Problem 1

(2+3+3+2 points) Given the following system of linear equations:

$$5x_1 - 2x_2 + 1x_3 = -1 \tag{1}$$
$$-3x_1 + 9x_2 + x_3 = 2 \tag{2}$$
$$2x_1 - x_2 - 7x_3 = 3 \tag{3}$$

a) Given the starting point $\vec{x}_0 = \vec{0}$ compute the next two iterates $\vec{x}_1$ and $\vec{x}_2$ of the Jacobi iteration.

Answer:

We are provided the following system of equations:

$$5x_1 - 2x_2 + 1x_3 = -1 \tag{1}$$
$$-3x_1 + 9x_2 + x_3 = 2 \tag{2}$$
$$2x_1 - x_2 - 7x_3 = 3 \tag{3}$$

From this, we can derive the following system:

$$\begin{bmatrix} 5 & -2 & 1 \\ -3 & 9 & 1 \\ 2 & -1 & -7 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix}$$

From this matrix system, we first need to extract some important information, as shown below:

$$\boldsymbol{A} = \begin{bmatrix} 5 & -2 & 1 \\ -3 & 9 & 1 \\ 2 & -1 & -7 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix}, \quad \boldsymbol{D} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & -7 \end{bmatrix}$$

$$\boldsymbol{U} = \begin{bmatrix} 0 & -2 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \quad \boldsymbol{L} = \begin{bmatrix} 0 & 0 & 0 \\ -3 & 0 & 0 \\ 2 & -1 & 0 \end{bmatrix}, \quad \vec{x}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

To perform Jacobi Iteration, we use the following formula:

$$\vec{x}_{k+1} = \boldsymbol{D}^{-1} \cdot \left(\vec{b} - (\boldsymbol{L} - \boldsymbol{U})\vec{x}_k\right)$$

The first two iterations are as follows:

$$\vec{x}_1 = \boldsymbol{D^{-1}} \cdot \vec{b} = \begin{bmatrix} \frac{1}{5} & 0 & 0 \\ 0 & \frac{1}{9} & 0 \\ 0 & 0 & \frac{-1}{7} \end{bmatrix} \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -\frac{1}{5} \\ \frac{2}{9} \\ -\frac{3}{7} \end{bmatrix}$$

$$\vec{x}_2 = \begin{bmatrix} \frac{1}{5} & 0 & 0 \\ 0 & \frac{1}{9} & 0 \\ 0 & 0 & \frac{-1}{7} \end{bmatrix} \cdot \left[\begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix} - \left(\begin{bmatrix} 0 & 0 & 0 \\ -3 & 0 & 0 \\ 2 & -1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -2 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \right) \cdot \right] = \begin{bmatrix} 0.0254 \\ 0.203 \\ -0.517 \end{bmatrix}$$

b) Given the starting point $\vec{x}_0 = \vec{0}$, compute the next two iterates $\vec{x}_1$ and $\vec{x}_2$ of the Gauss-Seidel iteration.

Answer:

Gauss-Seidel Formula:

$$\vec{x}_{k+1} = (\boldsymbol{D} + \boldsymbol{L})^{-1} \cdot \left(\vec{b} - \boldsymbol{U} \cdot \vec{x}_k\right)$$

We again start by listing all the parameters required for running the iterations:

$$\boldsymbol{A} = \begin{bmatrix} 5 & -2 & 1 \\ -3 & 9 & 1 \\ 2 & -1 & -7 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix}, \quad \boldsymbol{D} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & -7 \end{bmatrix}$$

$$\boldsymbol{U} = \begin{bmatrix} 0 & -2 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \quad \boldsymbol{L} = \begin{bmatrix} 0 & 0 & 0 \\ -3 & 0 & 0 \\ 2 & -1 & 0 \end{bmatrix}, \quad \vec{x}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The first step in this case is to calculate $(\boldsymbol{D} + \boldsymbol{L})^{-1}$.

$$(\boldsymbol{D} + \boldsymbol{L}) = \begin{bmatrix} 5 & 0 & 0 \\ -3 & 9 & 0 \\ 2 & -1 & -7 \end{bmatrix}$$

We can accomplish this through row operations (Gaussian Elimination) on the following relation:

$$\begin{aligned} &\boldsymbol{X} \cdot \boldsymbol{X}^{-1} = \boldsymbol{I} \\ &\Rightarrow [\boldsymbol{X}|\boldsymbol{I}] \Rightarrow \left[\boldsymbol{I}|\boldsymbol{X}^{-1}\right] \end{aligned}$$

