Q1) Implementing Gaussian Elimination Method

(i) Find the approximate time your computer takes for a single addition by adding first 106 positive integers using a for loop and dividing the time taken by 106. 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

Ans:

S.N.	Operation	Time Taken (sec.) (in seconds)
1.	Addition	1.5733×10^{-7}
2.	Multiplication	0.00057723
3.	Division	1.89120×10^{-7}

(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 the 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

Ans:

Q1_ii

December 7, 2021

[1]:
```
import sys
from math import floor, log10
```

[2]:
```
# With operation count & significant airthmatic
def gauss_elimination_without_pivoting(a, b, d):
    '''
    Gaussian elimination without pivoting.
    param: a is an n x n matrix
           b is an n x 1 vector
           d is significant digit
    return: x is the solution of Ax=b.
            row echelon form of a
            row echelon form of b
            addition, subtraction, multiply, divide operations count
    '''
    n = len(a)
    x = [0 for i in range(n)]
    add_ops_count, mul_ops_count, div_ops_count = 0, 0, 0

    # Significant digit conversion
    def tidy(x, sig):
        y = abs(x)
        if y <= sys.float_info.min:
            return 0.0000
        return round(x, sig-int(floor(log10(y)))-1)

    # Apply forward elimination
    for i in range(n-1):
        # Check for leading element as non-zero
        if a[i][i] == 0:
            sys.exit("Triangle leading element zero detected, Division by␣
↪Error!")
        for j in range(i+1, n):
            multiplier = tidy(a[j][i] / a[i][i], d)
            div_ops_count = div_ops_count + 1
            # Apply row operation on matrix a
            a[j][i] = tidy(0, d)
```

```
            for col in range(i+1, n):
                a[j][col] = tidy(a[j][col] - tidy(multiplier * a[i][col], d), d)
                add_ops_count = add_ops_count + 1
                mul_ops_count = mul_ops_count + 1
            # Apply row operation on vector b
            b[j] = tidy(b[j] - tidy(multiplier * b[i], d), d)
            add_ops_count = add_ops_count + 1
            mul_ops_count = mul_ops_count + 1

    # Apply back substitution
    # Calculate rank of a to check if unique solution is present
    rank = 0
    zero_rows_idx = []
    for r_idx in range(n):
        if any(a[r_idx]):
            rank = rank + 1
        else:
            zero_rows_idx.append(r_idx)

    if rank == n:
        # System has one unique solution
        x[n-1] = tidy(b[n-1] / a[n-1][n-1], d)
        div_ops_count = div_ops_count + 1
        for i in range(n-2, -1, -1):
            x[i] = b[i]
            for j in range(i+1, n):
                x[i] = tidy(x[i] - tidy(a[i][j]*x[j], d), d)
                add_ops_count = add_ops_count + 1
                mul_ops_count = mul_ops_count + 1
            x[i] = tidy(x[i] / a[i][i], d)
            div_ops_count = div_ops_count + 1
        return {"ref_a": a, "ref_b": b, "solution": x, "add_ops_count":␣
↪add_ops_count,
                "mul_ops_count": mul_ops_count, "div_ops_count": div_ops_count}
    else:
        # r < n, check if r+1, r+2, ... r+n rows in b has any non zero value
        for z_idx in zero_rows_idx:
            if b[z_idx] != 0:
                sys.exit("Incosistent system, there is no solution!")
        sys.exit("Consistent system, there may be infinitly many solutions!")
```

