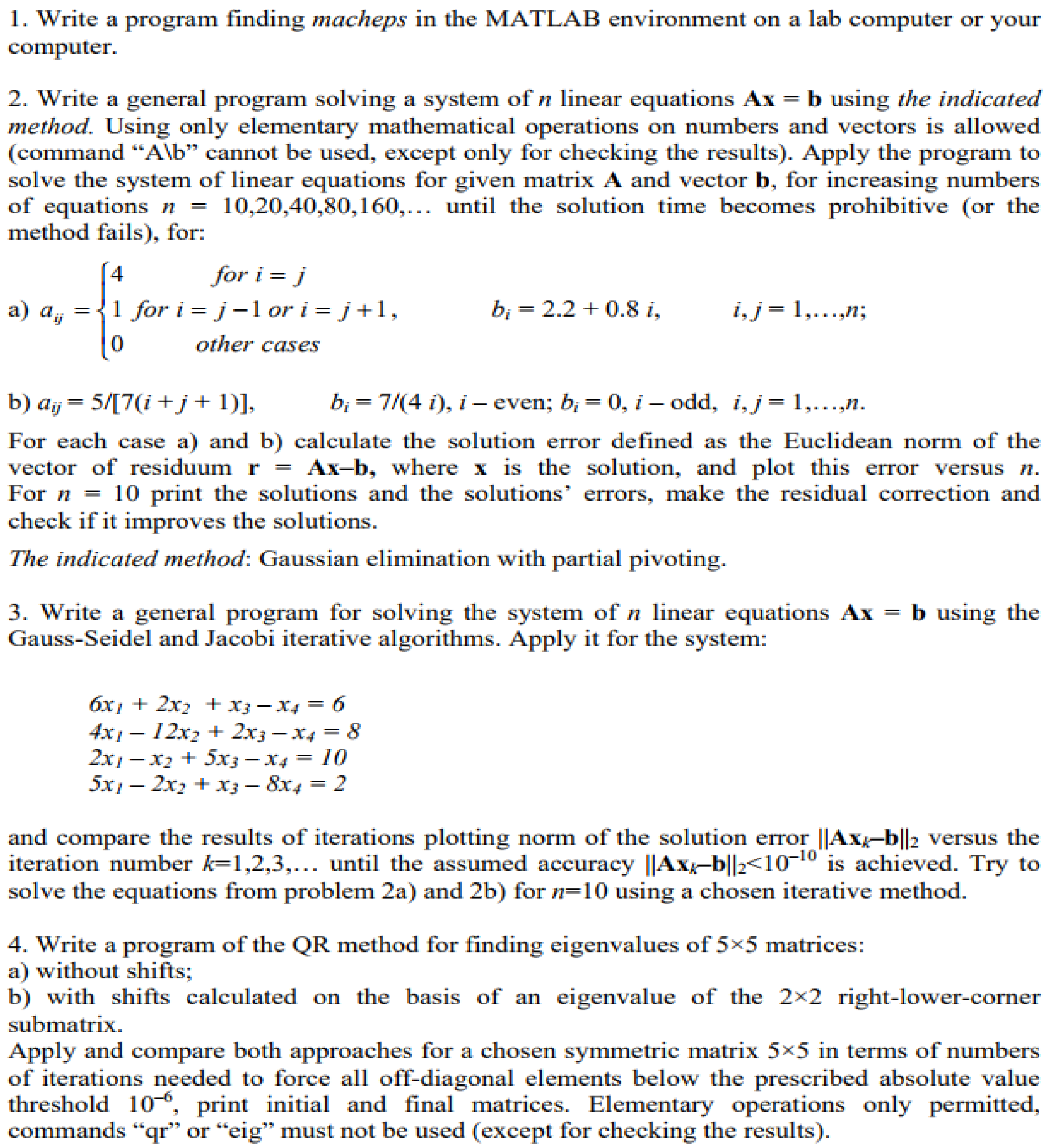
ENUME Project A

NO. 27

KORKMAZ BARAN 302809

Report Description



1. Write a program finding *macheps* in the MATLAB environment on a lab computer or your computer.

2. Write a general program solving a system of n linear equations $\mathbf{Ax} = \mathbf{b}$ using *the indicated method*. Using only elementary mathematical operations on numbers and vectors is allowed (command "A\b" cannot be used, except only for checking the results). Apply the program to solve the system of linear equations for given matrix **A** and vector **b**, for increasing numbers of equations n = 10,20,40,80,160,... until the solution time becomes prohibitive (or the method fails), for:

a) $a_{ij} = \begin{cases} 4 & \text{for } i = j \\ 1 & \text{for } i = j-1 \text{ or } i = j+1, \\ 0 & \text{other cases} \end{cases}$ $\quad b_i = 2.2 + 0.8\,i, \quad i, j = 1,\ldots,n;$

b) $a_{ij} = 5/[7(i + j + 1)]$, $\quad b_i = 7/(4\,i)$, i – even; $b_i = 0$, i – odd, $\; i, j = 1,\ldots,n$.

For each case a) and b) calculate the solution error defined as the Euclidean norm of the vector of residuum $\mathbf{r} = \mathbf{Ax}–\mathbf{b}$, where $\mathbf{x}$ is the solution, and plot this error versus n. For n = 10 print the solutions and the solutions' errors, make the residual correction and check if it improves the solutions.

The indicated method: Gaussian elimination with partial pivoting.

3. Write a general program for solving the system of n linear equations $\mathbf{Ax} = \mathbf{b}$ using the Gauss-Seidel and Jacobi iterative algorithms. Apply it for the system:

$$\begin{aligned} &6x_1 + 2x_2 + x_3 - x_4 = 6 \\ &4x_1 - 12x_2 + 2x_3 - x_4 = 8 \\ &2x_1 - x_2 + 5x_3 - x_4 = 10 \\ &5x_1 - 2x_2 + x_3 - 8x_4 = 2 \end{aligned}$$

and compare the results of iterations plotting norm of the solution error $||\mathbf{Ax}_k–\mathbf{b}||_2$ versus the iteration number k=1,2,3,... until the assumed accuracy $||\mathbf{Ax}_k–\mathbf{b}||_2<10^{-10}$ is achieved. Try to solve the equations from problem 2a) and 2b) for n=10 using a chosen iterative method.

4. Write a program of the QR method for finding eigenvalues of 5×5 matrices:
a) without shifts;
b) with shifts calculated on the basis of an eigenvalue of the 2×2 right-lower-corner submatrix.
Apply and compare both approaches for a chosen symmetric matrix 5×5 in terms of numbers of iterations needed to force all off-diagonal elements below the prescribed absolute value threshold 10^{-6}, print initial and final matrices. Elementary operations only permitted, commands "qr" or "eig" must not be used (except for checking the results).

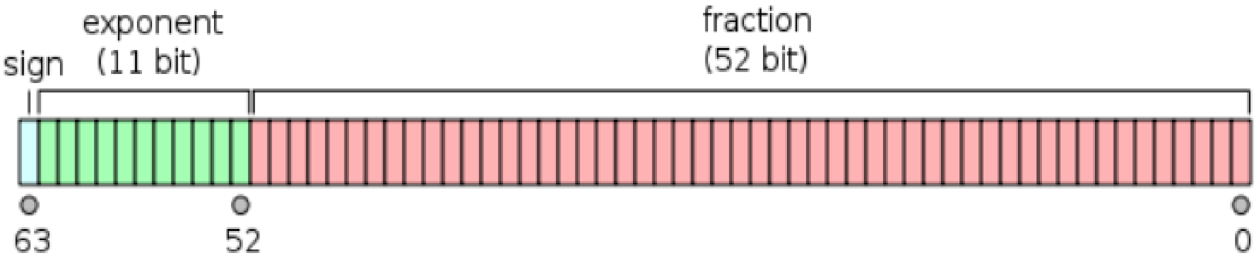
Table of Contents

Task 1

In this task, we are going to calculate the machine epsilon with a program and analyze how this value has obtained.

Theory

On most modern CPUs, the native data type for doing real valued arithmetic is the **IEEE 754 double precision floating point data type** where any value stored as a double requires **64 bits**. Double-precision floating-point numbers are stored in a **64-bit** word, with **52 bits** for mantissa, **11 bits** for exponent, and **1 bit** for the sign of the number.



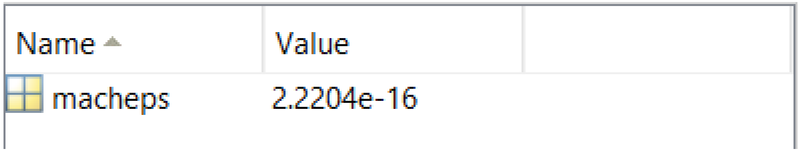
For our purposes, the most relevant fact about this structure is that it only allows us to represent a maximum of **16** decimal digits in a number. In order to fit in a number with over **16** digits, CPU round-off the digits beyond **16** which causes round-off errors in calculations.

Machine epsilon can be defined as the distance between 1 and the next largest floating-point number. In other words, machine epsilon calculates the maximum resolution difference for an increase of **1 bit** in binary between two numbers.

For **IEEE 754** double precision**,** machine epsilon is defined as $\mathbf{2^{-52}} \approx \mathbf{2.22e-16}$

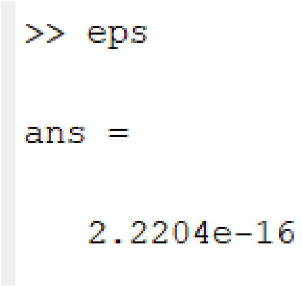
Results

After running the code on MATLAB, we get the value 2.2204e-16 as our machine epsilon.



Name ▲	Value
macheps	2.2204e-16

To verify of my results, I checked the value of machine epsilon by using built-in function **eps**.

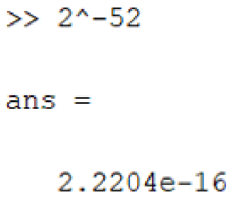


```
>> eps

ans =

   2.2204e-16
```

Analysis

From both **IEEE 744** and built-in **eps** function on MATLAB, we can see that our code obtained the correct machine epsilon value.

If we analyze the code for calculating machine epsilon, the division by **2** is **bit-shifting** the number to the right by **1** position at each iteration, starting at the whole value of $\mathbf{2^0}$and taking this number and add this to **1** until loop stops at when the number becomes **0** due to underflow. Loop stops at the least significant bit $(\mathbf{2^{-52}})$ which is the smallest value between 1 and the next floating number in **IEEE 744 double precision**.



```
>> 2^-52

ans =

   2.2204e-16
```

Task 2

In this task, we will be solving two different systems of linear equation (let them be **system a** and **system b** for convenience) with **Gaussian Elimination with Partial Pivoting Method**. In the second part, a residual correction will be applied to both methods at **n = 10** and check whether the outcome is improved.

Theory

Gaussian elimination is an algorithm for solving a system of linear equations. Like other finite methods, Gaussian elimination obtains a solution after a finite number of elementary arithmetical operations precisely defined by the method itself and the dimension of the problem has been done.

Gaussian elimination algorithm consists of the following steps repeated many times:

1. Construct a multiplier $\boldsymbol{m_{j,i} = a_{j,i}/a_{i,i}}$
2. Replace row $\boldsymbol{A_j}$ with $\boldsymbol{A_j - m_{j,i}\, A_i}$

We can see that in **step 2**, when doing the subtraction, we have to ensure that the numbers being subtracted do not have greatly differing magnitudes. That can happen if the multiplier gets too big or small relative the numbers in the two rows. The pivot value being a number too close to **0** produces a multiplier too large, which in turn can introduce a heightened danger of **round-off error**. To avoid using pivots that are too small, we can take advantage of the fact that interchanging any two rows in an augmented matrix does not change the result. This leads to the following strategy, called **partial pivoting:**

1. Before using $\boldsymbol{a_{i,i}}$ as a pivot, find the smallest **p** such that $\boldsymbol{a_{p,i} = max_{i\le k\le n}[a_{k,i}]}$
2. Swap row **p** with row **i**.

This guarantees that each pivot we use is as large as possible to minimize the potential for round-off error.

