ESCOLA POLITÉCNICA DA USP

MAP3121 - MÉTODOS NUMÉRICOS E APLICAÇÕES

Exercício Programa 2

Dupla:	*NUSP:*
Lucas Domingues Boccia	11262320
Murillo Hierocles A. S. Teixeira	11325264

21 de julho de 2021

Sumário

1 Introdução

Este exercício programa tem como objetivo estudar o cálculo de auto-valores e auto-vetores de matrizes simétricas fazendo o uso da Transformação de Householder. Além de aplicar esta transformação, vamos retomar algumas funções utilizadas no Exercício Programa 1.

Por fim, será possível usar o código montado para calcular os modos de vibração de uma treliça plana e dessa forma analisar a evolução de um sistema dadas as suas condições iniciais.

A dupla optou por resolver o problema usando a linguagem de programação **Python 3**.

2 Recapitulando algoritmos da atividade anterior

2.1 Algoritmo QR

Recapitulando o algoritmo QR implementado na atividade anterior, temos que este tem como principal função encontrar autovalores e autovetores de matrizes tridiagonais simétricas. Vamos destacar novamente as peculiaridades desse algoritmo:

2.2 Sem desclocamentos espectrais

O algoritmo QR para a determinação dos auto-valores consiste em usar a fatoração QR de matrizes tridiagonais iterativamente da forma descrita no pseudocódigo abaixo.

$A^{(0)} = A; V^{(0)} = I_{n\times n}$

$k = 0$

repita

$A^{(k)} \to Q^{(k)}R^{(k)}$

$A^{(k+1)} = R^{(k)}Q^{(k)}$

$V^{(k+1)} = V^{(k)}Q^{(k)}$

$k = k + 1$

até a convergência

2.3 Com desclocamentos espectrais

De forma a aumentar a taxa de convergência do algoritmo, serão feitos deslocamentos espectrais a cada iteração do processo fazendo o uso da heurística de Wilkinson. O Algoritmo decorrente dessa modificação ficará da seguinte forma:

Seja $A^{(0)} = A \in \mathbb{R}^{n\times n}$, $\mathrm{V}^{(0)} = I$ e $\mu_0 = 0$.

$k = 0$

para $m = n, n-1, ..., 2$ **faça**:

repita

se $k > 0$ calcule μ_k pela heuristica de Wilkinson

$A^{(k)} - \mu_k I \longrightarrow Q^{(k)}$ e $R^{(k)}$

$A^{(k+1)} = R^{(k)}Q^{(k)} + \mu_k I$

$V^{(k+1)} = V^{(k)}Q^{(k)}$

$k = k + 1$

até que $|\beta^{(k)}_{m-1}| < \epsilon$

fim do para

$\Lambda = A^{(k)}$

$V = V^{(k)}$

Para esta atividade, como estamos visando eficiência do algoritmo, vamos utilizar o **algoritmo QR com deslocamento espectral**, visto que este apresenta uma convergência muito mais rápida.

3 Transformação de Householder

De forma a trabalhar com matrizes mais genéricas, foi proposta a utilização de uma transformação capaz de tridiagionalizar matrizes simétricas, para então utilizarmos os algoritmos anteriormente desenvolvidos. Vamos agora detalhar as etapas do algoritmo.

Seja $\omega \in R^n$, definimos a transformação de Householder $H_w : R^n \to R^n$ como: $H_w = I - \frac{2ww^T}{w \cdot w}$. Sabemos então que a transformação linear H é ortogonal e simétrica, resultando em $H_w^{-1} = H_w = H_w^T$. Utilizaremos essa propriedade logo a seguir. Sejam x e y dois vetores não nulos em R^n, temos a seguinte equação:

$$H_w x = x - 2\frac{w \cdot x}{w \cdot w}$$

Dada $A_{n \times n}$ uma matriz simétrica, podemos encontrar a matriz tridiagonal T, originada de A, pela expressão abaixo:

$$T = HAH^T$$

onde $H = H_{w_{n-2}} H_{w_{n-3}} ... H_{w_1} = H^T$

Visando a eficiência computacional do algoritmo, **não utilizaremos a abordagem matricial**, ou seja, não desenvolveremos o algoritmo baseando-se em multiplicações matriciais. Para contornar esse problema, vamos dispor da utilização de vetores, fazendo-se então operações sucessivas por coluna da matriz A, até completarmos a transformação de Householder.

