MAP3121 - Métodos Numéricos e Aplicações

Exercício Programa 2

Autovalores e Autovetores de Matrizes Tridiagonais Simétricas

Data: 20/07/2021	
Membros:	
11261110 - Antonio Lago Araújo Seixas	**Turma** : 1
11259715 - Vanderson da Silva dos Santos	**Turma:** 2

São Paulo
2021

Índice

Introdução

Para a resolução do problema nesse exercício programa, por uma preferência de praticidade, a dupla optou pelo uso da linguagem de programação *python 3.7*, fazendo uso das bibliotecas *numpy*, a qual é utilizada principalmente para as operações entre vetores e matrizes, e *matplotlib*, usada para a construção dos gráficos exigidos.

Com isso, foram criado 5 arquivos principais sendo eles:

- “QR_algorithm.py”: O qual é implementado o algoritmo QR
- “householder_algorithm.py”: O qual é implementado o algoritmo householder
- “assignment_a.py”: O qual é implementado a tarefa A
- “assignment_b.py”: O qual é implementado a tarefa B
- “assignment_c.py”: O qual é implementado a tarefa C
- “_main_.py”: Arquivo main no qual o usuário escolhe qual das tarefas que pretende executar.

Implementação

Algoritmo QR

Rotações de givens e fatoração QR

As rotações de givens tratam-se de uma transformação algébrica ortogonal. Ela visa rotacionar uma certa matriz A no sentido anti-horário em um ângulo θ em relação à origem. Para isso, a matriz A é multiplicada pela matriz Q, a qual representa a transformação.

No problema proposto, a rotação de givens é utilizada especificamente para a aplicação da fatoração QR, esta a qual aplica sucessivas transformações de givens específicas a fim de transformar a matriz A em uma matriz R triangular superior. Portanto, a dupla implementou somente o caso alvo da transformação de givens a qual é utilizada no problema.

Neste caso específico, para uma matriz A_{nxn} tridiagonal simétrica, deve-se aplicar “n” vezes as transformações de givens, sendo que cada transformação Q_k; $1 < k < n$ admite o seguinte formato:

$$Qk = \begin{bmatrix} c_k & -s_k & 0 & \dots & 0 \\ s_k & c_k & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow Q_{k+1} = \begin{bmatrix} 0 & \dots & \dots & \dots & 0 \\ \vdots & c_{k+1} & -s_{k+1} & \dots & 0 \\ \vdots & s_{k+1} & c_{k+1} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Sendo c_k e s_k definidos pelas equações

$$c_k = \frac{\alpha_k}{\sqrt{\alpha_k^2 + \beta_k^2}} \quad \text{e} \quad s_k = -\frac{\beta_k}{\sqrt{\alpha_k^2 + \beta_k^2}}$$

Sendo α_k os valores na diagonal principal de A e β_k, os valores da subdiagonal de A.Nesta situação, a dupla implementou duas funções, sendo elas:

- *givens_rotations_Qk(k,A)*: Esta função recebe os parâmetros *k* e *A*, os quais correspondem ao número da interação e a matriz a ser transformada, respectivamente. A partir de tais parâmetros, ela calcula os valores de c_k e s_k e retorna a transformação de givens correspondente a interação. Sua implementação está mostrada na figura a seguir:

```
#Function to get the givens_rotation_Qk matrix
def givens_rotation_Qk(k,A):
    Q = np.eye(len(A))
    alfa_k = A[k,k]
    beta_k = A[k+1,k]
    ck = alfa_k/(np.sqrt((alfa_k**2)+(beta_k**2)))
    sk = -(beta_k)/(np.sqrt((alfa_k**2)+(beta_k**2)))
    Q[k,k] = ck
    Q[k,k+1] = -sk
    Q[k+1,k] = sk
    Q[k+1,k+1] = ck
    return Q
```

- *QR_fatoration(A)*: Esta recebe como parâmetro a matriz A, a qual é o alvo da fatoração e retorna a matriz R, triangular superior, e matriz Q a qual se refere a multiplicação de todas transformações de givens realizadas. Para isso a função, declara as matrizes R = A e Q sendo a identidade, e partir de um *for loop* tais valores são atualizados sendo multiplicados pelo retorno da chamada da função *givens_rotations(k,A)*. Segue abaixo a sua implementação:

```
## Function which returns a matrix after a givens rotation
def QR_fatoration(A):
    R = A
    matrix_i_size = len(A)
```

```
    Q =np.eye(matrix_i_size)
    Q_iteration_size = matrix_i_size - 1
    for i in range (0, Q_iteration_size,1):
        Q = Q@np.transpose(givens_rotation_Qk(i,R))
        R = givens_rotation_Qk(i,R)@R
    return [R,Q]
```

Algoritmo QR

O algoritmo QR visa determinar os autovalores e autovetores de uma matriz simétrica A. Sua aplicação pode ser feita de duas formas: com e sem deslocamentos espectrais.