Residual correction is an iterative method than can be used to improving the accuracy of a solution produced using Gaussian Elimination. First, the residuum **r** is calculated with the formula,

$$\boldsymbol{r}^{(1)} = \boldsymbol{A}\boldsymbol{x}^{(1)} - \boldsymbol{b}$$

Using factorization, the set $\boldsymbol{A}\boldsymbol{\delta x} = \boldsymbol{r}^{(1)}$ is solved to obtain $\boldsymbol{x}^{(1)}$. Then, using this $\boldsymbol{x}^{(1)}$ in the formula**,**

$$\boldsymbol{x}^{(2)} = \boldsymbol{x}^{(1)} - \boldsymbol{\delta x}$$

We can calculate $\boldsymbol{x}^{(2)}$ **.**

Finally, the residuum $\boldsymbol{r}^{(2)}$ is calculated with the formula,

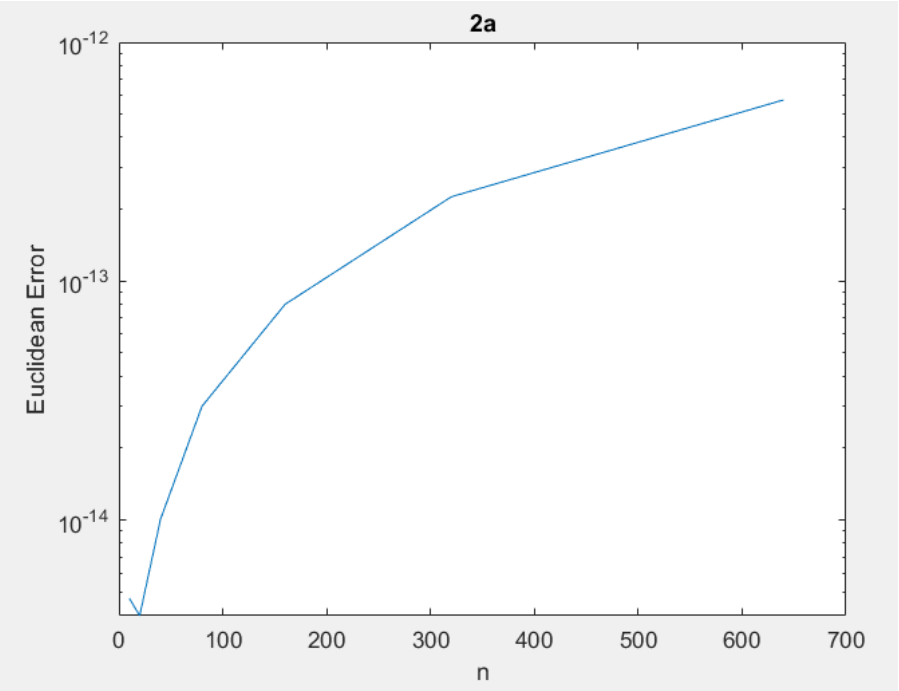
$$\boldsymbol{r}^{(2)} = \boldsymbol{A}\boldsymbol{x}^{(1)} - \boldsymbol{b}$$

If it is smaller than $\boldsymbol{r}^{(1)}$ and still too large, the procedure is repeated.

Results

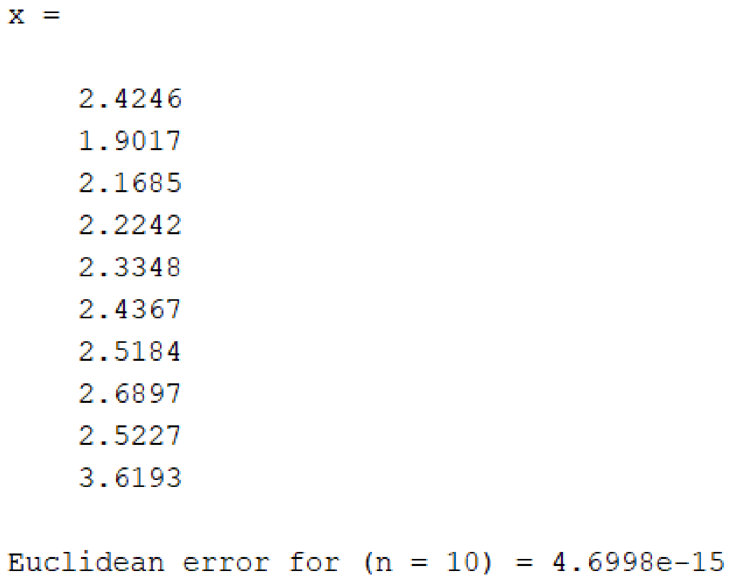
System a

Using the Gauss elimination with partial pivoting algorithm, **system a** solved and Euclidean Error plotted against number of equations **n.**



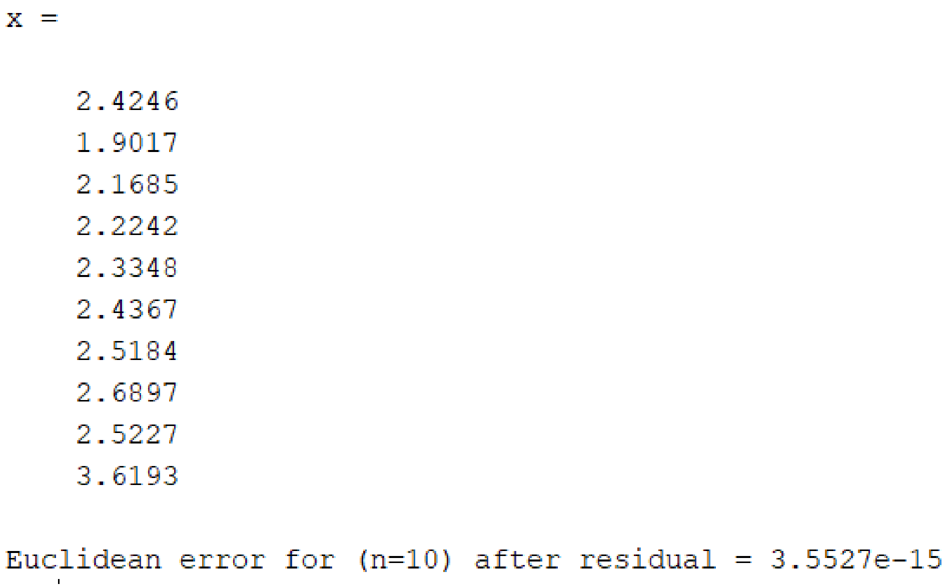
(graph 1)

Next, solution **x** and the Euclidean error printed for **n = 10**.



```
x =

    2.4246
    1.9017
    2.1685
    2.2242
    2.3348
    2.4367
    2.5184
    2.6897
    2.5227
    3.6193

Euclidean error for (n = 10) = 4.6998e-15
```

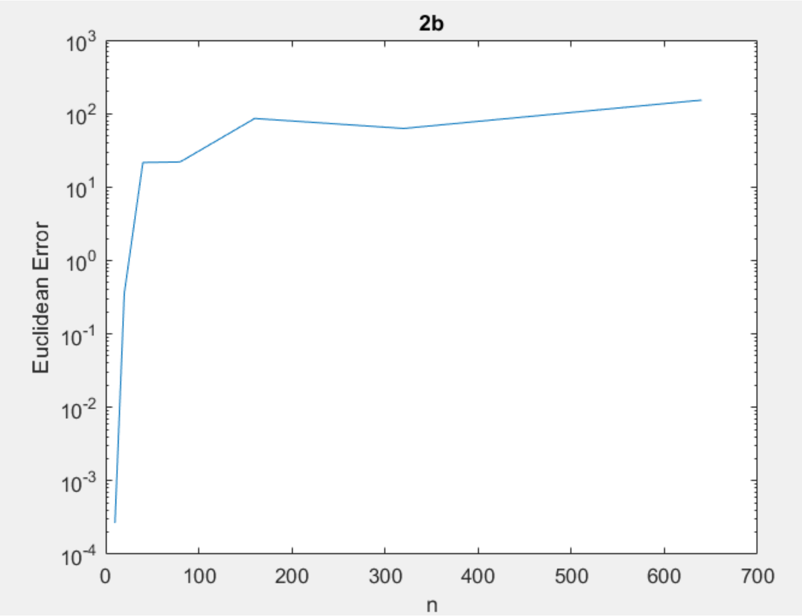
And finally, residual correction done for **n = 10.**



```
x =

    2.4246
    1.9017
    2.1685
    2.2242
    2.3348
    2.4367
    2.5184
    2.6897
    2.5227
    3.6193

Euclidean error for (n=10) after residual = 3.5527e-15
```

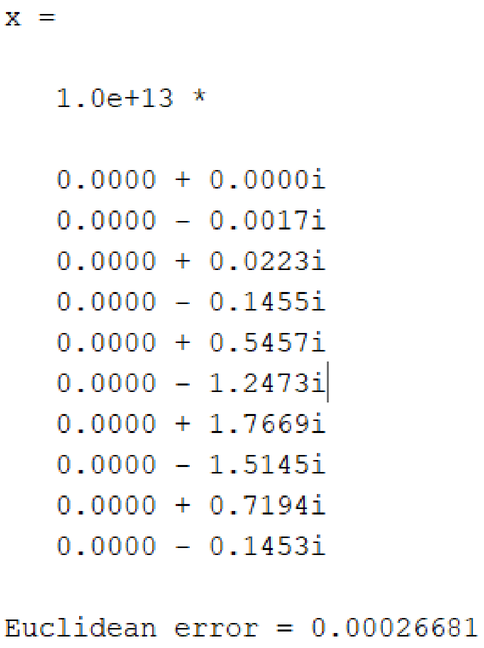
System b

Euclidean Error plotted against number of equations **n** for **system b.**



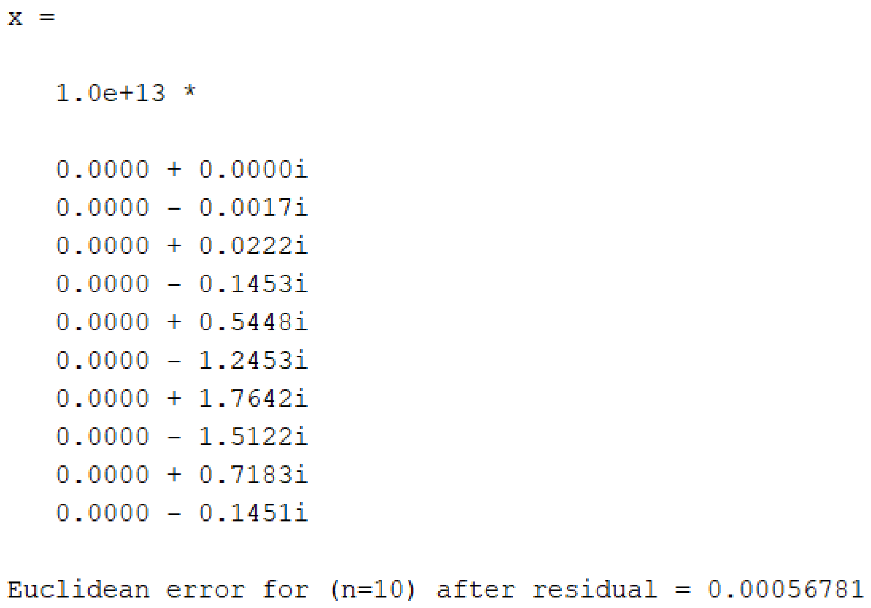
(graph 2)

Next, solution **x** and the **Euclidean error** printed for **n = 10**.



```
x =

   1.0e+13 *

   0.0000 + 0.0000i
   0.0000 - 0.0017i
   0.0000 + 0.0223i
   0.0000 - 0.1455i
   0.0000 + 0.5457i
   0.0000 - 1.2473i
   0.0000 + 1.7669i
   0.0000 - 1.5145i
   0.0000 + 0.7194i
   0.0000 - 0.1453i

Euclidean error = 0.00026681
```

And finally, residual correction done for **n = 10.**



```
x =

   1.0e+13 *

   0.0000 + 0.0000i
   0.0000 - 0.0017i
   0.0000 + 0.0222i
   0.0000 - 0.1453i
   0.0000 + 0.5448i
   0.0000 - 1.2453i
   0.0000 + 1.7642i
   0.0000 - 1.5122i
   0.0000 + 0.7183i
   0.0000 - 0.1451i

Euclidean error for (n=10) after residual = 0.00056781
```

Analysis

System a

From the **graph 1**, we can see that as number of equations increases, Euclidean error also increases. This is the result accumulation of **round-off error** that explained in the theory part.

Euclidean error when we run the program for **n=10** was **4.6998e-15**. After residual correction applied, error slightly decreased to **3.5527e-15**. It shows that residual correction improved the accuracy in this case.

System b

Gauss-Elimination method with partial pivoting breaks after a certain **n** for **system b**. For further analysis **condition number** of the equation is checked.

Condition number of the **system b** is **2.4261e+14.**

To compare the equations, condition number of the **system a** calculated at **2.8443** which is an astronomically smaller number than second equation's condition number**.**

Unfortunately, neither complete pivoting nor partial pivoting solves all problems of rounding error. Some systems of linear equations, called **ill-conditioned systems**, pivoting is not much help. A common type of system of linear equations that tends to be **ill-conditioned** is one for which **condition number** is much larger than **1** like **system b**. Practically, such a matrix is almost singular, and the computation of its inverse, or solution of a linear system of equations is prone to large numerical errors.

We can see that when **system b** solved with Gauss-Seidel for **n=10,** Euclidean error is quite large which supports my ill-conditioned system argument. Residual correction also does not work in this case.

Task 3

In this task, we will be solving the given linear equation system as well as the two systems of linear equations from **Task 2** with two different **iterative(indirect) methods** and analyze the outcome. Methods used will be compared and analyzed regarding to the results.

Theory

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 **nth** 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** $\boldsymbol{A}$. Initial point is usually generated according to the following formula:

$\boldsymbol{x}\ \mathbf{(}\boldsymbol{i}\mathbf{+1)} = \boldsymbol{Mx}\ \mathbf{(}\boldsymbol{i}\mathbf{)} + \boldsymbol{w}$ where **A** is a specific matrix and **w** is a vector.

For iterative methods to work, these two conditions must satisfy where **M** is the matrix and **w** is the vector.

