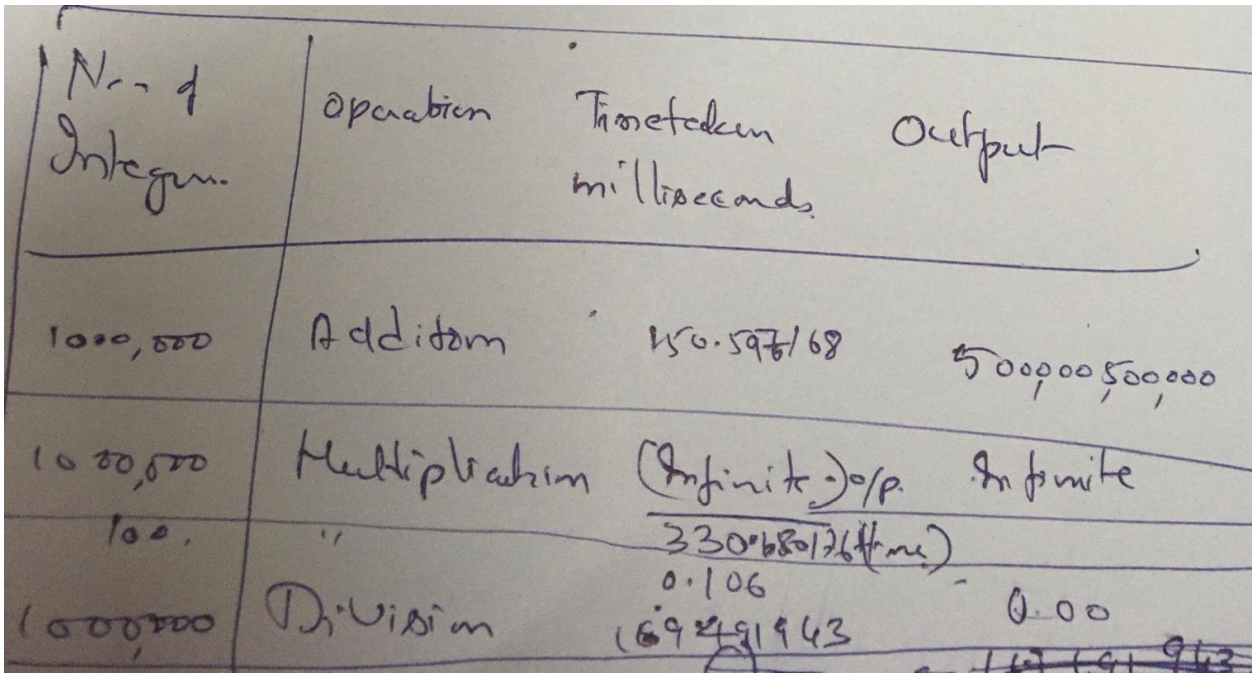
Student: Lakshmi Vardhani Sistla
ID: 2021FC04943
Section-2
EmailID:
2021FC04943@wilp.bits-pilani.ac.in

Work Integrated Learning Programmes Division
Assignment 1
DSECL ZC416 - Mathematical Foundations for Data Science

Q1) Implementing Gaussian Elimination Method
(i) Find the approximate time your computer takes for a single addition by adding first 10^6 positive integers using a for loop and dividing the time taken by 10^6. Similarly find the approximate time taken for a single multiplication and division. Report the result obtained in the form of a table. (0.5)
Deliverable(s): A tabular column indicating the time taken for each of the operations

Answer:

Time taken to execute below code for Addition, Multiplication and Division of 10^6 numbers



