

MAP3121 - Métodos Numéricos e Aplicações
Escola Politécnica da Universidade de São Paulo

Exercício Programa 2

Autovalores e Autovetores de Matrizes Reais
Simétricas

Gabriel Macias de Oliveira, NUSP 11260811, Eng. Elétrica
Rodrigo Ryuji Ikegami, NUSP 10297265, Eng. Elétrica

São Paulo,
2021.

Sumário

1	Introdução	2
1.1	Ferramentas Utilizadas	2
1.2	Execução dos <i>Scripts</i>	3
2	Implementação	6
2.1	As Transformações de Householder	6
2.1.1	Função de Implementação da Tridiagonalização de uma Matriz Real Simétrica	7
2.2	O Algoritmo QR	9
2.3	Leitura de Matrizes em Arquivos	9
2.4	Leitura de Treliças em Arquivos	10
3	Construção dos Testes	12
3.1	Teste A: Matriz de Autovalores Inteiros Conhecidos	12
3.2	Teste B: Matriz de Autovalores dados por Fórmula	14
3.3	Aplicação do Algoritmo a Treliças Planas	15
	Referências	22

1 Introdução

Matrizes reais simétricas surgem comumente no estudo de aplicações de métodos em engenharia. Além disso, seus autovalores e autovetores carregam informações sobre diversos modelos e descrições de muito interesse na análise e projeto de sistemas.

Neste Exercício Programa, implementamos as *Transformações de Householder* aplicadas a *Matrizes Reais Simétricas*, de modo a obter uma matriz semelhante tridiagonal simétrica. Além disso, utilizamos o *Algoritmo QR* implementado no EP anterior para obter os autovalores e autovetores da matriz original, utilizando a saída do algoritmo implementado neste EP. Abordaremos aspectos mais formais da implementação, como a descrição em código, bem como características de desempenho e acurácia. Por fim, o método será aplicado à solução de um Sistema de EDOs Lineares de Segunda Ordem.

Nosso objetivo é, dada uma matriz $A \in \mathbb{R}^{n \times n}$ real simétrica, encontrar seus autovalores $\{\lambda_1, \lambda_2, \dots, \lambda_n\}$ e seus respectivos autovetores $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$, de forma eficiente e atendendo limites de erro e convergência. Recordamos que, segundo [1], o Teorema Espectral garante que uma matriz real simétrica é ortogonalmente diagonalizável, todos os seus autovalores são reais e podemos escolher os respectivos autovetores de modo a formar uma base ortonormal do \mathbb{R}^n .

Na Seção 2, detalha-se a implementação do Algoritmo de tridiagonalização. Primeiro, há uma descrição da base matemática que orienta a construção da função, acompanhada pelo código e alguns comentários. A execução dos testes propostos é detalhada na Seção 3, em que se retrata especificamente como foram implementados e o que se espera observar nas variáveis de retorno. Também é descrita a interface de comando (CLI) que acompanha o programa. Ao final, a Seção ?? contém a exibição, análise e discussão dos resultados dos testes.

1.1 Ferramentas Utilizadas

Foram utilizadas as seguintes ferramentas para construção do código:

- Linguagem de Programação: *Python 3.7.9+*
- Bibliotecas Externas:
 - *numpy*, para trabalhar com aritmética de vetores
 - *matplotlib*, para produção de gráficos e animações
- IDE: *Visual Studio Code*
- Desenvolvimento Paralelo: *Git*

Além das bibliotecas externas, utilizaram-se as bibliotecas nativas *math*, para funções matemáticas básicas, *typing*, para utilizar tipos estáticos em *Python*, e *sys* para personalização da CLI.

Todos os testes em que são envolvidas métricas de tempo / número de iterações foram executados com base em um AMD Ryzen 5 3600X @ 4.2 GHz, portanto sendo suscetíveis a variações.

Todo o código está concentrado no arquivo `main.py`, cujos detalhes de execução se encontram em sequência e no arquivo `LEIA-ME.txt`.

Este relatório foi tipografado em L^AT_EX.

1.2 Execução dos *Scripts*

Estando o *Python* atualizado para uma versão compatível, isto é, 3.7.9 ou mais recente, deve-se certificar que ambas bibliotecas *numpy* e *matplotlib* estejam instaladas. Caso contrário, basta executar `pip install -r requirements.txt` em um terminal, para recebê-las.

O arquivo principal deve ser executado no mesmo diretório em que foi descompactado, utilizando o comando `python main.py`. A exibição do terminal deve ser da CLI que acompanha o programa, conforme a Figura 1.

[illegible]

Figura 1: Exibição inicial da Command Line Interface (CLI) do programa.

Na visualização principal do programa, deve-se escolher uma entre quatro rotinas de exibição. As 2 primeiras rotinas exibem autovalores, autovetores e outras grandezas relevantes para os 2 testes propostos no enunciado em [2]. A terceira rotina exibe autovalores, autovetores e animações relevantes para a aplicação ao problema de treliças planas descrito em [2], conforme se esclarecerá em sequência. Por último, a quarta rotina permite a diagonalização de uma matriz real simétrica arbitrária.

Ao escolher a primeira rotina, são exibidos, em sequência, os resultados para cada autovetor da matriz do primeiro teste. Entre uma execução e outra, deve-se pressionar [ENTER] para prosseguir para a próxima entrada, como se observa na Figura 2 abaixo. Desejamos que o usuário possa analisar os autovetores da matriz propostas e todas as suas características relevantes.

```
Windows PowerShell
>> Você selecionou o teste A proposto pelo relatório.
- Realizando leitura de arquivo: 'input-a'

Matriz de entrada:
[[ 2.  4.  1.  1.]
 [ 4.  2.  1.  1.]
 [ 1.  1.  1.  2.]
 [ 1.  1.  2.  1.]]

Matriz tridiagonalizada:
[[ 2.      -4.242641  0.      0.      ]
 [-4.242641  3.      1.414214  0.      ]
 [ 0.      1.414214  2.     -0.      ]
 [ 0.      0.      -0.     -1.     ]]

Autovalores:
- Encontrados: [ 7.  2. -1. -2.]
- Esperados:   [ 7.  2. -1. -2.]

Matriz de Autovetores:
      v1      v2      v3      v4
[[ 0.632456  0.787187  0.316228  0.      ]
 [ 0.632456 -0.787187  0.316228  0.      ]
 [ 0.316228  0.      -0.632456 -0.787187 ]
 [ 0.316228  0.      -0.632456  0.787187 ]]

OBS.: Caso A não seja renderizado corretamente em seu terminal, este
char é um Lambda, indicando o autovalor da matriz A.

A * v1 = [ 4.427189  4.427189  2.213594  2.213594]
λ * v1 = [ 4.427189  4.427189  2.213594  2.213594]

Proporção Entrada Depois/Antes Transformação:
Esperada: 7.0000
Obtida:   [ 7.  7.  7.  7.]

Pressione [ENTER] para continuar.
```

Figura 2: Resultado da escolha da primeira rotina de testes.