1. the convergence condition: $\boldsymbol{sr}(\boldsymbol{M}) < \mathbf{1}$ where $\boldsymbol{sr}(\boldsymbol{M})$ is spectral radius.
2. the coincidence condition: $\boldsymbol{\hat{x}} = \boldsymbol{M\hat{x}} + \boldsymbol{w}$

The **Jacobi Method** is as an iterative algorithm which is used for determining the solutions for the system of linear equations. In this method, an approximate value is filled in for each diagonal element and iterated until it converges. Jacobi method is a parallel computational scheme since all calculations can be done independently. In matrix terms, the definition of the Jacobi method can be expressed as

$$\boldsymbol{x^{(i+1)} = -D^{-1}(L+U)x^{(i)} + bD^{-1}, \ \ i = 0, 1, 2 \ldots}$$

where **L** is the sub-diagonal matrix, **D** the diagonal matrix and **U** the matrix with entries over the diagonal.

Or it can be written in this form which I will use in my code,

$$\boldsymbol{D^{-1}\left(b-(L+U)x^{(k)}\right) = Tx^{(k)} + C}$$

The **Gauss-Seidel Method** is similar to **Jacobi method** but with a modification. In the Gauss–Seidel method, instead of always using previous iteration values for all terms of the whenever an updated value becomes available, it is immediately used. This type of methods called sequential as calculations must be done in order. Assuming again that **D** is nonsingular, the following iterative method can be now proposed

$$\boldsymbol{(D+L)x^{(i+1)} = Ux^{(i)} + b\,, \qquad i = 0, 1, 2 \ldots}$$

where **L** is the sub-diagonal matrix, **D** the diagonal matrix and **U** the matrix with entries over the diagonal.

Another form of **Gauss-Seidel method** is,

$$\boldsymbol{Lx^{(i+1)} = b - Ux^{(i)}}$$

which I will be using in my code.

Since an iterative method computes successive approximations to the solution of a linear system, a **stop test** is needed to determine when to stop the iteration. These tests are as follows,

1) Checking difference between two subsequent iteration points $\boldsymbol{[\![x^{(i+1)} - x^{(i)}]\!] \leq \ \delta}$ where $\boldsymbol{\delta}$ is our assumed tolerance.

2) Checking a norm of the solution error vector $\llbracket \boldsymbol{Ax^{(i+1)}} - \boldsymbol{b} \rrbracket \leq \boldsymbol{\delta_2}$ if the test is not fulfilled within the assumed accuracy then the value of δ is diminished and we continue our iterations

As effectiveness goes, there are two different criteria for iterative methods:

1) The number of elementary arithmetic operations needed to perform a single iteration

$$\boldsymbol{x^{(i)}} > \boldsymbol{x^{(i+1)}}$$

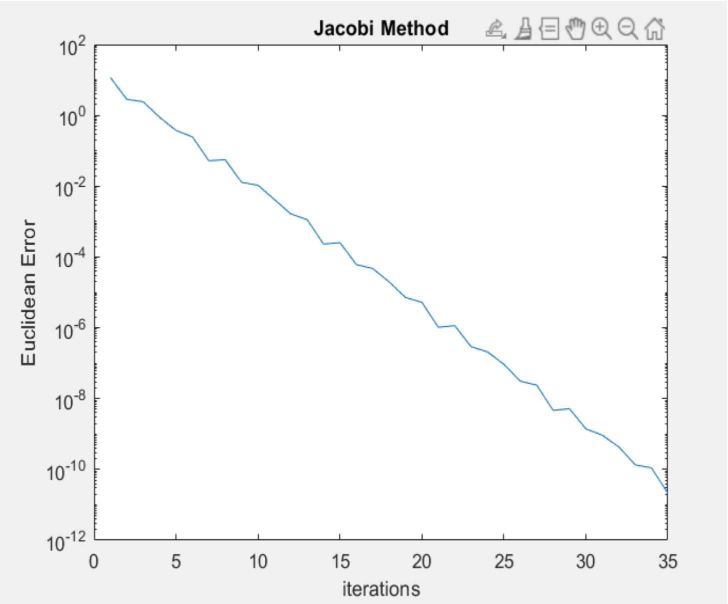
2) The speed of convergence, a measure indicating how fast the error decreases

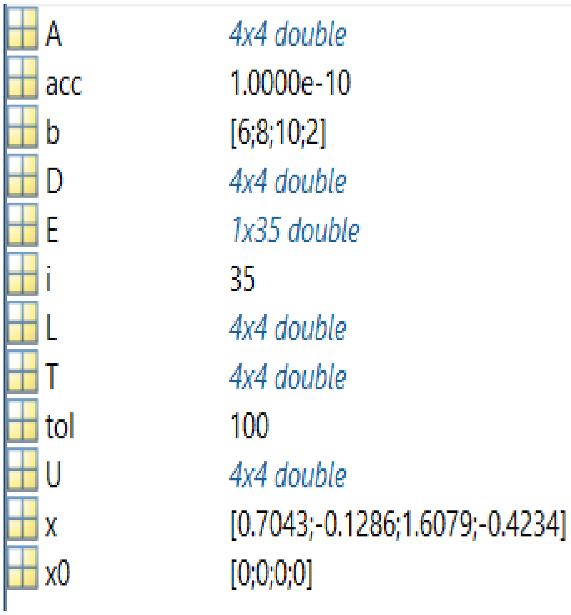
$$\boldsymbol{e^{i}} = \boldsymbol{x^{(i)}} - \boldsymbol{\hat{x}}$$

These methods do not always work. However, for the class of **strictly diagonally dominant** matrices, we can prove that they do work. A matrix is **diagonally dominant** if the absolute value of each diagonal element is at least as large as the sum of the absolute values of the remaining entries in the same row. A matrix is **strictly diagonally dominant** if the absolute value of each diagonal element is strictly greater than the sum of the absolute values of the remaining entries in the same row.

Results

First, given linear equation system solved by Jacobi and Gauss-Seidel methods.

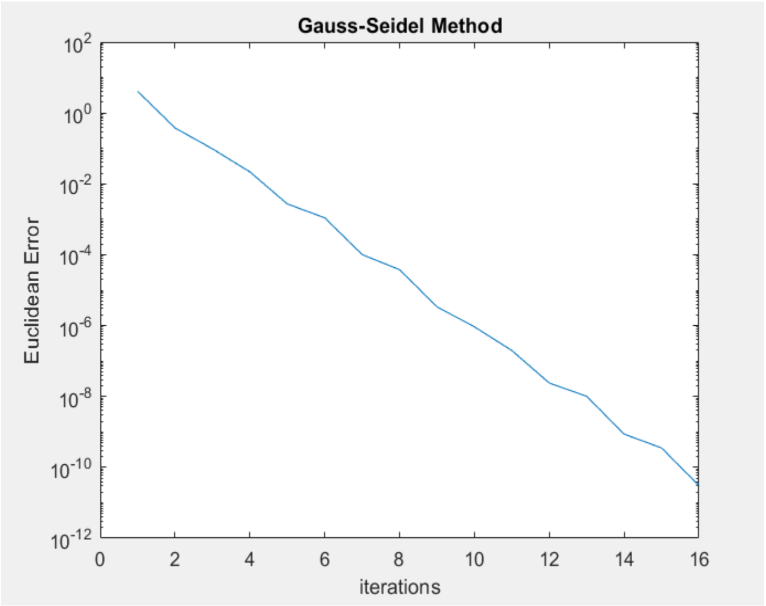


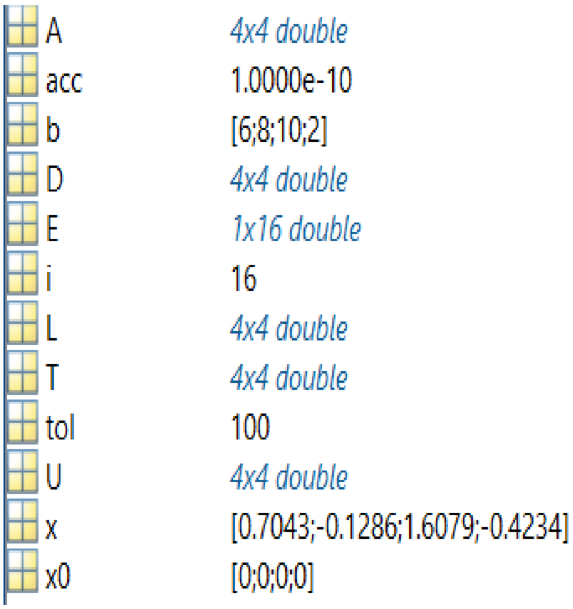


A	4x4 double
acc	1.0000e-10
b	[6;8;10;2]
D	4x4 double
E	1x35 double
i	35
L	4x4 double
T	4x4 double
tol	100
U	4x4 double
x	[0.7043;-0.1286;1.6079;-0.4234]
x0	[0;0;0;0]

(graph 3)

It takes **35** iterations to solve the system with **Jacobi method**





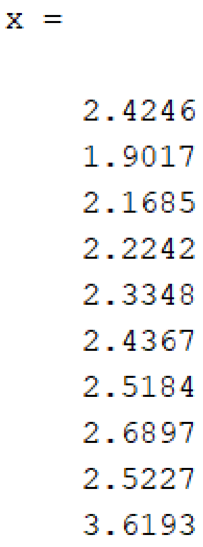
```
A      4x4 double
acc    1.0000e-10
b      [6;8;10;2]
D      4x4 double
E      1x16 double
i      16
L      4x4 double
T      4x4 double
tol    100
U      4x4 double
x      [0.7043;-0.1286;1.6079;-0.4234]
x0     [0;0;0;0]
```

(graph 4)

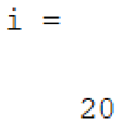
It takes **16** iterations to solve the given system with **Gauss-Seidel method.**

In the second part of the task, systems **a** and **b** from **Task 2** are solved by **Gauss-Seidel method**. Which resulted as:

From system a



```
x =

    2.4246
    1.9017
    2.1685
    2.2242
    2.3348
    2.4367
    2.5184
    2.6897
    2.5227
    3.6193
```

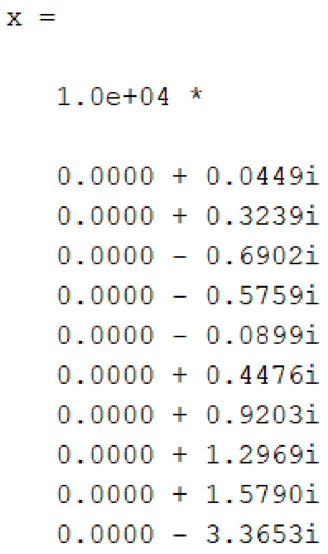


```
i =

    20
```

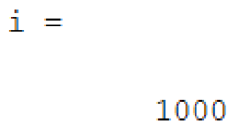
We can see that same result as in **Task 2** has obtained with **20** iterations.

For system b

Algorith for Gauss-Seidel method failed for **system b** because iterations kept reaching the maximum limit even when limit is increased from 100 to 1000.



```
x =

   1.0e+04 *

   0.0000 + 0.0449i
   0.0000 + 0.3239i
   0.0000 - 0.6902i
   0.0000 - 0.5759i
   0.0000 - 0.0899i
   0.0000 + 0.4476i
   0.0000 + 0.9203i
   0.0000 + 1.2969i
   0.0000 + 1.5790i
   0.0000 - 3.3653i
```



```
i =

        1000
```

Analysis

We can see that **Gauss-Seidel method** solved the system more accurately and with less iterations than **Jacobi Method**. Therefore, it can be said that it is the superior method which is why I choose to use it in the second part of the task.

In the second part, **system a** from **Task 2** solved successfully with **Gauss-Seidel method**. While elapsed time for **Gauss-Seidel** for **n=10** is only **0.005392** seconds, same time for **Gauss Elimination** for **n = 10** was **0.444152** seconds. So it safe to say Gauss-Seidel was the faster method in this case.

And for the **system b,** it was already established at the **Task 2** that this linear equation system is an **ill-conditioned** system with a very high **condition number**. Furthermore, since our iteration reaching the limit before solving the system, slow **convergence rate** is suspected. Since, from theory, we know that smaller the **spectral radius,** bigger the convergence rate will become, I decided to calculate spectral radius of **system a and system b**.

The calculations for **spectral radius** values showed that the **system a** spectral radius value is **0.0505** and the **system b** the value is **0.2005**. While the number for second equation is relatively large, it still satisfies the condition of convergence ($\boldsymbol{sr}(\boldsymbol{M}) < \mathbf{1}$)**.** Therefore, we might that **system b** can be solved by **Gauss-Seidel** but it will require huge amount of iterations.

Knowing **diagonal dominance** effects convergence for Jacobi and Gauss-Seidel methods, I checked if the **equation b** is diagonally dominant. The algorithm showed that **system b** is indeed not diagonally dominant.

Task 4

In this task, we will determine the **eigenvalues** of a chosen matrix by **QR method** and **QR method with shifts**.

Theory

QR method is a procedure to calculate the eigenvalues and eigenvectors of a matrix. Algorithm performs a **QR** decomposition which involves decomposing a matrix A into a product **A = QR** of **orthogonal matrix** Q and **upper triangular matrix R**.