Os deslocamentos espectrais atuam para minimizar as interações necessárias para a convergência do algoritmo. O deslocamento espectral são definidos a partir da heurística de Wilkinson, μ_k, sendo 'k' referente a interação correspondente ela é definida como:

$$d_k = (\alpha_{n-1}^{(k)} - \alpha_n^{(k)})/2 \text{ defina } \mu_k = \alpha_n^{(k)} + d_k - sgn(d_k)\sqrt{d_k^2 + (\beta_{n-1}^{(k)})^2}$$

$$\text{onde } sgn(d) = 1 \text{ se } d \geq 0 \text{ e } sgn(d) = -1 \text{ caso contrário.}$$

Sendo fornecido pelo enunciado, a dupla se pautou no pseudocódigo do algoritmo QR, o qual segue na figura abaixo:

1: Sejam $A^{(0)} = A \in \mathbb{R}^{n\times n}$ uma matriz tridiagonal simétrica, $V^{(0)} = I$ e $\mu_0 = 0$. O algoritmo calc
a sua forma diagonal semelhante $A = V\Lambda V^T$.

2: $k = 0$

3: **para** $m = n, n-1, \ldots, 2$ **faça**

4: **repita**

5: se $k > 0$ calcule μ_k pela heurística de Wilkinson

6: $A^{(k)} - \mu_k I \rightarrow Q^{(k)}R^{(k)}$

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

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

9: $k = k + 1$

10: **até que** $|\beta_{m-1}^{(k)}| < \epsilon$

11: **fim do para**

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

13: $V = V^{(k)}$

Sendo Λ a matriz de autovalores de A, V sua matriz de autovetores e μ_k o deslocamento espectral, sendo igual a 0 quando não se utiliza o deslocamento, foram implantadas mais duas funções a fim de se criar um código para tal algoritmo sendo elas:

- *wilkinson_heuristics(A,n):* Esta função recebe como parâmetros a matriz [A] e o inteiro n. A partir da matriz [A] é calculado o deslocamento espectral sendo que o inteiro "n" indica para qual posição da matriz deve ser calculado o deslocamento. A função retorna os valores de μ_k e de $\beta_{n-1}^{(k)}$. Segue abaixo sua implementação:

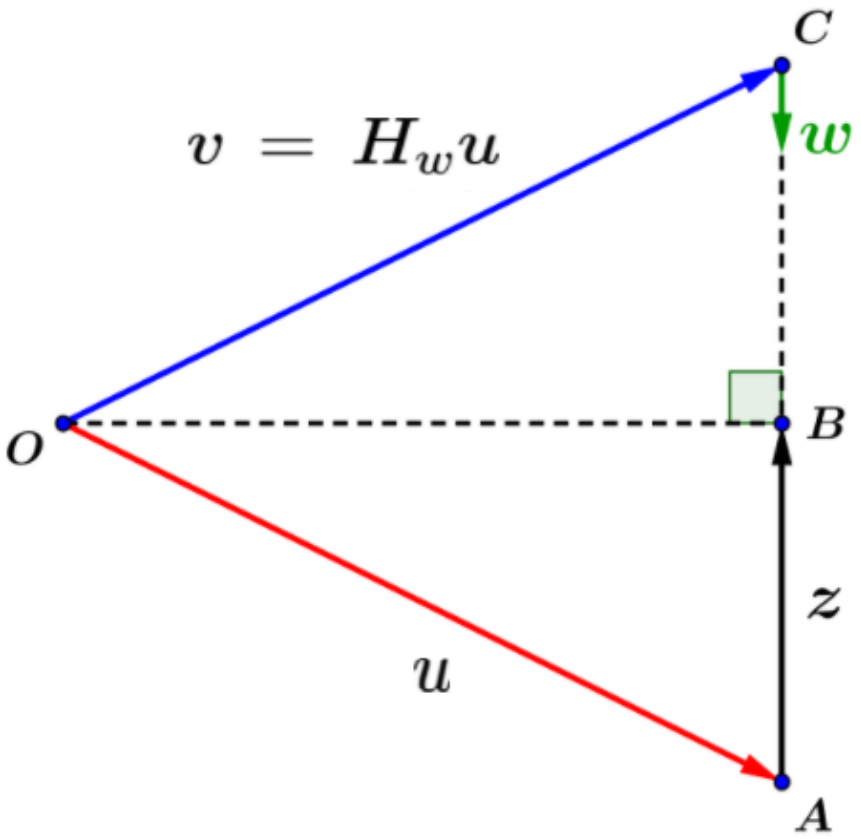
```
def wilkinson_heuristics(A,n):
    get_sgnd = lambda d: 1 if d >= 0 else -1
    alfa_ant = A[n-2,n-2]
    alfa = A[n-1,n-1]
    beta = A[(n-2)+1,n-2]
    dk = (alfa_ant - alfa)/2
    uk = alfa + dk - get_sgnd(dk)*(np.sqrt((dk**2)+(beta**2)))
    return [uk,beta]
```