Ao finalizar a execução do algoritmo, será possível obter as matrizes T e H^T. A partir disso, utilizaremos o **Algoritmo QR com deslocamento espectral** para a matriz T, obtendo: $T = V \Lambda V^T$. Assim teremos:

- Autovalores de A, dados por Λ
- Autovetores de A, dados por $H^T V$

4 Treliças planas

Visando desenvolver uma solução para um problema prático, foi sugerido aplicar os algoritmos desenvolvidos em um problema de treliças planas, composta por 28 barras, 12 nós articulados e 2 pontos de fixação, como mostra a figura abaixo:

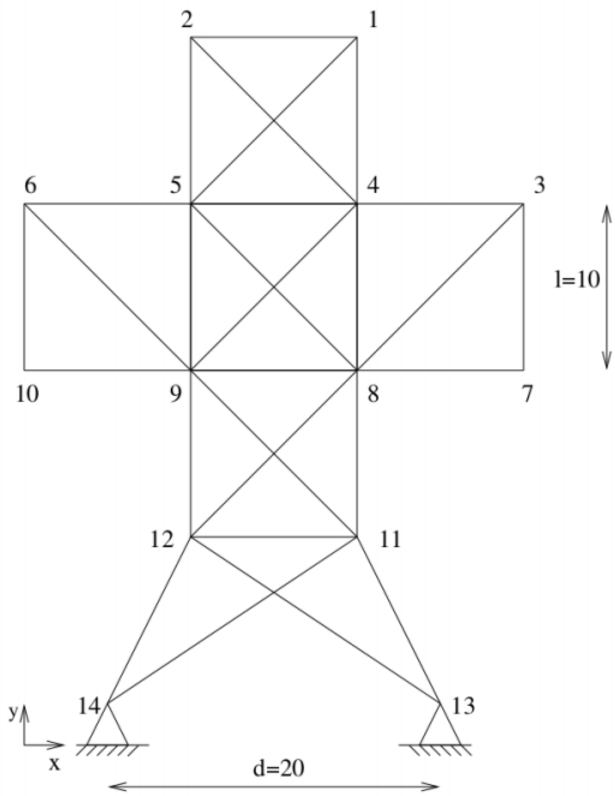

Figura 1: Treliça plana

Para a solução do problema, foi necessário desenvolver abordagem composta por diversas etapas, sendo estas: cálculo da matriz de massas, cálculo da matriz de rigidez individual de cada barra, cálculo da matriz de rigidez total da treliça e, por último, solução da equação diferencial de movimento. Estas etapas são descritas abaixo.

4.1 Matriz de massas

Como primeiro passo, vamos calcular a matriz de massas com base nos parâmetros fornecidos:

- ρ: densidade de massa
- A: seção transversal das barras
- L: valor padrão de comprimento

Dados esses parâmetros, e levando em consideração que as barras tem distribuição de massa homogênea, podemos considerar que a massa da barra se distribui igualmente para cada uma de suas extremidades. Sendo assim, isso irá facilitar o cálculo da massa presente em cada nó da treliça.

Como já foi passado o comprimento individual de cada barra, a partir do arquivo **input - c**, podemos elaborar um algoritmo simples capaz de calcular a massa de cada nó, totalizando uma matriz de massa M:

Para $i = 0, 1, 2, ...$ ***N° de barras - 1***

$$M(2(barras[i][0] - 1)) = M[i][0] + 0.5A\rho L[i]$$

$$M(2(barras[i][0] - 1) + 1) = M[i][0] + 0.5A\rho L[i]$$

$$M(2(barras[i][1] - 1)) = M[i][0] + 0.5A\rho L[i]$$

$$M(2(barras[i][1] - 1) + 1) = M[i][0] + 0.5A\rho L[i]$$

fim do para

4.2 Matriz de rigidez de cada barra

Dado a abordagem que descrevemos anteriormente, vamos dar continuidade na solução do problema, calculando então a *matriz de rigidez* $K_{4\times4}^{[i,j]}$. Para cada barra individualmente, vamos calcular uma matriz 4×4 correspondente. Assim, é possível chegar na seguinte equação matricial:

$$K^{[i,j]} = \frac{AE}{L_{[i,j]}} \begin{pmatrix} C^2 & CS & -C^2 & -CS \\ CS & S^2 & -CS & -S^2 \\ -C^2 & -CS & C^2 & CS \\ -CS & -S^2 & CS & S^2 \end{pmatrix}$$

com $C = cos(\theta)_{[i,j]}$ e $S = sin(\theta)_{[i,j]}$

Assim, é possível obter a matriz de rigidez 4×4 individual de cada barra, com base em seus respectivos ângulos $\theta_{[i,j]}$ com a horizontal, comprimento $L_{[i,j]}$, seção transversal A e módulo de elasticidade E. Como próximo passo, vamos calcular a matriz de rigidez total com a treliça completa, levando-se em consideração todas as barras.

4.3 Matriz de rigidez total da treliça

Para a construção da matriz de rigidez total da treliça, devemos pegar os valores de $K_{[i,j]}$ obtido na etapa anterior, em seguida vamos calculando seus valores de forma iterativa, ou seja, deve-se pegar a matriz $K_{[i,j]}$ e colocar na posição correspondente. As posições são as seguintes:

$$\begin{pmatrix} (2i-1, 2i-1) & (2i-1, 2i) & (2i-1, 2j-1) & (2i-1, 2j) \\ (2i, 2i-1) & (2i, 2i) & (2i, 2j-1) & (2i, 2j) \\ (2j-1, 2i-1) & (2j-1, 2i) & (2j-1, 2j-1) & (2j-1, 2j) \\ (2j, 2i-1) & (2j, 2i) & (2j, 2j-1) & (2j, 2j) \end{pmatrix}$$

Quando houver casos de sobreposição em que a matriz de uma barra sobreponha alguma outra matriz de outra barra, na composição da matriz total, deve-se então somar ambas as contribuições.