The setup and solution will look as follows:

$$\left[\begin{array}{ccc|ccc} 5 & 0 & 0 & 1 & 0 & 0 \\ -3 & 9 & 0 & 0 & 1 & 0 \\ 2 & -1 & -7 & 0 & 0 & 1 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0.2 & 0 & 0 \\ 0 & 1 & 0 & 0.06667 & 0.1111 & 0 \\ 0 & 0 & 1 & 0.04762 & -0.01587 & -0.14286 \end{array}\right]$$

$$(\boldsymbol{D} + \boldsymbol{L})^{-1} = \begin{bmatrix} 0.2 & 0 & 0 \\ 0.06667 & 0.1111 & 0 \\ 0.04762 & -0.01587 & -0.14286 \end{bmatrix}$$

With this, we are ready to perform the iterations:

$$\vec{x}_1 = (\boldsymbol{D} + \boldsymbol{L})^{-1} \cdot \vec{b} = \begin{bmatrix} 0.2 & 0 & 0 \\ 0.06667 & 0.1111 & 0 \\ 0.04762 & -0.01587 & -0.14286 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -0.2 \\ 0.1555 \\ -0.508 \end{bmatrix}$$

$$\vec{x}_2 = \begin{bmatrix} 0.2 & 0 & 0 \\ 0.06667 & 0.1111 & 0 \\ 0.04762 & -0.01587 & -0.14286 \end{bmatrix} \cdot \left[\begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix} - \begin{bmatrix} 0 & -2 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} -0.2 \\ 0.1555 \\ -0.508 \end{bmatrix}\right] = \begin{bmatrix} -0.036192 \\ 0.26659 \\ -0.476998 \end{bmatrix}$$

c) Given the starting point $\vec{x}_0 = \vec{0}$, compute the next two iterates $\vec{x}_1$ and $\vec{x}_2$ of the successive over-relaxation method for $w = 1.2$.
Answer:
SOR Formula:

$$\vec{x}_{k+1} = \left(\frac{1}{\omega} \cdot \boldsymbol{D} + \boldsymbol{L}\right)^{-1} \cdot \left(\vec{b} - \left(\boldsymbol{U} + \left(1 - \frac{1}{\omega}\right) \cdot \boldsymbol{D}\right) \cdot \vec{x}_k\right)$$

We again require the following values:

$$\boldsymbol{A} = \begin{bmatrix} 5 & -2 & 1 \\ -3 & 9 & 1 \\ 2 & -1 & -7 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix}, \quad \boldsymbol{D} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & -7 \end{bmatrix}$$

$$\boldsymbol{U} = \begin{bmatrix} 0 & -2 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \quad \boldsymbol{L} = \begin{bmatrix} 0 & 0 & 0 \\ -3 & 0 & 0 \\ 2 & -1 & 0 \end{bmatrix}, \quad \vec{x}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

We first calculate $\left(\frac{1}{\omega} \cdot \boldsymbol{D} + \boldsymbol{L}\right)^{-1}$ using the same method as before:

$$\left(\frac{1}{\omega} \cdot \boldsymbol{D} + \boldsymbol{L}\right)^{-1} = \begin{bmatrix} 0.22 & 0 & 0 \\ 0.080667 & 0.12222 & 0 \\ 0.056467 & -0.019206 & -0.15714 \end{bmatrix}$$

Now, we are ready to perform the iterations:

$$\vec{x}_1 = \left(\frac{1}{\omega} \cdot \boldsymbol{D} + \boldsymbol{L}\right)^{-1} \cdot \vec{b} = \begin{bmatrix} 0.22 & 0 & 0 \\ 0.080667 & 0.12222 & 0 \\ 0.056467 & -0.019206 & -0.15714 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -0.22 \\ 0.1638 \\ -0.5663 \end{bmatrix}$$

$$\vec{x}_2 = \begin{bmatrix} 0.22 & 0 & 0 \\ 0.080667 & 0.12222 & 0 \\ 0.056467 & -0.019206 & -0.15714 \end{bmatrix} \cdot \left(\begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix} - \left(\begin{bmatrix} 0 & -2 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \left(1 - \tfrac{1}{1.2}\right) \begin{bmatrix} 5 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & -7 \end{bmatrix} \right) \cdot \begin{bmatrix} -0.22 \\ 0.1638 \\ -0.5663 \end{bmatrix} \right)$$

$$\vec{x}_2 = \begin{bmatrix} -0.00135 \\ 0.2968 \\ -0.4619 \end{bmatrix}$$

d) Will the Jacobi and the Gauss-Seidel methods converge? Explain the answer.
Answer:
Jacobi Theorem: If $A \in \mathbb{R}^{n \times n}$ is strictly diagonally dominant, i.e. $|a_{ii}| > \sum_{j=1, j \neq i}^{n} |a_{ij}|$, for all rows $i = 1, ..., n$, then the Jacobi method converges for any initial guess x_0.