- *QR_algorithm(A, erro = 1/1000000, spectral_shift = true):* Esta função recebe como parâmetros a matriz A, a qual será aplicada o algoritmo, e o valor do erro, para fins de convergência, e o spcetral_shift, o qual especifica se deve utilizar ou não o deslocamento espectral, caso não seja especificado na chamada da função, o erro admite valor 0,000001 e admite-se o uso do deslocamento espectral. Dentro da implementação da função, é definida uma função auxiliar *get_uk* a qual chama a função *wilkinson_heuristics(A,n)* caso não seja a primeira interação ou spectral_shift = true, caso contrário é retornado o valor 0. A implementação da função seguiu fielmente o pseudocódigo fornecido, havendo dois loops um *for loop*, para iterar sobre subdiagonal da matriz A, e um *while loop*, para que se realize iterações até que o valor de β_k seja menor do que o erro estabelecido. A função, além de retornar as matrizes Λ e V, também é retornado a variável k, a qual indica o número de iterações que o algoritmo necessitou efetuar. A implementação da função está mostrada a seguir:

```python
def QR_algorithm(A, erro = 1/1000000, spectral_shift = True):
    #round_parameter = len(str(int(1/erro)))
    A = A.copy()
    get_uk = lambda A,n,k,spectral_shift: wilkinson_heuristics(A,n)[0] if ((k > 0) and (spectral_shift == True)) else 0
    V = I = np.eye(len(A))
    k = 0
    for m in range(len(A),1,-1):
        beta = wilkinson_heuristics(A,m)[1]
        while(abs(beta)>erro):
            uk = get_uk(A,m,k,spectral_shift)
            R,Q = QR_fatoration(A - (uk*I))
            A = R@Q + uk*I
            V = V@Q
            k = k + 1
            beta = wilkinson_heuristics(A,m)[1]
        A = np.matrix.round(A,6) # A = np.matrix.round(A,round_parameter - 1)
    lamb = A
    return [V,lamb,k]
```

Transformação de Householder

A transformação de Householder, H_w trata-se de uma operação algébrica na qual reflete um certo vetor ua um plano, ortogonal, w resultando em v. Segue abaixo sua sua representação geométrica:



Originalmente, utiliza-se matrizes na transformação de Householder, entretanto ela pode ser simplificada utilizando apenas vetores, o que a torna mais objetiva e a torna mais

eficiente para uma implementação computacional. Segue abaixo sua definição por meio de implementação vetorial:

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

No presente trabalho, a transformação de Householder foi utilizada para uma finalidade específica: aplicar transformação a fim de tornar uma matriz simétrica A_{nxn}em uma matriz tri-diagonal .Para tal finalidade, w_i, sendo i referente a cada iteração, assume, sendo a_i vetor referente a primeira coluna da matriz A da segunda linha em diante, $e$um vetor de mesmo tamanho de a_i com valor 1 em sua primeira posição e zero no restante e δ igual ao sinal de $A_{2,1}$, referente a seguinte equação:

$$w_i = a_i + \delta\|a_i\|e$$

Para que seja realizado a tridiagonalização da matriz, deve-se multiplicar a matriz A_{nxn} a direita e a esquerda por H_{w1}a fim de zerar sua primeira coluna a partir da terceira posição. Em seguida, deve-se aplicar as mesmas operações, porém à submatriz $A'_{n-1\,x\,n-1}$' repetindo tais operações até a matriz A se tornar tridiagonal.

A fim de executar tal algoritmo de tridiagonalização, a dupla implementou as seguintes funções:

- *get_e(n):* Tal função retorna o vetor e, sendo “n” o seu tamanho. Segue abaixo sua implementação:

```
---------------------------------------------------------------------------------------------------------------
def get_e(n):
    e = np.zeros((1,n))
    e[0,0] = 1
    return e
---------------------------------------------------------------------------------------------------------------
```

- *get_alfa(A,j)*: Tal função retorna o vetor α o qual equivale ao vetor correspondente aos n-1 valores da coluna “j” da matriz de entrada “A” nxn. Segue abaixo sua implementação

```
---------------------------------------------------------------------------------------------------------------
def get_alfa(A,j):
    alfa = np.array([])
    for i in range (1,len(A),1):
        alfa = np.append(alfa,A[i,j])
    return alfa
```

- *get_delta(A)*: Tal função retorna o sinal do elemento (2,1) da matriz de entrada “A”. Segue sua implementação:

```
def get_delta(A):
    return (A[1,0]/abs(A[1,0]))
```

- *norm(A)*: Tal função retorna a norma do vetor “A” de entrada. Segue sua implementação:

```
def norm(A):
    x=0
    for i in range (len(A)):
        x=x+(A[i]**2)
    x = np.sqrt(x)
    return x
```

- *get_wi(A)*: A partir da entrada da matriz “A”, esta função retorna o vetor w_i utilizado para a transformação de Householder. Segue sua implementação:

```
def get_wi(A):
    ai = np.array([])
    ai = np.append(ai,get_alfa(A,0))
    n = len(ai)
    wi = ai+(get_delta(A)*norm(ai)*get_e(n))
    return wi
```