A segunda rotina exibe as mesmas informações que a primeira, porém considerando a matriz de entrada do teste 2. O que varia, de uma execução para outra, é o tamanho da matriz exibida, bem como a própria matriz a ser diagonalizada.

A terceira rotina gera, além de informações relevantes à matriz da aplicação proposta no enunciado em [2], animações (caso desejado) da treliça com seus nós vibrando aos 5 modos naturais de menor frequência.

A quarta rotina exibe as mesmas informações que a primeira e a segunda, porém considerando uma matriz de entrada arbitrária. Ela permite ao usuário a inserção de uma matriz real simétrica, cujos elementos podem ser inseridos manualmente, 1 a 1, ou passados por um arquivo de entrada, com formatação igual à de *input-a* ou *input-b*.

Nota: Ao fornecer as entradas, é essencial que o usuário pressione [ENTER] entre uma entrada e outra. Logo, o padrão de digitação deve ser, por exemplo 1 [ENTER] 2 [ENTER] etc, para garantir que todas as entradas sejam lidas corretamente. Um exemplo de execução está na Figura 3 abaixo.

2 Implementação

2.1 As Transformações de Householder

Conforme [2], as *Transformações de Householder* são transformações lineares ortogonais $H_w : \mathbb{R}^n \rightarrow \mathbb{R}^n$ da forma $H_w = I - \frac{2ww^T}{w \cdot w}$ que operam sobre o espaço de vetores como uma reflexão em relação ao espaço w^\perp . Dado um vetor de interesse x , se $y = H_w x$, então:

$$y = x - 2 \frac{w \cdot x}{w \cdot w} w.$$

Dados dois vetores x e y , não nulos em \mathbb{R}^n , é possível definir uma transformação de Householder tal que $H_w x = \lambda y$, com $\lambda \in \mathbb{R}$. Para tanto basta se definir $w = x \pm \frac{\|x\|}{\|y\|} y$. (1)

Esta propriedade se torna extremamente útil para a tridiagonalização de matrizes reais simétricas. Para cada coluna i de uma matriz dada A , podemos definir uma transformação de Householder com:

$$w_i = \tilde{a}_i + \delta \frac{\|\tilde{a}_i\|}{\|e_{i+1}\|} e_{i+1} = \tilde{a}_i + \delta \|\tilde{a}_i\| e_{i+1}.$$

sendo $\delta = \pm 1$, e_{i+1} o $i+1$ -ésimo versor da base canônica de \mathbb{R}^n e \tilde{a}_i composta pelos elementos da coluna i de A , exceto os pertencentes à diagonal principal e acima dela. Isto é,

$$\tilde{a}_i = (0, 0, \dots, 0, A_{i+1,i}, A_{i+2,i}, \dots, A_{n-1,i}, A_{n,i})^T.$$

De acordo com o proposto em [2], utilizamos δ com sinal igual ao de $A_{i+1,i}$ para cada w_i .

Utilizando a propriedade em (1), podemos provar que

$$H_{w_i} \tilde{a}_i = \tilde{a}_i - \delta w_i H_{w_i} \tilde{a}_i = (0, 0, \dots, 0, -\delta \|\tilde{a}_i\|, 0, \dots, 0)^T.$$

Ou seja, a coluna i , após a transformação H_{w_i} , possui como único elemento não nulo o módulo de \tilde{a}_i na posição i , com sinal oposto ao de $A_{i+1,i}$.

Assim, temos que

$$H_{w_1} A = \begin{bmatrix} x & x & x & \dots & x \\ x & x & x & \dots & x \\ 0 & x & x & \dots & x \\ \dots & \dots & \dots & \dots & \dots \\ 0 & x & x & \dots & x \end{bmatrix}$$

onde os x representam valores quaisquer.

E, como H_{w_1} e A são simétricas, podemos fazer

$$H_{w_1}AH_{w_1} = \begin{bmatrix} x & x & 0 & \dots & 0 \\ x & x & x & \dots & x \\ 0 & x & x & \dots & x \\ \dots & \dots & \dots & \dots & \dots \\ 0 & x & x & \dots & x \end{bmatrix}$$

para zerar os elementos à direita da sobrediagonal na primeira coluna.

A matriz resultante, pelo mesmo motivo, também é simétrica. Assim, podemos aplicar sucessivas transformações de Householder à direita e à esquerda de A de forma a obter uma matriz semelhante a A , T , tridiagonal.

Vale ressaltar que, para cada H_{w_i} multiplicado à esquerda de A , apenas as linhas $i + 1$ a n têm seus elementos modificados. Simetricamente, quando se multiplica H_{w_i} à direita de A , apenas as colunas $i + 1$ a n têm seus elementos modificados. (2)

Ao fim das transformações, obteremos a expressão $T = HAH^T$, com:

$$H^T = H_{w_1}H_{w_2}\dots H_{w_{n-1}}H_{w_n}.$$

Para economizar tempo e memória, definimos \bar{w}_i e \bar{a}_i , que são equivalentes a w_i e \tilde{a}_i , respectivamente, mas sem os i primeiros valores, pois são todos zero.

2.1.1 Função de Implementação da Tridiagonalização de uma Matriz Real Simétrica

Feitas as considerações acima, criamos a função `tridiagonalization`, que recebe uma matriz simétrica A como entrada e devolve a matriz T tridiagonal, representada por dois vetores, `alphas` e `betas`, que representam sua diagonal principal e sua sobrediagonal, respectivamente, e a matriz `Ht`, que representa H^T , descrita anteriormente.