Gauss-Seidel Theorem: If $A \in \mathbb{R}^{n \times n}$ is diagonally dominant, i.e. $|a_{ii}| \geq \sum_{j=1, j \neq i}^{n} |a_{ij}|$, for all rows $i = 1, ..., n$, then the Gauss-Seidel method converges for any initial guess x_0. Let us examine $\boldsymbol{A}$:

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

$$\Rightarrow Row1 : |5| > |-2| + |1|$$

$$\Rightarrow Row2 : |9| > |-3| + |1|$$

$$\Rightarrow Row3 : |-7| > |2| + |-1|$$

We can see that A is strictly diagonally dominant. Therefore, both methods converge.

Some Notes:
For the first three parts of this question, performing each equation by hand can become quite hectic and prone to mistakes, so I prepared a simple python script to perform the iterations and verify my calculations. You can use it to verify your calculations for similar problems.

```python
import numpy as np
from typing import Union

# Algorithms:
class Jacobi:
  def __init__(self, *args, **kwargs) -> None:
    self.x = None          # Just for reference
    self.name = "Jacobi Iteration"  # Identifier

  def iteration(self, x, b, L, U, D):
    # Here we simply apply the formula
    self.x = np.matmul(np.linalg.inv(D), (b - np.matmul((L + U), x)))
    return self.x

class Gauss_Seidel:
  def __init__(self, *args, **kwargs) -> None:
    self.x = None
    self.name = "Gauss-Seidel Iteration"
    pass

  def iteration(self, x, b, L, U, D):
    # print("inverse(D + L): \n", np.linalg.inv(D + L))
    self.x = np.matmul(np.linalg.inv(D + L), (b - np.matmul(U, x)))
    return self.x

class Successive_Overrelaxation:
  def __init__(self, *args, **kwargs) -> None:
    self.x = None
    self.name = "Successive Overrelaxation"
    self.w = kwargs["w"]    # No better way of doing this for now
    print("w: ", self.w)      # since other algorithms don't have a 'w'

```

```python
  def iteration(self, x, b, L, U, D):
    # Divide and conquer (formula too big)
    first_part = np.linalg.inv(((1/self.w)*D) + L)
    second_part = b - np.matmul((U + (1 - (1/self.w))*D), x)
    # print("First part: \n", first_part)
    # print("Second part: \n", second_part)
    self.x = np.matmul(first_part, second_part)
    return self.x

# Interface through which the algorithms will be applied:
class Interface:
  def __init__(self,
         A: np.matrix,
         x: np.matrix,
         b: np.matrix,
         algo: Union[Jacobi, Gauss_Seidel, Successive_Overrelaxation],
         w: float = None) -> None:

    self.A = A
    self.x = x
    self.b = b
    self.D = np.diag(np.diag(A))  # Automating extraction of diagonal
    # self.P, self.L, self.U = sla.lu(A)
    self.L = np.tril(A) - self.D  # Extract lower triangular part
    self.U = np.triu(A) - self.D  # Extract upper triangular part
    self.step = 0          # Keep track of step number
    self.w = w             # Used for SOR
    self.algo = algo(w = w)     # Switch between algorithms here

  # Apply chosen algorithm
  def iteration(self, step_count = True):
    self.x = self.algo.iteration(x = self.x,
                                 b = self.b,
                                 L = self.L,
                                 U = self.U,
                                 D = self.D)
    if step_count == True:
      self.step += 1
    return self.x

  def iterate_n_times(self, steps = 5): # Automate
    i = 0
    while i < steps:
      self.step += 1
      self.print_step()
      self.iteration(step_count=False)
      self.print_x()
      i += 1

  def print_step(self):         # Used multiple times
    print(f"{self.algo.name}::Step {self.step}:")
```

```python

  def print_values(self):          # Print required values for reference
    self.print_step()
    print("A: \n", self.A)
    print()
    print("x: \n", self.x)
    print()
    print("b: \n", self.b)
    print()
    print("D: \n", self.D)
    print()
    print("D inverse: \n", np.linalg.inv(self.D))
    print()
    print("L: \n", self.L)
    print()
    print("U: \n", self.U)
    print()

  def print_x(self):
    # self.print_step()
    print("x: \n", self.x)
    print()

def test_set_1():             # Test set from hw5_2023_p1
  A = np.matrix([[5, -2, 1],
                 [-3, 9, 1],
                 [2, -1, -7]])
  b = np.transpose(np.matrix([-1, 2, 3]))
  x0 = np.transpose(np.matrix([0, 0, 0]))
  return A, b, x0

def test_set_2():
  A = np.matrix([[5, 2, -2],
                 [1, 6, -2],
                 [0, 1, 4]])
  b = np.transpose(np.matrix([5, 9, 10]))
  x0 = np.transpose(np.matrix([0, 0, 0]))
  return A, b, x0


def main(test_set, num_steps):
  A, b, x0 = test_set()
  print("Performing some steps of Jacobi Iteration...")
  jacobi = Interface(A, x0, b, Jacobi)
  jacobi.print_values()
  jacobi.iterate_n_times(num_steps)

  print("Performing some steps of Gauss-Seidel Iteration...")
  gsi = Interface(A, x0, b, Gauss_Seidel)
  gsi.print_values()
  gsi.iterate_n_times(num_steps)
```

```python

    print("Performing some steps of Successive Overrelaxation...")
    sor = Interface(A, x0, b, w = 1.1, algo = Successive_Overrelaxation)
    sor.print_values()
    sor.iterate_n_times(num_steps)


if __name__ == "__main__":
    main(test_set=test_set_1, num_steps=10)