- *householder_algorithm(A, debug = false)*: É a função do algoritmo de tridiagonalização em si. Dentro do seu corpo, é implementada a função auxiliar *get_hwi_x(w,x)* a qual retorna a transformação de Householder sobre um vetor x à um vetor w. O funcionamento do algoritmo é pautado principalmente em preencher uma matriz nova composta por zeros com os valores das transformações obtidos em três *for loop* implementados.O primeiro, o qual engloba os outros dois,é utilizado para gerar o vetor $w_i$de cada iteração, gerar a submatriz de A para a próxima iteração e preencher a nova matriz com os valores encontrados. Os dois outros loops são utilizado para fazer as multiplicações de Hwi à esquerda e à direita de A, respectivamente. Para que o valor original de A não seja alterado, é gerado uma cópia para ser apenas utilizado na função. O parâmetro debug é utilizado apenas

para verificar a execução da função, sendo printado a matriz que é retornada Segue abaixo sua implementação:

```
def householder_algorithm(A,debug = False):
    get_hwi_x = lambda w,x: (x -
(2*(np.inner(w,x)/np.inner(w,w))*w))
    A = A.copy()
    n = len(A)
    new_A = np.zeros((n,n))
    for i in range(0,n-2):
        m = len(A)
        wi = get_wi(A)
        for j in range(0,m):
            A[1:m,j] = get_hwi_x(wi,get_alfa(A,j))
        for j in range(0,m):
            A[j,1:m] = get_hwi_x(wi,get_alfa(np.transpose(A),j))
        new_A[i:n,i:n] = A
        A = np.delete(A,0,0)
        A = np.delete(A,0,1)
    new_A[n-2:n,n-2:n] = A
    if(debug==True):
        print("new_A = \n",np.matrix.round(new_A,4),"\n")
    return new_A
```

A fim de verificar a correta implementação das funções acima descritas, foram implementadas as seguintes funções de teste:

- *test_wi(A)*: Tal função recebe uma matriz A e testa se a função *get_alfa(A)* e *get_wi(A)* estão retornando os valores esperados, sendo aplicados na operação da transformação de householder. Segue sua implementação:

```
def test_wi(A):
    alfa = get_alfa(A,1)
    wi = get_wi(A)
    y = alfa - (2*(np.inner(wi,alfa)/np.inner(wi,wi))*wi)
    print("wi =",y)
```

- *test_householder_algorithm()*:Tal função testa a implementação da função *householder_algorithm(A, debug)*, utilizando como entrada a matriz:

$$A = \begin{bmatrix} 2 & -1 & 1 & 3 \\ -1 & 1 & 4 & 2 \\ 1 & 4 & 2 & -1 \\ 3 & 2 & -1 & 1 \end{bmatrix}$$

e a entrada “debug” como TRUE, para assim ser printado o resultado. Segue sua implementação:

```
def test_householder_algorithm():
    A = np.array([[2,-1,1,3],
                  [-1,1,4,2],
                  [1,4,2,-1],
                  [3,2,-1,1]], dtype=float)
    householder_algorithm(A,debug=True)
```

Executando as funções de teste acima, sendo a matriz:

$$A = \begin{bmatrix} 2 & -1 & 1 & 3 \\ -1 & 1 & 4 & 2 \\ 1 & 4 & 2 & -1 \\ 3 & 2 & -1 & 1 \end{bmatrix}$$

como entrada de *test_wi(A)*, são obtidas as seguintes saídas:

```
Execução da função test_wi(A)
wi = [[2.7136021  3.60302269 0.80906807]]

 Execução da função test_householder_algorithm()
new_A =
 [[ 2.      3.3166  0.      0.    ]
 [ 3.3166 -1.2727  2.8313  0.    ]
 [ 0.      2.8313  0.2057  1.5215]
 [ 0.      0.      1.5215  5.067 ]]
```

as quais correspondem aos valores esperados.

Tarefas

Foram construídas 4 diferentes teste para o programa. As duas primeiras são implementações dos **assignment_1_a** e **assignment_1_ b** descritos em no problema **[4.1]**. Os dois últimos testes foram feitos para os tópicos do item **[4.2]**, o terceiro corresponde à solução do problema das treliças planas e o quarto corresponde a visualização das treliças planas a partir de um gif.
Os problemas do tópico [4.1] foram resolvidos no arquivo “assignment1.py” e os do [4.2] no arquivo “assignment2.py”.

Tarefa 1

Nessa primeira tarefa desejamos resolver dois problemas [4.1.a] e [4.1.b], descritos no relatório, que foram implementados nas funções **assignment_1_a** e **assignment_1_ b**. Os dois scrits foram feitos no arquivo **assignment_1.py** pode ser vista abaixa:

assignment1.py - completa

```
--------------------------------------------------------------------------------
#!/usr/bin/env python3
# coding=utf-8

import numpy as np

from householder_method.householder_algorithm import householder_algorithm
from QR_method.QR_algorithm import QR_algorithm

INFO = '''
TAREFA 4.1:
(a) Tarefa 4.1.a: Calculo dos auto valores e autovetores da matriz:
    [2., 4., 1., 1.]
    [4., 2., 1., 1.]
    [1., 1., 1., 2.]
    [1., 1., 2., 1.]

(b) Tarefa 4.1.b: Calculo dos auto valores e autovetores da matriz:
    [ n  n-1 n-2 ...  2   1 ]
```

```
    [n-1 n-1 n-2 ...  2   1 ]
    [n-2 n-2 n-2 ...  2   1 ]
    [ :   :   :  ...  :   : ]
    [ 2   2   2   2   2   1 ]
    [ 1   1   1   1   1   1 ]
'''
def make_matrixB(n):
    B = np.zeros((n,n),dtype = float)
    for i in range (2,n+2,1):
        add = (i-1)*np.ones((n-i+2,n-i+2),dtype=float)
        B[0:n-i+2,0:n-i+2] = add[0:n-i+2:,0:n-i+2]
    return B

def make_matrixB_alt(n):
    B= np.zeros((n,n))
    B[0,0] = n
    for i in range (1,len(B),1):
        B[i,0:i] = n-i
        B[0:i,i] = n-i
        B[i,i] = n-i
    return B
def get_eigenvalues(n):
    lamb = np.array([])
    eigenvaluesi = lambda i: ((1/2)*((1 - np.cos((2*i-1)*np.pi/(2*n+1)))**(-1)))
    for i in range(1,n+1):
        lamb = np.append(lamb,eigenvaluesi(i))
    return lamb

def assignment_1_a():
    A = np.array([[2., 4., 1., 1.],
                  [4., 2., 1., 1.],
                  [1., 1., 1., 2.],
                  [1., 1., 2., 1.]])

    A = householder_algorithm(A)
    V,lamb = QR_algorithm(A)[0:2]
    print("V =\n",V)
    print("lamb =\n",np.diagonal(lamb))

def assignment_1_b():
    n = 20
    B = make_matrixB(n)
    B = householder_algorithm(B)
    V,lamb = QR_algorithm(B)[0:2]
    eigenvalues = get_eigenvalues(n)

    print("autovetores =\n",V)
    print("autovalores obtidos =\n",np.diagonal(lamb))
    print("autovalores esperados =\n",eigenvalues)

def assignment_1():
    print(INFO,"\n")
```

```
    item = input("Gostaria de ver qual item? ")
    if ((item == "a") or (item == "4.1.a")):
        assignment_1_a()
        print("\n")
    elif ((item == "b") or (item == "4.1.b")):
        assignment_1_b()
        print("\n")
    else:
        print("your input isn't valid \nplease, try again\n")

if __name__ == "__main__":
    try:
        assignment_1()

    except KeyboardInterrupt:
        print("\n Better luck next time")
```

Tarefa 1.a

A tarefa 1.a solicita que calcule os autovetores e autovalores da matriz abaixo:

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

Como podemos ver, se trata de uma matriz quadrada simétrica, logo, pode-se aplicar a equação de de Householder para tridiagonizar a matriz e depois usar o Algoritmo QR para diagonizar e obter os autovetores e autovalores.

A função feita para resolver o problema pode ser vista abaixo:

```
def assignment_1_a(debug = False):
    np.set_printoptions(precision=5,threshold=5) #print options
    A = np.array([[2., 4., 1., 1.],
                  [4., 2., 1., 1.],
                  [1., 1., 1., 2.],
                  [1., 1., 2., 1.]])

    A = householder_algorithm(A)
    V,lamb = QR_algorithm(A)[0:2]
    print("autovetores =\n",V)
    print("autovetores =\n",np.diagonal(lamb),"\n")

    if(debug == True):
```

```
        for i in range(0,len(A)):
            print('teste {j}'.format(j=i))
            print(A@V[:,i],"\n",np.diagonal(lamb)[i]*V[:,i],"\n")
-------------------------------------------------
```

Valores obtidos:

```
autovetores =
 [[ 6.32455495e-01  7.07106757e-01  3.16227895e-01  0.00000000e+00]
 [-7.45356019e-01  6.66666637e-01  8.24102641e-08  0.00000000e+00]
 [-2.10818529e-01 -2.35702417e-01  9.48683255e-01  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00]]
autovetores =
 [ 7.000001 -2.000001  2.        -1.      ]
```

No casos dos teste, foi testado a seguinte relação: Av = λv, que define um autovalor e autovetor

```
teste 0
[ 4.42719e+00 -5.21749e+00 -1.47573e+00 -1.17028e-16]
 [ 4.42719 -5.21749 -1.47573  0.     ]

teste 1
[-1.41421e+00 -1.33333e+00  4.71404e-01 -1.30841e-16]
 [-1.41421 -1.33333  0.47141 -0.     ]

teste 2
[ 6.32455e-01 -3.61363e-07  1.89737e+00 -1.75542e-16]
 [6.32456e-01 1.64821e-07 1.89737e+00 0.00000e+00]

teste 3
[-2.22045e-16  0.00000e+00 -4.44089e-16 -1.00000e+00]
 [-0. -0. -0. -1.]
```

Tarefa 1.b

A tarefa 1.b solicita que calcule os autovetores e autovalores da matriz abaixo:

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

Assim Como no item A, trata-se um de uma matriz simétrica, logo, podemos aplicar a equação de Householder para tridiagonalizar a matriz e depois usar o Algoritmo QR para diagonalizar e obter os autovetores e autovalores.