Code using Python:

```
import time

# Online Python compiler (interpreter) to run Python online.

# Write Python 3 code in this online editor and run it.

sumx=1

x=1000000

ms1 = time.time()*1000

print(ms1)

print("Current Time =", ms1)
```

```
while x != 1 :
    sumx=sumx+x
    x=x-1
print(sumx)
ms2 = time.time()*1000

print("end time taken=", (ms2))

print("Time taken=", (ms2-ms1))
```

output

Current Time = 1639044612353.9417

Value: 500000500000

end time taken= 1639044612504.5388

Time taken= 150.59716796875

--

```
import time
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
multx=1.0
x=1000000.0
ms1 = time.time()*1000
print(ms1)
print("Current Time =", ms1)

while x != 1 :
    multx=multx*x
    x=x-1
print(multx)
ms2 = time.time()*1000

print("end time taken=", (ms2))

print("Time taken=", (ms2-ms1))
```

Multiplication of 10^6 integers output

Current Time = 1639044556930.7104

inf

end time taken= 1639044557261.3906

Time taken= 330.68017578125

```
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
divx=1.0
x=1000000.0
ms1 = time.time()*1000
print(ms1)
print("Current Time =", ms1)

while x != 1 :
    divx=divx/x
    x=x-1
print(divx)
ms2 = time.time()*1000

print("Time taken=", (ms2-ms1))
```

Division of 10^6 integers Output

Current Time = 1639044472498.1187

Value: 0.0

end time taken= 1639044472667.6106

Time taken= 169.491943359375

(ii) Write a function to implement Gauss elimination with and without pivoting. Also write the code to count the number of additions, multiplications and divisions performed during Gaussian elimination. Ensure that the Gauss elimination performs 5S arithmetic which necessitates 5S arithmetic rounding for every addition, multiplication and division performed in the algorithm. If this is not implemented correctly, the rest of the answers will be considered invalid. Note that this is not same as simple 5 digit rounding at the end of the computation. Do not hardwire 5S arithmetic in the code and use dS instead. The code can then be run with various values of d. (0.5 + 0.5)

Deliverable(s): The code for the Gaussian elimination with and without partial pivoting with the rounding part

Answer

Basis for the arithmetic calculations :

$$\text{Total Number of Additions/Subtractions from A} \to \text{U} = \frac{n(n-1)(2n-1)}{6}$$

$$\text{Total Number of Multiplications/Divisions from A} \to \text{U} = \frac{n(n-1)(2n-1)}{6} + \frac{n(n-1)}{2} = \frac{n(n-1)(2n-1)}{6} + \frac{3n(n-1)}{6} = \frac{n(n^2} \tag{3}$$

We will now count the number of additions/ subtractions and the number of multiplications/divisions in going from $b \to g$. We have that:

$$\text{Total Number of Additions/Subtractions from b} \to \text{g} = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} \tag{4}$$

$$\text{Total Number of Multiplications/Divisions from b} \to \text{g} = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} \tag{5}$$

Lastly we count the number of additions/subtractions and multiplications/divisions for finding the solutions from the back-substitution method.

$$\text{Total Number of Additions/Subtractions from g} \to \text{x} = 0 + 1 + \ldots + (n-1) = \frac{n(n-1)}{2} \tag{6}$$

$$\text{Total Number of Multiplications/Divisons from g} \to \text{x} = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} \tag{7}$$

Therefore the total number of operations to obtain the solution of a system of n linear equations in n variables using Gaussian Elimination is:

$$\text{Total Number of Additions/Subtractions to Obtain Solution} = \frac{n(n-1)(2n+5)}{6} \tag{8}$$

$$\text{Total Number of Multiplications/Divisinons to Obtain Solution} = \frac{n(n^2+3n-1)}{3} \tag{9}$$

Gaussian Elimination without Pivoting

```
import numpy as np
import math


def forward_elimination(A, b, n):
    """
    Calculates the forward part of Gaussian elimination.
    """
    for row in range(0, n-1):
        for i in range(row+1, n):
            factor = A[i,row] / A[row,row]
            for j in range(row, n):
                A[i,j] = A[i,j] - factor * A[row,j]

            b[i] = b[i] - factor * b[row]

        print('A = \n%s and b = %s' % (A,b))
    return A, b

def back_substitution(a, b, n):
    """
    Does back substitution, returns the Gauss result.
    """
    x = np.zeros((n,1))
    x[n-1] = b[n-1] / a[n-1, n-1]
    for row in range(n-2, -1, -1):
        sums = b[row]
        for j in range(row+1, n):
            sums = sums - a[row,j] * x[j]
        x[row] = sums / a[row,row]
    return x

def gauss(A, b):
    """
    This function performs Gauss elimination without pivoting.
    """
    n = A.shape[0]
```