[3]:
```
# Test1
a = [[3.0,  2.0, -4.0],
     [2.0,  3.0,  3.0],
     [5.0, -3.0,  1.0]]
b = [3.0, 15.0, 14.0]
res = gauss_elimination_without_pivoting(a, b, 5)
```

```
print(f"REF of A: {res['ref_a']}\nREF of b: {res['ref_b']}\nSolution x:␣
↪{res['solution']}")
print(f"No. of Addition Performed: {res['add_ops_count']}\nNo. of␣
↪Multiplication Performed:: {res['mul_ops_count']}\nNo. of Division Performed:
↪: {res['div_ops_count']}")
```

```
REF of A: [[3.0, 2.0, -4.0], [0.0, 1.6667, 5.6667], [0.0, 0.0, 29.2]]
REF of b: [3.0, 13.0, 58.4]
Solution x: [2.9999, 1.0002, 2.0]
No. of Addition Performed: 11
No. of Multiplication Performed:: 11
No. of Division Performed:: 6
```

[4]:

```
# With operation count & significant airthmatic & partial pivoting
def gauss_elimination_with_pivoting(a, b, d):
    '''
    Gaussian elimination with partial pivoting.
    param: a is an n x n matrix
           b is an n x 1 vector
           d is significant digit
    return: x is the solution of Ax=b.
            row echelon form of a
            row echelon form of b
            addition, subtraction, multiply, divide operations count
    '''
    n =  len(a)
    x = [0 for i in range(n)]
    add_ops_count, mul_ops_count, div_ops_count = 0, 0, 0

    # Significant digit conversion
    def tidy(x, sig):
        y = abs(x)
        if y <= sys.float_info.min:
            return 0.0000
        return round(x, sig-int(floor(log10(y)))-1)

    # Apply forward elimination
    for i in range(n):
        max_idx = i
        max_val = a[max_idx][i]
        # Find the largest pivot element including i
        for j in range(i+1, n):
            if a[j][i] != 0.0 and abs(a[j][i]) > max_val:
                max_val, max_idx = a[j][i], j
```

```
        # Check if diagonal element is zero, which will cause divide by zero error
        if a[i][max_idx] == 0:
            if b[i] != 0:
                sys.exit("Singular matrix, Inconsistent system - no solutions!")
            else:
                sys.exit("Singular matrix, consistent system - may have infinitly many solutions!")

        # Swap the current row with larger value row
        if i != max_idx:
            for z in range(n):
                temp_a, temp_b = a[i][z], b[i]
                a[i][z], b[i] = a[max_idx][z], b[max_idx]
                a[max_idx][z], b[max_idx] = temp_a, temp_b
        for row in range(i+1, n):
            multiplier = tidy(a[row][i] / a[i][i], d)
            div_ops_count = div_ops_count + 1
            a[row][i] = tidy(0, d)
            for col in range(i+1, n):
                a[row][col] = tidy(a[row][col] - tidy(multiplier * a[i][col], d), d)
                mul_ops_count = mul_ops_count + 1
                add_ops_count = add_ops_count + 1
            b[row] = tidy(b[row] - tidy(multiplier * b[i], d), d)
            mul_ops_count = mul_ops_count + 1
            add_ops_count = add_ops_count + 1

    # Apply back substitution
    # Calculate rank of a to check if unique solution is present
    rank = 0
    zero_rows_idx = []
    for r_idx in range(n):
        if any(a[r_idx]):
            rank = rank + 1
        else:
            zero_rows_idx.append(r_idx)

    if rank == n:
        # System has one unique solution
        x[n-1] = tidy(b[n-1] / a[n-1][n-1], d)
        div_ops_count = div_ops_count + 1
        for i in range(n-2, -1, -1):
            x[i] = b[i]
            for j in range(i+1, n):
                x[i] = tidy(x[i] - tidy(a[i][j]*x[j], d), d)
                add_ops_count = add_ops_count + 1
```

```
                mul_ops_count = mul_ops_count + 1
            x[i] = tidy(x[i] / a[i][i], d)
            div_ops_count = div_ops_count + 1
        return {"ref_a": a, "ref_b": b, "solution": x, "add_ops_count":␣
↪add_ops_count,
                "mul_ops_count": mul_ops_count, "div_ops_count": div_ops_count}
    else:
        # r < n, check if r+1, r+2, ... r+n rows in b has any non zero value
        for z_idx in zero_rows_idx:
            if b[z_idx] != 0:
                sys.exit("Incosistent system, there is no solution!")
        sys.exit("Consistent system, there may be infinitly many solutions!")
```