A função feita para resolver o problema pode ser vista abaixo:

```
-----------------------------------------------------------
def assignment_1_b():
    n = 20
    B = make_matrixB(n)
    B = householder_algorithm(B)
    V,lamb = QR_algorithm(B)[0:2]
    eigenvalues = get_eigenvalues(n)

    print("autovetores =\n",V)
    print("autovetores =\n",np.diagonal(lamb))
    print("autovalores esperados =\n",eigenvalues)
-----------------------------------------------------------
```

Sendo que a as funções **make_matrixB(n)** e **get_eigenvalues(n)** foram feitas exclusivamente para essa função.

A função **make_matrixB(n)** ela recebe um valor n e produz a matriz descrita acima.

```
----------------------------------------
def make_matrixB_alt(n):
    B= np.zeros((n,n))
    B[0,0] = n
    for i in range (1,len(B),1):
        B[i,0:i] = n-i
        B[0:i,i] = n-i
        B[i,i] = n-i
    return B
----------------------------------------
```

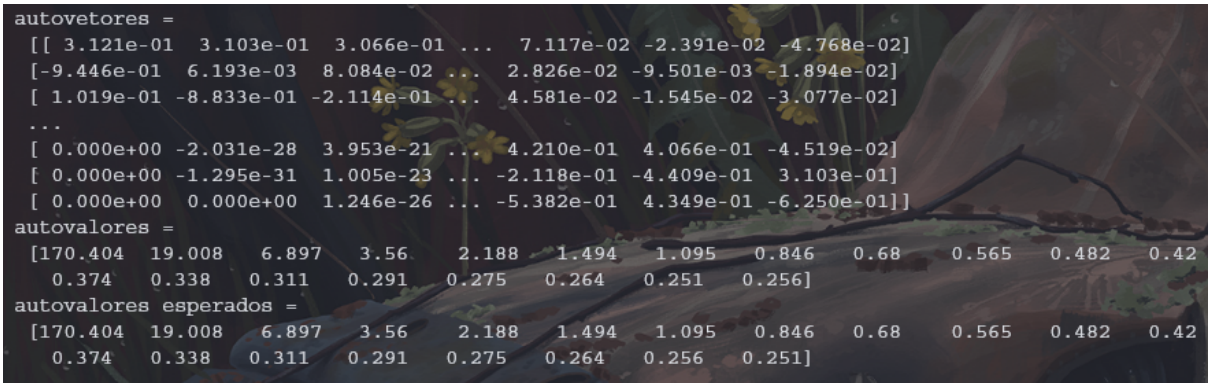
A função **get_eigenvalues(n)** ela recebe um valor n e produz os autovalores esperados para a matriz A solicitada para esse exercício seguindo a seguinte relação:

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

O código da função pode ser visto abaixo:

```
-------------------------------------------------
def get_eigenvalues(n):
    lamb = np.array([])
    eigenvaluesi = lambda i: ((1/2)*((1 - np.cos((2*i-1)*np.pi/(2*n+1)))**(-1)))
    for i in range(1,n+1):
        lamb = np.append(lamb,eigenvaluesi(i))
    return lamb
```

Valores obtidos na **assignment_1_b:**



```
autovetores =
 [[ 3.121e-01  3.103e-01  3.066e-01 ...  7.117e-02 -2.391e-02 -4.768e-02]
 [-9.446e-01  6.193e-03  8.084e-02 ...  2.826e-02 -9.501e-03 -1.894e-02]
 [ 1.019e-01 -8.833e-01 -2.114e-01 ...  4.581e-02 -1.545e-02 -3.077e-02]
 ...
 [ 0.000e+00 -2.031e-28  3.953e-21 ...  4.210e-01  4.066e-01 -4.519e-02]
 [ 0.000e+00 -1.295e-31  1.005e-23 ... -2.118e-01 -4.409e-01  3.103e-01]
 [ 0.000e+00  0.000e+00  1.246e-26 ... -5.382e-01  4.349e-01 -6.250e-01]]
autovalores =
 [170.404  19.008   6.897   3.56    2.188   1.494   1.095   0.846   0.68    0.565   0.482   0.42
  0.374   0.338   0.311   0.291   0.275   0.264   0.251   0.256]
autovalores esperados =
 [170.404  19.008   6.897   3.56    2.188   1.494   1.095   0.846   0.68    0.565   0.482   0.42
  0.374   0.338   0.311   0.291   0.275   0.264   0.256   0.251]
```

Para o teste, somente foi feita a comparação dos autovalores obtidos com os esperados.

Tarefa 2

Para essa tarefa, pede-se para resolver o problema de treliças planas a “n” nós móveis e “m” barras, com energia total 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 (hi , vi), de tal forma que podemos criar o vetor de deslocamento:

$$x_{2i-1} = h_i = \text{ deslocamento horizontal do nó i}$$
$$x_{2i} = v_i = \text{ deslocamento vertical do nó i}$$

Uma aproximação considerada para a resolução do problema é que a massa de cada barra está concentrada nos nós que a compõem. Sendo mi,j a massa da barra que conecta os nós i e j, então tal barra contribui com $0.5mi,j$ para mi e $0.5mi,j$ para mj , sendo mk a massa concentrada do nó.