A implementação feita se utiliza das propriedades descritas em (1) e (2) para aumentar a eficiência do código. Em cada iteração, podemos trabalhar sobre uma submatriz de A , sobre a qual faremos as contas, já que valores fora dela não são modificados, exceto a coluna/linha cuja maioria dos valores serão zerados. Os únicos valores que não são zero pertencem à diagonal principal e sua sobre/subdiagonal. Para os valores da diagonal, basta tomar $A_{i,i}$ na iteração i , pois seu valor não é afetado por nenhuma das transformações de Householder subsequentes. Para os valores da sobrediagonal, basta tomar $-\delta||\tilde{a}_i||$, como demonstrado anteriormente.

A implementação está no Código 2.1.1, abaixo, do qual se retiraram os comentários, mantidos no arquivo original do *script*.

```
1 def tridiagonalization(A: np.array) → Tuple[np.array, np.array, np.array]:
```



```

2     A = A.copy()
3     alphas = []
4     betas = []
5
6     H = np.identity(np.size(A, 0))
7
8     for m in reversed(range(2, np.size(A, 0))):
9         w_i = A[1:, 0]
10
11         alphas.append(A[0, 0])
12         betas.append(-sgn(w_i[0]) * np.sqrt(np.dot(w_i, w_i)))
13
14         w_i[0] -= betas[-1]
15         w_i2 = np.dot(w_i, w_i)
16
17         A = A[1:, 1:]
18
19         for col in np.transpose(A):
20             col -= 2 * np.dot(w_i, col) / w_i2 * w_i
21
22         for row in A:
23             row -= 2 * np.dot(w_i, row) / w_i2 * w_i
24
25         for row in H[:, -m:]:
26             row -= 2 * np.dot(w_i, row) / w_i2 * w_i
27
28         alphas.extend(np.diag(A))
29         betas.append(A[1, 0])
30
31     return (np.array(alphas), np.array(betas), H)

```

Código 2.1.1: Função que implementa a tridiagonalização de uma matriz dada, A .

O código segue a descrição formal apresentada anteriormente. A linha 9 define o vetor \bar{w}_i da Transformação de Householder, $H_{\bar{w}_i}$, de uma dada iteração i e o inicializa com \bar{a}_i . As linhas 11 e 12 adicionam os elementos calculados da diagonal principal e da sua sobrediagonal aos vetores `alphas` e `betas`, respectivamente. A linha 14 modifica o w_i de acordo com a expressão $\bar{w}_i = \bar{a}_i + \|\bar{a}_i\|\delta e_1$. A linha 15 define uma variável auxiliar `w_i2`, equivalente a $\bar{w}_i \cdot \bar{w}_i$. A linha 17 atualiza a variável `A` para armazenar a submatriz de uma dada iteração. As linhas 19 a 22 executam as multiplicações $H_{\bar{w}_i} \bar{A} H_{\bar{w}_i}$ e $H^T H_{\bar{w}_i}$. As linhas 24 e 25 adicionam os últimos elementos da diagonal principal e da sobrediagonal da matriz resultante em

alphas e betas, respectivamente.

2.2 O Algoritmo QR

Após se obter a matriz T , tridiagonal, a partir das transformações de Householder, podemos obter seus autovetores e autovalores utilizando o *Algoritmo QR*. Apesar de A e T serem matrizes semelhantes, isto é, possuem mesmos autovalores, seus autovetores são distintos. Ou seja, para se obter os autovalores de A , precisamos que $V^{(0)}$ seja equivalente a H^T , pois, utilizando-se a matriz identidade como $V^{(0)}$, obteríamos os autovetores de T .

Para a implementação do Algoritmo QR, foi utilizada a mesma função, `qr_algorithm`, do EP anterior. A única modificação feita sobre ela foi a adição de um parâmetro de entrada, `v0`, que é utilizado ao invés da identidade para o cálculo dos autovetores.

2.3 Leitura de Matrizes em Arquivos

Ambos testes A e B podem ter suas matrizes de entrada obtidas a partir da leitura de um arquivo, conforme detalhado em [2]. Neste arquivo, que utilizamos como padrão neste exercício-programa, a primeira linha contém o tamanho n da matriz $A \in \mathbb{R}^{n \times n}$. As linhas subsequentes contêm as entradas equivalentes de cada linha da matriz, sendo as entradas separadas por espaços, uma linha da matriz por linha do arquivo. O código 2.3 implementa essa função.

```
1 def matrix_from_file(filename):
2     with open(filename, encoding="utf-8") as file:
3         matrix_size: int = int(file.readline())
4         matrix = np.zeros((matrix_size, matrix_size))
5
6         treatline = lambda line: list(map(float, line.split()))
7         rows = list(filter(lambda line: len(line) > 0, map(treatline, file.readlines())))
8
9         for i, line in enumerate(rows):
10             matrix[i, :] = line
11
12     return matrix
```

Código 2.3: Função de leitura de uma matriz a partir de um arquivo.

Em resumo, abrimos os arquivos e extraímos o tamanho da matriz pelo valor (inteiro) da primeira linha. Criamos uma matriz preenchida com zeros do tamanho lido. Funcionalmente, transformamos cada linha de uma *array* de *strings* para uma *array* de *floats*, por meio da aplicação de dois mapeamentos nas linhas 6 e 7. Aplica-se um filtro que garante que as linhas lidas não são vazias, após o qual se converte a lista de *arrays* para uma matriz de retorno.

2.4 Leitura de Treliças em Arquivos

É possível também para a aplicação do algoritmo ao problema de treliças planas, descrever as estruturas para as quais desejamos solucionar por meio de arquivos. Em particular, utilizaremos a descrição dada em [2]. Para isso, implementamos duas funções. A primeira, `addBar` é uma função auxiliar que, dados os índices i e j dos nós que formam uma barra, seu comprimento L , o cosseno e seno do ângulo que forma com a horizontal, o módulo de Young em Pa do material da barra, bem como sua densidade p em kg/m^3 e a área da seção transversal da barra A em m^2 , adiciona a contribuição da barra correspondente às matrizes de massa M e de rigidez K , cuja descrição está no código 2.4.

```
1 def addBar(
2     i: int,
3     j: int,
4     L: float,
5     c: float,
6     s: float,
7     E: float,
8     p: float,
9     A: float,
10    M: np.array,
11    K: np.array,
12 ):
13     mass_contribution = 0.5 * p * A * L
14     M[i] += mass_contribution
15
16     local_stiffness = (A * E) / L * np.array([[c ** 2, c * s], [c * s, s ** 2]])
17     K[2 * i : 2 * i + 2, 2 * i : 2 * i + 2] += local_stiffness
18
19     if j in range(len(M)):
20         M[j] += mass_contribution
21         K[2 * i : 2 * i + 2, 2 * j : 2 * j + 2] += -local_stiffness
22         K[2 * j : 2 * j + 2, 2 * i : 2 * i + 2] += -local_stiffness
23         K[2 * j : 2 * j + 2, 2 * j : 2 * j + 2] += local_stiffness
```

Código 2.4: Função auxiliar que adiciona a contribuição de uma barra às matrizes que descrevem o sistema total.

