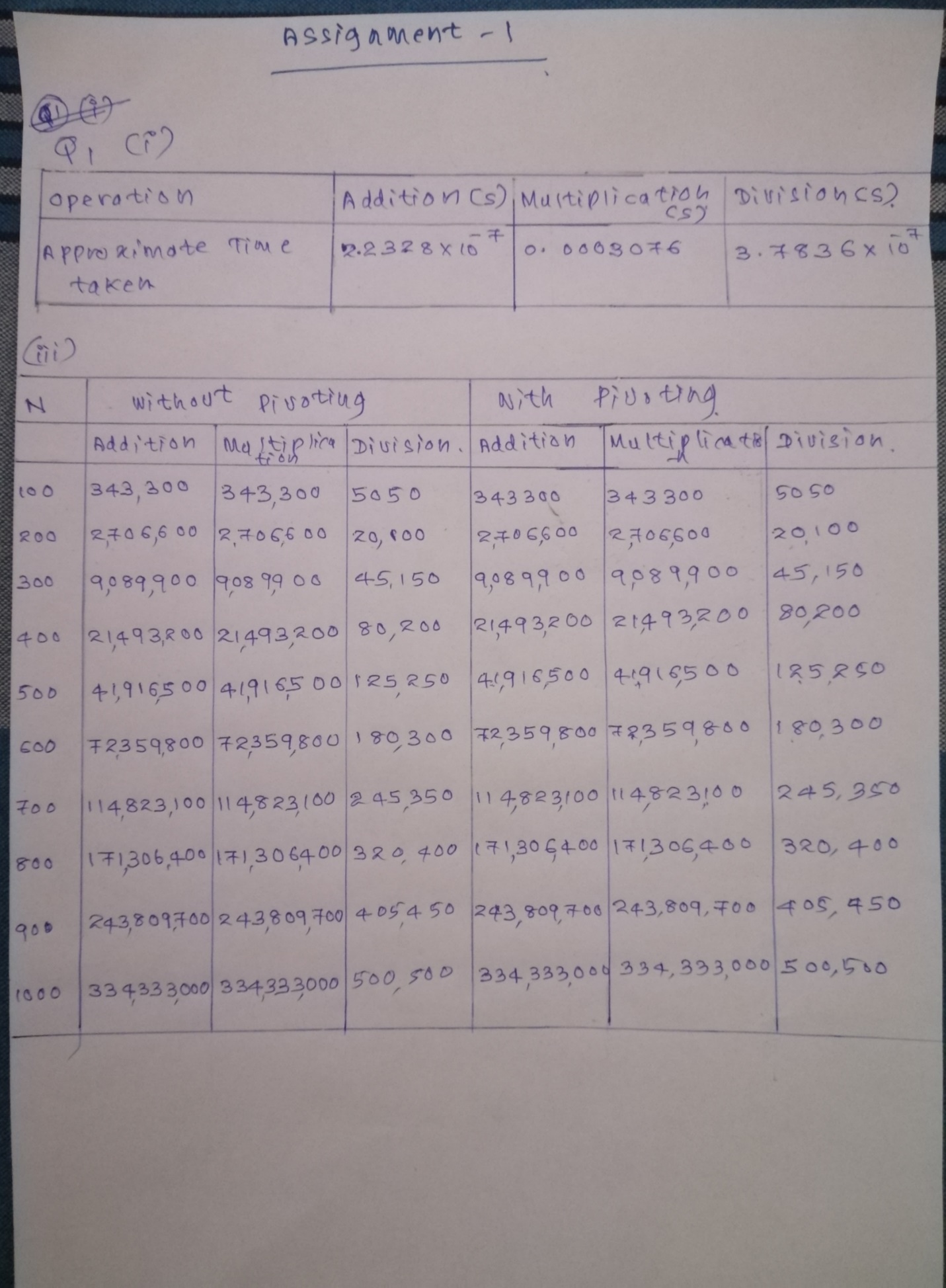
Q1) Implementing Gaussian Elimination Method



