EP2

VICENTE V. FIGUEIRA, NUSP: 11809301

Programa 1. Solução de Sistemas de Equações Lineares

1.(B)

Conforme mostrado no Item **a**, nosso problema consiste de resolver o seguinte sistema linear de equações,

$$\begin{bmatrix} 0.0 & 5.3 & -1.8 \\ 11.9 & 0.0 & 1.8 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} 3.1 \\ 15.0 \\ 0 \end{bmatrix} \tag{1.1}$$

Faremos isso neste item primeiro pelo método de Eliminação de Gauss utilizando Pivoteamento Parcial. Optamos por representar a matrix em nosso programa como uma matrix aumentada,

$$\left[\begin{array}{ccc||c} 0.0 & 5.3 & -1.8 & 3.1 \\ 11.9 & 0.0 & 1.8 & 15.0 \\ 1 & -1 & -1 & 0 \end{array}\right] \tag{1.2}$$

O programa que realiza os passos do Pivoteamento pode ser visto abaixo,

```python
def fTriangulacao(tMatrix):

    vDimensao = len(tMatrix) # Tamanho da matrix dada como input

    #########################################

    # Procura pela posição do Pivô

    for vColuna in range(vDimensao-1):

        vPivot = 0
        vLinhaPivot = 0

        for vLinha in range(vColuna, vDimensao):

            if abs( tMatrix[vLinha][vColuna] ) > abs( vPivot ):

                vPivot = tMatrix[vLinha][vColuna]
                vLinhaPivot = vLinha

    #########################################

    # Troca linhas de lugar para realocar o Pivô
    # e printa ao final de cada permutação de linhas

        vTemp1 = tMatrix[vColuna][:]
        vTemp2 = tMatrix[vLinhaPivot][:]

        tMatrix[vColuna] = vTemp2
        tMatrix[vLinhaPivot] = vTemp1

        print("Pivoteamento:")
        print(tMatrix)

    #########################################

    # Zera os espaços abaixos da posição do Pivô
    # e printa ao final de cada passo
```

```
        for vLinha in range(vColuna + 1, vDimensao):

            vMlc = -tMatrix[vLinha][vColuna] / tMatrix[vColuna][vColuna]

            for vTempCol in range(vColuna, vDimensao + 1):

                tMatrix[vLinha][vTempCol] = tMatrix[vColuna][vTempCol] * vMlc + tMatrix[vLinha][vTempCol]

            print("Introdução de zeros abaixo do pivô:")
            print(tMatrix)

    return tMatrix

def fSolucionar(tMatrix):

    vDimensao = len(tMatrix)
    tSolucao = []

    #########################################

    # Inicia a solução como um array nulo

    for vLinhaTemp in range(vDimensao):

        tSolucao.append(0)

    #########################################

    # Realiza o método de substituição para trás
    # para obter a solução final

    for vLinha in range(vDimensao-1, -1, -1):

        vFator = 1 / tMatrix[vLinha][vLinha]
        vSoma = 0

        for vColuna in range(vLinha+1, vDimensao):

            vSoma = vSoma + tSolucao[vColuna] * tMatrix[vLinha][vColuna]

        tSolucao[vLinha] = vFator * (tMatrix[vLinha][vDimensao] - vSoma)

    return tSolucao

    #########################################

    # Inicia os valores corretos para a matrix e o vetor b

tMatrix = [[0.0,5.3,-1.8],[11.9,0.0,1.8],[1,-1,-1]]
tB = [3.1,15.0,0]
tX = []

    #########################################

    # Transforma a Matrix em aumentada

vDimensao = len(tMatrix)

for vLinha in range(vDimensao):

    tMatrix[vLinha].append(tB[vLinha])

    #########################################

    # Triangula a matrix
```

```python

tMatrix = fTriangulacao(tMatrix)

print("Matrix Triangulada:")
print(tMatrix)

    ########################################

    # Resolve o sistema

tX = fSolucionar(tMatrix)

print("Solução Final:")
print(tX)
```