Na linha 13, calculamos a contribuição da massa da barra para os nós i e j que a definem. Notamos que j pode ser um nó fixo, portanto devemos verificar se este índice corresponde a um ponto móvel, isto é, se j é menor que o tamanho do vetor M . Na linha 16, calculamos a matriz de rigidez local $K_{i,j}$, adicionando essa contribuição à matriz de rigidez total conforme descrito em [2].

Considerando que a primeira linha do arquivo contém o número total de nós, o número de nós livres e o número de barras, e que a segunda linha contém a densidade, a área da seção transversal e o módulo de Young (em GPa), bem como as linhas subsequentes descrevem cada barra, cujas entradas são os nós que compõem a barra, o ângulo com a horizontal e o comprimento da barra, nesta ordem, separadas por espaços, criou-se a função 2.4 que implementa a leitura de uma treliça por um arquivo.

```

1 def truss_from_file(filename):
2     with open(filename, encoding="utf-8") as file:
3         total_nodes, free_nodes, _ = map(int, file.readline().split())
4         p, A, E = map(float, file.readline().split())
5         E *= 1e9
6
7         treatline = lambda line: tuple(map(int, line.split()[2:])) + tuple(
8             map(float, line.split()[2:]))
9         )
10        bars = list(
11            filter(lambda line: len(line) > 0, map(treatline, file.readlines()))
12        )
13
14        K = np.zeros((2 * free_nodes, 2 * free_nodes), float)
15        M = np.zeros(free_nodes, float)
16
17        for bar in bars:
18            (i, j, theta, L) = bar
19            theta = np.deg2rad(theta)
20            addBar(i - 1, j - 1, L, np.cos(theta), np.sin(theta), E, p, A, M, K)
21
22        return M, K, total_nodes, free_nodes, bars

```

Código 2.4: Função de leitura de uma treliça plana a partir de um arquivo.

Nas linhas 3 e 4 obtemos os dados da treliça, convertendo o módulo de Young de GPa em Pa pela multiplicação por 10^9 na linha 5. Convertemos as linhas do arquivo de *arrays* de *strings* para *arrays* de *floats* da mesma forma que realizado na última seção. Criamos a matriz de rigidez total K e o vetor de massas M utilizando o tamanho lido no arquivo. Para cada barra lida, adicionamos sua contribuição às matrizes de descrição do sistema nas linhas 18 a 20, convertendo, primeiramente, o ângulo de graus para radianos. Retorna-se o vetor de massa, a matriz de rigidez, o número de nós totais e livres e o número de barras. A razão pela representação de M como um vetor será descrita em seção posterior.

3 Construção dos Testes

Foram construídos 4 diferentes rotinas de teste para o programa. As duas primeiras são implementações dos testes A e B descritos em [2]. Já a terceira corresponde à aplicação para solução de treliças planas em oscilações de baixa energia total. Por fim, a quarta e última rotina permite ao usuário verificar a utilização do algoritmo para uma matriz qualquer, inserida manual- ou automaticamente. Em seguida, descreveremos as construções destes testes, na ordem que foram apresentados.

3.1 Teste A: Matriz de Autovalores Inteiros Conhecidos

Nesta instância, desejamos obter os autovalores e autovetores da matriz A descrita abaixo, cujos valores são conhecidos e valem $\Lambda = (7, 2, -1, -2)$, sendo:

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

Com os autovalores e autovetores calculados, verificamos se é válida a relação $Av_j = \lambda_j v_j$ para cada autovalor λ_j e seu autovetor correspondente, bem como realizar o teste de ortogonalidade dos autovetores, isto é, identificar se vale $VV^T = I$. O Código 3.1 abaixo implementa o teste.

```
1 def teste_1():
2     print(
3         """
4         >> Você selecionou o teste A proposto pelo relatório.
5
6         - Realizando leitura de arquivo: 'input-a'\n"""
7     )
8
9     A = matrix_from_file("input-a")
10
11     print("""      Matriz de entrada:\n""")
12     print("      ", np.array2string(A, prefix="      "))
13
14     alphas, betas, H = tridiagonalization(A)
15
16     print("""\n      Matriz tridiagonalizada:\n""")
17     print(
18         "      ",
19         np.array2string(
20             np.diag(betas, k=1) + np.diag(betas, k=-1) + np.diag(alphas),
21             prefix="      ",
22         ),
23     )
24
25     Lambda, _, V, _ = qr_algorithm(alphas, betas, H)
26
27     print(
28         f""\n      Autovalores:\n\t- Encontrados:\t{np.array(sorted(Lambda, reverse = True))}\n\t- Esperados:\t{np.array([7.0, 2.0, -1.0, -2.0])}"
29     )
```

```

30     print(f"\n      Matriz de Autovetores:\n")
31     print("\t      v1\t      v2\tv3\t      v4")
32     print("      ", np.array2string(V, prefix="      "))
33
34     print(
35         "\n      OBS.: Caso  $\lambda$  não seja renderizado corretamente em seu terminal, este \n      char é um Lambda, indicando o autovalor da matriz A."
36     )
37
38     for i in range(len(Lambda)):
39         result = np.matmul(A, V[:, i])
40
41         print(f"\n      A * v{i+1} = {result}")
42         print(f"       $\lambda$  * v{i+1} = {Lambda[i] * V[:, i]}")
43
44         def ratio(a, b):
45             return np.array(
46                 [Lambda[i] if a[j] < 1e-6 else a[j] / b[j] for j in range(len(a))]
47             )
48
49         print(
50             f"\n      Proporção Entrada Depois/Antes Transformação:\n\tEsperada: {Lambda[i]:.4f}\n\tObtida: {ratio(result, V[:, i])}"
51         )
52
53         input("\n      Pressione [ENTER] para continuar.")
54         sys.stdout.write("\x1b[1A")
55         sys.stdout.write("\x1b[2K")
56         sys.stdout.write("\x1b[1A")
57         sys.stdout.write("\x1b[2K")
58         print("      -")
59
60     print(f"\n      Teste de ortogonalidade:\n")
61     print(
62         "      VVt =",
63         np.array2string(np.matmul(V, np.transpose(V)), prefix="      "),
64     )
65
66     print("\n      Rotina de teste concluída! Obrigado pela execução!")