Sendo assim, é possível construir sucessivamente uma matriz de rigidez completa 24×24.

4.4 Solução da equação de movimento

A partir das etapas anteriores, é possível desenvolver uma equação diferencial de movimento, relacionando as contribuições de cada parcela da matriz K com a variação da posição de cada nó. Como vamos resolver esta matricialmente, primeiramente devemos ajustar as dimensões de cada matriz. Como a matriz M é uma matriz diagonal 12×12 e a matriz K é uma matriz simétrica 24×24, devemos então duplicar a matriz M, para que esta fique de acordo com as dimensões 24×24. A equação matricial obtida é mostrada abaixo:

$$M\ddot{x} + Kx = 0$$

Para desenvolvermos a solução da equação, devemos primeiramente fazer uma mudança na equação para termos coeficiente 1 apenas em $\ddot{x}$, assim a nossa equação ficará no seguinte formato:

$$\ddot{x} + \tilde{K}x = 0$$

onde $\tilde{K} = M^{-\frac{1}{2}}KM^{-\frac{1}{2}}$. Vale lembrar que poderíamos fazer $\tilde{K} = M^{-1}K$, porém essa transformação perderá a propriedade de simetria, tornando inviável a aplicação do algoritmo de Householder.

A partir de $\tilde{K}$ é possível aplicar a transformação de Householder, sendo possível obter as matrizes T e H.

$$\tilde{K} \xrightarrow{H_w} T, H$$

Ao se obter a matriz tridiagonal T e a matriz H, é possível encontrar os autovetores e autovalores originais de K, seguindo-se o raciocínio anteriormente apresentado, obtendo as seguintes expressões:

$$T = V\Lambda V^T$$

$$T \xrightarrow{QR} \Lambda, V$$

$$\text{Autovalores } (\omega) = \Lambda, \text{ Autovetores } (z) = H^T V$$

Dessa forma, a equação diferencial foi solucionada, sendo possível encontrar as frequências de vibração, que correspondem aos autovalores obtidos. Da mesma forma, os autovetores obtidos correspondem aos modos de vibração para suas respectivas frequências. Por último, é possível encontrar a solução x(t), tal que:

$$x(t) = ze^{i\omega t}, \text{ sendo } \omega \text{ a frequência de vibração e } z \text{ seu respectivo modo}$$

5 Código

Aqui serão descritas as funções criadas em Python para implementar o as transformações Householder e resolver o problema das treliças planas.

5.1 Bibliotecas usadas

Para este segundo exercício foi utilizada apenas a biblioteca NumPy, para trabalhar com aritmética de vetores, matrizes e para ler e escrever dados.

```
import numpy as np
```

Código 1: Importando as bibliotecas

5.2 Rotinas retiradas do EP1

5.2.1 Funções reaproveitadas de forma integral

Foram aproveitadas sem nenhuma mudança as funções *calculaQ*, responsável pelo cálculo das matriz de Rotação de Givens; *fatoracaoQR*, que retornava as matrizes Q e R dada uma matriz tridiagonal simétrica A de entrada; e por último também foi incluída a função *calculaMi*, responsável por calcular o fator μ da heurística de Wilkinson.

5.2.2 Funções modificadas

O código que implementava o algoritmo QR foi levemente modificado para este segundo exercício programa. Como as matrizes analisadas agora são simétricas cheias, é interessante passar a Matriz H^T como parâmetro para substituir a identidade no valor inicial da matriz de autovetores. Isso foi realizado como consta no Código 2.

```
def algoritmoQR(A, eps, H, desloc):
    Ld = A
    n = len(A)
    V = H
    k = 0
    uk = 0
    AutoValores = np.zeros(n)
    AutoVetores = np.zeros(n**2).reshape((n, n))
    for i in range(n-1):
      m = n - i
      while abs(Ld[m-1][m-2]) >= eps:
        if (k > 0 and desloc==True):
          uk = calculaMi(Ld, m)
        Desloc = uk*np.identity(m)
        [Q, R] = fatoracaoQR(Ld - Desloc)
        Ld = np.matmul(R, Q) + Desloc
        k += 1
        V[:, :m] = np.matmul(V[:, :m], Q)
      AutoValores[i] = Ld[-1][-1]
      Ld = Ld[:len(Ld)-1,:len(Ld)-1]
    AutoValores[i+1] = Ld[0]
    AutoValores = np.flip(AutoValores)
    AutoVetores = V

    return [AutoValores, AutoVetores, k]
```

Código 2: Algoritmo QR modificado

5.3 Transformação de Householder

A Transformação de Householder foi implementada como descrito na Seção 3. Ela está dividida em 4 funções distintas que serão apresentadas aqui. A primeira calcula o vetor w_i usado para a transformação. A função seguinte é responsável por calcular o produto $H^T H_{wi}$ a cada iteração do algoritmo de tridiagonalização. A próxima calcula a matriz AH_{w_i} usando operações entre vetores.