```

The results from 10 iterations for each algorithm is shown below:

```
Performing some steps of Jacobi Iteration...
Jacobi Iteration::Step 0:
A:
 [[ 5 -2  1]
 [-3  9  1]
 [ 2 -1 -7]]

x:
 [[0]
 [0]
 [0]]

b:
 [[-1]
 [ 2]
 [ 3]]

D:
 [[ 5  0  0]
 [ 0  9  0]
 [ 0  0 -7]]

D inverse:
 [[ 0.2         0.          0.        ]
 [ 0.          0.11111111  0.        ]
 [-0.         -0.         -0.14285714]]

L:
 [[ 0  0  0]
 [-3  0  0]
 [ 2 -1  0]]
```

```
U:
 [[ 0 -2  1]
 [ 0  0  1]
 [ 0  0  0]]

Jacobi Iteration::Step 1:
x:
 [[-0.2       ]
 [ 0.22222222]
 [-0.42857143]]

Jacobi Iteration::Step 2:
x:
 [[-0.02539683]
 [ 0.2031746 ]
 [-0.51746032]]

Jacobi Iteration::Step 3:
x:
 [[-0.0152381 ]
 [ 0.2712522 ]
 [-0.46485261]]

Jacobi Iteration::Step 4:
x:
 [[ 0.0014714 ]
 [ 0.26879315]
 [-0.47167549]]

Jacobi Iteration::Step 5:
x:
 [[ 0.00185236]
 [ 0.27512108]
 [-0.46655005]]

Jacobi Iteration::Step 6:
x:
 [[ 0.00335844]
 [ 0.27467857]
 [-0.4673452 ]]
```

```
Jacobi Iteration::Step 7:
x:
 [[ 0.00334047]
 [ 0.27526895]
 [-0.46685167]]

Jacobi Iteration::Step 8:
x:
 [[ 0.00347791]
 [ 0.27520812]
 [-0.46694114]]

Jacobi Iteration::Step 9:
x:
 [[ 0.00347148]
 [ 0.27526388]
 [-0.46689318]]

Jacobi Iteration::Step 10:
x:
 [[ 0.00348419]
 [ 0.2752564 ]
 [-0.46690299]]

Performing some steps of Gauss-Seidel Iteration...
Gauss-Seidel Iteration::Step 0:
A:
 [[ 5 -2  1]
 [-3  9  1]
 [ 2 -1 -7]]

x:
 [[0]
 [0]
 [0]]

b:
 [[-1]
 [ 2]
 [ 3]]

D:
```

```
 [[ 5  0  0]
 [ 0  9  0]
 [ 0  0 -7]]

D inverse:
 [[ 0.2         0.          0.        ]
 [ 0.          0.11111111  0.        ]
 [-0.         -0.         -0.14285714]]

L:
 [[ 0  0  0]
 [-3  0  0]
 [ 2 -1  0]]

U:
 [[ 0 -2  1]
 [ 0  0  1]
 [ 0  0  0]]

Gauss-Seidel Iteration::Step 1:
x:
 [[-0.2       ]
 [ 0.15555556]
 [-0.50793651]]

Gauss-Seidel Iteration::Step 2:
x:
 [[-0.03619048]
 [ 0.26659612]
 [-0.47699672]]

Gauss-Seidel Iteration::Step 3:
x:
 [[ 0.00203779]
 [ 0.27590112]
 [-0.46740365]]

Gauss-Seidel Iteration::Step 4:
x:
 [[ 0.00384118]
 [ 0.27543635]
 [-0.466822  ]]
```

```
Gauss-Seidel Iteration::Step 5:
x:
 [[ 0.00353894]
 [ 0.27527098]
 [-0.46688473]]

Gauss-Seidel Iteration::Step 6:
x:
 [[ 0.00348534]
 [ 0.27526008]
 [-0.46689849]]

Gauss-Seidel Iteration::Step 7:
x:
 [[ 0.00348373]
 [ 0.27526108]
 [-0.46689909]]