```

Código 3.1: Implementação do Teste A.

Na linha 8, lemos a matriz do arquivo `input-a`, fornecido com o enunciado, bem como enviado no arquivo compactado da solução do exercício. Com a matriz lida, executamos o processo de tridiagonalização por transformações de Householder na linha 13, decompondo a matriz em seus vetores de α s e β s, de acordo com a descrição do Exercício Programa 1. Na linha 24, aplica-se o Algoritmo QR, cujo resultado contém os autovalores e autovetores da matriz A . Nas linhas 40 e 41, verificamos a definição para os autovalores e vetores encontrados, isto é, se verificamos $Av = \lambda v$. Por fim, a linha 62 executa o teste de ortogonalidade.

3.2 Teste B: Matriz de Autovalores dados por Fórmula

Neste teste, encontraremos os autovalores e autovetores da matriz A abaixo, cujos autovalores são dados pela fórmula $\lambda_j = \frac{1}{2} \left[1 - \cos \frac{(2i-1)\pi}{2n+1} \right]^{-1}$ com $i = 1, 2, \dots, n$.

$$A = \begin{bmatrix} n & n-1 & n-2 & \dots & 2 & 1 \\ n-1 & n-1 & n-2 & \dots & 2 & 1 \\ n-2 & n-2 & n-2 & \dots & 2 & 1 \\ \vdots & \vdots & \vdots & \dots & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Apresenta-se as mesmas verificações descritas no teste acima. O Código 3.2 abaixo detalha a implementação do teste.

```

1  def teste_2():
2      print(
3          """
4          >> Você selecionou o teste B proposto pelo relatório.
5
6          - Realizando leitura de arquivo: 'input-b'\n"""
7      )
8
9      A = matrix_from_file("input-b")
10
11     print("""      Matriz de entrada:\n""")
12     print("      ", np.array2string(A, prefix="      "))
13
14     alphas, betas, H = tridiagonalization(A)
15
16     n, _ = A.shape
17     expectedEigenvalues = np.array(
18         [1.0 / (2 * (1 - np.cos((2 * i + 1) * pi / (2 * n + 1)))) for i in range(n)]
19     )
20     print("""\n      Matriz tridiagonalizada:\n""")
21     print(
22         "      ",
23         np.array2string(
24             np.diag(betas, k=1) + np.diag(betas, k=-1) + np.diag(alphas),
25             prefix="      ",
26         ),
27     )
28
29     Lambda, _, V, _ = qr_algorithm(alphas, betas, H)
30
31     print(
32         f"\n      Autovalores:\n\t- Encontrados:\t{np.array(sorted(Lambda, reverse = True))}\n\t- Esperados:\t{expectedEigenvalues}"
33     )
34     print(f"\n      Matriz de Autovetores:\n")
35     print(
36         "\t v1\t v2\tv3\t v4\t v5\t ... \t v16\t v17\t v18\tv19\t v20"
37     )
38     print("      ", np.array2string(V, prefix="      "))
39
40     print(

```

```

41         "\n      OBS.: Caso  $\lambda$  não seja renderizado corretamente em seu terminal, este \n      char é um Lambda, indicando o autovalor da matriz A."
42     )
43
44     for i in range(len(Lambda)):
45         result = np.matmul(A, V[:, i])
46
47         print(f"\n      A * v{i+1} = {result}")
48         print(f"       $\lambda$  * v{i+1} = {Lambda[i] * V[:, i]}")
49
50     def ratio(a, b):
51         return np.array(
52             [Lambda[i] if a[j] < 1e-6 else a[j] / b[j] for j in range(len(a))]
53         )
54
55     print(
56         f"\n      Proporção Entrada Depois/Antes Transformação:\n\tEsperada: {Lambda[i]:.6f}\n\tObtida: {ratio(result, V[:, i])}"
57     )
58
59     input("\n      Pressione [ENTER] para continuar.")
60     sys.stdout.write("\x1b[1A")
61     sys.stdout.write("\x1b[2K")
62     sys.stdout.write("\x1b[1A")
63     sys.stdout.write("\x1b[2K")
64     print("      -")
65
66     print(f"\n      Teste de ortogonalidade:\n")
67     print(
68         "      VVt =",
69         np.array2string(np.matmul(V, np.transpose(V)), prefix="      ",
70     )
71
72     print("\n      Rotina de teste concluída! Obrigado pela execução!")

```

Código 3.2: Implementação do Teste B.

A implementação é análoga ao Teste A, diferindo apenas na construção dos autovalores para comparação, pois são dados pela fórmula apresentada.

3.3 Aplicação do Algoritmo a Treliças Planas

Desejamos solucionar o problema de treliças planas a n nós móveis e m barras, com energia total suficientemente baixa de modo que possamos aproximar o modelo por equações lineares. O objetivo é encontrar as frequências naturais de vibração e os modos de oscilação associados a tais frequências. As barras têm mesmo material, com densidade ρ , módulo de elasticidade E e área da seção transversal A . Particularizando para o teste desenvolvido, trabalharemos com uma treliça a 12 nós móveis, 2 nós fixos e 28 barras.

O estado do sistema é caracterizado pelo deslocamento de cada nó (h_i, v_i) , de tal forma que podemos criar o vetor de deslocamento \mathbf{x} em que $\mathbf{x}_{2i} = h_i$ e $\mathbf{x}_{2i+1} = v_i$, $i = 0, 1, \dots, n-1$ são os deslocamentos horizontal e vertical de cada nó, respectivamente. A resposta dinâmica do sistema não depende da deformação estática pela gravidade, portanto consideraremos apenas os efeitos da deformação elástica e da energia cinética para a evolução do sistema.