Consider for the moment a **QR-factorization** of the matrix A,

$$\boldsymbol{A} = \boldsymbol{QR}$$

where **Q * Q = 1** and **R** is **upper triangular**. We will now reverse the order of multiplication product of **Q** and **R** and eliminate **R**,

$$\boldsymbol{RQ} = \boldsymbol{Q}^{*}\boldsymbol{AQ}$$

Since $\boldsymbol{Q}^*$**AQ** is a similarity transformation of **A**, **RQ** has the same eigenvalues as **A**. More importantly, by repeating this process, the matrix **RQ** will become closer and closer to upper triangular, such that we eventually can read off the eigenvalues from the diagonal.

QR-method generates a sequence of matrices $\boldsymbol{A_k}$ initiated with $\boldsymbol{A_0 = A}$ and given by

$$\boldsymbol{A_k = Q_k R_k}$$

where $\boldsymbol{Q_k}$ and $\boldsymbol{R_k}$ represents a **QR-factorization** of $\boldsymbol{A_{k-1}}$,

$$\boldsymbol{A_{k-1} = Q_k R_k}$$

In order to achieve an efficient algorithm is to improve the **convergence speed** of the QR method. Convergence can be dramatically improved by considering a QR-step applied to the matrix formed by subtracting a multiply of the identity matrix. This type of acceleration is called **shifting**.

Algorithm for **QR method with shift** works as follows,

1. The eigenvalue $\boldsymbol{\lambda_n}$ is found, as a closer to $\boldsymbol{d_n^{(k)}}$ eigenvalue of the 2×2 submatrix from the right lower corner of $\boldsymbol{A^{(k)}}$,

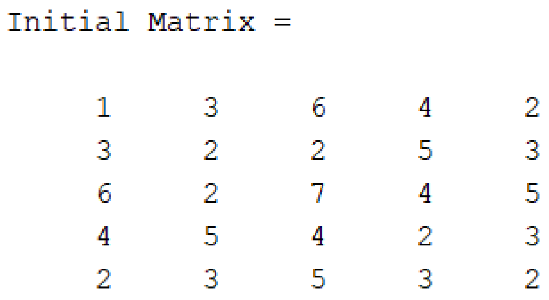
2. The last row and the last column of the actual matrix $\boldsymbol{A^{(k)}}$ are deleted and then only $\boldsymbol{A_{n-1}^{(k)}}$ is compared with the full matrix.

3 - Next eigenvalue $\boldsymbol{\lambda_{n-1}}$is found using the same procedure – i.e., the matrix $\boldsymbol{A_{n-1}^{(k)}}$ A (k) n−1 is transformed using the QR procedure until $\boldsymbol{e_{n-2}^{(k)} = 0}$. In the case of full (not tridiagonal) matrix transformations, the iterations are performed until all the elements of the last matrix row, except the diagonal one $\boldsymbol{d_{n-1}^{(k)}}$ is zero.

Iteration will continue until all eigenvalues are found.

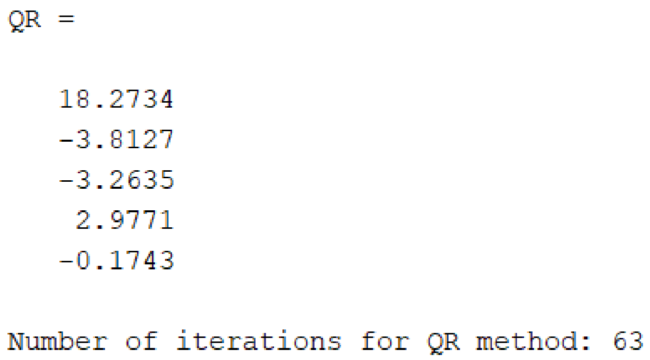
Even though, the method is also applicable and very effective for nonsymmetric matrices, the **QR method** is always convergent and very effective (when using the algorithm with shifts) for **symmetric matrices**. With this information in mind, I have decided to use a symmetric matrix in this task.

Results



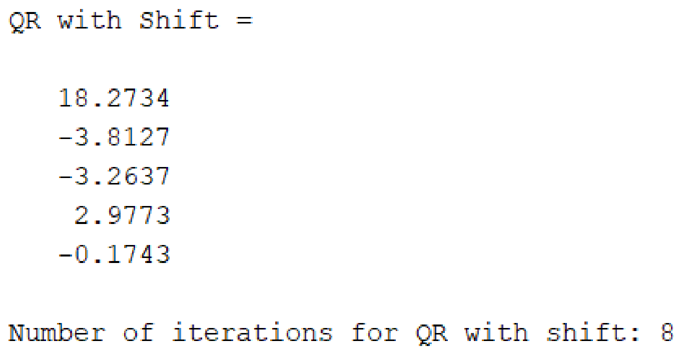
```
Initial Matrix =

     1     3     6     4     2
     3     2     2     5     3
     6     2     7     4     5
     4     5     4     2     3
     2     3     5     3     2
```

Eigenvalues obtained by **QR** algorithm with **63** iterations.



```
QR =

   18.2734
   -3.8127
   -3.2635
    2.9771
   -0.1743

Number of iterations for QR method: 63
```

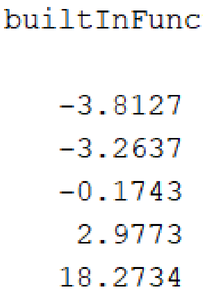
Eigenvalues obtained by **QR with shift** algorithm with **8** iterations



```
QR with Shift =

   18.2734
   -3.8127
   -3.2637
    2.9773
   -0.1743

Number of iterations for QR with shift: 8
```

Analysis

Results are proven to be correct after checking them with built-in function **eig().**



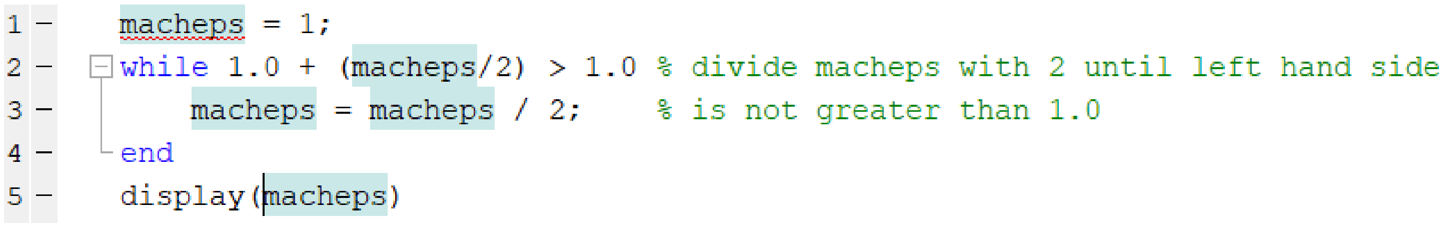
```
builtInFunction =

   -3.8127
   -3.2637
   -0.1743
    2.9773
   18.2734
```

While the iterations required to find the eigenvalues is **63** with QR method, it is only **8** for QR method with shifts. According to this result, **QR method with shifts** proven to be more efficient method.

Codes

Task 1

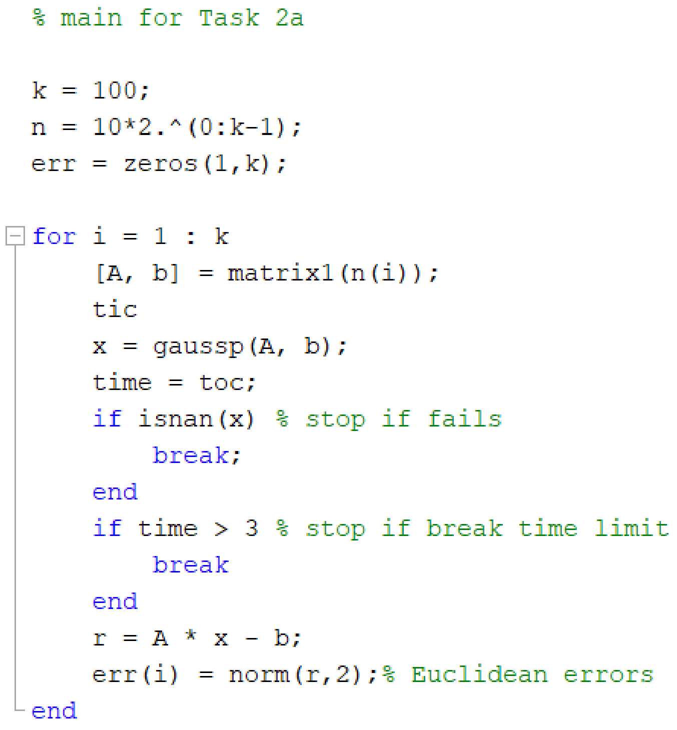
1.Program for calculating machine epsilon



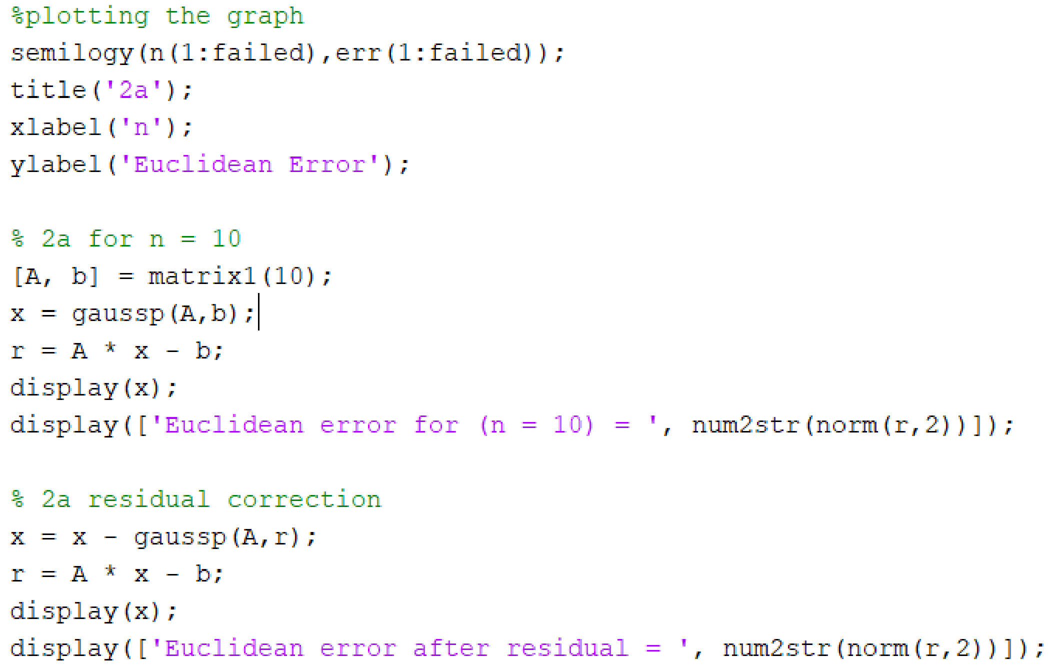
```
macheps = 1;
while 1.0 + (macheps/2) > 1.0 % divide macheps with 2 until left hand side
    macheps = macheps / 2;     % is not greater than 1.0
end
display(macheps)
```