Gauss-Seidel Iteration::Step 8:
x:
 [[ 0.00348425]
 [ 0.27526131]
 [-0.46689897]]

Gauss-Seidel Iteration::Step 9:
x:
 [[ 0.00348432]
 [ 0.27526133]
 [-0.46689895]]

Gauss-Seidel Iteration::Step 10:
x:
 [[ 0.00348432]
 [ 0.27526132]
 [-0.46689895]]

Performing some steps of Successive Overrelaxation...
w:  1.1
Successive Overrelaxation::Step 0:
A:
 [[ 5 -2  1]
```

```
 [-3  9  1]
 [ 2 -1 -7]]

x:
 [[0]
 [0]
 [0]]

b:
 [[-1]
 [ 2]
 [ 3]]

D:
 [[ 5  0  0]
 [ 0  9  0]
 [ 0  0 -7]]

D inverse:
 [[ 0.2         0.          0.        ]
 [ 0.          0.11111111  0.        ]
 [-0.         -0.         -0.14285714]]

L:
 [[ 0  0  0]
 [-3  0  0]
 [ 2 -1  0]]

U:
 [[ 0 -2  1]
 [ 0  0  1]
 [ 0  0  0]]

Successive Overrelaxation::Step 1:
x:
 [[-0.22      ]
 [ 0.16377778]
 [-0.56630794]]

Successive Overrelaxation::Step 2:
x:
 [[-0.00135003]
```

```
 [ 0.29678707]
 [-0.46186004]]

Successive Overrelaxation::Step 3:
x:
 [[ 0.01233052]
 [ 0.27573649]
 [-0.46469728]]

Successive Overrelaxation::Step 4:
x:
 [[ 0.0023244 ]
 [ 0.27451941]
 [-0.46736708]]

Successive Overrelaxation::Step 5:
x:
 [[ 0.00337686]
 [ 0.27535333]
 [-0.46690037]]

Successive Overrelaxation::Step 6:
x:
 [[ 0.00353586]
 [ 0.2752712 ]
 [-0.46688417]]

Successive Overrelaxation::Step 7:
x:
 [[ 0.00348026]
 [ 0.27525704]
 [-0.46690104]]

Successive Overrelaxation::Step 8:
x:
 [[ 0.0034833 ]
 [ 0.27526163]
 [-0.46689912]]

Successive Overrelaxation::Step 9:
x:
 [[ 0.00348459]
```

```
 [ 0.27526141]
 [-0.46689887]]

Successive Overrelaxation::Step 10:
x:
 [[ 0.00348431]
 [ 0.2752613 ]
 [-0.46689896]]
```

As we can see, in all three cases the algorithms converge to fixed values, but Jacobi iteration requires more number of steps to converge when compared to the rest.

Problem 2

(2+3+2+3 points) Let $A = L + D + U$ be a non-singular matrix with L being a lower triangular, D being a diagonal, and U being an upper triangular matrix.

a) Based on the lecture, show that the error for the iterative solver $e_k = x - x_k$ for iteration step k, approximation x_k and the linear system of equations $Ax = b$ with solution x, can be written as
$e_k = (I - Q^{-1}A)^k e_0$
with initial error $e_0 = x - x_0$ and initial guess x_0. Use the general iteration formula $x_{k+1} = (I - Q^{-1}A)x_k + Q^{-1}b$ without specifying Q.
Answer:

$$\begin{aligned}
\vec{x}_{k+1} &= (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})\vec{x}_k + \boldsymbol{Q}^{-1}\vec{b} \\
\vec{x}_{k+1} &= \vec{x}_k - \boldsymbol{Q}^{-1} \cdot \boldsymbol{A} \cdot \vec{x}_k + \boldsymbol{Q}^{-1} \cdot \vec{b} \\
\vec{x} - \vec{x}_{k+1} &= \boldsymbol{A}^{-1} \cdot \vec{b} - \vec{x}_k + \boldsymbol{Q}^{-1} \cdot \boldsymbol{A} \cdot \vec{x}_k - \boldsymbol{Q}^{-1} \cdot \vec{b} \\
\vec{x} - \vec{x}_{k+1} &= (\vec{x} - \vec{x}_k) + \boldsymbol{Q}^{-1}\boldsymbol{A}(\vec{x}_k - \boldsymbol{A}^{-1}\vec{b}) \\
\vec{x} - \vec{x}_{k+1} &= (\vec{x} - \vec{x}_k) - \boldsymbol{Q}^{-1}\boldsymbol{A}(\vec{x} - \vec{x}_k) \\
\vec{x} - \vec{x}_{k+1} &= (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})(\vec{x} - \vec{x}_k) \\
\vec{e}_{k+1} &= (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})\vec{e}_k
\end{aligned}$$

Now, evaluate this equation for some values:

$$
\begin{aligned}
e_1 &= (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})e_0 = (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})^1 e_0 \\
e_2 &= (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})e_1 = (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})^2 e_0 \\
e_3 &= (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})e_2 = (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})^3 e_0 \\
e_4 &= (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})e_3 = (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})^4 e_0 \\
e_5 &= (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})e_4 = (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})^5 e_0 \\
\vdots & \qquad\quad \vdots \\
e_k &= (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})^k e_0
\end{aligned}
$$

b) Show that the Jacobi iteration for solving $A\vec{x} = \vec{b}$ can be written as

$$\vec{x}_{k+1} = D^{-1} \cdot \left(\vec{b} - (L+U)\vec{x}_k\right)$$

Answer:
We know,

$$\vec{x}_{k+1} = (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})\vec{x}_k + \boldsymbol{Q}^{-1}\vec{b}$$

First, we rearrange the formula to a more suitable state:

$$
\begin{aligned}
\vec{x}_{k+1} &= (\boldsymbol{Q}^{-1}\boldsymbol{Q} - \boldsymbol{Q}^{-1}\boldsymbol{A})\vec{x}_k + \boldsymbol{Q}^{-1}\vec{b} \\
\vec{x}_{k+1} &= \boldsymbol{Q}^{-1}(\boldsymbol{Q} - \boldsymbol{A})\vec{x}_k + \boldsymbol{Q}^{-1}\vec{b} \\
\vec{x}_{k+1} &= \boldsymbol{Q}^{-1}(\vec{b} + (\boldsymbol{Q} - \boldsymbol{A})\vec{x}_k) \\
\vec{x}_{k+1} &= \boldsymbol{Q}^{-1}(\vec{b} - (\boldsymbol{A} - \boldsymbol{Q})\vec{x}_k)
\end{aligned}
$$

We know that $\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U}$, and for Jacobi Iteration, $\boldsymbol{Q} = \boldsymbol{D}$. Replacing these values into the derivation above, we obtain:

$$
\begin{aligned}
&\vec{x}_{k+1} = \boldsymbol{D^{-1}}(\vec{b} - (\boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U} - \boldsymbol{D})\vec{x}_k) \\
&\Rightarrow \vec{x}_{k+1} = \boldsymbol{D^{-1}}(\vec{b} - (\boldsymbol{L} + \boldsymbol{U})\vec{x}_k)
\end{aligned}
$$

c) Show that the Gauss-Seidel iteration for solving A x = b can be written as

$$\vec{x}_{k+1} = (D+L)^{-1} \cdot \left(\vec{b} - U\vec{x}_k\right)$$

Answer:
We know,

$$\vec{x}_{k+1} = (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})\vec{x}_k + \boldsymbol{Q}^{-1}\vec{b} = \boldsymbol{Q}^{-1}(\vec{b} - (\boldsymbol{A} - \boldsymbol{Q})\vec{x}_k) \quad \text{(Derived in previous solution)}$$