Uma aproximação considerada para a resolução do problema é que a massa de cada barra está concentrada nos nós que a compõem. Sendo $m_{i,j}$ a massa da barra que conectando os nós i e j , então tal barra contribui com $0.5m_{i,j}$ para m_i e $0.5m_{i,j}$ para m_j , sendo m_k a massa concentrada do nó k .

Poderíamos armazenar as massas concentradas em uma matriz diagonal $M \in \mathbb{R}^{24 \times 24}$, em que m_{2i} e m_{2i+1} , $i = 0, 1, \dots, n-1$ são as massas concentradas e aparecem aos pares devido à forma com que armazenamos o deslocamento vertical e horizontal combinados em um vetor único. Todavia, da perspectiva de otimização do consumo de memória, é mais eficiente armazenar tais massas em um único vetor de massas M , em que m_i é a massa concentrada do nó i , em que se observa uma correspondência direta para os índices do vetor de deslocamento. Ou seja, a massa concentrada associada a uma entrada k do vetor \mathbf{x} é observada na entrada $\lfloor \frac{k}{2} \rfloor$.

Assumindo pequenos deslocamentos e comportamento elástico linear das barras, temos que a energia total de deformação D é dada por: $D = \frac{1}{2} \mathbf{x}^T k \mathbf{x}$ em que K é a matriz de rigidez total da treliça. Obtemos K por meio da contribuição de cada barra para o sistema, adicionando sua matriz de rigidez local $K^{i,j}$ nas posições equivalentes, isto é:

$$\begin{bmatrix} \vdots & \vdots & \cdots & \vdots & \vdots & \\ \cdots & K_{2i,2i} & K_{2i,2i+1} & \cdots & K_{2i,2j} & K_{2i,2j+1} & \cdots \\ \cdots & K_{2i+1,2i} & K_{2i+1,2i+1} & \cdots & K_{2i+1,2j} & K_{2i+1,2j+1} & \cdots \\ \cdots & \vdots & \vdots & \cdots & \vdots & \vdots & \cdots \\ \cdots & K_{2j,2i} & K_{2j,2i+1} & \cdots & K_{2j,2j} & K_{2j,2j+1} & \cdots \\ \cdots & K_{2j+1,2i} & K_{2j+1,2i+1} & \cdots & K_{2j+1,2j} & K_{2j+1,2j+1} & \cdots \\ \vdots & \vdots & \cdots & \vdots & \vdots & \end{bmatrix} + = \begin{bmatrix} \vdots & \vdots & \cdots & \vdots & \vdots & 0 \\ \cdots & K_{0,0}^{i,j} & K_{0,1}^{i,j} & \cdots & K_{0,2}^{i,j} & K_{0,3}^{i,j} & \cdots \\ \cdots & K_{1,0}^{i,j} & K_{1,1}^{i,j} & \cdots & K_{1,2}^{i,j} & K_{1,3}^{i,j} & \cdots \\ \cdots & \vdots & \vdots & \cdots & \vdots & \vdots & \cdots \\ \cdots & K_{2,0}^{i,j} & K_{2,1}^{i,j} & \cdots & K_{2,2}^{i,j} & K_{2,3}^{i,j} & \cdots \\ \cdots & K_{3,0}^{i,j} & K_{3,1}^{i,j} & \cdots & K_{3,2}^{i,j} & K_{3,3}^{i,j} & \cdots \\ 0 & \vdots & \vdots & \cdots & \vdots & \vdots & \end{bmatrix}$$

para cada barra, incluindo aquelas que conectam nós móveis a nós fixos, pois tais também são deformáveis.

A rigidez local de uma barra é definida por

$$K^{i,j} = \frac{AE}{L_{i,j}} \begin{bmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix}$$

onde A é a área da seção transversal da barra em m^2 , E o módulo de elasticidade em Pa , $L_{i,j}$ o comprimento da barra e $c = \cos\theta_{i,j}$ e $s = \sin\theta_{i,j}$ sendo $\theta_{i,j}$ o ângulo entre a barra e o eixo horizontal. Não precisamos armazenar todos os senos e cossenos na memória. Basta notar que se

$$Q = \begin{bmatrix} c^2 & cs \\ cs & s^2 \end{bmatrix}$$

então

$$K^{i,j} = \frac{AE}{L_{i,j}} \begin{bmatrix} Q & -Q \\ -Q & Q \end{bmatrix}$$

que é, de fato, o que armazenamos na função `addBar`, descrita anteriormente.

A energia cinética de um nó i é $T_i = \frac{1}{2}m_i(\dot{h}_i^2 + \dot{v}_i^2)$. Somando a contribuição de cada nó, temos a energia cinética do sistema:

$$T = \frac{1}{2}\dot{\mathbf{x}}^T M' \dot{\mathbf{x}}$$

em que $M' \in \mathbb{R}^{24 \times 24}$ é diagonal e $M'_{k,k} = M_{\lfloor \frac{k}{2} \rfloor}$, como descrito anteriormente.

A descrição do movimento dada pela energia é, portanto $M' \ddot{\mathbf{x}} + K \mathbf{x} = 0$. As frequências naturais de vibração e os modos associados são encontrados ao se fazer $\mathbf{x}(t) = \mathbf{z}e^{i\omega t}$, sendo \mathbf{z} modo natural de oscilação associado à frequência ω , levando-nos à equação generalizada de autovalores

$$K \mathbf{z} = \omega^2 M' \mathbf{z}$$

Sendo M' diagonal, com entradas reais positivas, podemos substituir $\mathbf{z} = M'^{-\frac{1}{2}} \mathbf{y}$, chegando a $K M'^{-\frac{1}{2}} \mathbf{y} = M'^{\frac{1}{2}} \omega^2 \mathbf{y}$, que se manipula em

$$\tilde{K} \mathbf{y} = \omega^2 \mathbf{y}$$

onde $\tilde{K} = M'^{-\frac{1}{2}} K M'^{-\frac{1}{2}}$ simétrica definida positiva e real.