Task 2

2.1 Main for 2a

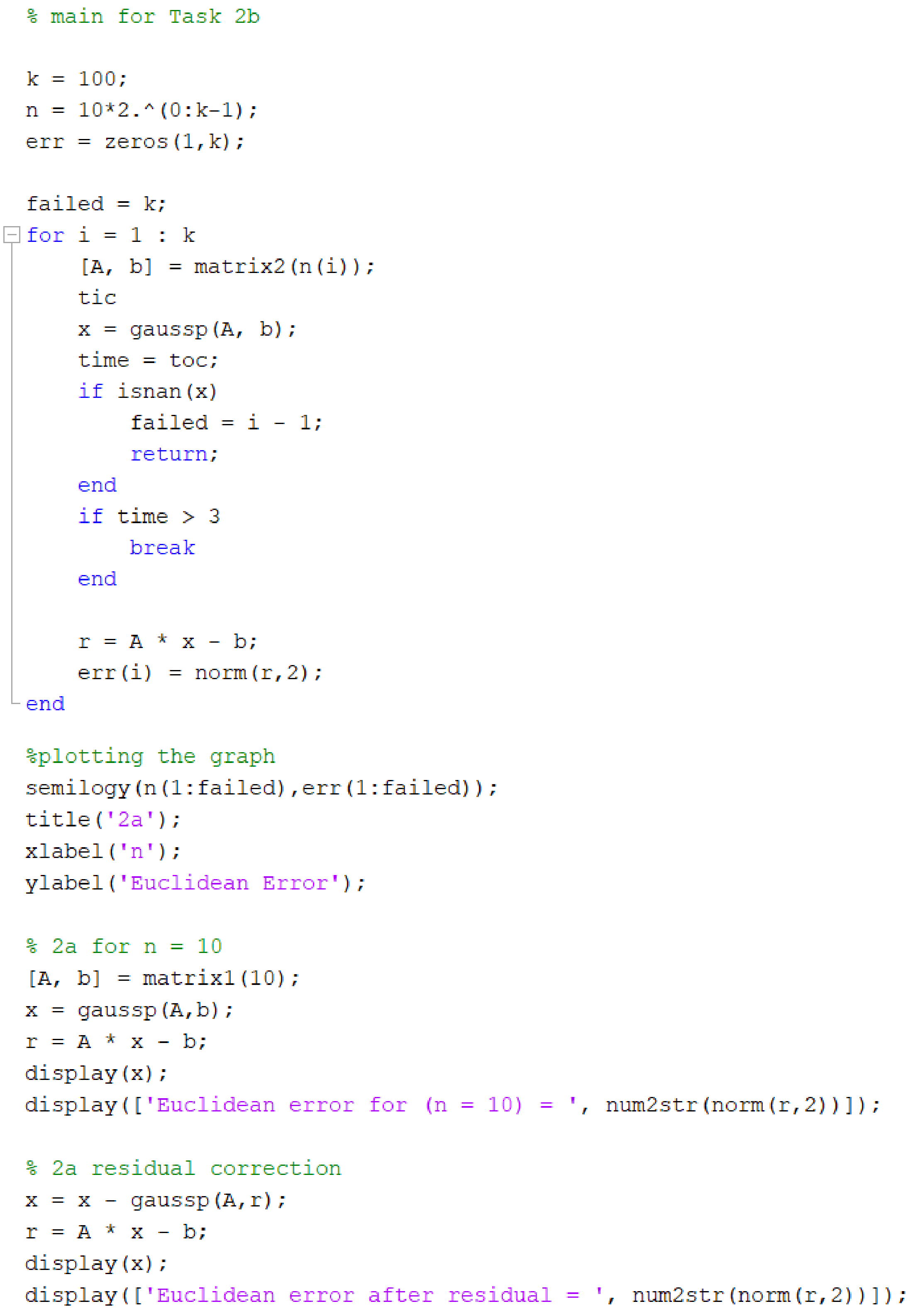


```
% main for Task 2a

k = 100;
n = 10*2.^(0:k-1);
err = zeros(1,k);

for i = 1 : k
    [A, b] = matrix1(n(i));
    tic
    x = gaussp(A, b);
    time = toc;
    if isnan(x) % stop if fails
        break;
    end
    if time > 3 % stop if break time limit
        break
    end
    r = A * x - b;
    err(i) = norm(r,2);% Euclidean errors
end
```



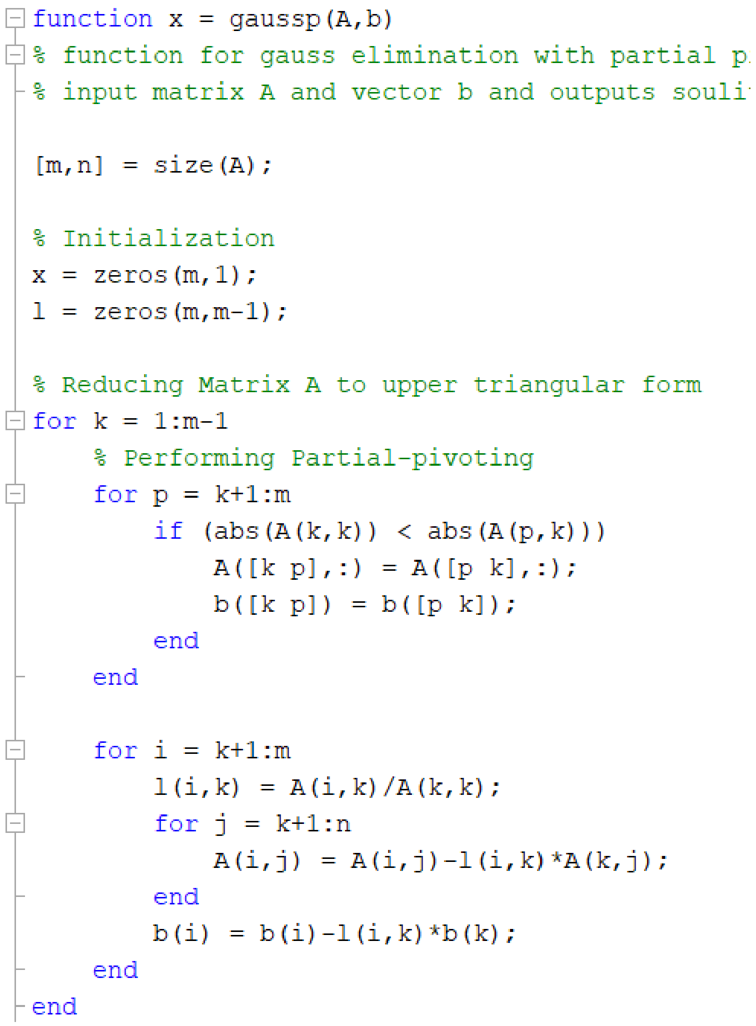
```
%plotting the graph
semilogy(n(1:failed),err(1:failed));
title('2a');
xlabel('n');
ylabel('Euclidean Error');

% 2a for n = 10
[A, b] = matrix1(10);
x = gaussp(A,b);
r = A * x - b;
display(x);
display(['Euclidean error for (n = 10) = ', num2str(norm(r,2))]);

% 2a residual correction
x = x - gaussp(A,r);
r = A * x - b;
display(x);
display(['Euclidean error after residual = ', num2str(norm(r,2))]);
```

2.2 Main for 2b

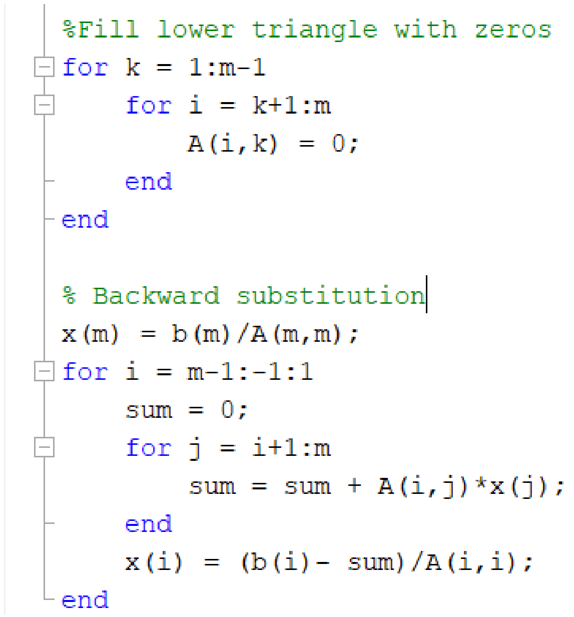


```
% main for Task 2b

k = 100;
n = 10*2.^(0:k-1);
err = zeros(1,k);

failed = k;
for i = 1 : k
    [A, b] = matrix2(n(i));
    tic
    x = gaussp(A, b);
    time = toc;
    if isnan(x)
        failed = i - 1;
        return;
    end
    if time > 3
        break
    end

    r = A * x - b;
    err(i) = norm(r,2);
end

%plotting the graph
semilogy(n(1:failed),err(1:failed));
title('2a');
xlabel('n');
ylabel('Euclidean Error');

% 2a for n = 10
[A, b] = matrix1(10);
x = gaussp(A,b);
r = A * x - b;
display(x);
display(['Euclidean error for (n = 10) = ', num2str(norm(r,2))]);

% 2a residual correction
x = x - gaussp(A,r);
r = A * x - b;
display(x);
display(['Euclidean error after residual = ', num2str(norm(r,2))]);
```

2.3. Gaussian Elimination with partial pivoting

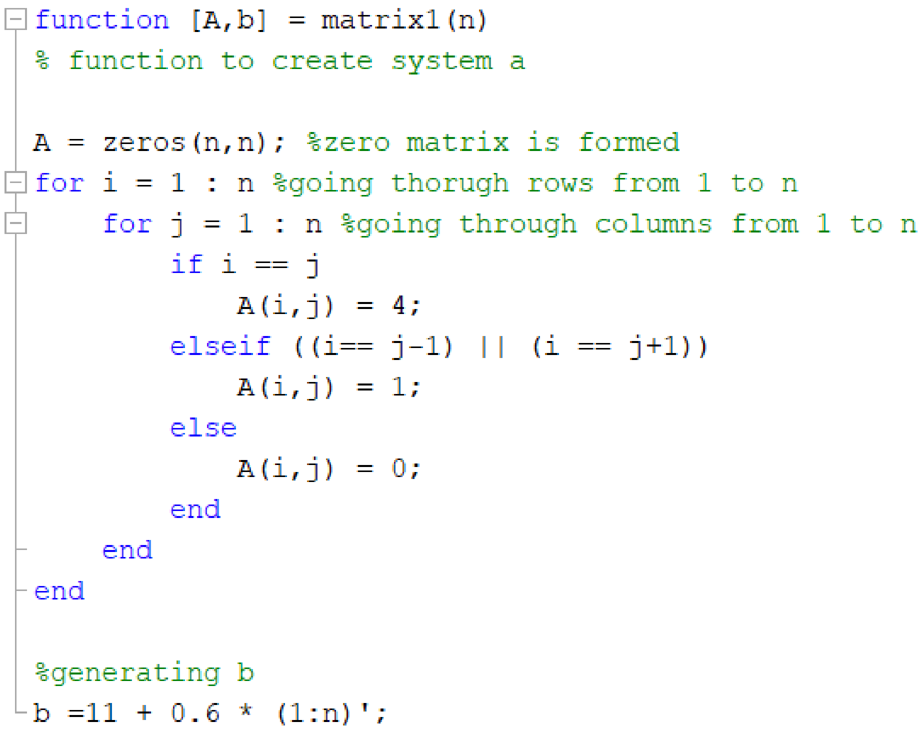


```
function x = gaussp(A,b)
% function for gauss elimination with partial p
% input matrix A and vector b and outputs souli

[m,n] = size(A);

% Initialization
x = zeros(m,1);
l = zeros(m,m-1);

% Reducing Matrix A to upper triangular form
for k = 1:m-1
    % Performing Partial-pivoting
    for p = k+1:m
        if (abs(A(k,k)) < abs(A(p,k)))
            A([k p],:) = A([p k],:);
            b([k p]) = b([p k]);
        end
    end

    for i = k+1:m
        l(i,k) = A(i,k)/A(k,k);
        for j = k+1:n
            A(i,j) = A(i,j)-l(i,k)*A(k,j);
        end
        b(i) = b(i)-l(i,k)*b(k);
    end
end
```

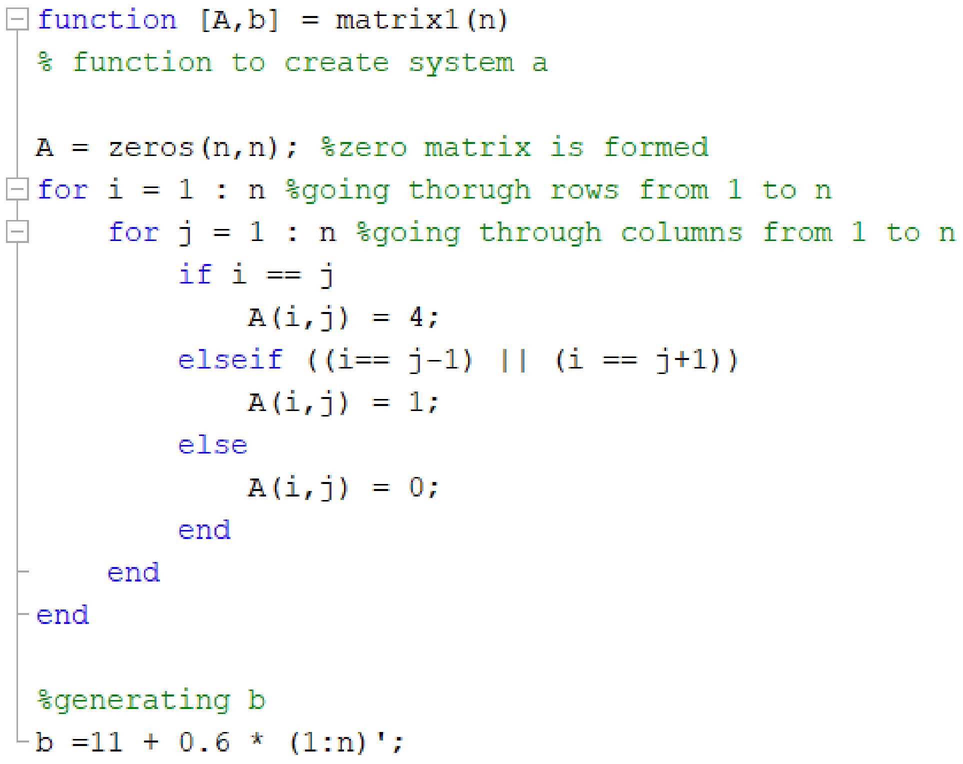


```
%Fill lower triangle with zeros
for k = 1:m-1
    for i = k+1:m
        A(i,k) = 0;
    end
end

% Backward substitution
x(m) = b(m)/A(m,m);
for i = m-1:-1:1
    sum = 0;
    for j = i+1:m
        sum = sum + A(i,j)*x(j);
    end
    x(i) = (b(i)- sum)/A(i,i);
end
```