Este programa também printa os resulados passo a passo, que são — Mostrando até a 4ª casa decimal — :

Pivoteamento:
$[[11.9, 0.0, 1.8, 15.0],$
$[0.0, 5.3, -1.8, 3.1],$
$[1, -1, -1, 0]]$
Introdução de zeros abaixo do pivô :
$[[11.9, 0.0, 1.8, 15.0],$
$[0.0, 5.3, -1.8, 3.1],$
$[1, -1, -1, 0]]$
Introdução de zeros abaixo do pivô :
$[[11.9, 0.0, 1.8, 15.0],$
$[0.0, 5.3, -1.8, 3.1],$
$[0.0, -1.0, -1.1513, -1.2605]]$
Pivoteamento :
$[[11.9, 0.0, 1.8, 15.0],$
$[0.0, 5.3, -1.8, 3.1],$
$[0.0, -1.0, -1.1513, -1.2605]]$
Introdução de zeros abaixo do pivô:
$[[11.9, 0.0, 1.8, 15.0],$
$[0.0, 5.3, -1.8, 3.1],$
$[0.0, 0.0, -1.4909, -0.6756]]$
Matrix Triangulada:
$[[11.9, 0.0, 1.8, 15.0],$
$[0.0, 5.3, -1.8, 3.1],$
$[0.0, 0.0, -1.4909, -0.6756]]$
Solução Final:
$[1.1920, 0.7388, 0.4532]$

Isso nos dá um resultado da matrix triangular final como:

$$\begin{bmatrix} 11.9 & 0.0 & 1.8 \\ 0.0 & 5.3 & -1.8 \\ 0 & 0 & -1.4909 \end{bmatrix} \tag{1.3}$$

E com a solução do sistema sendo:

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} 1.1920 \\ 0.7388 \\ 0.4532 \end{bmatrix} \tag{1.4}$$

1.(C)

Agora iremos resolver o mesmo sistema de equações utilizando do método de Jacobi. Para isso vamos utilizar da matrix já com suas duas primeiras linhas permutadas.

$$\begin{bmatrix} 11.9 & 0.0 & 1.8 \\ 0.0 & 5.3 & -1.8 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} I_2 \\ I_1 \\ I_3 \end{bmatrix} = \begin{bmatrix} 15.0 \\ 3.1 \\ 0 \end{bmatrix} \tag{1.5}$$

O critério de parada utilizado foi a precisão de '$\epsilon = 10^{-3}$' entre passos da solução com o "chute" para a solução inicial sendo $[0, 0, 0]$. O programa pode ser visto abaixo,

```python
# Precisão para a parada do programa
vPrecisao = 1.E-3

def fJacobi(tMatrix, tX0, tB):

    ##########################################

    # Definição de Varráveis inicias. Erro inicial
    # é escolhido arbitrariamente

    vPasso = 0
    vErro = 10
    tXPasso = tX0[:]
    vDimensao = len(tX0)

    print("Passo: ", vPasso)
    print("Erro: --")
    print("Solução neste passo: ", tXPasso)

    ##########################################

    # Loop principal

    while vErro > vPrecisao:

    # Salva o passo atual para o calculo de erros

        tXProxPasso = tXPasso[:]

    ##########################################

    # Atualiza o Passo atual

        for vLinha in range(vDimensao):

            vSoma = 0

            for vColuna in range(vDimensao):

    # Ignora os elementos da diagonal

                if vColuna == vLinha:

                    continue

                vSoma = vSoma + tXProxPasso[vColuna] * tMatrix[vLinha][vColuna]

            vFator = 1 / tMatrix[vLinha][vLinha]
            tXPasso[vLinha] = vFator * (tB[vLinha] - vSoma)

    ##########################################

    # Calcula o erro