Por conseguinte, tridiagonalizando \tilde{K} e aplicando o Algoritmo QR para matrizes tridiagonais simétricas desenvolvido no primeiro exercício-programa, encontramos os autovalores e autovetores de \tilde{K} . Os autovalores λ_j de \tilde{K} são tais que $\omega_j = \sqrt{\lambda_j}$. Seus autovetores, por sua vez, correspondem a \mathbf{y}_j , portanto encontramos os modos de oscilação tomando $\mathbf{z}_j = M'^{-\frac{1}{2}} \mathbf{y}_j$, pois M' e \mathbf{y}_j são conhecidos. Transformamos o problema de EDOs da treliça plana em um problema de autovalores e autovetores.

Limitações de Modelo: É importante notar que, pela construção aplicada, o modelo se restringe a pequenos níveis de energia, no qual a treliça pode ser bem descrita por uma equação diferencial linear. Se partirmos para níveis maiores, aparecem não-linearidades devido a diferentes fenômenos físicos que cabem a um curso mais avançado. Destarte, daremos maior atenção às 5 menores frequências de vibração encontradas, que correspondem a modos dentro do limite de linearidade.

Solução a partir de componentes cossenodais: Da mesma forma que feito com molas no primeiro exercício-programa, podemos observar que a solução-geral da equação é uma família de exponenciais complexas da forma

$$\mathbf{x}(t) = V^T \begin{bmatrix} e^{i\omega_1 t} \\ e^{i\omega_2 t} \\ \vdots \\ e^{i\omega_n t} \end{bmatrix}$$

sendo V a matriz de autovetores. No caso particular, em que a excitação inicial é múltipla de um

autovetor, temos uma solução particular de componentes que vibram à mesma frequência, ou seja,

$$\mathbf{x}_p(t) = a\mathbf{z}_j e^{i\omega_j t}$$

Para a construção da animação da treliça vibrando, podemos escrever a solução particular em termos puramente cossenodais reais. Primeiro, mostraremos que $\mathbf{z}e^{-i\omega t}$ também leva à mesma conclusão sobre os autovalores e autovetores que $\mathbf{z}e^{i\omega t}$.

Tome $\mathbf{x}_2(t) = \mathbf{z}e^{-i\omega t}$, logo $\ddot{\mathbf{x}}_2(t) = -\omega^2\mathbf{x}_2(t)$ e, portanto, $M'\ddot{\mathbf{x}} + K\mathbf{x} = 0 \iff \ddot{\mathbf{y}} = -\tilde{K}\mathbf{y}$, com \tilde{K} definido da mesma forma que para \mathbf{x}_1 , o qual nos leva à mesma família de autovalores, autovetores e soluções. Portanto, para cada autovalor λ_j , ambos $\mathbf{z}_j e^{-i\omega_j t}$ e $\mathbf{z}_j e^{i\omega_j t}$ são soluções da EDO. Ou seja, podemos escrever a solução particular para o caso em que a excitação inicial é múltipla de um autovetor como

$$\mathbf{x}(t) = a\mathbf{z}_j e^{-i\omega_j t}$$

□

Da linearidade da equação, construímos:

$$\mathbf{x}_3(t) = V^T \begin{bmatrix} e^{i\omega_1 t} + e^{-i\omega_1 t} \\ e^{i\omega_2 t} + e^{-i\omega_2 t} \\ \vdots \\ e^{i\omega_n t} + e^{-i\omega_n t} \end{bmatrix} \text{ e } \mathbf{x}_4(t) = V^T \begin{bmatrix} e^{i\omega_1 t} - e^{-i\omega_1 t} \\ e^{i\omega_2 t} - e^{-i\omega_2 t} \\ \vdots \\ e^{i\omega_n t} - e^{-i\omega_n t} \end{bmatrix}$$

as quais, pela relação de Euler-Moivre, $e^{i\omega t} = \cos \omega t + i \sin \omega t$ equivalem a:

$$\mathbf{x}_3(t) = V^T \begin{bmatrix} 2 \cos \omega_1 t \\ 2 \cos \omega_2 t \\ \vdots \\ 2 \cos \omega_n t \end{bmatrix} \text{ e } \mathbf{x}_4(t) = V^T \begin{bmatrix} 2i \sin \omega_1 t \\ 2i \sin \omega_2 t \\ \vdots \\ 2i \sin \omega_n t \end{bmatrix}$$

O que nos leva à solução-geral em termos cossenodais:

$$\begin{aligned} \mathbf{x}(t) &= C_1 V^T \begin{bmatrix} 2 \cos \omega_1 t \\ 2 \cos \omega_2 t \\ \vdots \\ 2 \cos \omega_n t \end{bmatrix} + C_2 V^T \begin{bmatrix} 2i \sin \omega_1 t \\ 2i \sin \omega_2 t \\ \vdots \\ 2i \sin \omega_n t \end{bmatrix} \\ &= C'_1 V^T \begin{bmatrix} \cos \omega_1 t \\ \cos \omega_2 t \\ \vdots \\ \cos \omega_n t \end{bmatrix} + C'_2 V^T \begin{bmatrix} \sin \omega_1 t \\ \sin \omega_2 t \\ \vdots \\ \sin \omega_n t \end{bmatrix} \end{aligned}$$

com $C_1, C_2 \in \mathbb{C}$ e $C'_1 = 2C_1, C'_2 = 2iC_2$. No caso particular em que $\mathbf{x}(0) = \mathbf{z}_j$ e $\dot{\mathbf{x}}(0) = 0$, temos que $\mathbf{x}(t) = C'_1 \mathbf{z}_j \cos \omega_j t + C'_2 \mathbf{z}_j \sin \omega_j t$ e $\mathbf{x}(0) = C'_1 \mathbf{z}_j = \mathbf{z}_j \iff C'_1 = 1$. Como $\dot{\mathbf{x}}(0) = \omega C'_2 \mathbf{z}_j = 0$, logo $C'_2 = 0$ e concluímos que $\mathbf{x}(t) = \mathbf{z}_j \cos \omega_j t$!

