Universidade Federal de Goiás
Goiânia, maio de 2021

Trabalho 2 - Resolução de Sistemas Lineares

Aluno: Matheus Lázaro Honório da Silva
Matrícula: 201801523
Disciplina: Cálculo Numérico

1 - Escreva um algoritmo para a resolução de um sistema linear triangular inferior.

Um sistema triangular inferior é tal que Ax = b em que A[i][j] = 0, para todo j > i, isto é

A[1][1]x1 = b1

A[2][1]x1 + A[2][2]x2 = b2

. . . . .
. . . . .
. . . . .

A[n][1]x1 + A[n][2]x2 + … + A[n][n]xn = bn

Algoritmo para resolver um sistema linear triangular inferior em Python:

```
import numpy as np

def resolverSistemaLinearTriangularInferior(A, b):
   n = len(A)
   x = np.zeros(n)

   for i in range(0, n): # 0 até n-1
       soma = 0.0
       for j in range(0, i): # 0 até i-1
           soma = soma + A[i, j] * x[j]
       x[i] = (b[i] - soma)/A[i, i]
   return x

A = np.array(
   [
           [3.0,   0.0,    0.0],
           [1.0,   2.0,    0.0],
           [2.0,   -4.0,   1.0],
   ]
)
b = np.array(
   [9.0,   5.0,    7.0]
)

x = resolverSistemaLinearTriangularInferior(A, b)
print(x)
# solução : x = [3.0, 1.0, 5.0]
```

4 - Seja Ax = b um sistema n x n com matriz tridiagonal (a[i][j] = 0 se |i - j| > 1).

a) Escreva um algoritmo para resolver Ax = b através da Eliminação de Gauss com estratégia de pivoteamento parcial de modo que a estrutura especial da matriz A seja explorada.

```
import numpy as np

def eliminacaoGaussComPivParcial(A, b):
    n = len(b)
    x = np.zeros(n)

    for k in range(n - 1):
        # pivoteamento
        if(abs(A[k, k]) < 0.00000001):
            for i in range(k + 1, n):
                if(abs(A[i, k]) > abs(A[k, k])):
                    A[k, i] = A[i, k]
                    b[k, i] = b[i, k]
                    break

        for i in range(k + 1, n):
            if(A[i, k] == 0):
                continue
            m = A[k, k]/A[i, k]
            for j in range(k, n):
                A[i, j] = A[k, j] - A[i, j] * m
            b[i] = b[k] - b[i] * m

    x[n - 1] = b[n - 1] / A[n - 1, n - 1]

    for i in range(n - 2, -1, -1):
        soma = 0
        for j in range(i + 1, n):
            soma += A[i, j] * x[j]
        x[i] = (b[i] - soma) / A[i, i]
    return x
```

b) Compare o "custo" de resolvê-lo por Eliminação de Gauss via algoritmo tradicional, com o de resolvê-lo pelo algoritmo do item (a).

Algoritmo de Eliminação de Gauss tradicional em Python:

```
import numpy as np

def eliminacaoGauss(A, b):
    n = len(b)
    x = np.zeros(n)
    for k in range(n - 1):
        for i in range(k + 1, n):
            if(A[i, k] == 0):
                continue
            m = A[k, k]/A[i, k]
            for j in range(k, n):
                A[i, j] = A[k, j] - A[i, j] * m
            b[i] = b[k] - b[i] * m

    x[n - 1] = b[n - 1] / A[n - 1, n - 1]
    for i in range(n - 2, -1, -1):
        soma = 0
        for j in range(i + 1, n):
            soma += A[i, j] * x[j]
        x[i] = (b[i] - soma) / A[i, i]
    return x
```

c) Teste seus resultados com o sistema:

$$\begin{cases} 2x_1 - x_2 = 1 \\ -x_{i-1} + 2x_i - x_{i+1} = 0, \quad 2 \leq i \leq (n-1) \\ -x_{n-1} + 2x_n = 0 \end{cases}$$

para n = 10.