Por último é implementado o algoritmo proposto para a tridiagonalização, fazendo uso de sucessivas Transformações de Householder.

5.3.1 Cálculo do vetor $\boldsymbol{w}_i$

```
def create_wi(A):
    col1 = A[:,0].copy()
    signal = -1 if col1[1] >=0 else 1
    factor = signal*np.sqrt(col1[1:].dot(col1[1:]))
    w = col1; w[0] = 0; w[1] = col1[1] - factor
    return w
```

Código 3: Rotina para o cálculo de w_i

5.3.2 Cálculo de H^T

```
def createHT(X, w):
    IH = np.zeros(X.T.shape)
    for i in range(0, len(X.T)):
        col = X[:, i].copy()
        IH[i] = col - (2*w.dot(col)/w.dot(w))*w
    return IH
```

Código 4: Rotina para o cálculo de H^T a cada iteração

5.3.3 Produto de uma matriz por H_{w_i} usando vetores

```
def householder(A, w):
    col1 = A[:,0].copy()
    col1 = col1 - (2*(w.dot(col1))/w.dot(w))*w
    col1[2:] = np.zeros(len(col1)-2)

    AH = np.zeros(A.shape)
    AH[0] = col1
    for i in range(len(A)-1):
        col = A[:, i+1]
        AH[i+1] = col - (2*(w.dot(col))/w.dot(w))*w
    return AH
```

Código 5: Transformação de Householder usando vetores

5.3.4 Tridiagonalização de matrizes simétricas

```python
def tridiagonalize(A):
    n = len(A)
    T = A.copy()
    HT = np.eye(n)

    for i in range(n-2):
        w = create_wi(T[i:, i:])
        tempT = householder(T[i:, i:], w)
        T[i:, i:] = householder(tempT, w)
        HT[:, i:] = createHT(HT[:, i:].T, w)

    return T, HT
```

Código 6: Procedimento para a tridiagonalização de matrizes simétricas

5.4 Testes

Para realizar o teste dos algoritmos anteriores foram lidos dois arquivos contendo matrizes simétricas e para cada um deles o programa apresenta os autovalores e autovetores da matriz. Também foram foi feita a verificação de que $Av = \lambda v$ e da ortogonalidade da matriz de autovetores. Os códigos abaixo realizam esses procedimentos, o primeiro lê as matrizes e o seguinte realiza os testes.

5.4.1 Leitura das matrizes

```python
def readMatrix(fileName):
    f = open(fileName, 'r')
    n = int(f.readline().strip())
    M = []
    for i in range(n):
        line = f.readline().strip().replace('  ', ' ').split(' ')
        M.append([float(element) for element in line])
    f.close()
    M = np.array(M)
    return M
```

Código 7: Leitura dos inputs de matrizes

5.4.2 Realização dos testes

```
def task1(question):
    if(question == 'A'):
        M = readMatrix('./inputs/input-a')

    if(question == 'B'):
        M = readMatrix('./inputs/input-b')

    print('Matriz em análise = ')
    print(M)
    T, H = tridiagonalize(M)
    L, V, _ = algoritmoQR(T, 1e-6, H, True)
    print('\nAutovalores calculados = ')
    print(np.around(L, 5))
    print('\nAutovetores calculados = ')
    print(np.around(V, 5))
    for i in range(len(M)):
        print(f'\nVerificação Av = lv para l = {np.around(L[i], 5)}')
        print(f'\n     A x {i+1}° autovetor  = ')
        print('     ', np.around(M@V[:, i], 5))
        print(f'\n     {i+1}° autovalor x {i+1}° autovetor  = ')
        print('     ', np.around(L[i]*V[:, i], 5))
    print('\nVerificação da ortogonalidade: V x Vt = I => Vt = V^(-1)')
    print('V x Vt = ')
    print(np.around((np.matmul(V, V.T)), 5))
```

Código 8: Rotina de testes

5.5 Solução do problema de treliças planas

Vamos agora implementar numericamente a solução proposta na seção anterior, com base em cada etapa desenvolvida. Inicialmente devemos ler os dados do arquivo **input-c**. Em seguida, devemos desenvolver cada etapa descrita anteriormente, ou seja, devemos implementar os passos de construção das matrizes de massa, rigidez e rigidez total, além da solução propriamente dita, em que há necessidade de se calcular a matriz $\tilde{K} = M^{-\frac{1}{2}} K M^{-\frac{1}{2}}$, e seguidamente utilizar os algoritmos desenvolvidos no EP1.