2.4 Function for creating system a and system b



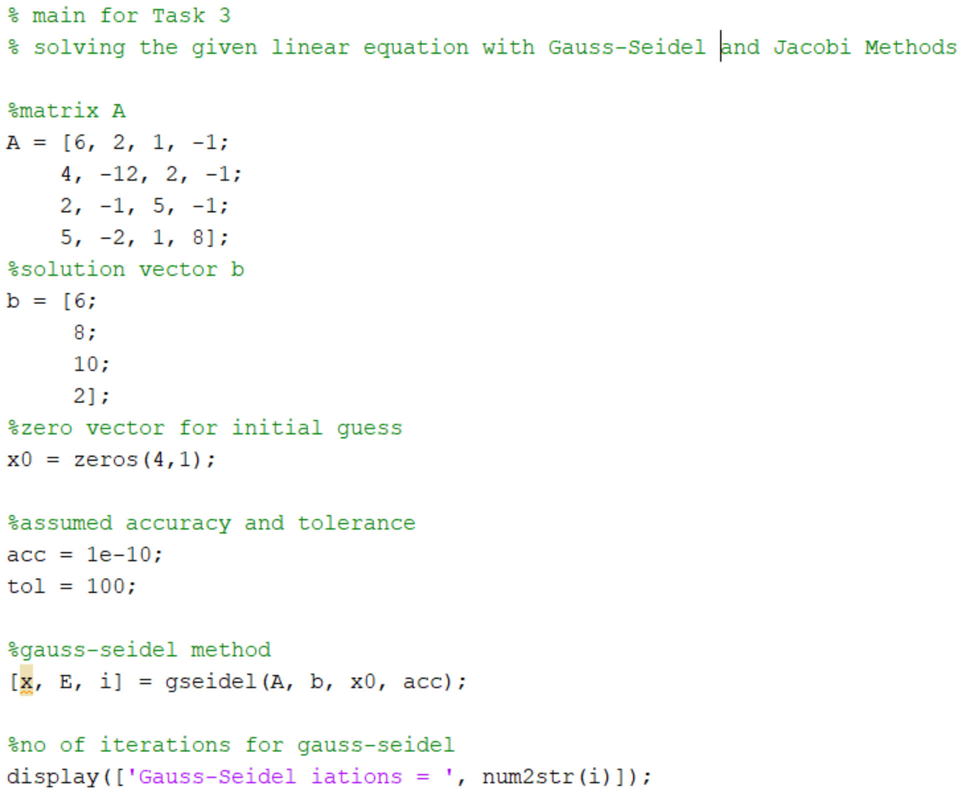
```
function [A,b] = matrix1(n)
% function to create system a

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) = 4;
        elseif ((i== j-1) || (i == j+1))
            A(i,j) = 1;
        else
            A(i,j) = 0;
        end
    end
end

%generating b
b =11 + 0.6 * (1:n)';
```



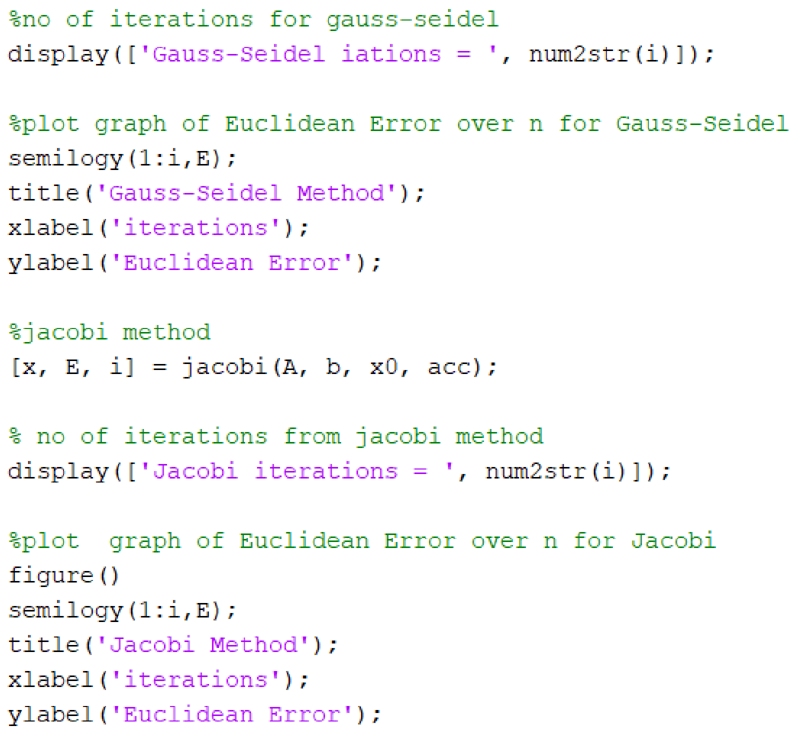
```
function [A,b] = matrix1(n)
% function to create system a

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) = 4;
        elseif ((i== j-1) || (i == j+1))
            A(i,j) = 1;
        else
            A(i,j) = 0;
        end
    end
end

%generating b
b =11 + 0.6 * (1:n)';
```

Task 3

3.1 Main for 3a

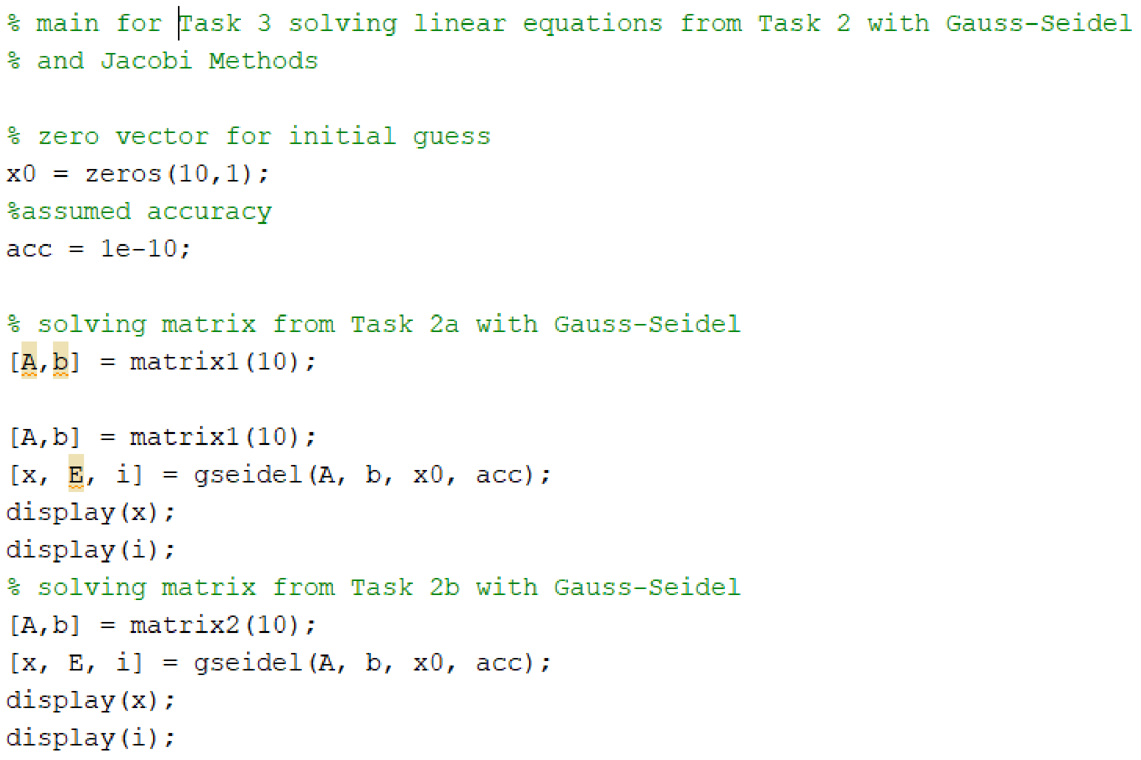


```
% main for Task 3
% solving the given linear equation with Gauss-Seidel and Jacobi Methods

%matrix A
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 for initial guess
x0 = zeros(4,1);

%assumed accuracy and tolerance
acc = 1e-10;
tol = 100;

%gauss-seidel method
[x, E, i] = gseidel(A, b, x0, acc);

%no of iterations for gauss-seidel
display(['Gauss-Seidel iations = ', num2str(i)]);
```



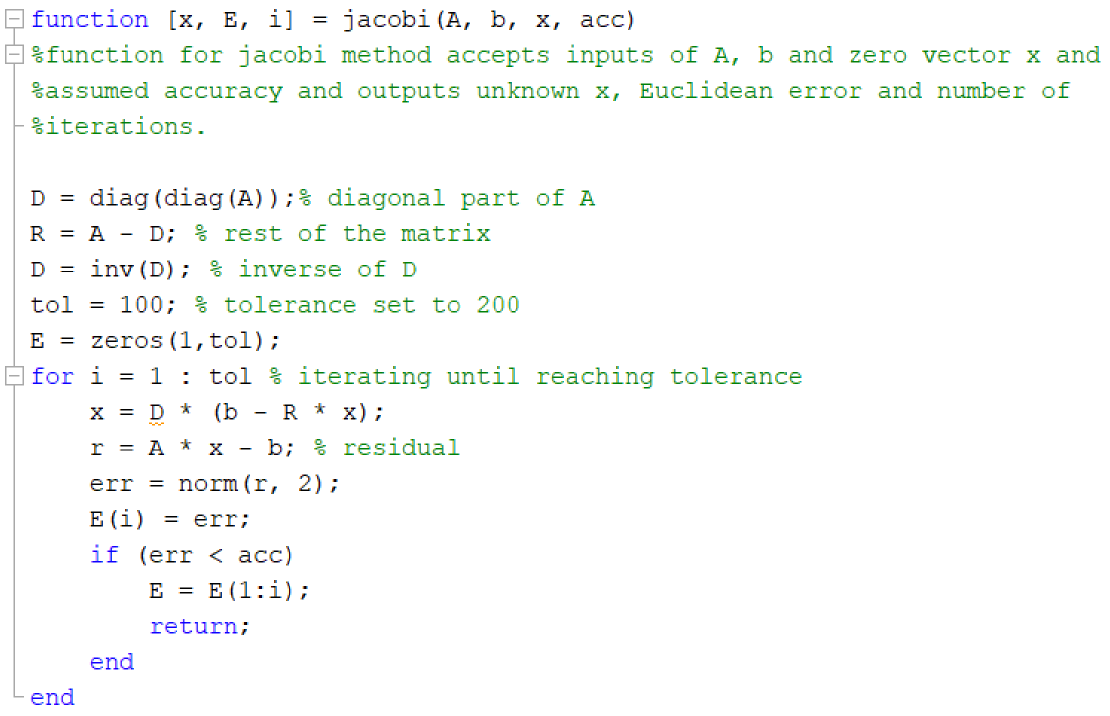
```
%no of iterations for gauss-seidel
display(['Gauss-Seidel iations = ', num2str(i)]);

%plot graph of Euclidean Error over n for Gauss-Seidel
semilogy(1:i,E);
title('Gauss-Seidel Method');
xlabel('iterations');
ylabel('Euclidean Error');

%jacobi method
[x, E, i] = jacobi(A, b, x0, acc);

% no of iterations from jacobi method
display(['Jacobi iterations = ', num2str(i)]);

%plot  graph of Euclidean Error over n for Jacobi
figure()
semilogy(1:i,E);
title('Jacobi Method');
xlabel('iterations');
ylabel('Euclidean Error');
```