Resultados deste sistema utilizando Eliminação de Gauss:
X = [0.8 0.6 0.4 0.2]

Para comparar os custos, foi utilizada a biblioteca time do Python, importando no início do programa e calculando a diferença de tempo entre antes e depois da execução de cada função.
Foi realizada a comparação através do cálculo da média de 10 execuções de cada função para o sistema fornecido em (c).

Tempo de execução Eliminação de Gauss tradicional:
0.000294
0.000338
0.000299
0.000226
0.000138
0.000350
0.000138
0.000139
0.000329
0.000166
media: 0,0002417 segundos ≃ 241,7 microssegundos

Tempo de execução Eliminação de Gauss com Pivoteamento Parcial
0.000155
0.000159
0.000271
0.000247
0.000162
0.000152
0.000268
0.000155
0.000164
media: 0,0001733 segundos ≃ 173,3 microssegundos

Desta forma, temos que Eliminação de Gauss com Pivoteamento Parcial é mais eficiente.
Cabe ressaltar que os resultados pode ser diferentes conforme o computador utilizado na execução.

11 -

Estratégia usada para obter A^-1 pelo método da fatoração LU:
Sendo Ax = e[i], i = 1, 2, …, n
sendo e[i] a coluna i da matriz identidade de ordem n.

- **Entre o método da Eliminação de Gauss e fatoração LU, qual o mais indicado para o cálculo de A^-1?**

  Método da Eliminação de Gauss apresentado em 4- b).
  Método da fatoração LU em Python mostrado no Projeto.

  Método mais indicado para o cálculo de A^-1 é a fatoração LU.
  - O método da fatoração LU é melhor, visto que, sendo a fatoração de A completa, fica mais simples obter o inverso de A, usando o mesmo processo de substituição para frente e para trás usado para resolver o vetor arbitrário de b. Podemos usar as colunas da matriz identidade como vetores individuais de b, para resolver o inverso coluna por coluna.

a) Aplique o método escolhido no item (c) para obter a inversa da matriz

A =

4	**-1**	**0**	**-1**	**0**	**0**
-1	**4**	**-1**	**0**	**-1**	**0**
0	**-1**	**4**	**0**	**0**	**-1**
-1	**0**	**0**	**4**	**-1**	**0**
0	**-1**	**0**	**-1**	**4**	**-1**
0	**0**	**-1**	**0**	**-1**	**4**

Resultado obtido para A^-1 pelo método da fatoração LU:

A^-1 =

0.29482402	0.0931677	0.02815735	0.08612836	0.04968944	0.0194617
0.0931677	0.32298137	0.0931677	0.04968944	0.10559006	0.04968944
0.02815735	0.0931677	0.29482402	0.0194617	0.04968944	0.08612836
0.08612836	0.04968944	0.0194617	0.29482402	0.0931677	0.02815735
0.04968944	0.10559006	0.04968944	0.0931677	0.32298137	0.0931677
0.0194617	0.04968944	0.08612836	0.02815735	0.0931677	0.29482402

22 - Em cada caso:

a) Verifique se o critério de Sassenfeld é satisfeito;

Algoritmo para verificar se o critério de Sassenfeld é satisfeito, em Python:

```
import numpy as np
def criterioSassenfeld(A):
    coeficientes = np.zeros(len(A))
    for i in range(len(A)):
        b = 0
        for j in range(len(A)):
            if((i != j and i == 0) or i < j):
                b = b + A[i, j]
            elif(i != j and i != 0):
                b = b + A[i, j] * coeficientes[j]
        b = b / A[i, i]
        np.append(coeficientes, b) # anexar b ao fim
    maiorCoeficiente = max(coeficientes)
    if(maiorCoeficiente < 1):
        print("O método de Gauss-Seidel converge para o sistema")
    else:
        print("Não é possível afirmar a convergência do método de
Gauss-Seidel no sistema")
```