```

```python
        vErro = 0

        for vLinha in range(vDimensao):

            vErroTemp = abs(tXPasso[vLinha] - tXProxPasso[vLinha])

            if vErroTemp > vErro:

                vErro = vErroTemp

    # Atualiza os contadores e printa o resultado
    # de cada passo

        vPasso = vPasso + 1

        print("Passo: ", vPasso)
        print("Erro: ", vErro)
        print("Solução neste passo: ", tXPasso)

    return(tXPasso)

    #########################################

    # Define os parametros de nosso problema

tMatrix= [[11.9,0.0,1.8],[0.0,5.3,-1.8],[1,-1,-1]]
tB = [15.0,3.1,0]
tX0 = [0,0,0]

    #########################################

    # Resolve o sistema

tSolucao = fJacobi(tMatrix, tX0, tB)

print("Solução Final: ", tSolucao)
```

O retorno do programa é,

Passo	Erro	$[I_1, I_2, I_3]$
1	1.261	[1.261, 0.585, -0.0]
2	0.671	[1.261, 0.585, 0.676]
3	0.229	[1.158, 0.814, 0.676]
4	0.332	[1.158, 0.814, 0.344]
5	0.113	[1.208, 0.702, 0.344]
6	0.163	[1.208, 0.702, 0.507]
7	0.055	[1.184, 0.757, 0.507]
8	0.080	[1.184, 0.757, 0.427]
9	0.027	[1.196, 0.730, 0.427]
10	0.039	[1.196, 0.730, 0.466]
11	0.013	[1.190, 0.743, 0.466]
12	0.019	[1.190, 0.743, 0.447]
13	0.007	[1.193, 0.737, 0.447]
14	0.009	[1.193, 0.737, 0.456]
15	0.003	[1.191, 0.740, 0.456]
16	0.005	[1.191, 0.740, 0.452]
17	0.001	[1.192, 0.738, 0.452]
18	0.002	[1.192, 0.738, 0.454]
19	0.0007	[1.192, 0.739, 0.4543]

Na tabela mostramos apenas até o 3º algarismo significativo, a solução final é,

$$\text{Solução Final: } [1.1918465707994006, 0.7390614731107796, 0.45390322638173963]$$

$$\text{Erro Final: } 0.00077358903768087774$$

Que é compatível com o resultado obtido pelo método anterior.

1.(D)

O nosso sistema é, após permutar a primeira com a segunda linha,

$$\begin{bmatrix} 11.9 & 0.0 & 1.8 \\ 0.0 & 5.3 & -1.8 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} I_2 \\ I_1 \\ I_3 \end{bmatrix} = \begin{bmatrix} 15.0 \\ 3.1 \\ 0 \end{bmatrix} \tag{1.6}$$

Que resulta em uma matrix

$$\begin{bmatrix} 11.9 & 0.0 & 1.8 \\ 0.0 & 5.3 & -1.8 \\ 1 & -1 & -1 \end{bmatrix} \tag{1.7}$$

Assim podemos obter cada 'a_{ij}', onde 'i' refere a linha e j a coluna. A construção da matrix '$\mathbb{J}$' é então,

$$\begin{bmatrix} 0 & 0 & \frac{1.8}{11.9} \\ 0 & 0 & -\frac{1.8}{5.3} \\ -1 & 1 & 0 \end{bmatrix} \tag{1.8}$$

Cujos auto-valores calculados são,

$$\lambda_1 = 0 \tag{1.9}$$

$$\lambda_2 = \frac{6\sqrt{542402}}{6307}\mathrm{i} \tag{1.10}$$

$$\lambda_3 = -\frac{6\sqrt{542402}}{6307}\mathrm{i} \tag{1.11}$$

E portanto o raio espectral é,

$$\rho_s = \|\lambda_2\| = \frac{6\sqrt{542402}}{6307} \tag{1.12}$$

We have then to solve for,

$$\rho_s^k = 10^{-3} \tag{1.13}$$

$$k \ln\left(\rho_s^k\right) = -3\ln(10) \tag{1.14}$$

$$k = -3\frac{\ln(10)}{\ln\left(\rho_s^k\right)} \tag{1.15}$$

$$k = -3\frac{\ln(10)}{\ln\left(\frac{6\sqrt{542402}}{6307}\right)} \tag{1.16}$$

$$k = 19.4 \tag{1.17}$$

O que é extremamente compatível com o resultado obtido no item anterior, que convergiu em exatos 19 passos!