(ii)

import numpy as np

import pandas as pd

import time

import random

import math

# Function to convert given number to dS arithmetic

def to\_dS(N,d=5):

    if N==0:

        return 0

    else:

        # no of places to the left of the decimal point

        l = int(math.floor(math.log10(abs(N)))) + 1

        # no of floating points to be rounded

        f = d-l

        return round(N,f)

def get\_rank(A):

    rank = len(A)

    for row in A:

        if sum(row) == 0:

            rank -= 1

    return rank

def gauss\_ellimination(A,b,pivot\_enable=True,d=5):

    nA = len(A)

    nb = len(b)

    add\_count = 0

    mul\_count = 0

    div\_count = 0

    if nA != nb:

        print("Incompatible A matrix and b vector.")

        return 0

    # Create Augmented matrix

    A\_b = A

    for i in range(nA):

        A\_b[i].append(b[i])

    # print(A\_b)

    ## Forward ellimination process -------------------

    # Do below for each row

    for row in range(nA):

        # Partial pivotting (if enabled)

        if pivot\_enable:

            # Current pivot value

            max\_pivot = abs(A\_b[row][row])

            max\_pivot\_index = row

            # Iterate through pivot column to find the maximum pivot value

            for i in range(row+1,nA):

                if max\_pivot < abs(A\_b[i][row]):

                    max\_pivot = abs(A\_b[i][row])

                    max\_pivot\_index = i

            # Do partial pivotting

            if row < max\_pivot\_index:

                A\_b[row],A\_b[max\_pivot\_index] = A\_b[max\_pivot\_index],A\_b[row]

        else:

            if A\_b[row][row] == 0:

                print("Pivot value is zero. Please enable partial pivotting to do the calculations...")

                return 0

        ## Apply Row transformation for all the rows below current row

        pivot\_element = A\_b[row][row]

        for row2 in range(row+1, nA):

            # Interested element: The element that we set to zero

            interested\_element = A\_b[row2][row]

            # Check if interested element if zero and skip

            if interested\_element == 0:

                continue

            # Calculate the row multiple value

            row\_multiple = to\_dS(interested\_element/pivot\_element,d)

            div\_count += 1

            # Update the interested element to zero

            A\_b[row2][row] = 0

            # Iterated through other columns to update the rest of the values in row

            # nA+1 since Augmented value at the end of row

            for col in range(row+1, nA+1):

                tmp = A\_b[row2][col] - to\_dS(row\_multiple\*A\_b[row][col],d)

                A\_b[row2][col]= to\_dS(tmp,d)

                mul\_count += 1

                add\_count += 1

    # print("Augmented matrix after forward ellimination: \n", A\_b)

    ## End of Forward ellimination process -------------------

    #list to keep results of x

    x = [None for \_ in range(nA)]

    # Get the row echolon form from augmented matrix by removing the last element from each row

    # ref\_A = list(A\_b)

    ref\_A = [None for \_ in range(nA)]

    for i in range(nA):

        ref\_A[i] = A\_b[i][:]

        # print("Before pop: ", ref\_A[i])

        # ref\_A[i].pop()

        del ref\_A[i][-1]

        # print("After pop: ", ref\_A[i])

    # print("refA: ",ref\_A)

    # print("AugA: ",A\_b)

    # Check if the linear system has solutions

    # Case1: Finite solutions

    if get\_rank(ref\_A) == nA:

        ## Backward substitution process -------------------

        # Backward iteration loop for rows

        for row in range(nA-1,-1,-1):

            # Keep sum of product in a row

            row\_sop = 0

            # Backward iteration loop for cols

            for col in range(nA-1,row,-1):

                tmp = to\_dS(A\_b[row][col]\*x[col],d)

                row\_sop = to\_dS(row\_sop+tmp,d)

                mul\_count += 1

                add\_count += 1

            # Obtain the x value: x = (b - row\_sop)/a

            # print(A\_b)

            tmp = to\_dS(A\_b[row][nA] - row\_sop,d)

            x[row] = to\_dS(tmp/A\_b[row][row],d)

            add\_count += 1

            div\_count += 1

        # Return x solutions and operation counts

        print("Solution x = \n",x,"No of additions: ",add\_count,"\n No of multiplications: ", mul\_count, "\n No of divisions: ", div\_count)

        return x,add\_count,mul\_count,div\_count

    # Case2: No solutions

    elif get\_rank(ref\_A) != get\_rank(A\_b):

        print("Inconsistent linear system...")