Generates random matrix with given condition number

```
def gen_rand_matrix(m,n,kappa):
#
#
    k= min(m,n)
    (Q1,R1) = np.linalg.qr(np.random.rand(m,m))
    (Q2,R2) = np.linalg.qr(np.random.rand(n,n))

    U = Q1[0:m,0:k]
    V = Q2[0:k,0:n]
    j=k-1

    l = kappa**(1/j)
    x1 = np.arange(0,j+1)
    x2 = np.flip(x1,axis=0)
    sing = np.power(l,x2)
    S = np.diag(sing)
    Vt = np.transpose(V)

        M = np.dot(np.dot(U,S),Vt)
        return M
```

This part here is testing

```
kappa = 1/(math.exp(1e-3)-1)
m=100

A  = gen_rand_matrix(m,m,kappa)
x = np.random.rand(m,1)
b = np.dot(A,x)
xhat1 = gauss(A,b)

xerror1 = np.linalg.norm(xhat1-x)/np.linalg.norm(x)
```

For Arithmetic below is the explanation.

Consider a system of n linear equations in n unknowns and the corresponding $n \times n$ coefficient matrix $A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$, the $n \times 1$ solution matrix $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$, and the $n \times 1$ constant matrix $b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$. Then $Ax = b$. Earlier we saw that at the $(n-1)^{\text{th}}$ step of Gaussian Elimination, we obtain the following system of equations:

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1n}x_n &= g_2 \\ u_{22}x_2 + \cdots + u_{2n}x_n &= g_2 \\ &\vdots \\ u_{nn}x_n &= g_n \end{aligned} \tag{1}$$

The values of the u's and the g's were described on the The Gaussian Elimination Algorithm page. Let $U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{nn} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$ be the $n \times n$ coefficient matrix for this equivalent system, and let $g = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}$ be the $n \times 1$ equivalent constant matrix. Then $Ux = g$. Then the following table states the

operations count from going from A to U at each step $1, 2, \ldots, n-1$.

Step Number	Number of Additions/Subtractions from $A \to U$	Number of Multiplications from $A \to U$	Number of Divisions from $A \to U$
1	$(n-1)^2$	$(n-1)^2$	$n-1$
2	$(n-2)^2$	$(n-2)^2$	$n-2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	4	4	2
$n-1$	1	1	1
Totals:	Total Num. of Add/Sub $= \frac{n(n-1)(2n-1)}{6}$	Total Num. of Multiplications $= \frac{n(n-1)(2n-1)}{6}$	Total Num. of Divisions $= \frac{n(n-1}{2}$

We will count the number of additions/subtractions together and the number of multiplications/divisions together. The total number of additions/subtractions and multiplications/divisions in going from $A \to U$ is:

$$\text{Total Number of Additions/Subtractions from A} \to \text{U} = \frac{n(n-1)(2n-1)}{6}$$

$$\text{Total Number of Multiplications/Divisions from A} \to \text{U} = \frac{n(n-1)(2n-1)}{6} + \frac{n(n-1)}{2} = \frac{n(n-1)(2n-1)}{6} + \frac{3n(n-1)}{6} = \frac{n(n^2} {3} \tag{3}$$

We will now count the number of additions/ subtractions and the number of multiplications/divisions in going from $b \to g$. We have that:

$$\text{Total Number of Additions/Subtractions from b} \to \text{g} = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} \tag{4}$$

$$\text{Total Number of Multiplications/Divisions from b} \to \text{g} = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} \tag{5}$$

