < 1 where sr (M) is spectral radius. This helps us to determine the rate of convergence as well
2) Coincidence condition : $\hat{x} = M\hat{x} + w$

The way to compare two 2 iterative methods is by measuring their effectiveness and the criteria we use for that are

1) The number of elementary arithmetic operations needed to perform a single iteration $x^{(i)}$ -> $x^{(i+1)}$
2) The speed of convergence, a measure indicating how fast the error decreases. $e^{(i)} = x^{(i)} - \hat{x}$

Another thing we should implement in our iterative algorithm when solving system of linear equations is stop tests which includes the following criteria to check when to terminate the iterations of iterative method :

1) Checking difference between two subsequent iteration points $\left\|x^{(i+1)} - x^{(i)}\right\| \leq \delta$ where δ is our assumed tolerance.
2) Checking a norm of the solution error vector $\left\|Ax^{(i+1)} - b\right\| \leq \delta_2$ if the test is not fulfilled within the assumed accuracy then the value of δ is diminished and we continue our iterations.

So the way Jacobi's Method works is as follows :

Matrix A is decomposed into A = L+D+U where L is sub-diagonal matrix, D is diagonal matrix and U is the matrix with entries above diagonal

Assuming diagonal entries of Matrix A are nonzero then a system of linear equations Ax = b can be written in this form $Dx^{(i+1)} = -(L+U)x^{(i)} + b$, i = 0,1,2,..

An equivalent form is as follows $x^{(i+1)} = -D^{-1}(L+U)x^{(i)} + bD^{-1}$

Since Jacobi's method is based on parallel computation the following scalar equation can be used $x_j^{(i+1)} = -\frac{1}{d_{jj}}(\sum_{k=1}^{n} l_{jjk} + u_{jk})x_k^{(i)} + b_j$. In order for Jacobi method to converge a strong diagonal dominance of the matrix A is required.

Next we will discuss Gauss-Seidel method as follows :

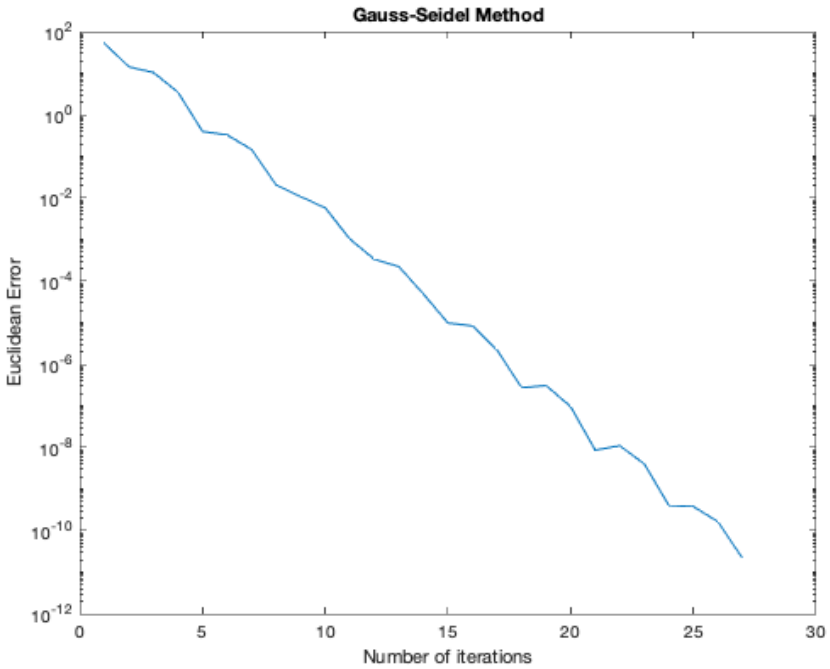
The matrix A is decomposed into L and D and U just like in Jacobi.

Assuming D is nonsingular a single iteration concept in Gauss-Seidel is based on the this formula $Dx^{(i+1)} = -Lx^{(i+1)} - Ux^{(i)} + b$, where i = 0,1,2,….

Gauss-Seidel is a sequential method which makes it faster than Jacobi and this will be proved in my analysis later on.