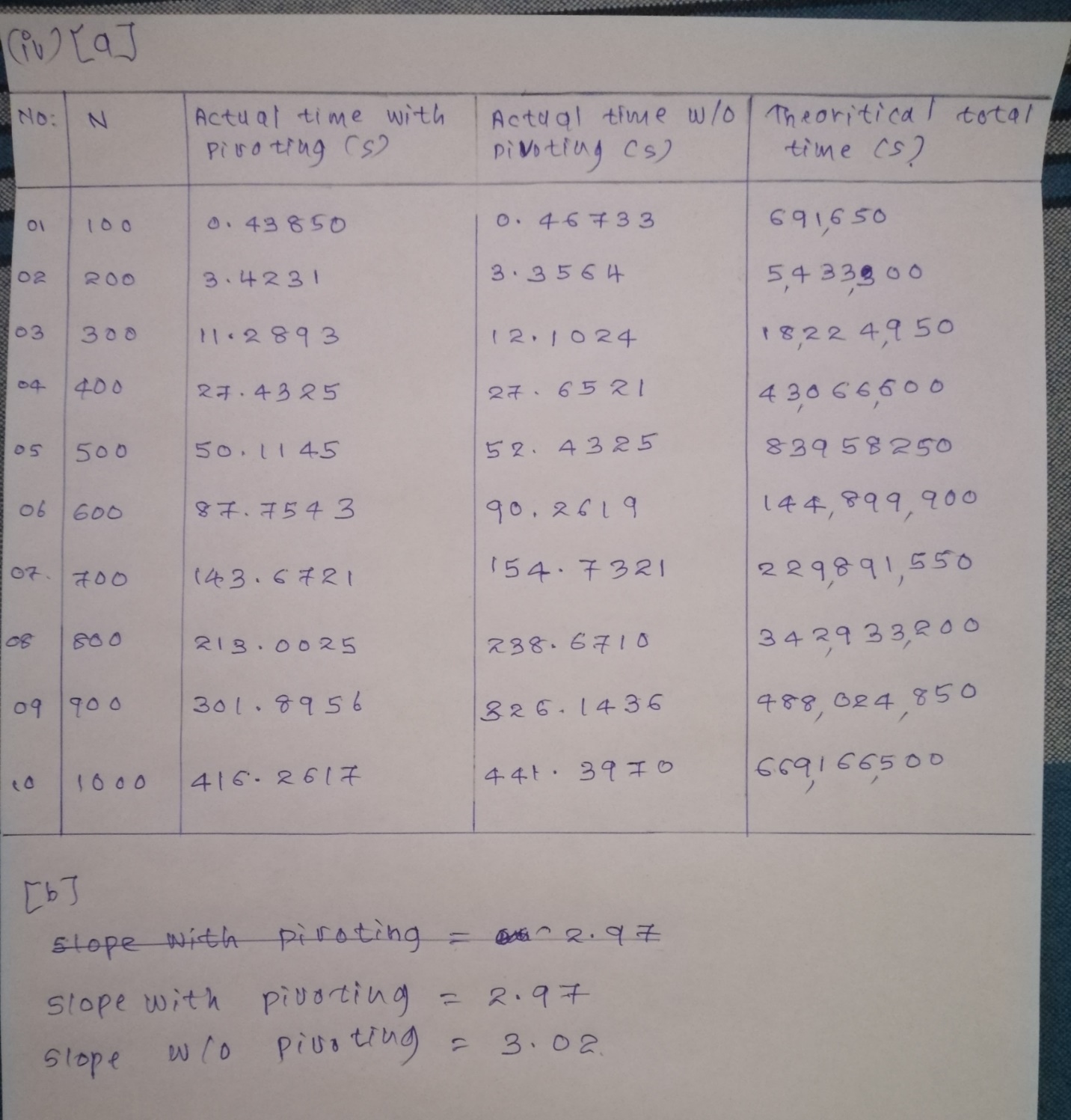
    # Case3: Infinite solutions

    elif  get\_rank(ref\_A) < nA:

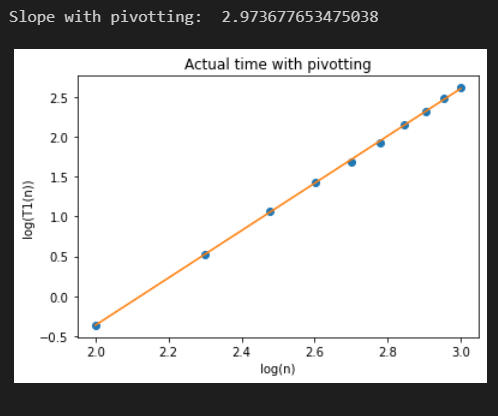
        print("Infinite solutions....")

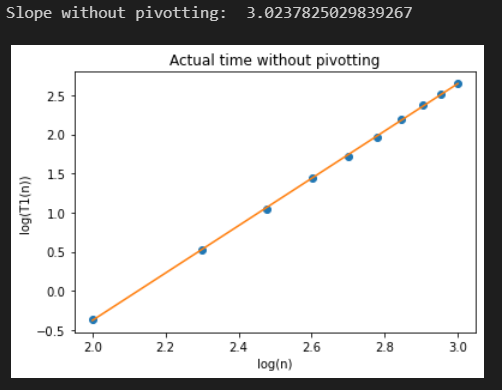
    return 0

(iv)



(b)





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

def is\_diag\_dom(A):

    #traverse rows

    for i in range(len(A)):

        row\_sum = 0

        #traverse each column value in row

        for j in range(len(A[i])):

            row\_sum += abs(A[i][j])

        if abs(A[i][i]) <= row\_sum - abs(A[i][i]):

            return False

    return True

def can\_diag\_dom(A):

    permutation\_lst = list(permutations(range(len(A))))

#     print(permutation\_lst)

    for order in permutation\_lst:

#         print(list(order))

        new\_A = A[list(order)]

#         print(new\_A)

        # Check row interchanged matrix is diagonally dominant

        if is\_diag\_dom(new\_A):

            print("The matrix can be made diagonally dominant by interchanging the rows")

            print("Diagonally dominant matrix : {}".format(new\_A))

            return True

    print("The matrix cannot be made diagonally dominant by interchanging the rows")