Lastly we count the number of additions/subtractions and multiplications/divisions for finding the solutions from the back-substitution method.

$$\text{Total Number of Additions/Subtractions from g} \to \text{x} = 0 + 1 + \ldots + (n-1) = \frac{n(n-1)}{2} \tag{6}$$

$$\text{Total Number of Multiplications/Divisons from g} \to \text{x} = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} \tag{7}$$

Therefore the total number of operations to obtain the solution of a system of n linear equations in n variables using Gaussian Elimination is:

$$\text{Total Number of Additions/Subtractions to Obtain Solution} = \frac{n(n-1)(2n+5)}{6} \tag{8}$$

$$\text{Total Number of Multiplications/Divisinons to Obtain Solution} = \frac{n(n^2+3n-1)}{3} \tag{9}$$

1. iii) Generate random matrices of size n × n where n = 100, 200, . . . , 1000. Also generate a random b ∈ R_n for each case. Each number must be of the form m.dddd (Example : 4.5444) which means it has 5 Significant digits in total. Perform Gaussian elimination with and without partial pivoting for each n value (10 cases) above. Report the number of additions, divisions and multiplications for each case in the form of a table. No need of the code and the matrices / vectors. (0.5 + 0.5)
Deliverable(s): Two tabular columns indicating the number of additions, multiplications and divisions for each value of n, for with and without pivoting

Step Number	Number of Additions/Subtractions from $A \to U$	Number of Multiplications from $A \to U$	Number of Divisions from $A \to U$
1	$(n-1)^2$	$(n-1)^2$	$n-1$
2	$(n-2)^2$	$(n-2)^2$	$n-2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	4	4	2
$n-1$	1	1	1
Totals:	Total Num. of Add/Sub $= \frac{n(n-1)(2n-1)}{6}$	Total Num. of Multiplications $= \frac{n(n-1)(2n-1)}{6}$	Total Num. of Divisions $= \frac{n(n-1}{2}$

Div: n(n^2 +3n-1)/3

Table for values 100, 200, 300, 1000

N	Addition	multiplication	Division		
100	328,350	328,350	3233		
200	2.646,700	2.646,700	13,133		

300	8,955,050	8,955,050	30,299.67		
1000	332,833,500	332,833,500	334,333		

Gaussian elimination python program:

```
# Importing NumPy Library
import numpy as np
import sys

# Reading number of unknowns


n = int(input('Enter number of unknowns: '))

# Making numpy array of n x n+1 size and initializing
# to zero for storing augmented matrix
a = np.zeros((n,n+1))

# Making numpy array of n size and initializing
# to zero for storing solution vector
x = np.zeros(n)

# Reading augmented matrix coefficients
print('Enter Augmented Matrix Coefficients:')
for i in range(n):
    for j in range(n+1):
        a[i][j] = float(input( 'a['+str(i)+']['+ str(j)+']='))

# Applying Gauss Elimination
for i in range(n):
    if a[i][i] == 0.0:
        sys.exit('Divide by zero detected!')

    for j in range(i+1, n):
        ratio = a[j][i]/a[i][i]

        for k in range(n+1):
            a[j][k] = a[j][k] - ratio * a[i][k]

# Back Substitution
x[n-1] = a[n-1][n]/a[n-1][n-1]

for i in range(n-2,-1,-1):
    x[i] = a[i][n]

    for j in range(i+1,n):
        x[i] = x[i] - a[i][j]*x[j]

    x[i] = x[i]/a[i][i]

# Displaying solution
print('\nRequired solution is: ')
for i in range(n):
    print('X%d = %0.2f' %(i,x[i]), end = '\t')
```

Output

```
Enter number of unknowns: 3
Enter Augmented Matrix Coefficients:
a[0][0]=1
a[0][1]=1
a[0][2]=1
a[0][3]=9
a[1][0]=2
a[1][1]=-3
a[1][2]=4
a[1][3]=13
a[2][0]=3
a[2][1]=4
a[2][2]=5
a[2][3]=40



 Required solution is:
 X0 = 1.00      X1 = 3.00      X2 = 5.00
```

Gaussian Elimination without pivot:

```
mport numpy as np

class GEPP():

 """

 Gaussian elimination with partial pivoting.

 input: A is an n x n numpy matrix

 b is an n x 1 numpy array

 output: x is the solution of Ax=b
```

with the entries permuted in

accordance with the pivoting

done by the algorithm

post-condition: A and b have been modified.