[5]:
```
# Test1
a = [[2.0,  2.0, 1.0],
     [4.0,  2.0, 3.0],
     [1.0, -1.0, 1.0]]
b = [6.0, 4.0, 0.0]
res = gauss_elimination_with_pivoting(a, b, 5)

print(f"REF of A: {res['ref_a']}\nREF of b: {res['ref_b']}\nSolution x:␣
↪{res['solution']}")
print(f"No. of Addition Performed: {res['add_ops_count']}\nNo. of␣
↪Multiplication Performed:: {res['mul_ops_count']}\nNo. of Division Performed:
↪: {res['div_ops_count']}")
```

```
REF of A: [[4.0, 2.0, 3.0], [0.0, -1.5, 0.25], [0.0, 0.0, -0.33333]]
REF of b: [4.0, -1.0, 3.3333]
Solution x: [9.0, -1.0, -10.0]
No. of Addition Performed: 11
No. of Multiplication Performed:: 11
No. of Division Performed:: 6
```

(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 for 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

Ans:

S. No.	n	without Pivoting			with Pivoting		
		Additions	Multiply	Division	Additions	Multiply	Division
1.	100	338250	338250	5050	338250	338250	5050
2.	200	2686500	2686500	20100	2686500	2686500	20100
3.	300	9044750	9044750	45150	9044750	9044750	45150
4.	400	21413000	21413000	80200	21413000	21413000	80200
5.	500	41791250	41791250	125250	41791250	41791250	125250
6.	600	72179500	72179500	180300	72179500	72179500	180300
7.	700	114577750	114577750	245350	114577750	114577750	245350
8.	800	170986000	170986000	320400	170986000	170986000	320400
9.	900	243404250	243404250	405450	243404250	243404250	405450
10	1000	333832500	333832500	500500	333832500	333832500	500500

(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 T1(n) is the actual time calculated for an n ×n matrix, plot a graph of log(T1(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

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

Ans:

S. NO.	n	Actual time with Pivoting (sec.)	Actual time without Pivoting (sec.)	Theoretical time (seconds)
1.	100	0.32371	0.66628	0.10144
2.	200	2.48801	5.2246	0.80269
3.	300	8.4440	17.826	2.6991
4.	400	19.485	40.767	6.3860
5.	500	38.274	79.695	12.459
6.	600	65.868	138.15	21.513
7.	700	109.53	224.90	34.143
8.	800	161.54	334.66	50.945
9.	900	230.87	480.84	72.515
10.	1000	314.32	653.53	99.447

log (T(n)) vs. log(n) Line Slope:

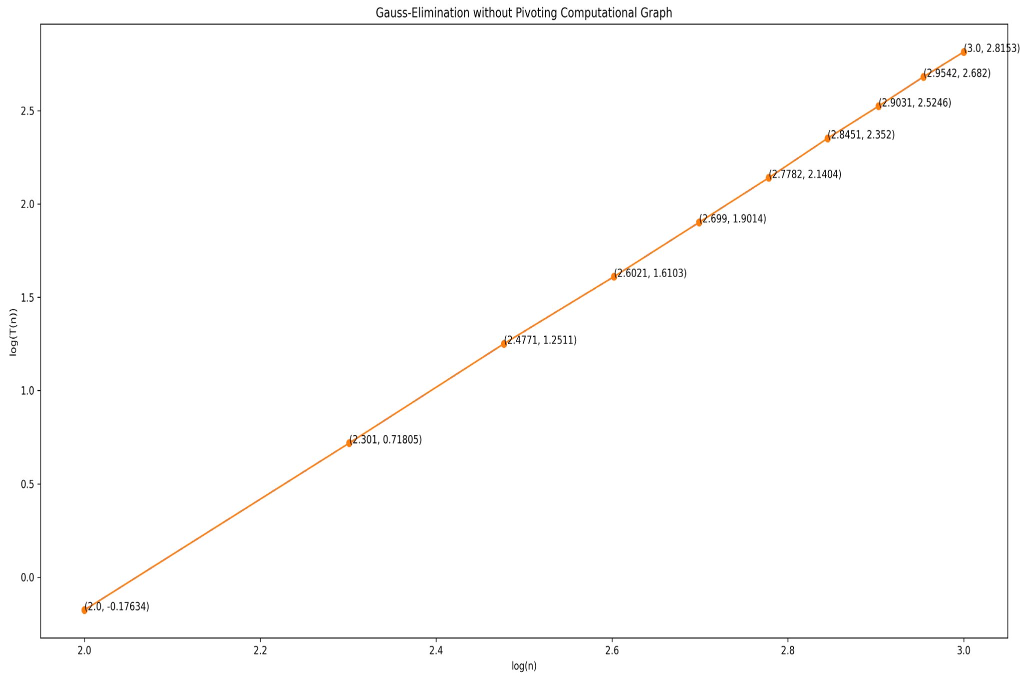
With Pivoting:

Slope = 2.9944

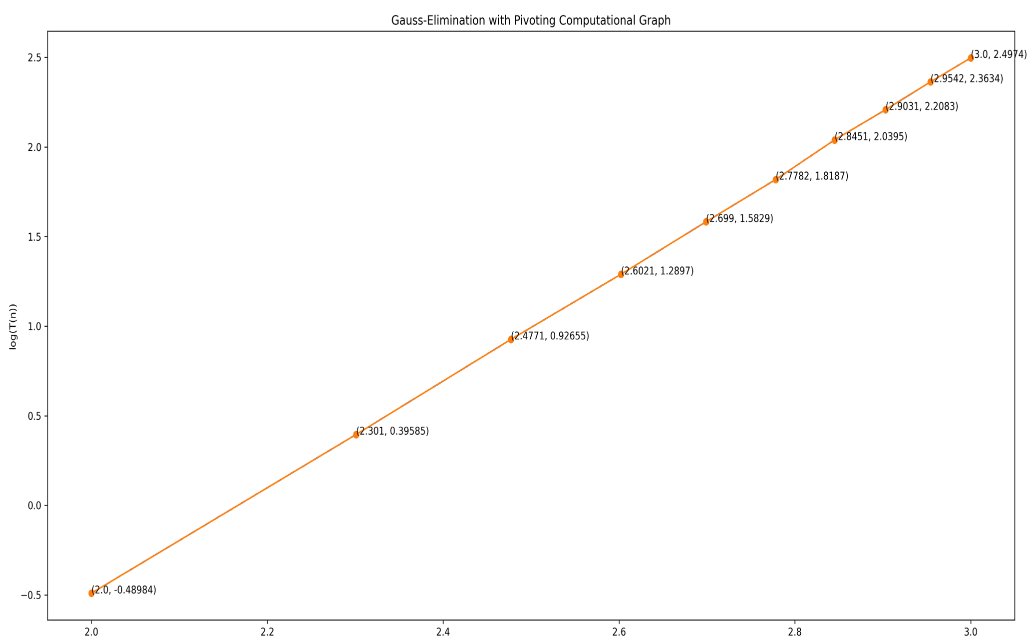
without Pivoting:

Slope = 2.9937

Gauss-Elimination without Pivoting log(T(n)) vs. log(n) plot:



Gauss-Elimination with Pivoting log(T(n)) vs. log(n) plot:



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
```

In [1]:

```
def is_diagonally_dominant(mat, size):
    for row_idx in range(size):
        row_sum = 0
        [row_sum := row_sum + abs(elm) for elm in mat[row_idx]]
        dia_val = abs(mat[row_idx][row_idx])
        if (row_sum - dia_val) > dia_val:
            return False
    return True
```

In [2]:

```
def is_diagonally_dominant_after_row_ops(mat, size):
    max_row_vals, max_row_idxs, row_sums = [], [], []
    for row_idx in range(size):
        max_val, max_idx, row_sum = -1, -1, 0
        for col_idx in range(size):
            elm = abs(mat[row_idx][col_idx])
            row_sum = row_sum + elm
            if elm > max_val:
                max_val, max_idx = elm, col_idx
        max_row_vals.append(max_val)
        max_row_idxs.append(max_idx)
        row_sums.append(row_sum)
    max_row_idxs.sort()
    if all([True if max_row_vals[x] >= (row_sums[x] - max_row_vals[x]) else Fa
        return True
    else:
        return False
```

In [3]:

```
def diagonally_dominant_analysis(mat):
    size = len(mat)
    dd_status = is_diagonally_dominant(mat, size)
    print(f"Is matrix diagonally dominant?: {dd_status}")
    if not dd_status:
        print(f"Is matrix can be made diagonally dominant by interchanging the
```

In [4]:

```
# Test1
m = [[ -8, 1, 45 ], [ 14, 9, 2 ], [ 3, 10, -4 ]]
diagonally_dominant_analysis(m)
```