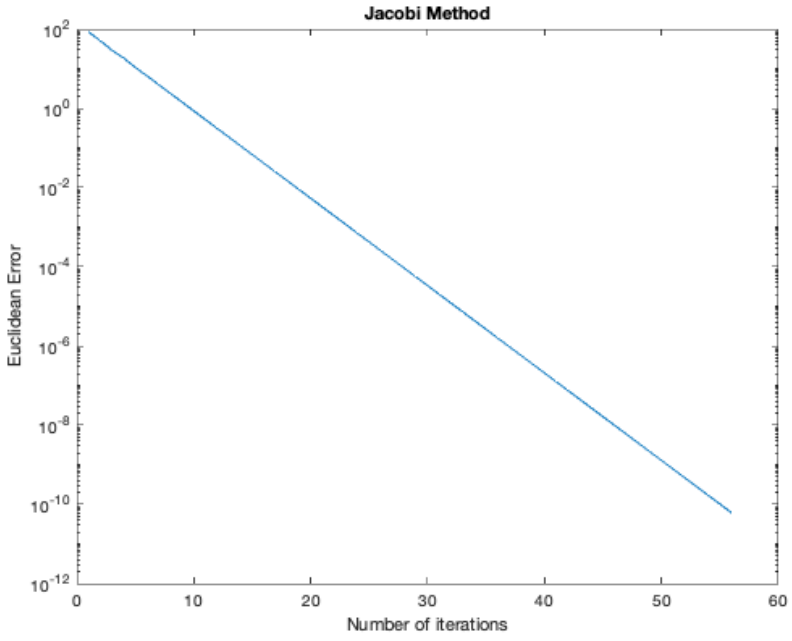
Analysis

Gauss-Seidel



It takes **27** iterations with Euclidean error of **2.176e-11** to solve the given system

Jacobi



It takes **56** iterations with Euclidean error of **6.133e-11**

so looking at both results we can obviously see that it takes Gauss-Seidel less number of iterations to solve the system and also it has lower error so it satisfies both of our measure of effectiveness criteria mentioned above in the theory.

So next thing we will solve the systems from tasks 2a and 2b using Gauss-Seidel for n = 10 and observe the results.

The attempt to solve 2a using Gauss-Seidel succeeded and it took **29** iterations to solve the system
But for 2b we kept reaching our max iteration instead of the tolerance so it failed.

Checking the Spectral radius for task 3 using the following formulas **M= -inv(D+L) * U** for Gauss Seidel and **`M = -inv(D) * (L+U)`** `for` Jacobi. The SRs for the given system of equation in task 3 were **0.0492** for Gauss-Seidel method and **0.1089** for Jacobi.

Next Calculating the SR for Task 2a and 2b using Gauss-Seidel I got the following the results **0.0743** for 2a

And **0.4059** for 2b. Although I stated above that the method failed to solve the matrix 2b our SR was found to be 0.4059 was makes it theoretically possible to solve the matrix but it will need a lot of time for Gauss-Seidel to solve it.

QR Method

Theory

The QR method is an algorithm to find the eigenvalues and eigenvectors of a matrix. The idea behind the algorithm is to perform a QR decomposition which involves decomposing a matrix A into a product A = QR of orthogonal matrix Q and upper triangular matrix R.

Now there are different algorithms to perform the QR decomposition but for this task I will be using the MGS (Modified Gram-Schmidt) and the reason I am choosing the modified version over the classical version is because classical version suffers from numerical instability and roundoff error issues.

The main difference between the modified and classical versions of Gram-Schmidt is in classical the vectors are orthogonalized one after another but in modified a new vector is orthogonalized then all the next vectors immediately orthogonalized with the respect of this first vector.

To compute the eigenvalues for a given matrix there are 2 ways to do it. First way is with no shifts and how the algorithm works is as follows

$$A^{(k)} = Q^{(k)}R^{(k)}$$
$$A^{(k+1)} = R^{(k)}Q^{(k)} \text{ since } Q^{(k)} \text{ is orthogonal then ,}$$
$$R^{(k)} = (Q^{(k)})^{-1}A^{(k)} = Q^{(k)T}A^{(k)}$$
$$A^{(k+1)} = Q^{(k)T}A^{(k)}Q^{(k)}$$

$$A^{(k)} \rightarrow V^{-1}AV = \text{diag}\{\lambda_i\}$$

The problem with this algorithm is that it has slow rate of convergence since the convergence of an off diagonal element to zero is linear with convergence ration $\left|\frac{\lambda_{i+1}}{\lambda_i}\right|$