5.5.1 Leitura dos dados das treliças

```
def loadTrussesData(fileName):
    f = open(fileName, "r")
    totalJoints, freeJoints, numberOfBars = f.readline().split(" ")[:3]
    density, A, E = f.readline().split(" ")[:3]
    params = [int(totalJoints),
              int(freeJoints),
              int(numberOfBars),
              float(density),
              float(A),
              float(E)*1e9]
    lengths = []
    angles = []
    bars = []
    for i in range(int(numberOfBars)):
        infos = f.readline().rstrip().split(" ")
        bars.append(
            [int(infos[0]), int(infos[1])]
        )
        angles.append(float(infos[2]))
        lengths.append(float(infos[3]))
    f.close()
    return params, lengths, angles, bars
```

Código 9: Rotina de leitura dos dados das treliças

5.5.2 Cálculo da matriz de massas

```
def createMassMatrix(numberOfFreeJoints, numberOfBars, density, A, bars, lengths):
    M = np.zeros(numberOfBars)

    for i in range(numberOfBars):
        M[2*(bars[i][0]-1)] += 0.5*A*density*lengths[i]
        M[2*(bars[i][0]-1)+1] += 0.5*A*density*lengths[i]
        M[2*(bars[i][1]-1)] += 0.5*A*density*lengths[i]
        M[2*(bars[i][1]-1)+1] += 0.5*A*density*lengths[i]
    return M[:-4]
```

Código 10: Rotina para o cálculo da matriz de massas

5.5.3 Cálculo da matriz de rigidez de cada barra

```
def createBarStiffnessMatrix(A, E, angle, length):
    C = np.cos(np.deg2rad(angle))
    S = np.sin(np.deg2rad(angle))
    vectorCS = np.array([-C, -S, C, S]).reshape(4, 1)

    K = (A*E/length)*vectorCS@vectorCS.T
    return K
```

Código 11: Rotina para o cálculo da matriz de rigidez de uma barra

5.5.4 Cálculo da matriz de rigidez total das treliças (K)

```
def createTrussesStiffnessMatrix(params, bars, lengths, angles):
    numberOfFreeJoints = params[1]
    numberOfBars = params[2]
    A = params[4]
    E = params[5]

    K = np.zeros((2*numberOfFreeJoints, 2*numberOfFreeJoints))
    for i in range(numberOfBars):
        firstJoint = bars[i][0];
        secondJoint = bars[i][1];
        length, angle = [lengths[i], angles[i]]
        Kij = createBarStiffnessMatrix(A, E, angle, length)

        K[2*(firstJoint-1), 2*(firstJoint-1)] += Kij[0, 0]
        K[2*(firstJoint-1)+1, 2*(firstJoint-1)] += Kij[1, 0]
        K[2*(firstJoint-1), 2*(firstJoint-1)+1] += Kij[0, 1]
        K[2*(firstJoint-1)+1, 2*(firstJoint-1)+1] += Kij[1, 1]

        if(secondJoint <= numberOfFreeJoints):
            K[2*(firstJoint-1), 2*(secondJoint-1)] += Kij[0, 2]
            K[2*(firstJoint-1)+1, 2*(secondJoint-1)] += Kij[1, 2]
            K[2*(firstJoint-1), 2*(secondJoint-1)+1] += Kij[0, 3]
            K[2*(firstJoint-1)+1, 2*(secondJoint-1)+1] += Kij[1, 3]

            K[2*(secondJoint-1), 2*(secondJoint-1)] += Kij[2, 2]
            K[2*(secondJoint-1)+1, 2*(secondJoint-1)] += Kij[3, 2]
            K[2*(secondJoint-1), 2*(secondJoint-1)+1] += Kij[2, 3]
```

```
            K[2*(secondJoint-1)+1, 2*(secondJoint-1)+1] += Kij[3, 3]

            K[2*(secondJoint-1), 2*(firstJoint-1)] += Kij[2, 0]
            K[2*(secondJoint-1)+1, 2*(firstJoint-1)] += Kij[3, 0]
            K[2*(secondJoint-1), 2*(firstJoint-1)+1] += Kij[2, 1]
            K[2*(secondJoint-1)+1, 2*(firstJoint-1)+1] += Kij[3, 1]
    return K
```

Código 12: Rotina para o cálculo da matriz de rigidez da estrutura

5.5.5 Cálculo da matriz $\tilde{K}$

```
def createKtilmatrix(K, M):
    Ktil = np.zeros(K.shape, dtype='float')
    for i in range(len(K)):
        Ktil[i] = np.sqrt(1/M[i])*K[i]
    for i in range(len(K)):
        Ktil[:, i] = np.sqrt(1/M[i])*Ktil[:, i]
    return Ktil
```