Caso 1:

$$A = \begin{bmatrix} 10 & 1 & 1 \\ 1 & 10 & 1 \\ 1 & 1 & 10 \end{bmatrix}; \quad b = \begin{bmatrix} 12 \\ 12 \\ 12 \end{bmatrix}$$

Resposta: O método de Gauss-Seidel converge para o sistema

Caso 2:

$$A = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix}; \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Resposta: O método de Gauss-Seidel converge para o sistema

b) Resolva por Gauss-Seidel, se possível:

Método de Gauss-Seidel:

```
import numpy as np
def gaussSeidel(A, b, iteracoes):
    iteracao = 0
    x = np.zeros(len(A))
    solucoes = np.zeros(len(A))
    while(iteracao < iteracoes):
        for i in range(len(A)):
            x = b[i]
            for j in range(len(A)):
                if(i != j):
                    x -= A[i, j] * solucoes[j]
            x /= A[i, i]
            solucoes[i] = x
        iteracao += 1
    print("sol:")
    print(solucoes)
A = np.array(
    [
        [10.0,   1.0,    1.0],
        [1.0,   10.0,   1.0],
        [1.0,   1.0,    10.0],
    ]
)
b = np.array(
    [12.0, 12.0, 12.0]
)
print("solucao:")
gaussSeidel(A, b, 250)
```

```
Caso 1:
A =  10    1    1         b =  12
      1   10    1    ;         12
      1    1   10              12
Solução obtida: x = [1.0, 1.0, 1.0]
```

```
Caso 2:
A =   4   -1    0    0         b =  1
     -1    4   -1    0    ;         1
      0   -1    4   -1              1
      0    0   -1    4              1
Solução obtida: x = [0.3636, 0.4545, 0.4545, 0.3636]
```

25 -

a) Aplique analítica e graficamente os métodos de Gauss-Jacobi e Gauss-Seidel no sistema:

2x1 + 5x2 = -3

3x1 + x2 = 2

b) Repita o item (a) para o sistema obtido permutando as equações.

c) Analise seus resultados.

Método de Gauss-Jacobi em python:

```
import numpy as np
def gaussJacobi(A, b, solucao, iteracoes):
    iteracao = 0
    vetAux = np.zeros(len(A))
    while(iteracao < iteracoes):
        for i in range(len(A)):
            x = b[i]
            for j in range(len(A)):
                if(i != j):
                    x = x - A[i, j] * solucao[j]
            x = x / A[i, i]
            vetAux[i] = x
        iteracao += 1
        for p in range(len(vetAux)):
            solucao[p] = vetAux[p]
    print(solucao)
```

Método de Gauss-Seidel apresentado em 22 - b)

a) O sistema não converge por Gauss-Jacobi nem por Gauss-Seidel.

- Através do critério da linhas.

b)

Permutando as equações:

Solução do método de Gauss-Seidel para o sistema com um máximo de 60 iterações:

x ≃ [1.0 -1.0]

Solução do método de Gauss-Jacobi para o sistema com um máximo de 60 iterações:

x ≃[1.0 -1.0]

33 - Resolva os sistemas lineares abaixo usando a fatoração de Cholesky:

Método da fatoração de Cholesky em Python:

```
import numpy as np
import math
def metodoCholesky(matriz):
    n = len(matriz)
    L = np.zeros((n, n))
    for i in range(n):
        for j in range(i + 1):
            soma = 0
            if(j != i):
                for k in range(j):
                    soma = soma + (L[i, k] * L[j, k])
                if(L[j, j] > 0):
                    L[i, j] = int((matriz[i, j] - soma)/L[j, j])
            else:
                for k in range(j):
                    soma = soma + pow(L[j, k], 2)
                L[j, i] = int(math.sqrt(matriz[j, j] - soma))
    #print(L)
    return L
def resolveSistema(G, gTransposta, b):
    n = len(G)
    y = np.zeros(n)
    for i in range(n):  # laço de repetição decrescente de n-1 até 0
        soma = 0
        for j in range(i):
            soma = soma + G[i, j] * y[j]
        y[i] = (b[i] - soma) / G[i, i]
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        soma = 0
        for j in range(i, n):
            soma = soma + gTransposta[i, j] * x[j]
        x[i] = (y[i] - soma) / gTransposta[i, i]
    return x
matriz = np.array([
        [16, 4, 8, 4],
        [4, 10, 8, 4],
        [8, 8, 12, 10],
        [4, 4, 10, 12]
])
b = np.array([32, 26, 38, 30])
L = metodoCholesky(matriz)
x = resolveSistema(L, np.transpose(L), b)
print(x)
```