The solution to the slow convergences of the QR method without shift is the second way which is with shifts based on 2x2 lower right corner submatrix. The algorithm works as follows :

1) The eigenvalue λ_n is found as closest as possible to $d_n^{(k)}$ eigenvalue of the 2x2 submatrix from right lower corner of $A^{(k)}$.
2) The last row and last column of actually matrix $A^{(k)}$ are deleted and then only submatrix $A_{n-1}^{(k)}$ is compared with the full matrix
3) Next eigenvalue λ_{n-1} is found using same process the new submatrix obtained is transformed using QR method until $e_{n-2}^{(k)} = 0$. If no tridiagonal matrix transformation was made, the iterations go on until the elements in last matrix row are zeros except the diagonal element in the lower right corner. Then the process goes on until all the eigenvalues are found.

The convergence ratio with shifts is $\left|\frac{\lambda_{i+1}-p_k}{\lambda_i-p_k}\right|$ p_k should be chosen based on estimate of λ_{i+1}.

I will be using the following symmetric matrix to do my analysis.

3	5	1	4	2
5	2	2	4	3
1	2	0	2	2
4	4	2	2	3
2	3	2	3	4

Analysis

After running my Algorithm in MatLab and comparing my results with the built in function eig() I got the following results.

Initial Matrix =

3	5	1	4	2
5	2	2	4	3
1	2	0	2	2
4	4	2	2	3
2	3	2	3	4

WithNoShift =

14.1579
-2.8405
2.1762
-1.8129
-0.6807

WithShift =

14.1579
2.1762
-0.6807
-1.8129
-2.8405

builtInFunction =

-2.8405
-1.8129
-0.6807
2.1762
14.1579

Number of iterations with no shifts: 81
Number of iterations with shifts: 25

And as we can see we were able to find the eigenvalues faster with shifts and this confirms my assumption in theory.

Appendix

Task 2.

Main.m

```
k = 8;

n = 10*2.^(0:k-1);



%part a
err = zeros(1,k);
failed = k;
for i = 1 : k
    [A, b] = matrix1(n(i));
    x = gaussppelim(A, b);
    if isnan(x)
        display(['Failed for n = ', num2str(n(i))]);
        failed = i - 1;
        break;
end
r = A * x - b;
    err(i) = norm(r,2);
end
%plot the graph
semilogy(n(1:failed),err(1:failed));
title('Error for 2.a)');
xlabel('n');
ylabel('Euclidean Error');

%part b
err = zeros(1,k);
```

```
failed = k;
for i = 1 : k
    [A, b] = matrix2(n(i));
    x = gaussppelim(A, b);
    if isnan(x)
        display(['Failed for n = ', num2str(n(i))]);
        failed = i - 1;
        break;
end
r = A * x - b;
    err(i) = norm(r,2);
end
%plot the graph
figure()
semilogy(n(1:failed),err(1:failed));
title('Error for 2.b)');
xlabel('n');
ylabel('Euclidean Error');

%part a for n = 10
[A, b] = matrix1(10);
x = gaussppelim(A,b);
r = A * x - b;
display(x);
c = norm(in(A)) * norm(A);
display(c);
display(['Euclidean error = ', num2str(norm(r,2))]);
%residual correction for n = 10
x = x - gaussppelim(A,r);
r = A * x - b;
display(x);
display(['Euclidean error = ', num2str(norm(r,2))]);

%part b for n = 10
[A, b] = matrix2(10);
x = gaussppelim(A,b);
r = A * x - b;
display(x);
c = norm(inv(A)) * norm(A);
display(c);
display(['Euclidean error = ', num2str(norm(r,2))]);
%residual correction for n = 10
x = x - gaussppelim(A,r);
r = A * x - b;
display(x);
display(['Euclidean error = ', num2str(norm(r,2))]);
```

Matrix 1 :

```
function [A,b] = matrix1(n)
% function to create matrix for first case
A = zeros(n,n);  %zero matrix is formed
for i = 1 : n    %going thorugh rows from 1 to n
    for j = 1 : n %going through columns from 1 to n
        if i == j
            A(i,j) = 12; %aij = 7
        elseif ((i== j-1) || (i == j+1))
            A(i,j) = 4;
        end
    end
end
%b is created, 1 dim vector is created then transposed
b =11 + 0.6 * (1:n)';
```