We know that $\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U}$, and for Gauss-Seidel Iteration, $\boldsymbol{Q} = \boldsymbol{L} + \boldsymbol{D}$. Replacing these values into the derivation above, we obtain:

$$\begin{aligned}
&\vec{x}_{k+1} = \boldsymbol{Q}^{-1}(\vec{b} - (\boldsymbol{A} - \boldsymbol{Q})\vec{x}_k) \\
&\Rightarrow \vec{x}_{k+1} = (\boldsymbol{L} + \boldsymbol{D})^{-1}(\vec{b} - (\boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U} - \boldsymbol{L} - \boldsymbol{D})\vec{x}_k) \\
&\Rightarrow \vec{x}_{k+1} = (\boldsymbol{L} + \boldsymbol{D})^{-1}(\vec{b} - \boldsymbol{U}\vec{x}_k)
\end{aligned}$$

d) Derive how the successive over-relaxation (SOR) iteration for solving $A\vec{x} = \vec{b}$ with weight w can be rewritten in a similar form to the one in (c).
Answer:
Once again, we use the following relation:

$$\vec{x}_{k+1} = (\boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{A})\vec{x}_k + \boldsymbol{Q}^{-1}\vec{b} = \boldsymbol{Q}^{-1}(\vec{b} - (\boldsymbol{A} - \boldsymbol{Q})\vec{x}_k) \quad \text{(Derived in part (b))}$$

For Successive Over-Relaxation, we use the following parameter changes:

$$\begin{aligned}
\boldsymbol{A} &= \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U} \\
\boldsymbol{Q} &= \boldsymbol{L} + \frac{1}{\omega}\boldsymbol{D}
\end{aligned}$$

The next steps look as follows:

$$\begin{aligned}
\vec{x}_{k+1} &= \boldsymbol{Q}^{-1}(\vec{b} - (\boldsymbol{A} - \boldsymbol{Q})\vec{x}_k) \\
\vec{x}_{k+1} &= \left(\boldsymbol{L} + \frac{1}{\omega}\boldsymbol{D}\right)^{-1} \left(\vec{b} - \left(\boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U} - \boldsymbol{L} - \frac{1}{\omega}\boldsymbol{D}\right)\vec{x}_k\right) \\
\vec{x}_{k+1} &= \left(\boldsymbol{L} + \frac{1}{\omega}\boldsymbol{D}\right)^{-1} \left(\vec{b} - \left(\boldsymbol{U} + \left(1 - \frac{1}{\omega}\right)\boldsymbol{D}\right)\vec{x}_k\right)
\end{aligned}$$

Problem 3

(6+4 points)

a) Generate the Bézier curve to the data points $(0, 2)$, $(1, 3)$, $(2, 0)$, and $(3, 0)$ and find the vector at the point $u = \frac{3}{4}$. Remember to convert u into the interval $[0, 1]$ first.
Answer:

Points: $(0,2), \quad (1,3), \quad (2,0), \quad (3,0)$

$u = \frac{3}{4}, \quad u \in [0,3] \quad \Rightarrow \quad t = \frac{u}{3}$

$$B(t) = \sum_{i=0}^{n} \left(\binom{n}{i} \cdot (1-t)^{(n-i)}\, t^i P_i \right), \quad 0 \le t \le 1$$

$$\Rightarrow B(t) = (1-t)^n P_0 + \binom{n}{1}(1-t)^{n-1} t P_1 + ... + \binom{n}{n-1}(1-t)t^{n-1} P_{n-1} + t^n P_n, \quad 0 \le t \le 1$$

For $n = 3$,

$$B(t) = \sum_{i=0}^{3} \left(\binom{3}{i} \cdot (1-t)^{(3-i)}\, t^i P_i \right)$$
$$\Rightarrow B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3$$

For the x-coordinates:

$$\begin{aligned} B_x(t) &= (1-t)^3(0) + 3(1-t)^2 t(1) + 3(1-t)t^2(2) + t^3(3) \\ \Rightarrow B_x(t) &= 3(t - 2t^2 + t^3) + 6t^2 - 6t^3 + 3t^3 \\ \Rightarrow B_x(t) &= 3t - 6t^2 + 3t^3 + 6t^2 - 3t^3 \\ \Rightarrow B_x(t) &= 3t \end{aligned}$$

For the y-coordinates:

$$\begin{aligned} B_y(t) &= (1-t)^3(2) + 3(1-t)^2 t(3) + 3(1-t)t^2(0) + t^3(0) \\ \Rightarrow B_y(t) &= 2(1 - 3t + 3t^2 - t^3) + 9t(1 - 2t + t^2) \\ \Rightarrow B_y(t) &= 2 + 3t - 12t^2 + 7t^3 \end{aligned}$$

$$\vec{B}(t) = \begin{bmatrix} 3t \\ 2 + 3t - 12t^2 + 7t^3 \end{bmatrix}$$

Now, we calculate the values:

$$\begin{aligned} B_x\left(t = \frac{u}{3} = \frac{1}{4}\right) &= \frac{3}{4} \\ B_y\left(t = \frac{u}{3} = \frac{1}{4}\right) &= \frac{135}{64} \\ \vec{B}\left(t = \frac{u}{3} = \frac{1}{4}\right) &= \begin{bmatrix} \frac{3}{4} \\ \frac{135}{64} \end{bmatrix} \end{aligned}$$

b) Sketch the Bézier points, curve and polygon.