Código 13: Rotina para o cálculo da matriz $\tilde{K}$

5.5.6 Resolução da segunda tarefa

```
def task2():
    # Solução da equação da Tarefa 2
    # Import dos parâmetros dos arquivos
    params, lengths, angles, bars = loadTrussesData("./inputs/input-c")
    numberOfJoints = params[0]
    numberOfFreeJoints = params[1]
    numberOfBars = params[2]
    density = params[3]
    A = params[4]

    # Parâmetros da solução da equação:
    # MX" + KX = 0
    M = createMassMatrix(numberOfFreeJoints, numberOfBars,  density, A, bars, lengths)
    K = createTrussesStiffnessMatrix(params, bars, lengths, angles)

    Ktil = createKtilmatrix(K, M)
```

```python

    T, H = tridiagonalize(Ktil)

    L, Y, _ = algoritmoQR(T, 1e-6, H, desloc=True)
    w = np.sqrt(L)

    Z = np.zeros(Y.shape)
    for i in range(len(K)):
        Z[i] = np.sqrt(1/M[i])*Y[i]

    n = 5
    freqs = np.sort(w)[0:n]
    modosDeVibracao = np.array([Z[:, list(w).index(freqs[0])],
                                Z[:, list(w).index(freqs[1])],
                                Z[:, list(w).index(freqs[2])],
                                Z[:, list(w).index(freqs[3])],
                                Z[:, list(w).index(freqs[4])]])

    for i in range(n):
        print(f'\nFrequência {i+1} =', np.around(freqs[i], 3), 'rad/s')
        print(f'Modo de vibração {i+1} = ')
        print(modosDeVibracao[i])

    resposta = input('\nDeseja conferir os dados para animação (S/N)? ')
    if resposta.upper().strip()=='S':
        createOutputFilesForAnimation(freqs, modosDeVibracao)
```

Código 14: Código usado para resolver o problema das treliças

5.5.7 Interface com o usuário

```
def userInterface():
  print('EP2 - Algoritmo QR para matrizes simétricas')
  print('Dupla:')
  print(' Lucas Domingues Boccia, NUSP 11262320')
  print(' Murillo Hierocles Alves de Sá Teixeira, NUSP 11325264')
  print('\n-------------')
  querAnalisar = True

  while(querAnalisar):
    print('\nQuestões:')
    print('A. Teste da tridiagonalização, item A')
    print('B. Teste da tridiagonalização, item B')
    print('C. Aplicação: Treliças Planas')
    print('\n-------------')

    questaoInvalida = True
    while(questaoInvalida):
        questao = input('\nQual a questão de interesse (A/B/C)? ')
        print('\n')

        questaoInvalida = False
        if (questao.upper().strip() == 'A' or questao.upper().strip() == 'B'):
            task1(questao.upper().strip())

        elif (questao.upper().strip() == 'C'):
            task2()

        else:
            questaoInvalida = True
            print('Questão inválida!')

    print('\n-------------')
    resposta = input('\nDeseja testar outra questão (S/N)? ')
    querAnalisar = True if resposta.upper().strip()=='S' else False
  print('\n-------------')
  print('\nFim da execução!')
```

Código 15: Código usado para interagir com o usuário do programa

6 Resultados

6.1 Tarefa 1 - Cálculo de auto-valores e auto-vetores

6.1.1 Item 1.a)

A primeira tarefa do exercício programa era a de calcular os auto-valores e auto-vetores para uma matriz simétrica A que com o seguinte formato:

$$A = \begin{pmatrix} 2 & 4 & 1 & 1 \\ 4 & 2 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 1 & 2 & 1 \end{pmatrix}$$

Com o algoritmo apresentado, foi possível encontrar os seguintes autovalores:

$$\begin{pmatrix} 7 \\ -2 \\ 2 \\ -1 \end{pmatrix}$$

A partir desses autovalores, encontramos os seguintes autovetores:

$$\begin{pmatrix} 0.63246 & 0.70711 & 0.31623 & 0.00000 \\ 0.63246 & -0.70711 & 0.31623 & 0.00000 \\ 0.31623 & 0.00000 & -0.63246 & -0.70711 \\ 0.31623 & 0.00000 & -0.63246 & 0.70711 \end{pmatrix}$$

6.1.2 Item 1.b)

Seguindo-se um raciocínio semelhante ao item a), temos que agora a matriz A é dada por:

$$A = \begin{pmatrix} n & n-1 & n-2 & \cdots & 2 & 1 \\ n-1 & n-1 & n-2 & \cdots & 2 & 1 \\ n-2 & n-2 & n-2 & \cdots & 2 & 1 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 2 & 2 & 2 & \cdots & 2 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{pmatrix}$$

Essa matriz A possui autovalores dados por:

$$\lambda_i = \frac{1}{2}\left[(1 - cos\frac{(2i-1)\pi}{2n+1}\right]^{-1}, i = 1, 2, ..., n$$

Usando n = 20, como sugerido no enunciado, foi possível obter os seguintes autovalores:

$$\begin{pmatrix} 170.40427 \\ 19.00810 \\ 6.89678 \\ 3.56048 \\ 2.18808 \\ 1.49399 \\ 1.09545 \\ 0.84612 \\ 0.68025 \\ 0.56477 \\ 0.48156 \\ 0.42003 \\ 0.37369 \\ 0.33836 \\ 0.31129 \\ 0.29061 \\ 0.27504 \\ 0.26369 \\ 0.25147 \\ 0.25596 \end{pmatrix}$$

Considerando os valores obtidos pelo algoritmo e comparando estes com os valores teóricos obtidos pela expressão na página anterior, foi possível concluir que o algoritmo está de fato funcional, isto é, os valores estão sendo compatíveis com uma precisão decente.