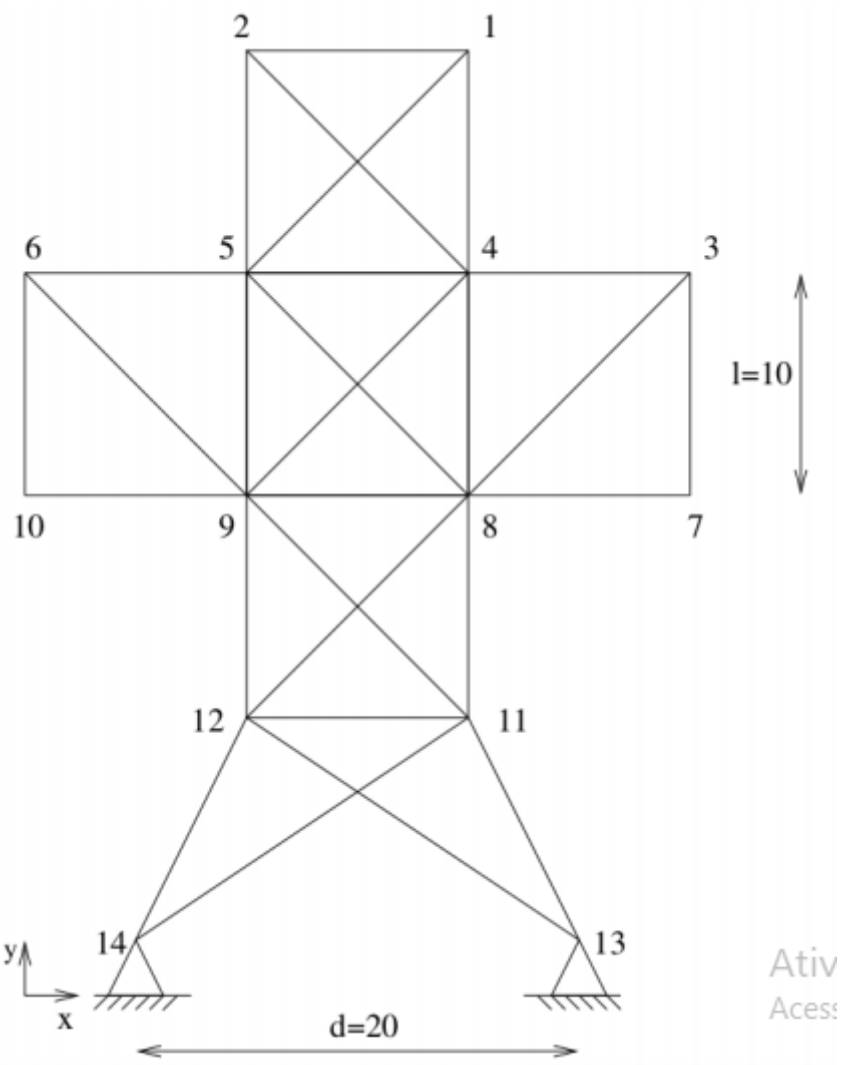
Para cada barra, foi definido uma equação de rigidez é dada por:

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

Sendo Lij é o comprimento da barra e C e S:

$$C = \cos\theta_{\{i,j\}}, \quad S = \operatorname{sen}\theta_{\{i,j\}},$$

Dada a treliça:



A sua matriz de rigidez total K pode ser construída da seguinte forma: inicie K com zeros em todas as entradas. Depois, para cada barra {i, j}, adicione a sua contribuição de (1) nas posições.

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

Sendo que a contribuição das barras {11,14} e {12,13} não pode ser considerada, pois os nós 13 e 14 não se movimentam.

A matrix das massas totais , 24 x 24, pode ser construída a partir da seguinte relação:

$$M_{2i-1,2i-1} = m_i,$$
$$M_{2i,2i} = m_i,$$

para $i = 1, 2, \ldots, 12$.

A equação do movimento das treliças é dada pela equação:

$$M\ddot{\boldsymbol{x}} + K\boldsymbol{x} = 0.$$

A equação generalizada dos autovalores:

$$\boldsymbol{x}(t) = \boldsymbol{z}e^{i\omega t}$$

Como a matriz M é diagonal com entradas diagonais positivas, podemos transformar o problema acima em um problema convencional de autovalores fazendo-se a substituição:

$$\boldsymbol{z} = M^{-\frac{1}{2}}\boldsymbol{y},$$

Assim, podemos que concluir que:

$$\tilde{K}\boldsymbol{y} = \omega^2\boldsymbol{y},$$

Sendo:

$$\tilde{K} = M^{-\frac{1}{2}}KM^{-\frac{1}{2}}$$

Com essas relações bem estabelecidas, podemos concluir que:

Para resolver o exercício foi feito o seguinte script:

Assignment_2:

Para resolução desse exercício, a princípio é lido os valores do input C em relação ao valores fornecidos para cada barra:

```
info = read_input_c()
```

`info = read_input_c()`

sendo info um matrix, com cada linha representando os valores em float lidos no input-c