3.2 Main for 3b

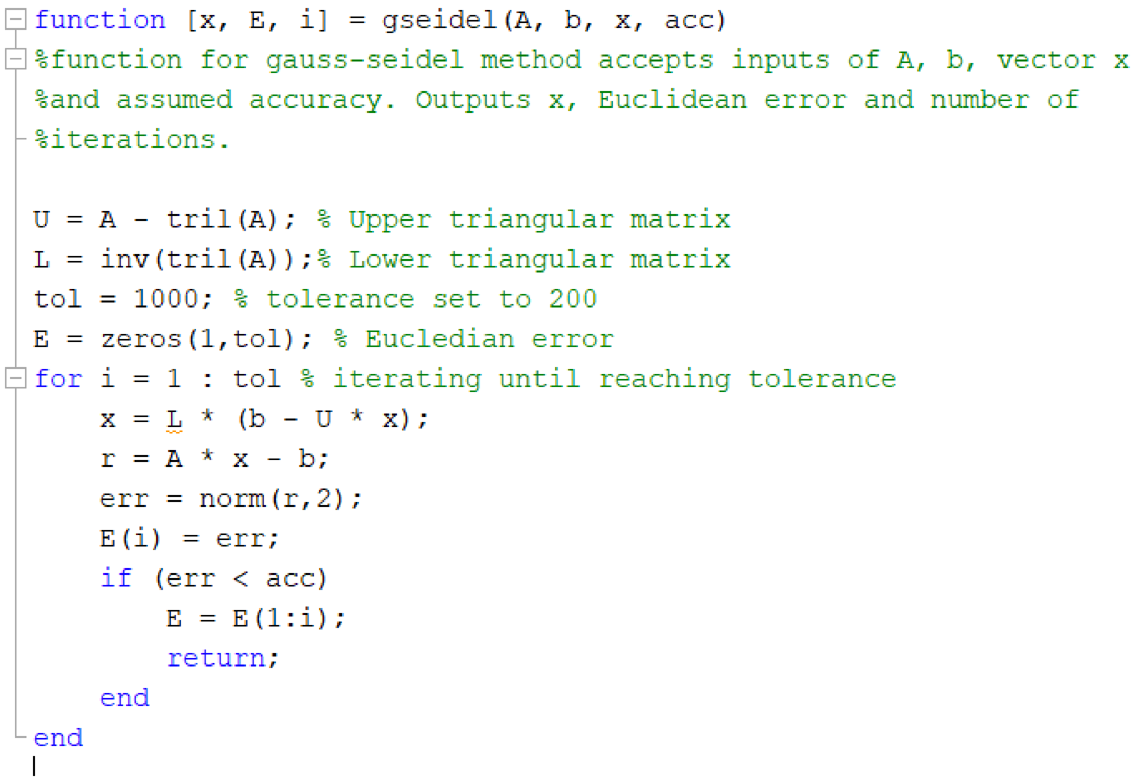


```
% main for Task 3 solving linear equations from Task 2 with Gauss-Seidel
% and Jacobi Methods

% zero vector for initial guess
x0 = zeros(10,1);
%assumed accuracy
acc = 1e-10;

% solving matrix from Task 2a with Gauss-Seidel
[A,b] = matrix1(10);

[A,b] = matrix1(10);
[x, E, i] = gseidel(A, b, x0, acc);
display(x);
display(i);
% solving matrix from Task 2b with Gauss-Seidel
[A,b] = matrix2(10);
[x, E, i] = gseidel(A, b, x0, acc);
display(x);
display(i);
```

3.3 Jacobi and Gauss-Seidel Methods

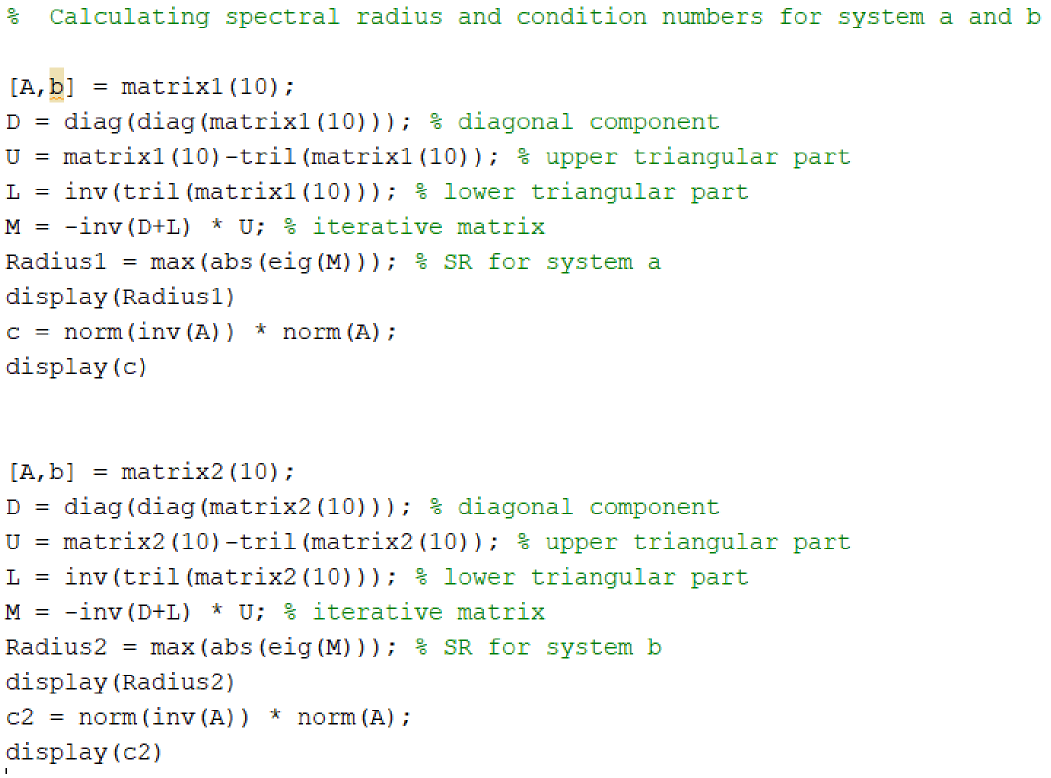


```
function [x, E, i] = jacobi(A, b, x, acc)
%function for jacobi method accepts inputs of A, b and zero vector x and
%assumed accuracy and outputs unknown x, Euclidean error and number of
%iterations.

D = diag(diag(A));% diagonal part of A
R = A - D; % rest of the matrix
D = inv(D); % inverse of D
tol = 100; % tolerance set to 200
E = zeros(1,tol);
for i = 1 : tol % iterating until reaching tolerance
    x = D * (b - R * x);
    r = A * x - b; % residual
    err = norm(r, 2);
    E(i) = err;
    if (err < acc)
        E = E(1:i);
        return;
    end
end
```



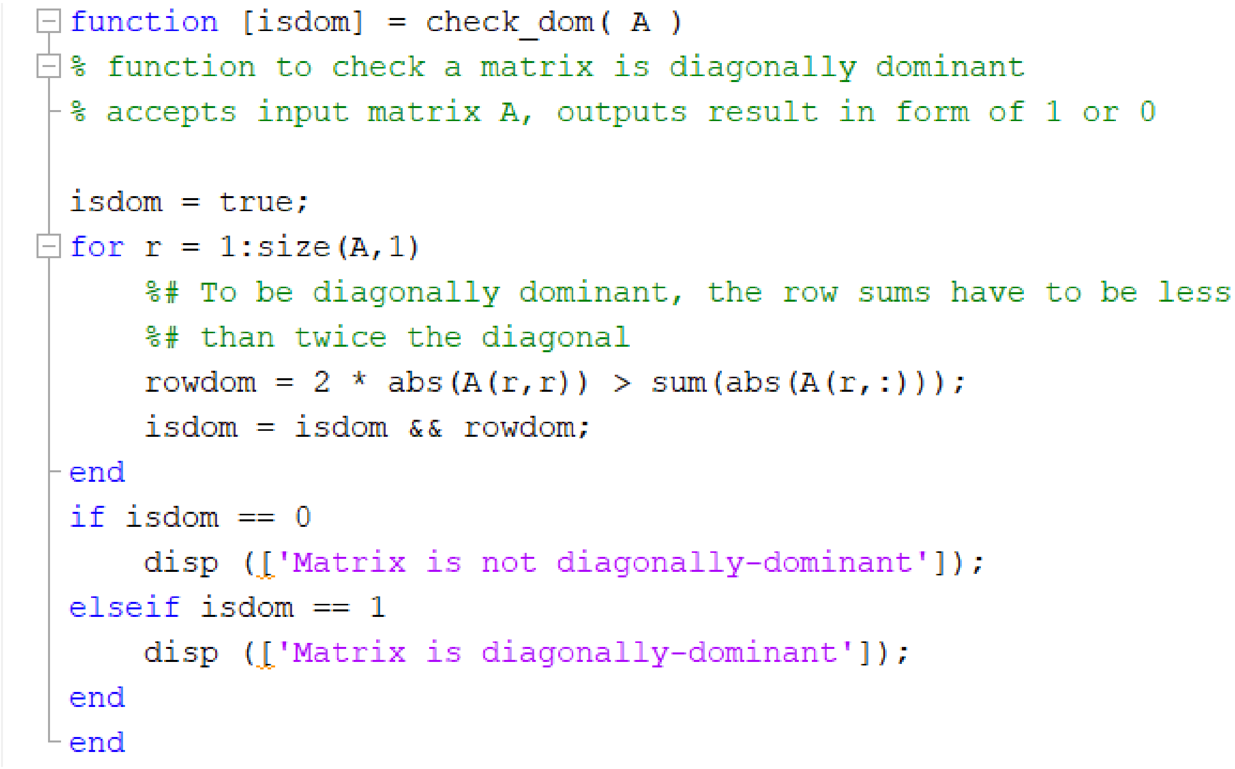
```
function [x, E, i] = gseidel(A, b, x, acc)
%function for gauss-seidel method accepts inputs of A, b, vector x
%and assumed accuracy. Outputs x, Euclidean error and number of
%iterations.

U = A - tril(A); % Upper triangular matrix
L = inv(tril(A));% Lower triangular matrix
tol = 1000; % tolerance set to 200
E = zeros(1,tol); % Eucledian error
for i = 1 : tol % iterating until reaching tolerance
    x = L * (b - U * x);
    r = A * x - b;
    err = norm(r,2);
    E(i) = err;
    if (err < acc)
        E = E(1:i);
        return;
    end
end
```

3.4 Calculating Spectral Radius and Condition Number



```
%  Calculating spectral radius and condition numbers for system a and b

[A,b] = matrix1(10);
D = diag(diag(matrix1(10))); % diagonal component
U = matrix1(10)-tril(matrix1(10)); % upper triangular part
L = inv(tril(matrix1(10))); % lower triangular part
M = -inv(D+L) * U; % iterative matrix
Radius1 = max(abs(eig(M))); % SR for system a
display(Radius1)
c = norm(inv(A)) * norm(A);
display(c)


[A,b] = matrix2(10);
D = diag(diag(matrix2(10))); % diagonal component
U = matrix2(10)-tril(matrix2(10)); % upper triangular part
L = inv(tril(matrix2(10))); % lower triangular part
M = -inv(D+L) * U; % iterative matrix
Radius2 = max(abs(eig(M))); % SR for system b
display(Radius2)
c2 = norm(inv(A)) * norm(A);
display(c2)
```

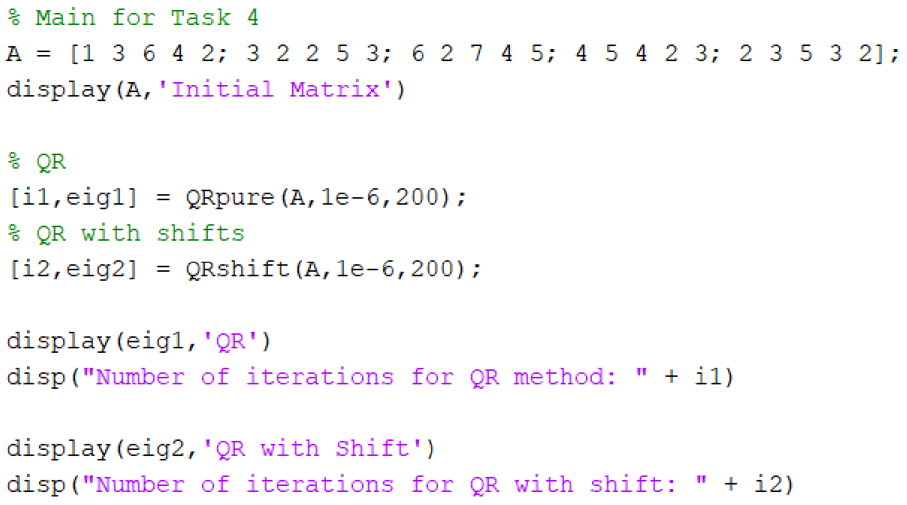
3.5 Function checking if matrix diagonally dominant



```
function [isdom] = check_dom( A )
% function to check a matrix is diagonally dominant
% accepts input matrix A, outputs result in form of 1 or 0

isdom = true;
for r = 1:size(A,1)
    %# To be diagonally dominant, the row sums have to be less
    %# than twice the diagonal
    rowdom = 2 * abs(A(r,r)) > sum(abs(A(r,:)));
    isdom = isdom && rowdom;
end
if isdom == 0
    disp (['Matrix is not diagonally-dominant']);
elseif isdom == 1
    disp (['Matrix is diagonally-dominant']);
end
end
```