```
Is matrix diagonally dominant?: False
Is matrix can be made diagonally dominant by interchanging the rows?: True
```

(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 $\times$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

Ans:

Input Matrices:

$$A = \begin{bmatrix} -7 & -8 & 4 & -3 \\ 1 & -5 & 3 & -1 \\ 9 & -8 & -9 & 0 \\ -1 & 7 & 2 & 4 \end{bmatrix}$$

$$b^T = \begin{bmatrix} 8 & 7 & 4 & -1 \end{bmatrix}$$

Norms:

1-Norm = 2.8952

infinity-Norm = 2.1429

frobenius-Norm = 2.2597

Iteration matrix:

$$-(I+L)^{-1} * U = \begin{bmatrix} 0.0 & -1.1429 & 0.57143 & -0.42857 \\ 0.0 & -0.22857 & 0.71429 & -0.28571 \\ 0.0 & -0.93968 & -0.063492 & -0.17460 \\ 0.0 & 0.58413 & -1.0754 & 0.48016 \end{bmatrix}$$

(iii) Repeat part (ii) for the Gauss Jacobi iteration. (1)

Deliverable(s): The input matrix, iteration matrix and the three norms obtained

Ans:

Input matrices:

$$A = \begin{bmatrix} -12 & 21 & 13 & 18 \\ 2 & 30 & -3 & 25 \\ -20 & -3 & 12 & -17 \\ -8 & -8 & -3 & 28 \end{bmatrix}$$

$$b^T = \begin{bmatrix} -14 & 27 & 29 & -5 \end{bmatrix}$$

Norms:

1-Norm: 3.7560

Infinity-Norm: 4.3333

Frobenius-Norm: 3.4952

Iteration matrix:

$$-D^{-1} * (L+U) = \begin{bmatrix} 0.0 & 1.75 & 1.0833 & 1.5 \\ -0.066667 & 0.0 & 0.1 & -0.83333 \\ 1.6667 & 0.25 & 0.0 & 1.4167 \\ 0.28571 & 0.28571 & 0.10714 & 0.0 \end{bmatrix}$$

(iv) Write a function that performs 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 iterations 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

Ans:

Input matrices:

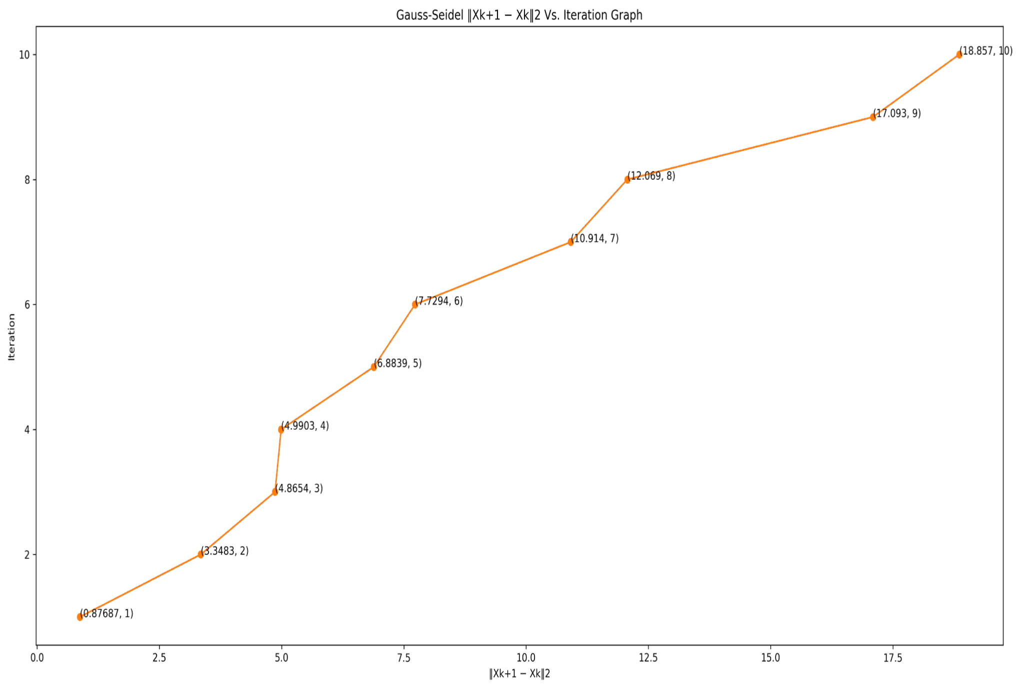
$$A = \begin{bmatrix} 2 & -4 & -4 & 2 \\ 1 & 5 & 2 & 4 \\ -1 & 2 & 2 & -2 \\ 3 & -2 & 5 & 4 \end{bmatrix}$$

$$b^T = \begin{bmatrix} 1 & -3 & -2 & 2 \end{bmatrix}$$

Does it converge? NO