Para os autovetores encontramos o resultado, entretanto por ser uma matriz muito extensa, optamos por não colocar esta no relatório.

6.2 Tarefa 2 - Treliças Planas

Para a resolução do problema de Treliças Planas proposto, seguimos o raciocínio e equacionamento expostos na seção 4. Além disso, a implementação numérica para resolução dessa equação está detalhada na seção 5. Dessa forma, foi possível obter as 5 menores frequências e seus respectivos modos de vibração. Vamos apresentar estes abaixo.

Frequências

$$\begin{pmatrix} 24.592548 & 92.012445 & 94.703365 & 142.809697 & 150.822127 \end{pmatrix}$$

Modos de vibração

$$\begin{pmatrix}
0.003496 & -0.000006 & -0.000779 & 0.003869 & -0.000090 \\
-0.000612 & -0.002186 & 0.000808 & -0.001930 & 0.001923 \\
0.003496 & 0.000006 & -0.000779 & 0.003869 & 0.000090 \\
0.000612 & -0.002186 & -0.000808 & 0.001930 & 0.001923 \\
0.002269 & 0.000473 & 0.000641 & -0.001488 & 0.001505 \\
-0.001813 & -0.002985 & 0.002838 & 0.002681 & -0.003388 \\
0.002248 & 0.000176 & 0.000875 & -0.000690 & 0.000434 \\
-0.000596 & -0.002051 & 0.000714 & -0.001143 & 0.001797 \\
0.002248 & -0.000176 & 0.000875 & -0.000690 & -0.000434 \\
0.000596 & -0.002051 & -0.000714 & 0.001143 & 0.001797 \\
0.002269 & -0.000473 & 0.000641 & -0.001488 & -0.001505 \\
0.001813 & -0.002985 & -0.002838 & -0.002681 & -0.003388 \\
0.000998 & -0.000004 & 0.002484 & -0.000557 & -0.000689 \\
-0.001817 & -0.003087 & 0.002941 & 0.002913 & -0.003717 \\
0.000996 & -0.000004 & 0.002397 & -0.000512 & -0.000628 \\
-0.000507 & -0.001744 & 0.000311 & 0.000020 & 0.001129 \\
0.000996 & 0.000004 & 0.002397 & -0.000512 & 0.000628 \\
0.000507 & -0.001744 & -0.000311 & -0.000020 & 0.001129 \\
0.000998 & 0.000004 & 0.002484 & -0.000557 & 0.000689 \\
0.001817 & -0.003087 & -0.002941 & -0.002913 & -0.003717 \\
0.000042 & 0.000042 & 0.001157 & -0.000231 & 0.000050 \\
-0.000296 & -0.001097 & -0.000056 & 0.000045 & 0.000679 \\
0.000042 & -0.000042 & 0.001157 & -0.000231 & -0.000050 \\
0.000296 & -0.001097 & 0.000056 & -0.000045 & 0.000679
\end{pmatrix}$$

7 Tarefa Bônus - Animação de treliças vibrando

A tarefa bônus tinha como proposta a visualização dos resultados encontrados até agora com os algoritmos desenvolvidos. Devido à possibilidade de fazer em outras linguagens, a dupla optou por realizar a animação usando a biblioteca p5.js do JavaScript. No entanto, os dados foram criados no programa principal em Python, sem usar outras bibliotecas além do NumPy para exportar arquivos.