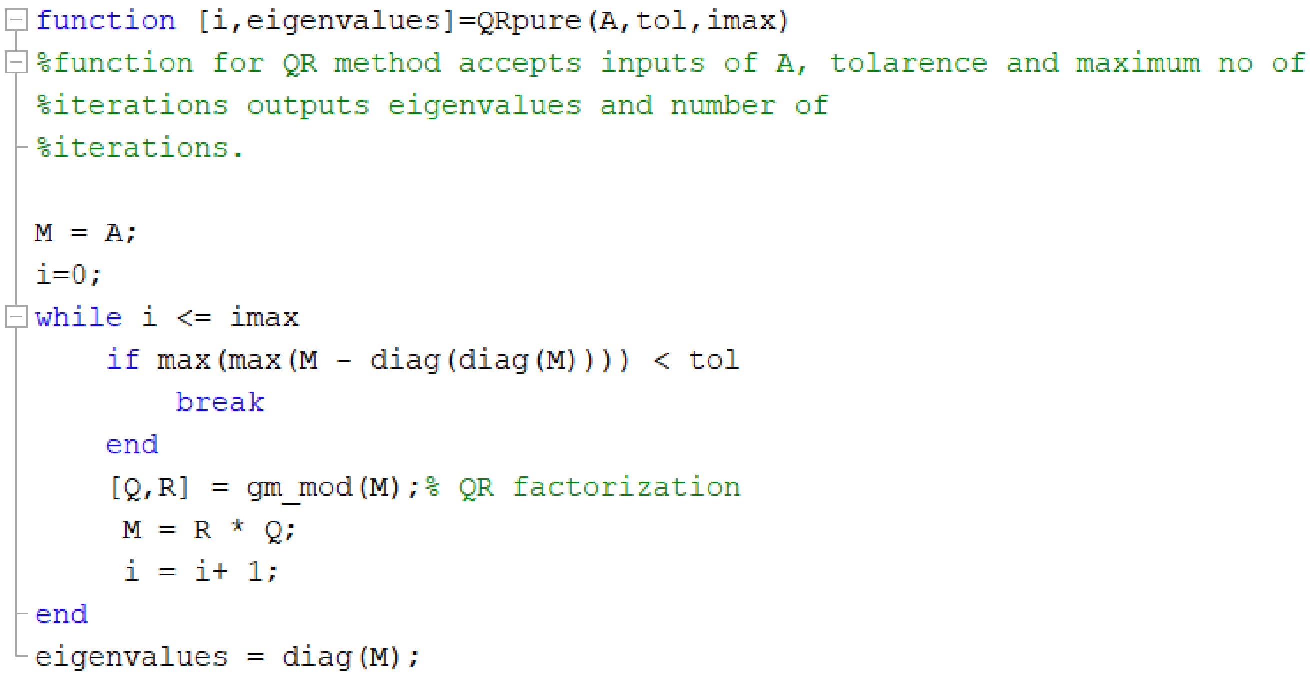
Task 4

4.1 Main for Task 4



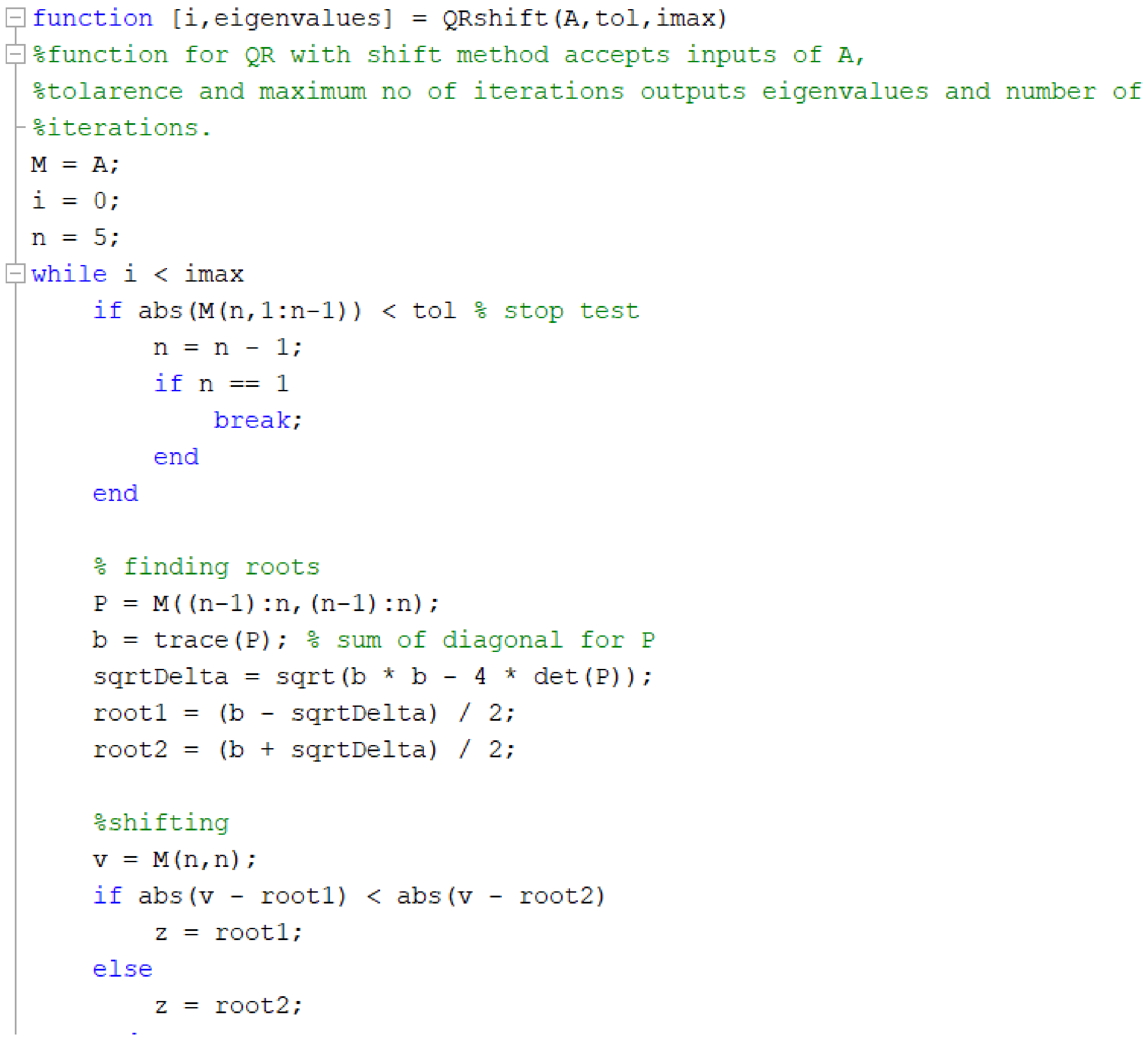
```
% Main for Task 4
A = [1 3 6 4 2; 3 2 2 5 3; 6 2 7 4 5; 4 5 4 2 3; 2 3 5 3 2];
display(A,'Initial Matrix')

% QR
[i1,eig1] = QRpure(A,1e-6,200);
% QR with shifts
[i2,eig2] = QRshift(A,1e-6,200);

display(eig1,'QR')
disp("Number of iterations for QR method: " + i1)

display(eig2,'QR with Shift')
disp("Number of iterations for QR with shift: " + i2)
```

4.2 QR

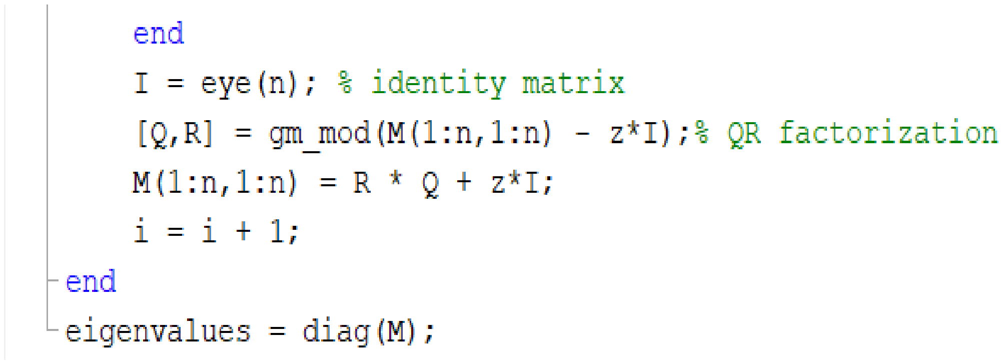


```
function [i,eigenvalues]=QRpure(A,tol,imax)
%function for QR method accepts inputs of A, tolarence and maximum no of
%iterations outputs eigenvalues and number of
%iterations.

M = A;
i=0;
while i <= imax
    if max(max(M - diag(diag(M)))) < tol
        break
    end
    [Q,R] = gm_mod(M);% QR factorization
     M = R * Q;
     i = i+ 1;
end
eigenvalues = diag(M);
```

4.3 QR with shifts

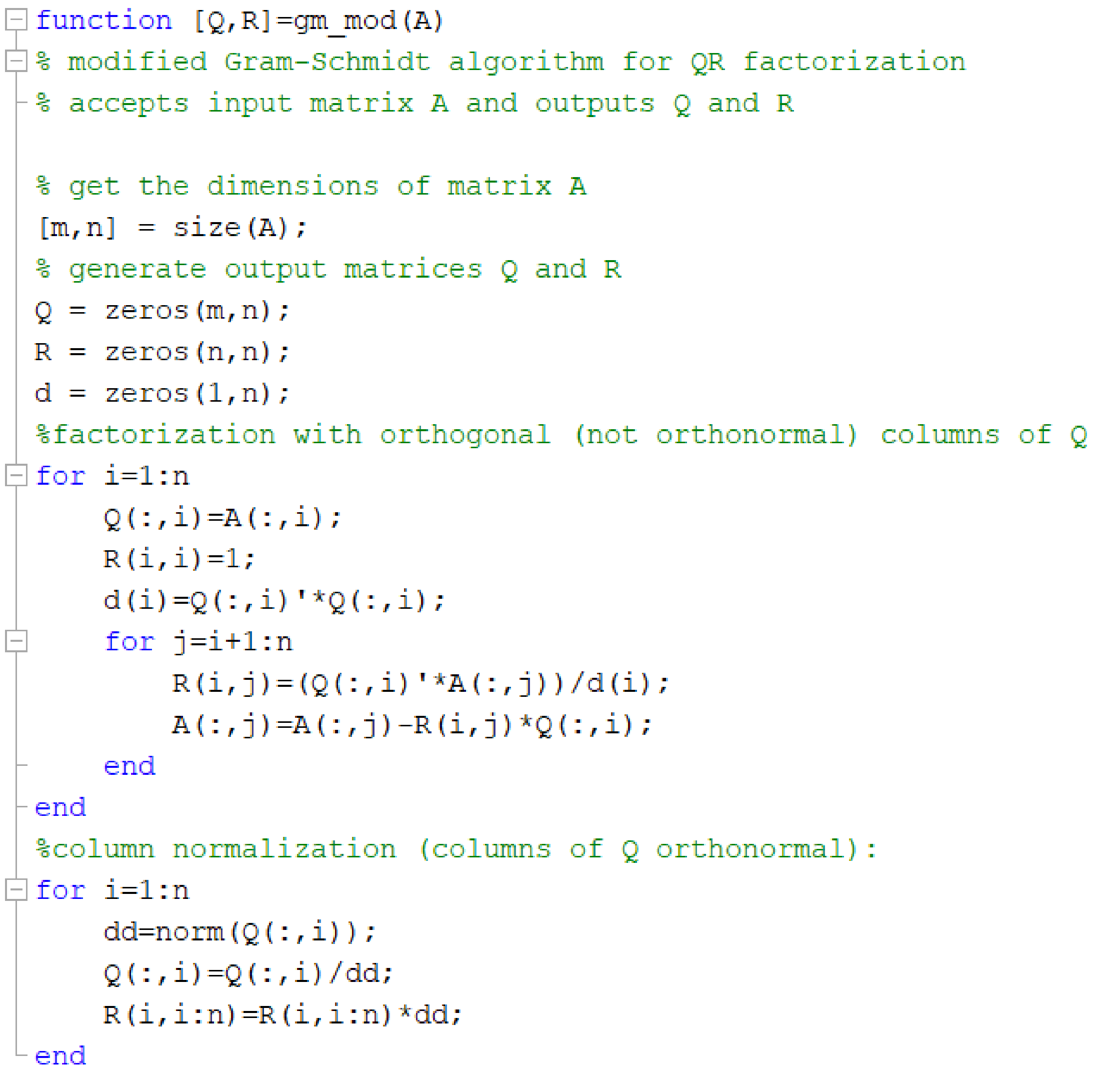


```
function [i,eigenvalues] = QRshift(A,tol,imax)
%function for QR with shift method accepts inputs of A,
%tolarence and maximum no of iterations outputs eigenvalues and number of
%iterations.
M = A;
i = 0;
n = 5;
while i < imax
    if abs(M(n,1:n-1)) < tol % stop test
        n = n - 1;
        if n == 1
            break;
        end
    end

    % finding roots
    P = M((n-1):n,(n-1):n);
    b = trace(P); % sum of diagonal for P
    sqrtDelta = sqrt(b * b - 4 * det(P));
    root1 = (b - sqrtDelta) / 2;
    root2 = (b + sqrtDelta) / 2;

    %shifting
    v = M(n,n);
    if abs(v - root1) < abs(v - root2)
        z = root1;
    else
        z = root2;
```



```
    end
    I = eye(n); % identity matrix
    [Q,R] = gm_mod(M(1:n,1:n) - z*I);% QR factorization
    M(1:n,1:n) = R * Q + z*I;
    i = i + 1;
end
eigenvalues = diag(M);
```

4.4 Modified Gram-Schmidt Method



```
function [Q,R]=gm_mod(A)
% modified Gram-Schmidt algorithm for QR factorization
% accepts input matrix A and outputs Q and R

% get the dimensions of matrix A
[m,n] = size(A);
% generate output matrices Q and R
Q = zeros(m,n);
R = zeros(n,n);
d = zeros(1,n);
%factorization with orthogonal (not orthonormal) columns of Q
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
%column normalization (columns of Q orthonormal):
for i=1:n
    dd=norm(Q(:,i));
    Q(:,i)=Q(:,i)/dd;
    R(i,i:n)=R(i,i:n)*dd;
end
```