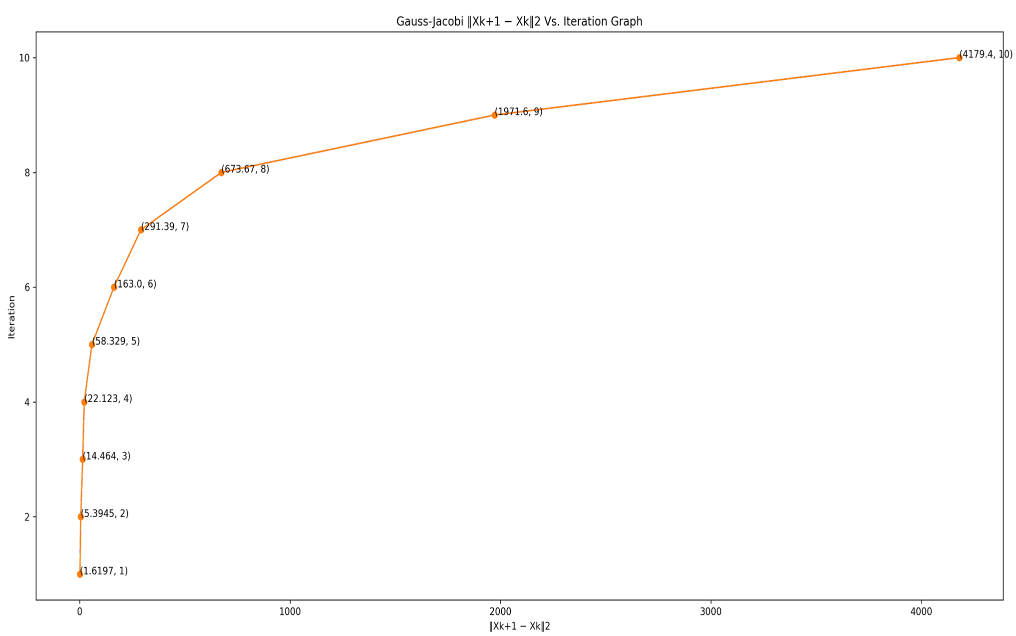
Iterations

1. $[x_1^1, x_2^1, x_3^1, x_4^1] = [0.5, -0.7, -0.05, -0.1625]$
2. $[x_1^2, x_2^2, x_3^2, x_4^2] = [-0.8375, -0.2825, -1.2987, 2.6103]$
3. $[x_1^3, x_2^3, x_3^3, x_4^3] = [-5.2728, -1.1142, 0.088094, 3.7874]$
4. $[x_1^4, x_2^4, x_3^4, x_4^4] = [-5.3396, -2.5972, 2.7148, -0.18749]$
5. $[x_1^5, x_2^5, x_3^5, x_4^5] = [0.9227, -1.7205, 0.99435, -2.2952]$
6. $[x_1^6, x_2^6, x_3^6, x_4^6] = [1.3429, 0.56984, -3.1936, 3.7697]$
7. $[x_1^7, x_2^7, x_3^7, x_4^7] = [-8.5172, -0.63489, -0.854, 7.6379]$
8. $[x_1^8, x_2^8, x_3^8, x_4^8] = [-10.116, -4.3456, 5.9257, -1.4931]$
9. $[x_1^9, x_2^9, x_3^9, x_4^9] = [5.1533, -2.8064, 2.89, -8.3806]$
10. $[x_1^{10}, x_2^{10}, x_3^{10}, x_4^{10}] = [9.0477, 3.139, -7.9918, 5.2784]$



(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



Input Matrices:

$$A = \begin{bmatrix} 6 & 3 & -8 & 3 \\ 7 & -7 & 9 & 5 \\ 2 & -3 & -6 & -6 \\ -5 & -2 & 1 & 1 \end{bmatrix}$$

$$b^T = \begin{bmatrix} -2 & 6 & -8 & 0 \end{bmatrix}$$

Does it converge? No

Iterations:

1. $[x_1^1, x_2^1, x_3^1, x_4^1] = [-0.33333, -0.85714, 1.3333, 0.0]$
2. $[x_1^2, x_2^2, x_3^2, x_4^2] = [1.873, 0.52381, 1.6508, -4.7143]$
3. $[x_1^3, x_2^3, x_3^3, x_4^3] = [3.963, -0.22902, 6.4101, 8.7619]$
4. $[x_1^4, x_2^4, x_3^4, x_4^4] = [3.947, 17.606, -5.9931, 12.947]$
5. $[x_1^5, x_2^5, x_3^5, x_4^5] = [-23.6, 4.6321, -19.101, 60.94]$
6. $[x_1^6, x_2^6, x_3^6, x_4^6] = [-58.587, -5.4872, -69.789, -89.637]$
7. $[x_1^7, x_2^7, x_3^7, x_4^7] = [-45.823, -213.2, 74.185, -234.12]$
8. $[x_1^8, x_2^8, x_3^8, x_4^8] = [322.24, -118.53, 326.78, -729.7]$
9. $[x_1^9, x_2^9, x_3^9, x_4^9] = [859.48, 220.31, 897.71, 1047.4]$
10. $[x_1^{10}, x_2^{10}, x_3^{10}, x_4^{10}] = [562.78, 2760.9, -869.69, 3840.3]$