1.(E)

Agora vamos utilizar do método de Gauss-Seidel, a principal diferença deste método com o método de Jacobi está na linha 50. Foi utilizado também uma precisão de '10^{-3}' e valores inicial da solução como sendo '$[0, 0, 0]$'. O código pode ser visto abaixo,

```
1  # Precisão para a parada do Programa
2  vPrecisao = 1.E-3
3
4  def fGaussSeidel(tMatrix, tX0, tB):
5
6      ##########################################
7
8      # Definição dos parâmetros do problema,
9      # a escolha do erro é arbitrária
10
11     vPasso = 0
12     vErro = 10
13     tXPasso = tX0[:]
14     vDimensao = len(tX0)
```

```python

    print("Passo: ", vPasso)
    print("Erro: --")
    print("Solução neste passo: ", tXPasso)

    ########################################

    # Loop principal do método

    while vErro > vPrecisao:

    # Salva o Valor do passo atual para
    # calcular o erro posteriormente

        tXProxPasso = tXPasso[:]

    ########################################

    # Calculo do próximo passo

        for vLinha in range(vDimensao):

            vSoma = 0

    # Ignora os elementos da diagonal

            for vColuna in range(vDimensao):

                if vColuna == vLinha:

                    continue

    # Diferença do método é utilizar nesta soma tXPasso
    # ao invés de utilizar tXProxPasso

                vSoma = vSoma + tXPasso[vColuna] * tMatrix[vLinha][vColuna]

            vFator = 1 / tMatrix[vLinha][vLinha]
            tXPasso[vLinha] = vFator * (tB[vLinha] - vSoma)

    ########################################

    # Cálculo do erro

        vErro = 0

        for i in range(vDimensao):

            vErroTemp = abs(tXPasso[i] - tXProxPasso[i])

            if vErroTemp > vErro:

                vErro = vErroTemp

    ########################################

    # Aumenta o contador de passos e printa

        vPasso = vPasso + 1

        print("Passo: ", vPasso)
        print("Erro: ", vErro)
        print("Solução neste passo: ", tXPasso)

    return(tXPasso)

```

```python
    ########################################

    # Define os valores do problema

tMatrix= [[11.9,0.0,1.8],[0.0,5.3,-1.8],[1,-1,-1]]
tB = [15.0,3.1,0]
tX0 = [0,0,0]

    ########################################

    # Resolve o sistema

tSolucao = fGaussSeidel(tMatrix, tX0, tB)

print("Solução Final: ", tSolucao)
```

O retorno do programa é,

Passo	Erro	$[I_1, I_2, I_3]$
0	–	[0, 0, 0]
1	1.261	[1.261, 0.585, 0.671]
2	0.332	[1.158, 0.814, 0.344]
3	0.163	[1.208, 0.702, 0.507]
4	0.080	[1.184, 0.757, 0.427]
5	0.039	[1.196, 0.730, 0.466]
6	0.0193	[1.190, 0.743, 0.447]
7	0.009	[1.193, 0.737, 0.456]
8	0.005	[1.191, 0.740, 0.452]
9	0.002	[1.192, 0.738, 0.454]
10	0.001	[1.192, 0.739, 0.453]
11	0.0005	[1.192, 0.739, 0.453]

Novamente na tabela mostramos apenas os 3 primeiros algarismos significativos para facilitar a leitura. O resultado final é,

$$\text{Solução Final: } [1.192015699509284, 0.7386817312904751, 0.453333968218809]$$
$$\text{Erro Final: } 0.0005488705301880392$$

Que é também compatível com os outros métodos, porém converge mais rapidamente.