:return

"""

def __init__(self, A, b, doPricing=True):

#super(GEPP, self).__init__()

self.A = A # input: A is an n x n numpy matrix

self.b = b # b is an n x 1 numpy array

self.doPricing = doPricing

self.n = None # n is the length of A

self.x = None # x is the solution of Ax=b

self._validate_input() # method that validates input

self._elimination() # method that conducts elimination

self._backsub() # method that conducts back-substitution

def _validate_input(self):

self.n = len(self.A)

if self.b.size != self.n:

raise ValueError("Invalid argument: incompatible sizes between" +

"A & b.", self.b.size, self.n)

def _elimination(self):

"""

k represents the current pivot row. Since GE traverses the matrix in the

upper right triangle, we also use k for indicating the k-th diagonal

column index.

:return

"""

Elimination

for k in range(self.n - 1):

if self.doPricing:

Pivot

maxindex = abs(self.A[k:, k]).argmax() + k

if self.A[maxindex, k] == 0:

raise ValueError("Matrix is singular.")

Swap

```
 if maxindex != k:
 self.A[[k, maxindex]] = self.A[[maxindex, k]]
 self.b[[k, maxindex]] = self.b[[maxindex, k]]
 else:
 if self.A[k, k] == 0:
 raise ValueError("Pivot element is zero. Try setting doPricing to True.")
 # Eliminate
 for row in range(k + 1, self.n):
 multiplier = self.A[row, k] / self.A[k, k]
 self.A[row, k:] = self.A[row, k:] - multiplier * self.A[k, k:]
 self.b[row] = self.b[row] - multiplier * self.b[k]
 def _backsub(self):
 # Back Substitution
 self.x = np.zeros(self.n)
 for k in range(self.n - 1, -1, -1):
 self.x[k] = (self.b[k] - np.dot(self.A[k, k + 1:], self.x[k + 1:])) / self.A[k, k]
def main():
 A = np.array([[1., -1., 1., -1.],
 [1., 0., 0., 0.],
 [1., 1., 1., 1.],
 [1., 2., 4., 8.]])
 b = np.array([[14.],
 [4.],
 [2.],
 [2.]])
 GaussElimPiv = GEPP(np.copy(A), np.copy(b), doPricing=False)
 print(GaussElimPiv.x)
 print(GaussElimPiv.A)
 print(GaussElimPiv.b)
 GaussElimPiv = GEPP(A, b)
 print(GaussElimPiv.x)
if __name__ == "__main__":
 main()
```