Na função abaixo são criados os arquivos. Nas linhas 195 a 199 é implementado o algoritmo para a solução do problema (descrita no enunciado do exercício programa). Nele são gerados os deslocamentos de cada nó em função do tempo.

```
181 def createOutputFilesForAnimation(freqs, modosDeVibracao):
182         modo = input('\nEscolha um modo de vibração (1, 2, 3, 4 ou 5): ')
183         numeroDeFiguras = input('\nQual o número de figuras que se deseja criar? ')
184         tempoEntreFiguras = input('\nQual o tempo entre figuras (em segundos)? ')
185 
186         modo = int(modo)
187         numeroDeFiguras = int(numeroDeFiguras) + 1
188         tempoEntreFiguras = float(tempoEntreFiguras)
189 
190 
191         tempo = np.linspace(0, numeroDeFiguras*tempoEntreFiguras, numeroDeFiguras)
192         x = np.zeros(24*(len(tempo)+1)).reshape((24, len(tempo)+1))
193 
194         headerStr = "       i"
195         x[:, 0] = range(1, 25)
196         for t in range(len(tempo)):
197             headerStr += f'   X({t+1:05d})'
198             print()
199             x[:, t+1] = (modosDeVibracao[modo-1]*np.cos(freqs[modo]*tempo[t])).T
200 
201         # Criação dos arquivos que serão utilizados pela
202         # aplicação web para a execução da animação
203 
204         # Arquivo para a verificação do algoritmo utilizado
205         np.savetxt('./outputs/x(t)', x, delimiter=" ", fmt=("%10.7f"), header=headerStr)
206 
207 
208 
```

```python
        #Arquivo JavaScript com os modos de vibração
        # (para evitar o uso de bibliotecas adicionais!)
        np.savetxt('./outputs/modos_de_vibracao.js', modosDeVibracao, delimiter=" ")
        with open('./outputs/modos_de_vibracao.js','r') as contents:
            save = contents.read()
        with open('./outputs/modos_de_vibracao.js','w') as contents:
            contents.write("let modosDeVibracao = `")
        with open('./outputs/modos_de_vibracao.js','a') as contents:
            contents.write(save)
            contents.write('`')

        #Arquivo JavaScript com as frequências de vibração
        # (para evitar o uso de bibliotecas adicionais!)
        np.savetxt('./outputs/frequencias.js', freqs, delimiter=" ")
        with open('./outputs/frequencias.js','r') as contents:
            save = contents.read()
        with open('./outputs/frequencias.js','w') as contents:
            contents.write("let frequencias = `")
        with open('./outputs/frequencias.js','a') as contents:
            contents.write(save)
            contents.write('`')

        #Arquivo JavaScript com o input-c
        # (para evitar o uso de bibliotecas adicionais!)
        file = open("./outputs/input-c.js", "w")
        with open('./inputs/input-c','r') as contents:
            save = contents.read()
        with open("./outputs/input-c.js", "w") as contents:
            contents.write("let inputC = `")
        with open("./outputs/input-c.js",'a') as contents:
            contents.write(save)
            contents.write('`')
```

Código 16: Rotina de testes

De forma a conseguir visualizar a animação em uma taxa de quadros alta, o mesmo algoritmo foi implementado em JavaScript, na função calculateX abaixo, usando os arquivos JavaScript gerados na função acima.

```
const calculateX = (vibrationMode, frequency, time) => {
  x = []
  for(i = 0; i < vibrationMode.length; i++)
    x[i] = vibrationMode[i]*cos(frequency*time)
  return x
}
```

Código 17: Algoritmo para o cálculo da posição dos nós em função do modo de vibração e de um instante no tempo

Esta função cria subsídios para a atualização da posição dos nós da estrutura em função do tempo.

```
function Joint(x, y, numero) {
  this.xCentral = x;
  this.yCentral = y;
  this.x = x;
  this.y = y;
  this.size = 10;
  this.numero = str(numero);

  this.show = () => {
    strokeWeight(this.size)
    stroke(255);
    point(this.x, this.y);
    noStroke();
    fill(0);
  }

  this.update = (dx, dy) => {
    this.x = this.xCentral + dx;
    this.y = this.yCentral + dy;
  }
}
```

Código 18: Objeto do nó da estrutura. A função *this.update* atualiza é responsável por atualizar a posição dos nós

A parte central da aplicação que gera a animação é a que segue, onde são calculados e dispostos os nós na tela.

```
// Desenho dos nós
for (k = 1; k <= nosTotais; k++)
  joints[k].show();

angleMode(RADIANS)

x = calculateX(vibrationMode, frequency, t);
// Deslocamentos
for (k = 1; k <= nosTotais - 2; k++) {
    joints[k].update(
      scl * amplitude * x[ 2*(k - 1) ],
      scl * amplitude * x[ 2*(k - 1) + 1 ]
    );
}
```

Código 19: Trecho da função draw() do arquivo animation.js.

A animação e o resto do código que a gera estão junto ao programa.

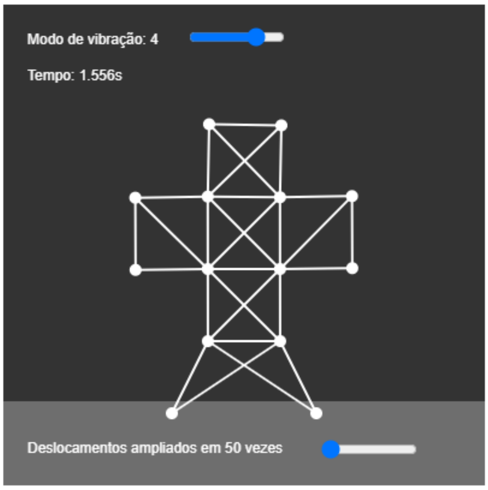

Figura 2: Print da animação gerada

8 Conclusão

A partir dos resultados obtidos, é possível concluir que o algoritmo desenvolvido nesse exercício programa é mais robusto e genérico do que o anterior, isso se dá por conta da propriedade da Transformação de Householder ter como entrada uma matriz simétrica, e não uma matriz tridiagonal simétrica. Além disso, percebe-se que a nova implementação está mais otimizada, resultando em melhor desempenho e uma resposta com boa precisão. Também é interessante reparar que o algoritmo e o ferramental desenvolvido durante a execução do exercício programa traz um entendimento maior da aplicação dos métodos numéricos na resolução de problemas.