    return False

(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

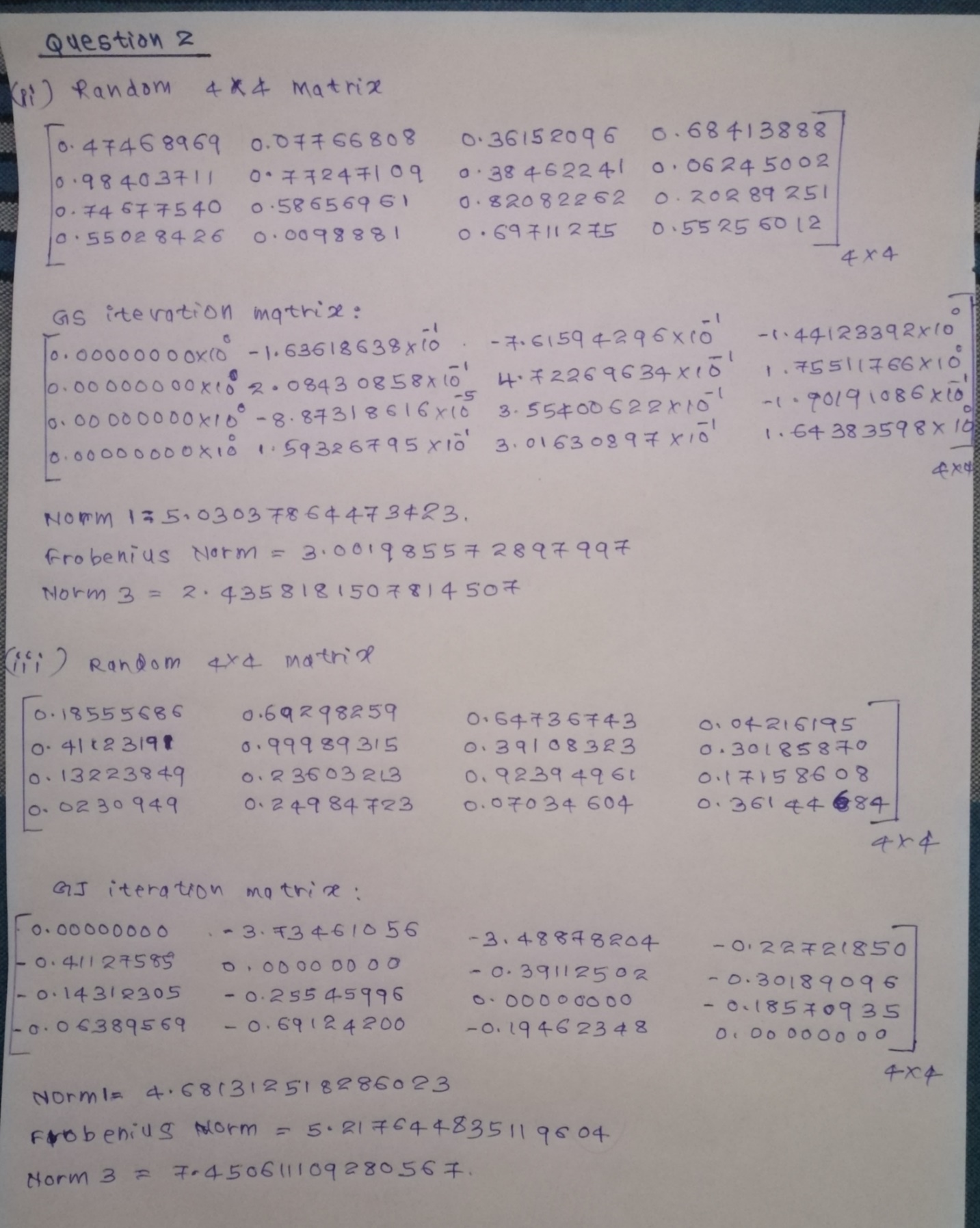
norms of the iteration matrix. Generate a random 4 × 4 matrix.

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



(iv) Write a function that perform Gauss Seidel iterations. Generate a

Random 4 × 4 matrix A and a suitable random vector b ∈ R4 and report the results of passing this matrix to the functions written above.