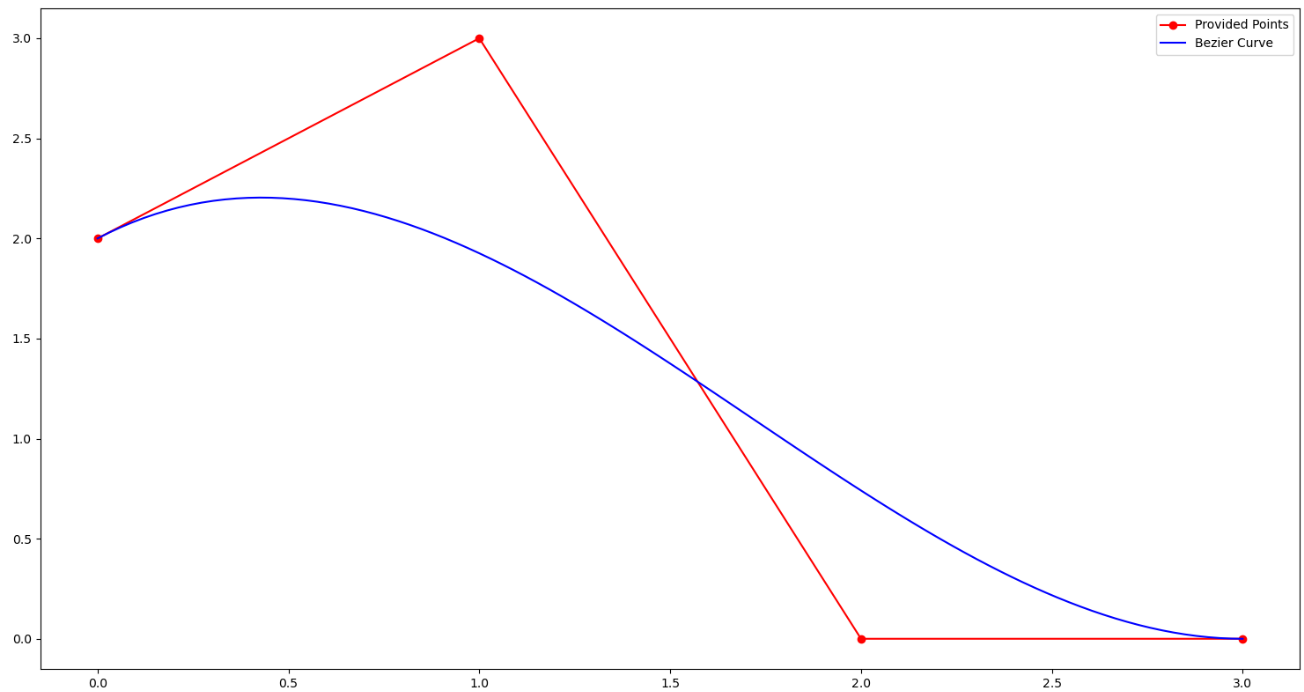
Answer:

We will be plotting the following points, and the subsequent vector valued function.

Points: $(0, 2), \quad (1, 3), \quad (2, 0), \quad (3, 0)$

$$\vec{B}(t) = \begin{bmatrix} 3t \\ 2 + 3t - 12t^2 + 7t^3 \end{bmatrix}$$

The plots look as follows:



The python script used to generate this plot is provided below:

```
import numpy as np
import matplotlib.pyplot as plt

def bx(t):           # Bezier formula for x-axis
    return 3*t

def by(t):           # Bezier formula for y-axis
    return 2 + 3*t - 12*t**2 + 7*t**3


x = [0, 1, 2, 3]         # Provided x coordinates
y = [2, 3, 0, 0]         # Provided y coordinates

# Evaluating points over the given range
u = np.linspace(0, 3, 100)
```

```

t = u/3                                                      # Normalize

# Calculate Bezier values for both axes
bxt = [bx(i) for i in t]
byt = [by(i) for i in t]

plt.plot(x, y, 'ro-', label = 'Provided Points')
plt.plot(bxt, byt, 'b-', label = 'Bezier Curve')
plt.legend()
plt.show()

```