GEPP.py

```
import numpy as np
class GEPP():
 """
 Gaussian elimination with partial pivoting.
```

```
input: A is an n x n numpy matrix
b is an n x 1 numpy array
output: x is the solution of Ax=b
with the entries permuted in
accordance with the pivoting
done by the algorithm
post-condition: A and b have been modified.
:return
"""
def __init__(self, A, b, doPricing=True):
#super(GEPP, self).__init__()
self.A = A # input: A is an n x n numpy matrix
self.b = b # b is an n x 1 numpy array
self.doPricing = doPricing
self.n = None # n is the length of A
self.x = None # x is the solution of Ax=b
self._validate_input() # method that validates input
self._elimination() # method that conducts elimination
self._backsub() # method that conducts back-substitution
def _validate_input(self):
self.n = len(self.A)
if self.b.size != self.n:

raise ValueError("Invalid argument: incompatible sizes between" +
"A & b.", self.b.size, self.n)
def _elimination(self):
"""
k represents the current pivot row. Since GE traverses the matrix in the
upper right triangle, we also use k for indicating the k-th diagonal
column index.
:return
"""
# Elimination
for k in range(self.n - 1):
if self.doPricing:
# Pivot
maxindex = abs(self.A[k:, k]).argmax() + k
if self.A[maxindex, k] == 0:
```

```
 raise ValueError("Matrix is singular.")
 # Swap
 if maxindex != k:
 self.A[[k, maxindex]] = self.A[[maxindex, k]]
 self.b[[k, maxindex]] = self.b[[maxindex, k]]
 else:
 if self.A[k, k] == 0:
 raise ValueError("Pivot element is zero. Try setting doPricing to True.")
 # Eliminate
 for row in range(k + 1, self.n):
 multiplier = self.A[row, k] / self.A[k, k]
 self.A[row, k:] = self.A[row, k:] - multiplier * self.A[k, k:]
 self.b[row] = self.b[row] - multiplier * self.b[k]
 def _backsub(self):
 # Back Substitution
 self.x = np.zeros(self.n)
 for k in range(self.n - 1, -1, -1):
 self.x[k] = (self.b[k] - np.dot(self.A[k, k + 1:], self.x[k + 1:])) / self.A[k, k]
def main():
 A = np.array([[1., -1., 1., -1.],
 [1., 0., 0., 0.],
 [1., 1., 1., 1.],
 [1., 2., 4., 8.]])
 b = np.array([[14.],
 [4.],
 [2.],
 [2.]])
  GaussElimPiv = GEPP(np.copy(A), np.copy(b), doPricing=False)
 print(GaussElimPiv.x)
 print(GaussElimPiv.A)
 print(GaussElimPiv.b)
 GaussElimPiv = GEPP(A, b)
 print(GaussElimPiv.x)
if __name__ == "__main__":
 main()
----- [The test code] ----
import numpy as np
from GEPP import *
```

```
print("Create matrix")
A = np.array([ [1,2,3],
[3,4,5],
[3,2,1]])
y = np.array([ [1],
[3],
[4]])
Gause = GEPP(np.copy(A), np.copy(y), doPricing=False)
print(Gause.x)
print(Gause.A)
print(Gause.b)
```

Generates the output:

```
self.x[k] = (self.b[k] - np.dot(self.A[k, k + 1:], self.x[k + 1:])) / self.A[k, k]
[ nan -inf inf] ## in case of zero divide error can be eliminated
[[ 1 2 3]
[ 0 -2 -4]
[ 0 0 0]]
[[1]
[0]
[1]]
```

Gaussian Elimination with partial Pivoting

IV) Using the code determine the actual time taken for Gaussian elimination with and without partial pivoting for the 10 cases and compare this with the theoretical time. Present this data in a tabular form. Assuming $T_1(n)$ is the actual time calculated for an $n \times n$ matrix, plot a graphs of $\log(T_1(n))$ vs $\log(n)$ (for the 10 cases) and fit a straight line to the observed curve and report the slope of the lines. Ensure that separate graphs are to be plotted for the method with and without partial pivoting. (0.5 + 1 + 1)

Deliverable(s):

(a) A table

(a) A table

S. No.	n	Actual time with pivoting	Actual time without pivoting	Theoretical time

(b) two log log plots and the slope in both the cases

n	10	100	1000	10000	Theotrical time
Gaussian Elimination without Pivot	3.28	25.83	250.8	2501	$T\left(\frac{8n^4}{3}+12n^3+\frac{4n^2}{3}\right)$
					$CT\mid_{BS}=T\left(4n^2+12n\right)$