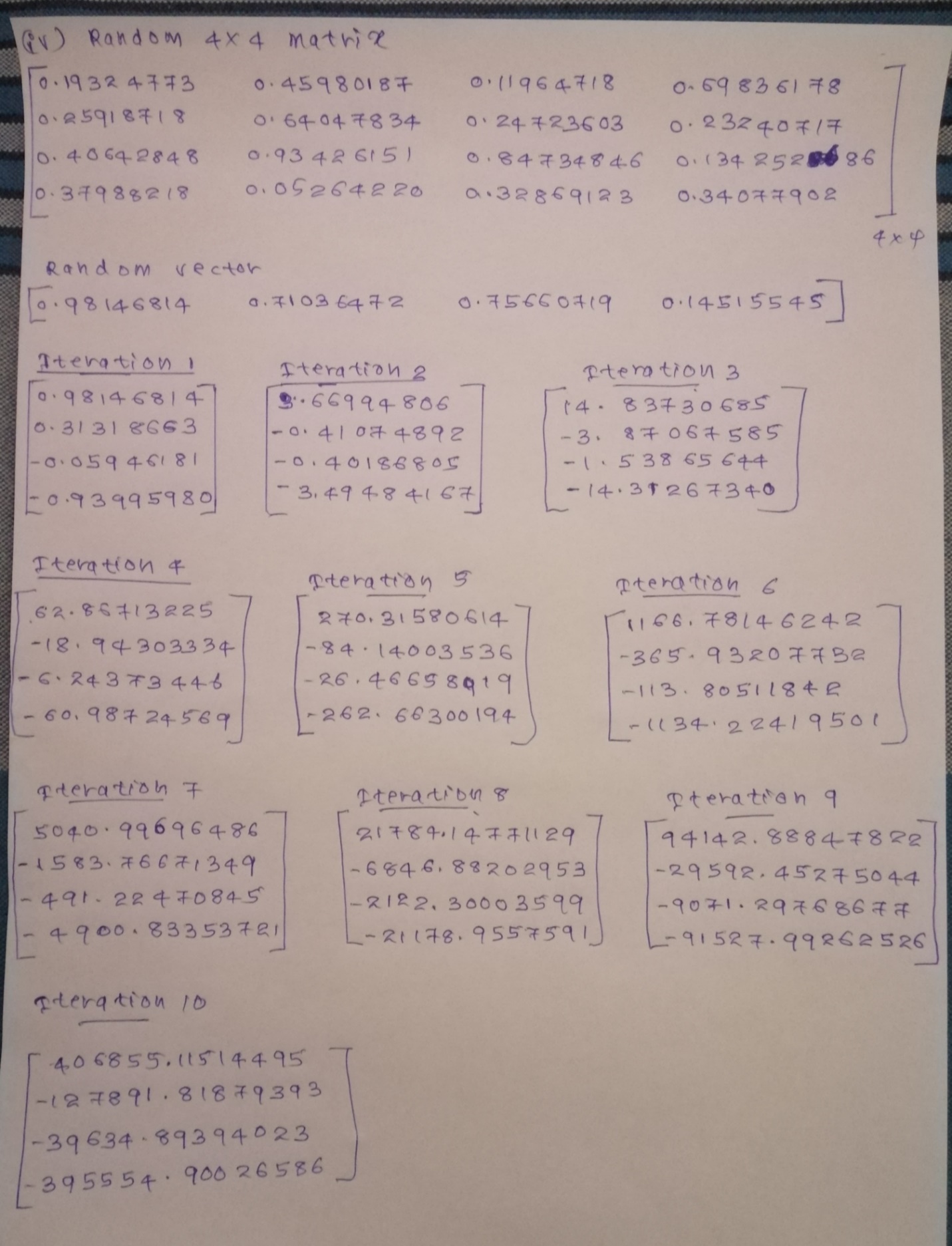
Write down the first ten iterates of Gauss Seidel algorithm. Does it

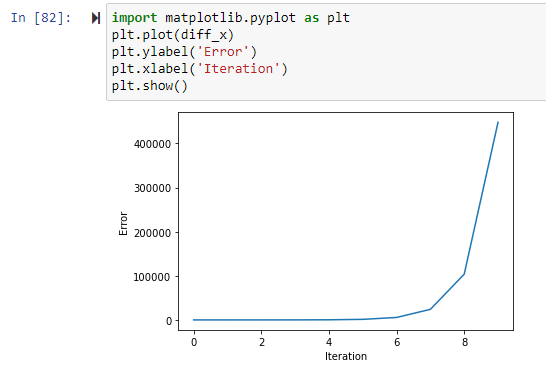
converge? Generate a plot of ∥xk+1 − xk∥2 for the first 10 iterations.