Matrix 2 :

```
function [A,b] = matrix2(n)
% function to create matrix for b)
A = zeros(n,n);
for i = 1 : n
    for j = 1 : n
        A(i,j) = 5/(6*(i+j-1)) ;
    end
end
%b is created 1 dim vector is created and then transposed
b = 1./(1.5 * (1:n)' );
b(2:2:end) = 0;
```

Gauss-elimination (Partial Pivoting) :

```
function [b] = gaussppelim(A,b)
% function to solve system of linear equations using gauus elimination
%partial pivoting
n = length (b);
% here we enter the first phase of Gauss Elim Partial Pvioting which we
% transfer it into upper triangle
for i = 1:(n-1)
    [d ,maxim] = max (abs(A(i:n , i)));
    maxim = maxim + i -1;
    %if (d < 1e-15)
        %b = NaN;
        %return;
    %end
    A([i;maxim],:) = A([maxim;i],:);
    b([i;maxim]) = b([maxim;i]);
    for j = (i+1) :n
        d = A(j,i)/A(i,i);
        A(j,:) = A(j,:) - d * A(i,:);
        b(j) = b(j) - d * b(i);
    end
end

for i = n : -1 : 2
    b(i) = b(i) / A(i,i);
    j = 1 : (i-1);
    b(j) = b(j) - A(j,i) * b(i);
end
```

```
b(1) = b(1)/A(1,1);
```

Task 3

Main.m

```
%matrix A, coeffients of x variables in system of linear equations
A = [6, 2, 1, -1;
    4, -12, 2, -1;
    2, -1, 5, -1;
    5, -2, 1, 8];
%solution vector b
b = [6;
    8;
    10;
    2];
%zero vector x0
x0 = zeros(4,1);
%assumed accuracy
asacry = 1e-10;

%using gauss-seidel method
D = diag(diag(A));
U = A-tril(A);
L = inv(tril(A));
M = -inv(D+L) * U;
SR = max(abs(eig(M)));
display(SR);

[x, E, iter] = gseidel(A, b, x0, asacry);
%print number of iterations from gauss-seidel method
display(['Gauss-Seidel iterations = ', num2str(iter)]);
%plot on semilogarithmic graph
semilogy(1:iter,E);
title('Gauss-Seidel Method');
xlabel('Number of iterations');
ylabel('Euclidean Error');


%using jacobi method
D = diag(diag(A));
U = A-tril(A);
L = inv(tril(A));
M = -inv(D) * (L+U);
SR = max(abs(eig(M)));
display(SR);

[x, E, iter] = jacobi(A, b, x0, asacry);
%print number of iterations from jacobi method
display(['Jacobi iterations = ', num2str(iter)]);
%plot on semilogarithmic graph
figure()
semilogy(1:iter,E);
title('Jacobi Method');
xlabel('Number of iterations');
ylabel('Euclidean Error');

%solving equations from previous task using gauss-seidel
x0 = zeros(10,1);

% %part A
[A,b] = matrix1(10);
D = diag(diag(matrix1(10)));
```

```
U = matrix1(10)-tril(matrix1(10));
L = inv(tril(matrix1(10)));
M = -inv(D+L) * U;
SR = max(abs(eig(M)));
display(SR);

[A,b] = matrix1(10);
[x, E, iter] = gseidel(A, b, x0, asacry);
display(x);
display(iter);

%part B
[A,b] = matrix2(10);
D = diag(diag(matrix2(10)));
U = matrix2(10)-tril(matrix2(10));
L = inv(tril(matrix2(10)));
M = -inv(D+L) * U;
SR = max(abs(eig(M)));
display(SR);

[A,b] = matrix2(10);
[x, E, iter] = gseidel(A, b, x0, asacry);
display(x);
display(iter);
```

Gauss-Seidel :

```
function [x, Eerr, iter] = gseidel(A, b, x, asacry)
%function for gauss-seidel method accepts inputs of A, b and zero vector x
%and assumed accuracy and outputs unknown x, Euclidean error and number of
%iterations.
U = A - tril(A);        %U is upper triangular matrix
L = inv(tril(A));
mi = 100;
Eerr = zeros(1,mi);
for iter = 1 : mi
    x = L * (b - U * x);
    r = A * x - b;
    err = norm(r,2);
    Eerr(iter) = err;
    if (err < asacry)
        Eerr = Eerr(1:iter);
        return;
    end
end
```