a) **16x1 + 4x2 + 8x3 + 4x4 = 32**
4x1 + 10x2 + 8x3 + 4x4 = 26
8x1 + 8x2 + 12x3 + 10x4 = 38
4x1 + 4x2 + 10x3 + 12x4 = 30

Solução obtida pelo método da fatoração de Cholesky:
x = [1.0, 1.0, 1.0, 1.0]

b) **20x1 + 7x2 + 9x3 = 16**
7x1 + 30x2 + 8x3 = 38
9x1 + 8x2 + 30x3 = 38
Solução obtida pelo método da fatoração de Cholesky:
x = [0.0, 1.0, 1.0]

Projeto

a) Compare as soluções dos sistemas lineares

Utilizando o método da Fatoração LU com Pivoteamento Parcial:

```
import numpy as np
def imprimirMatriz(A):
    n = len(A)
    for i in range(n):
        for j in range(n):
            print("%.3f\t" %A[i][j], end="")
        print("")
    print("")
def fatoracaoLUcomPivotamentoParcial(A, b):
    n = len(A)
    cofator = np.zeros(n)
    L = np.zeros(n)
    U = np.zeros(n)
    for i in range(0, n):
        cofator[i] = i
    print("A")
    imprimirMatriz(A)
    for k in range(0, n - 1):
        pivo = abs(A[k, k])
        r = k
        for i in range(k, n):
            if(abs(A[i, k]) > pivo):
                pivo = abs(A[i, k])
                r = i
        if(pivo == 0):
            print("A matriz é singular. O sistema não admite uma única
solução.")
            break
        if(r != k):
            aux = cofator[k]
            cofator[k] = cofator[r]
            cofator[r] = aux
            for j in range(0, n):
                aux = A[k, j]
                A[k, j] = A[r, j]
                A[r, j] = aux
        for i in range(k+1, n):
            m = A[i, k] / A[k, k]
            A[i, k] = m
            for j in range(k+1, n):
                A[i, j] = A[i, j] - m * A[k, j]
        L = np.eye(n)
        U = np.zeros((n, n))
        for i in range(n):
            for j in range(n):
                if(i > j):
                    L[i, j] = A[i, j]
                elif(j >= i):
```

```
                U[i, j] = A[i, j]
    print("L")
    imprimirMatriz(L)
    print("U")
    imprimirMatriz(U)
    c = np.zeros(n)
    for i in range(0, n):
        r = int(cofator[i])
        c[i] = b[r]
    y = np.zeros(n)
    for i in range(0, n):
        soma = 0
        for j in range(0, i):
            soma = soma + A[i, j] * y[j]
        y[i] = c[i] - soma
    print("Y")
    print(y)
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        soma = 0
        for j in range(i, n):
            soma = soma + A[i, j] * x[j]
        x[i] = (y[i] - soma) / A[i, i]
    print("Resultado: ")
    for i in range(n):
        print("%.3f\t" %x[i], end="")
A = np.array(
    [
        [1, -1],
        [1, -0.99999],

    ]
)
b = np.array(
        [1, 0]
)
fatoracaoLUcomPivotamentoParcial(A, b)
```

```
x    -              y    =    1
x    -    1.00001   y    =    0

Solução obtida:
X = [100001.000,    100000.000]


x    -              y    =    1
x    -    0.99999   y    =    0

Solução obtida:
X = [-99999.000,    -100000.000]
```

- As soluções obtidas evidenciam que a precisão dos valores fornecidos influencia diretamente nos resultados obtidos na resolução de sistemas lineares.