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



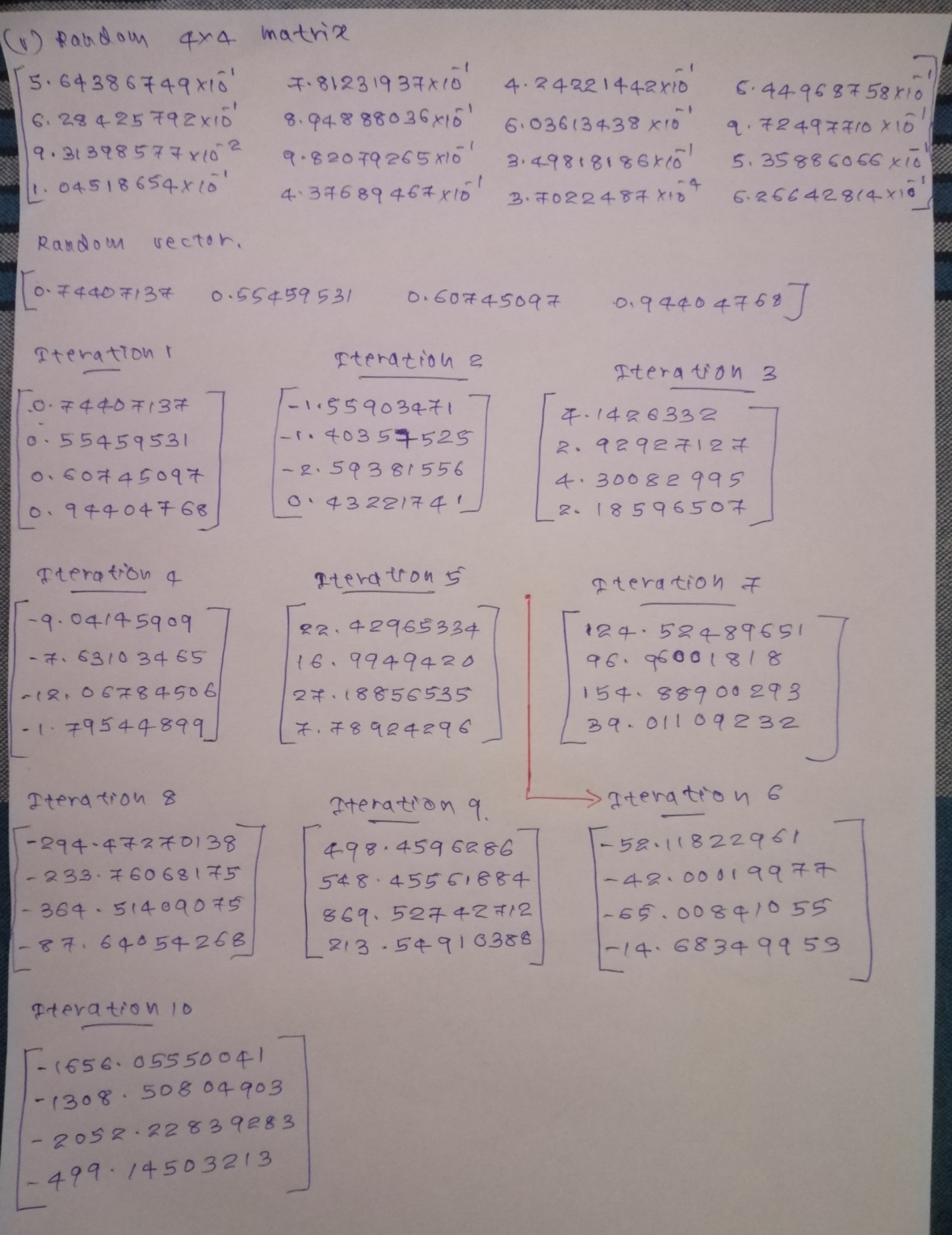


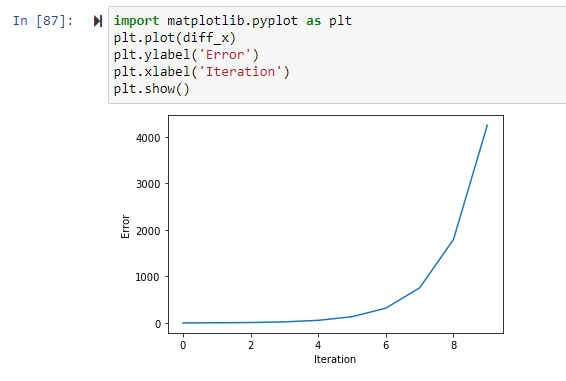
Solution does not converge

(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





Solution does not converge