Q2) Implementing Gauss Seidel and Gauss Jacobi Methods
(i) Write a function to check whether a given square matrix is diagonally dominant or not. If not, the function should indicate if the matrix can be made diagonally dominant by interchanging the rows? Code to be written and submitted. (1)
Deliverable(s): The code

```
.    # Create a function to say checkdiagnolydominant_matx() which takes the given matrix
2.    # and the number of rows of the given matrix as the arguments and returns true or false
3.5.    def checkdiagnolydominant_matx(mtrx, mtrxrows):
6.        # Inside the function, Loop till the given number of rows using the For loop.
7.        for n in range(0, mtrxrows):
8.           # Take a variable to say rslt_summ and initialize its value to 0.
9.            rslt_summ = 0
10.           # Inside the For loop, Iterate till the given number of rows using another
11.           # Nested For loop(Inner For loop).
12.
13.           for m in range(0, mtrxrows):
14.             # Add the absolute of mtrx[n][m] to the above-initialized rslt_summ and store
15.             # it in  the same variable.
16.            # Check if the abs(mtrx[n][n]) (diagonal element) is less than the rslt_summ
22.           # (Which is the #sum of non diagonal elements) using the if conditional statement.
23.           if (abs(mtrx[n][n]) < rslt_summ):
24.             # If it is true, then return False.
25.               return False
26.       # Return True.
27.       return True
28.
30.   # Give the number of rows of the matrix as user input using the int(input()) function
31.   # and store it in a variable.
32.   mtrxrows = int(input('Enter some random number of rows of the matrix = '))
33.   # Give the number of columns of the matrix as user input using the int(input()) function
34.   # and store it in another variable.
35.   mtrxcols = int(input('Enter some random number of columns of the matrix = '))
36.   # Take a list and initialize it with an empty value using [] or list() to say gvnmatrix.
37.   mtrx = []
38.   # Loop till the given number of rows using the For loop
39.   for n in range(mtrxrows):
40.       # Inside the For loop, Give all the row elements of the given Matrix as a list using
```

```
        # the list(),map(),int(),split() functions and store it in a variable.
        l = list(map(int, input(
            'Enter {'+str(mtrxcols)+'} elements of row {'+str(n+1)+'} separated by spaces = ').split()))
        # Add the above row elements list to gvnmatrix using the append() function.
        mtrx.append(l)
    if((checkdiagnolydominant_matx(mtrx, mtrxrows))):
      # If it is true, print "Yes, the given matrix is a diagonally dominant matrix".
        print("Yes, the given matrix is a diagonally dominant matrix")
    else:
      # Else print "No, the given matrix is not a diagonally dominant matrix".
        print("No, the given matrix is not a diagonally dominant matrix")
```

Enter some random number of rows of the matrix = 3

Enter some random number of columns of the matrix = 3

Enter {3} elements of row {1} separated by spaces = 1 3 5

Enter {3} elements of row {2} separated by spaces = 2 4 6

Enter {3} elements of row {3} separated by spaces = 7 8 9

No, the given matrix is not a diagonally dominant matrix

(ii) Write a function to generate Gauss Seidel iteration for a given square matrix. The function should also return the values of 1,∞ and Frobenius norms of the iteration matrix. Generate a random 4 × 4 matrix.
Report the iteration matrix and its norm values returned by the function along with the input matrix. (1)
Deliverable(s): The input matrix, iteration matrix and the three norms obtained

```
    # Defining our function as seidel which takes 3 arguments
    # as A matrix, Solution and B matrix


    def seidel(a, x ,b):
        #Finding length of a(3)
        n = len(a)
         for loop for 3 times as to calculate x, y , z
for j in range(0, n):
            # to calculate respective xi, yi, zi
            for i in range(0, n):
                if(j != i):
                    d-=a[j][i] * x[i]
            # updating the value of our solution
            x[j] = d / a[j][j]
    # int(input())input as number of variable to be solved
    n = 3
    a = []
```

```
b = []
# initial solution depending on n(here n=3)
x = [0, 0, 0]
a = [[4, 1, 2],[3, 5, 1],[1, 1, 3]]
b = [4,7,3]
print(x)


#loop run for m times depending on m the error value
for i in range(0, 25):
    x = seidel(a, x, b)
    #print each time the updated solution
    print(x)
```