Fatos como este ocorrem quando a matriz A do sistema está próxima de uma matriz singular e então o sistema é mal condicionado.

Dizemos que um sistema linear é bem condicionado se pequenas mudanças nos coeficientes e/ou nos termos independentes acarretarem pequenas mudanças na solução do sistema. Caso contrário, o sistema é dito mal condicionado.

Embora saibamos que uma matriz A pertence ao conjunto das matrizes não inversíveis se, e somente se, det(A) = 0, o fato de uma matriz A ter det(A) = 0 não implica necessariamente que o sistema linear que tem A por matriz de coeficientes seja mal-condicionado.

O número de condição de A, cond(A) = || A || || A^-1 ||, onde || . || é uma norma de matrizes, é uma medida precisa do bom ou mau condicionamento do sistema que tem A por matriz de coeficientes, pois demonstra-se que $\frac{1}{Cond(A)} = min\left\{\frac{||A-B||}{||A||}, \text{ tais que } B \text{ é não inversível}\right\}$

b) As matrizes de Hilbert, Hn, onde

h[i][j] = $\frac{1}{i+j-1}$

(b.1) - Use pacotes computacionais, que estimam ou calculam cond(A), para verificar que quanto maior for n, mais mal condicionada é Hn.

- Algoritmo em Python:

```python
import numpy as np

def matrizHilbert(n):
    H = np.zeros((n, n))
    for i in range(1, n+1):
        for j in range(1, n+1):
            H[i-1, j-1] = 1 / (i + j - 1)
    return H

for i in range(2, 11):
    # print("Matriz:")
    H = matrizHilbert(i)
    print("n = ", i)
    print("Cond = ",np.linalg.cond(H, float("inf")))
```

$$Hn = \begin{matrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \vdots & & & & \\ \frac{1}{n} & \frac{1}{n+1} & \cdots & & \frac{1}{2n-1} \end{matrix}$$

Exemplo matriz de Hilbert, para n = 5

$$Hn = \begin{matrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{matrix}$$

```
Resultados obtidos para norma 2
n =  2 Cond =  19.28147006790397
n =  3 Cond =  524.0567775860644
n =  4 Cond =  15513.73873892924
n =  5 Cond =  476607.25024259434
n =  6 Cond =  14951058.642254665
n =  7 Cond =  475367356.583129
n =  8 Cond =  15257575538.060041
```

```
n =  9 Cond =  493153756446.8762
n =  10 Cond =  16024416992541.715


Resultados obtidos para norma infinita
n =  2 Cond =  27.00000000000001
n =  3 Cond =  748.0000000000027
n =  4 Cond =  28375.00000000183
n =  5 Cond =  943656.0000063627
n =  6 Cond =  29070279.00379062
n =  7 Cond =  985194889.577766
n =  8 Cond =  33872792385.924484
n =  9 Cond =  1099651994744.017
n =  10 Cond =  35356847610517.12
```

Podemos ver nos resultados que o número de condição cresce conforme aumenta a ordem da matriz. Essa proporcionalidade pode ser observada até a matriz de Hilbert de ordem 10 conforme observado nos resultados

Portanto vemos que este tipo de matriz apresenta um maior mal condicionamento à medida que a ordem da matriz aumenta.

(b.2) - Resolva os sistemas Hnx = bn, n = 3, 4, 5, …, 10, onde bn é o vetor cuja i-ésima componente é

$$\sum_{j=1}^{n} \frac{1}{i+j-1}$$ **Desta forma a solução exata será x* = (1 1 … 1)^T.**

(b.3) - Analise seus resultados.