Jacobi :

```
function [x, Eerr, iter] = jacobi(A, b, x, asacry)
%function for jacobi method accepts inputs of A, b and zero vector x and
%assumed accuracy and outputs unknown x, Euclidean error and number of
%iterations.
D = A .* eye(length(b));
%dot product of A with identity vector creates diagonal matrix of A
R = A - D;              %rest of the matrix
D = inv(D);
mi = 100;          %maximum number of iterations
Eerr = zeros(1,mi);
for iter = 1 : mi
    x = D * (b - R * x);
    r = A * x - b;
    err = norm(r, 2);
    Eerr(iter) = err;
    if (err < asacry)
        Eerr = Eerr(1:iter);
        return;
    end
end
```

Task 4

Main.m

```
% the matrix i have chose
A=[3 5 1 4 2; 5 2 2 4 3; 1 2 0 2 2; 4 4 2 2 3; 2 3 2 3 4];
display(A,'Intial Matrix')
% qr-eigenvalue algrorithm with no shifts
[it1,eig1] = QRNoShift(A,1e-6,100);
% qr-eigenvalue algrorithm with shifts
[it2,eig2] = QRshift(A,1e-6,100);
% standard qr-eigenvalue algorithm
eig3 = eig(A);
display(eig1,'WithNoShift')
display(eig2,'WithShift')
display(eig3,'builtInFunction')
disp("Number of iterations with no shifts: " + it1)
disp("Number of iterations with shifts: " +it2)
```

Quadpoly :

```
function [x1,x2] = quadpoly(A)
b = - A(1,1)-A(2,2);
c = A(1,1)*A(2,2)-A(1,2)*A(2,1);
D = b^2 - 4*c;
x1 = (-b+sqrt(D))/2;
x2 = (-b-sqrt(d))/2;
```

Gram Schmidt modified :

```
function [Q,R]=gram_mod(A)
    % get the dimensions of matrix A
    [m,n] = size(A);
    % generate output matrices Q and R
    Q = zeros(m,n);
    R = zeros(n,n);
    % generate
    d = zeros(1,n);
    for i=1:n
       Q(:,i)=A(:,i);
       R(i,i)=1;
       d(i)=Q(:,i)'*Q(:,i);
       for j=i+1:n
           R(i,j)=(Q(:,i)'*A(:,j))/d(i);
           A(:,j)=A(:,j)-R(i,j)*Q(:,i);
       end
    end

    for i=1:n
        dd=norm(Q(:,i));
        Q(:,i)=Q(:,i)/dd;
        R(i,i:n)=R(i,i:n)*dd;
    end
```

QRNoShift :

```
function [i,eigenvalues]=QRNoShift(D,tol,imax)
%tol - tolerance (upper bound) for nulled elements
%imax - max no. of iterations

i=0;
while i <= imax && max(max(D-diag(diag(D)))) > tol
    [Q1,R1] = gram_mod(D);
    D=R1*Q1;
    i=i+1;
end
if i > imax
    error('imax exceeded program terminated');
end
eigenvalues = diag(D);
```

QRshift:

```
function [i,eigenvalues] = QRshift(A,tol,imax)
%tol tolerance
%imax max iteration for calculate eigenvalue
n = size(A,1);
eigenvalues = diag(ones(n));
INITIALsubmatrix = A; %intial matrix
for k = n:-1:2
    DK = INITIALsubmatrix; %initial matrix to calculate a single eigenvalue
    i = 0;
    while i<imax && max(abs(DK(k,1:k-1)))>tol
        DD=DK(k-1:k,k-1:k);
        [ev1,ev2]=quadpoly(A);
        if abs(ev1-DD(2,2)) < abs(ev2-DD(2,2))
            shift=ev1;
        else
            shift=ev2;
        end
        DP=DK-eye(k)*shift;
        [Q1,R1]=gram_mod(DP);
        DK=R1*Q1+eye(k)*shift;
        i=i+1;
    end
    if i > imax
        error('imax exceeded program terminated');
    end
    eigenvalues(k)=DK(k,k);
    if k > 2
        INITIALsubmatrix=DK(1:k-1,1:k-1);
    else
        eigenvalues(1)=DK(1,1);
    end
end
```