An example for the matrix version

A linear system shown as Ax=b is given by

Output:

$$x^{(1)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.5000 \\ -0.8636 \end{bmatrix}.$$

$$x^{(2)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.5000 \\ -0.8636 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8494 \\ -0.6413 \end{bmatrix}.$$

$$x^{(3)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8494 \\ -0.6413 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8077 \\ -0.6678 \end{bmatrix}.$$

$$x^{(4)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8077 \\ -0.6678 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8127 \\ -0.6646 \end{bmatrix}.$$

$$x^{(5)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8127 \\ -0.6646 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8121 \\ -0.6650 \end{bmatrix}.$$

$$x^{(6)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8121 \\ -0.6650 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8122 \\ -0.6650 \end{bmatrix}.$$

$$x^{(7)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8122 \\ -0.6650 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8122 \\ -0.6650 \end{bmatrix}.$$

(iii) Repeat part (ii) for the Gauss Jacobi iteration. (1)
Deliverable(s): The input matrix, iteration matrix and the three norms Obtained

```
# Defining equations to be solved

# in diagonally dominant form

f1 = lambda x,y,z: (17-y+2*z)/20
f2 = lambda x,y,z: (-18-3*x+z)/20
f3 = lambda x,y,z: (25-2*x+3*y)/20


# Initial setup
x0 = 0

y0 = 0 z0
= 0 count
= 1


# Reading tolerable error

e = float(input('Enter tolerable error: '))


# Implementation of Jacobi Iteration

print('\nCount\tx\ty\tz\n')


condition = True


while condition:

    x1 = f1(x0,y0,z0)
    y1 = f2(x0,y0,z0)
    z1 = f3(x0,y0,z0)

    print('%d\t%0.4f\t%0.4f\t%0.4f\n' %(count, x1,y1,z1))

    e1 = abs(x0-x1);
    e2 = abs(y0-y1);
    e3 = abs(z0-z1);


    count += 1
    x0 = x1

    y0 = y1

    z0 = z1


    condition = e1>e and e2>e and e3>e
```

```
print('\nSolution: x=%0.3f, y=%0.3f and z = %0.3f\n'% (x1,y1,z1))
```

Python Program Output:
Jacobi Method

```
Enter tolerable error: 0.00001

Count   x       y       z

1       0.8500  -0.9000 1.2500

2       1.0200  -0.9650 1.0300

3       1.0012  -1.0015 1.0032

4       1.0004  -1.0000 0.9996

5       1.0000  -1.0001 1.0000

6       1.0000  -1.0000 1.0000

7       1.0000  -1.0000 1.0000


Solution: x=1.000, y=-1.000 and z = 1.000
```

(iv) Write a function that perform Gauss Seidel iterations. Generate a random 4×4 matrix A and a suitable random vector $b \in R^4$ and report the results of passing this matrix to the functions written above. Write down the first ten iterates of Gauss Seidel algorithm. Does it converge? Generate a plot of $\|x_{k+1} - x_k\|_2$ for the first 10 iterations. Take a screenshot and paste it in the assignment document. (1)

Deliverable(s): The input matrix and the vector, the 10 successive iterates and the plot

(v) Repeat part (iv) for the Gauss Jacobi method. (1)
Deliverable(s): The input matrix and the vector, the 10 successive iterates and the plot

Jacobi method described in III) is having the output as below.

```
  Enter tolerable error: 0.00001


  Count   x       y       z


  1       0.8500  -0.9000 1.2500


  2       1.0200  -0.9650 1.0300


  3       1.0012  -1.0015 1.0032


  4       1.0004  -1.0000 0.9996


  5       1.0000  -1.0001 1.0000


  6       1.0000  -1.0000 1.0000


  7       1.0000  -1.0000 1.0000



Solution: x=1.000, y=-1.000 and z = 1.000
```