Algoritmo em Python:

```
import numpy as np
def fatoracaoLUcomPivotamentoParcial(A, b):
    n = len(A)
    cofator = np.zeros(n)
    L = np.zeros(n)
    U = np.zeros(n)
    for i in range(0, n):
        cofator[i] = i
    for k in range(0, n - 1):
        pivo = abs(A[k, k])
        r = k
        for i in range(k, n):
            if(abs(A[i, k]) > pivo):
                pivo = abs(A[i, k])
                r = i
        if(pivo == 0):
            print("A matriz é singular. O sistema não admite uma única
solução.")
            break
        if(r != k):
            aux = cofator[k]
            cofator[k] = cofator[r]
```

```
            cofator[r] = aux
            for j in range(0, n):
                aux = A[k, j]
                A[k, j] = A[r, j]
                A[r, j] = aux
        for i in range(k+1, n):
            m = A[i, k] / A[k, k]
            A[i, k] = m
            for j in range(k+1, n):
                A[i, j] = A[i, j] - m * A[k, j]
        L = np.eye(n)
        U = np.zeros((n, n))
        for i in range(n):
            for j in range(n):
                if(i > j):
                    L[i, j] = A[i, j]
                elif(j >= i):
                    U[i, j] = A[i, j]
    c = np.zeros(n)
    for i in range(0, n):
        r = int(cofator[i])
        c[i] = b[r]
    y = np.zeros(n)
    for i in range(0, n):
        soma = 0
        for j in range(0, i):
            soma = soma + A[i, j] * y[j]
        y[i] = c[i] - soma

    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        soma = 0
        for j in range(i, n):
            soma = soma + A[i, j] * x[j]
        x[i] = (y[i] - soma) / A[i, i]
    print("Resultado: ")
    # print(x)
    for i in range(n):
        print("%.3f\t" %x[i], end="")
def matrizHilbert(n):
    H = np.zeros((n, n))
    for i in range(1, n+1):
        for j in range(1, n+1):
            H[i-1, j-1] = 1 / (i + j - 1)
    return H
def formarBn(n):
    B = np.zeros(n+1)
    for i in range(1, n+1):
        soma = 0
        for j in range(1, n+1):
            soma += 1 / (i + j - 1)
            B[i-1] = soma
    # print(B)
```

```
        return B
    for i in range(3, 11):
        A = matrizHilbert(i)
        b = formarBn(i)
        fatoracaoLUcomPivotamentoParcial(A, b)
```

```
n =  3
Resultado: 1.000    1.000    1.000

n =  4
Resultado: 1.000    1.000    1.000    1.000

n =  5
Resultado: 1.000    1.000    1.000    1.000    1.000

n =  6
Resultado: 1.000    1.000    1.000    1.000    1.000    1.000

n =  7
Resultado: 1.000    1.000    1.000    1.000    1.000    1.000    1.000

n =  8
Resultado: 1.000    1.000    1.000    1.000    1.000    1.000    1.000    1.000

n =  9
Resultado:1.000    1.000    1.000    1.000    1.000    1.000    1.000    1.000
1.000

n =  10
Resultado:1.000    1.000    1.000    1.000    1.000    1.000    1.000    1.000
1.000    1.000
```

Para n = 3, temos:

```
H3 =  [1.              0.5              0.33333333 ]
      [0.5             0.33333333       0.25       ]
      [0.33333333      0.25             0.2        ]
b3 = [1.83333333      1.08333333        0.78333333 ]^T
H3 * (1 1 1)^T = b3
[1.              0.5              0.33333333 ] [1]      [1.83333333]
[0.5             0.33333333       0.25       ]*[1] =    [1.08333333]
[0.33333333      0.25             0.2        ] [1]      [0.78333333]
```

Temos que a soma dos elementos da linha i da matriz de Hilbert é igual elemento da linha i do vetor b.

Sendo h[i][j] = $\frac{1}{i+j-1}$ e

bn o vetor cuja i-ésima componente é $\sum_{j=1}^{n} \frac{1}{i+j-1}$