Utilizaremos esta propriedade para construir a animação da treliça. Ou seja, dado um nó i e seu deslocamento inicial, podemos escrever sua posição no tempo como

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} x_{0,i} \\ y_{0,i} \end{pmatrix} + \begin{pmatrix} \mathbf{z}_{j,2i} \\ \mathbf{z}_{j,2i-1} \end{pmatrix} \cos(\omega_j t) \quad i = 0, 1, \dots, n-1$$

O Código 3.3 abaixo exibe a execução completa do teste.

```

1  def teste_3():
2      print(
3          """
4          >> Você selecionou a Aplicação em Trelças Planas.
5
6          - Realizando leitura de arquivo: 'input-c'\n"""
7      )
8
9      M, K, total_nodes, free_nodes, bars = truss_from_file("input-c")
10
11     print("""\n      Matriz de Rigidez (K):\n""")
12     print("      K = ", np.array2string(K, prefix="      "))
13
14     print("""\n      Matriz de Massa (M):\n""")
15     print("      M = ", np.array2string(np.diag(M), prefix="      "))
16
17     M = 1.0 / np.sqrt(M)
18
19     for i in range(2 * len(M)):
20         for j in range(2 * len(M)):
21             K[i, j] += M[i // 2] * M[j // 2]
22
23     print("""\n      Matriz da Equação Diferencial (K-):\n""")
24     print("      K- = ", np.array2string(K, prefix="      "))
25
26     alphas, betas, H = tridiagonalization(K)
27     Lambda, _, V, _ = qr_algorithm(alphas, betas, H)
28
29     print("""\n      K- Tridiagonalizado:\n""")
30     print(
31         "      HK-Ht = ",
32         np.array2string(
33             np.diag(alphas) + np.diag(betas, -1) + np.diag(betas, 1),
34             prefix="      ",
35         ),
36     )
37     print(f"""\n      Autovalores Encontrados: {np.array(sorted(Lambda))}""")
38
39     frequencies = list(map(np.sqrt, sorted(Lambda)))
40     eigenvalues = sorted(Lambda)[:5]
41     modes = []
42
43     for i in map(lambda eig: list(Lambda).index(eig), eigenvalues):
44         modes.append(V[:, i])
45
46     for mode in modes:
47         for i in range(len(mode)):
48             mode[i] += M[i // 2]
49
50     print(f"""\n      5 menores Frequências Encontrados: {frequencies[:5]}""")
51     print("""\n      Modos de vibração associados às 5 menores frequências:\n""")
52
53     if (
54         str(
55             input(
56                 "\n      Deseja visualizar uma animação das treliças vibrando conforme cada modo de vibração? (S/n): "
57             )
58         ).lower()

```

```

59     #= "n"
60 ):
61     titles = ["primeira", "segunda", "terceira", "quarta", "quinta"]
62     scale = [100, 150, 100, 100, 150]
63
64     for i in range(5):
65         print(f"\n      Exibindo a {titles[i]} animação!")
66
67         freq = frequencias[i]
68         mode = modos[i]
69
70         for j in range(len(mode)):
71             mode[j] *= scale[i]
72
73         print(f"\n      Frequência natural de oscilação: {freq:.4f} rad/s")
74         print(f"      Modo de vibração:")
75         print("      z = ", np.array2string(mode, prefix="      "))
76
77         print(
78             f"\n      Iremos excitar a treliça com uma condição inicial igual a {scale[i]} vezes o modo de oscilação associado a esta frequência."
79         )
80
81         print(
82             f"\n      Animação aberta. Por favor, feche a janela para prosseguir."
83         )
84
85         fig = plt.figure()
86         fig.set_size_inches(18.5, 10.5)
87         fig.suptitle(f"Evolução do Sistema no Tempo - Frequência: {freq:.4f}")
88
89         X0 = [15, 5, 25, 15, 5, -5, 25, 15, 5, -5, 15, 5, 20, 0]
90         Y0 = [40, 40, 30, 30, 30, 30, 20, 20, 20, 20, 10, 10, 0, 0]
91
92         bar_lines = [0] * len(bars)
93
94         ax = fig.add_subplot(111)
95
96         mat = ax.plot(X0, Y0, "o")
97
98         for k, (i, j, _, _) in enumerate(bars):
99             bar_lines[k] = ax.plot(
100                 [X0[i - 1], X0[j - 1]], [Y0[i - 1], Y0[j - 1]], c="k"
101             )
102
103         patch = reduce(lambda a, b: a + b, bar_lines) + list(mat)
104
105         def animate(index):
106             t = index / (2 * pi * freq)
107             solution = [mode[j] * np.cos(freq * t) for j in range(24)]
108             solution += (0, 0, 0, 0)
109
110             X = np.array([X0[i // 2] + solution[i] for i in range(0, 28, 2)])
111
112             Y = np.array([Y0[i // 2] + solution[i] for i in range(1, 28, 2)])
113
114             for k, (i, j, _, _) in enumerate(bars):
115                 bar_lines[k][0].set_data([X[i - 1], X[j - 1]], [Y[i - 1], Y[j - 1]])
116
117             mat[0].set_data(X, Y)
118
119             return patch
120
121         anim = FuncAnimation(fig, animate, frames=600000, interval=1, blit=True)
122         plt.show()
123     print(f"\n      Rotina de teste concluída! Obrigado pela execução!")

```

Código 3.3: Implementação da Aplicação para Trelças.

Na linha 9, carregamos as matrizes M e K do arquivo `input-c`. Na linha 21, calculamos \tilde{K} , armazenando na mesma instância de memória que K (pois não precisaremos mais dela). Nas linhas 26 e 27, tridigitalizamos \tilde{K} , aplicando, em sequência, o Algoritmo QR. Nas linhas 39 a 48, encontramos as frequências de oscilação natural a partir dos autovalores de \tilde{K} e os modos associados por meio de seus autovetores,

para as 5 menores frequências de oscilação, respeitando a condição de linearidade. As linhas seguintes, até o final, constroem a animação e exibição dos dados, que podem ser vistas pela execução do programa.

NOTA: As oscilações iniciais foram tomadas como múltiplos dos modos de oscilação, de modo que se pudesse visualizar bem a movimentação, portanto pode haver exageros no fator de escala. *Recomendamos a visualização da animação, pois gostamos muito dela!*

Referências

- [1] JÚNIOR, M. B. **Álgebra Linear**. 3. ed. São Paulo: Publicações do IME, 1988.
- [2] MAP3121. **EP2: Autovalores e Autovetores de Matrizes Reais Simétricas**. Disponível em: https://edisciplinas.usp.br/pluginfile.php/6318233/mod_resource/content/2/ep2_2021.pdf. Acesso em: 30 jun. 2021.