Depois disso, é criado um vetor de objetos “Beam”, sendo que cada objeto recebe como parâmetro o 1° e o 2° nó, o ângulo e a largura da barra. Esse objeto já calcula o K individual de cada e barra e sua respectiva massa.

```
 beams = np.array([Beam(info[i][0],info[i][1],info[i][2],info[i][3]) for i in
range(0,NUM_TRE)]) #create beams objects
```

A partir desse dos parâmetros K do vetor de objetos é criado o K total e a matriz M

```
    K = make_total_k_matrix(beams) #create K matrix
    M = make_M_matrix(beams) #create M matrix
```

A partir de K e M, podemos encontrar o valor de Ktil seguindo a seguinte relação:

$$\tilde{K} = M^{-\frac{1}{2}} K M^{-\frac{1}{2}}$$

```
inv_sqrt_m = inv_diagonal_matrix(np.sqrt(M)) # invert M matrix
K_til = inv_sqrt_m@K@inv_sqrt_m #built Ktil
```

Depois disso, seguindo o seguinte raciocínio:

$$Kz = \omega^2 Mz$$
$$z = M^{-1/2} y \ , \ \tilde{K} = M^{-1/2} K M^{-1/2}$$
$$\tilde{K} y = \omega^2 y$$
$$y \Rightarrow \text{autovetor de } \tilde{K}$$
$$\omega^2 \Rightarrow \text{autovalor de } \tilde{K}$$

Dessa forma, como sabemos que K til é simétrica, aplicamos a equação de Householder junto com a QR para encontrar os autovalores e autovetores de K til

```
build_diagonal_matrix = lambda matrix: QR_algorithm(householder_algorithm(matrix))
y_vector,w_vector,ite = build_diagonal_matrix(K_til) #get y and w² with Ktil

w_vector = np.sqrt(np.diagonal(w_vector)) #get w
```

Como temos todos os valores da frequência angular, pode-se encontrar a frequência seguindo a seguinte relação:

$$\omega = 2.\pi.f$$

```
f_vector = (w_vector)/(2*np.pi) #get frequencies
per_vector = 1/f_vector # get periods
```

Como já temos todas frequências, já pode encontrar as cinco menores e seus respectivos modos de vibração:

```
len_f = len(f_vector)
print("menores frequencias = ", f_vector[len_f-5:len_f]) #print "menores frequencias"
```

```
    print("modo de vibração = ", np.array([1,4,5,3,2])) #print "menores frequencias"
    print("menores frequencias angulares = ", w_vector[len_f-5:len_f]) #print "menores
frequencias angulares"
```

Depois disso, seguindo a seguinte relação podemos encontrar a posição de cada um dos nós:

$$
\begin{aligned}
&M\ddot{x} + Kx = 0 \\
&x(t) = z\,e^{j\omega t} = z(A\cos(\omega t) + Bjsen(\omega t)) \\
&x(0) = zA = z.\,e^{0} \Rightarrow A = 1 \\
&\dot{x}(0) = 0 = jz\omega B \Rightarrow B = 0 \\
&x = z\cos(\omega t)
\end{aligned}
$$

O código para a tarefa 2, com tudo implementado, pode ser visto abaixo:

```
---------------------------------------------------------------------------------------
def assignment_2():
    np.set_printoptions(precision=3,threshold=5) #print options
    dt = 0.0001
    time = 1

    info = read_input_c() #read file and get the informations

    beams = np.array([Beam(info[i][0],info[i][1],info[i][2],info[i][3]) for i in range(0,NUM_TRE)]) #create
beams objects

    K = make_total_k_matrix(beams) #create K matrix
    M = make_M_matrix(beams) #create M matrix
    inv_sqrt_m = inv_diagonal_matrix(np.sqrt(M)) # invert M matrix
    K_til = inv_sqrt_m@K@inv_sqrt_m #built Ktil

    build_diagonal_matrix = lambda matrix: QR_algorithm(householder_algorithm(matrix))
    y_vector,w_vector,ite = build_diagonal_matrix(K_til) #get y and w² with Ktil

    w_vector = np.sqrt(np.diagonal(w_vector)) #get w
    z_vector = inv_sqrt_m@y_vector #get z

    f_vector = (w_vector)/(2*np.pi) #get frequencies
    per_vector = 1/f_vector # get periods

    x_vector = lambda t: z_vector*np.cos(w_vector*t)
    x_vector_vertical = lambda t,pos: get_x_component(x_vector,t,pos,type = 0) #get vertical moviment
    x_vector_horizontal = lambda t,pos: get_x_component(x_vector,t,pos,type = 1) #get horizontal moviment
    #test_x_moviment(x_vector,x_vector_vertical,x_vector_horizontal)

    len_f = len(f_vector)
    print("menores frequencias = ", f_vector[len_f-5:len_f]) #print "menores frequencias"
    print("modo de vibração = ", np.array([1,4,5,3,2])) #print "menores frequencias"
    print("menores frequencias angulares = ", w_vector[len_f-5:len_f]) #print "menores frequencias
angulares"
```

```
    print("periodo das menores frequencias = ", per_vector[len_f-5:len_f]) #print "periodo das menores
frequencias"

    t_points,x_points = get_points(dt,x_vector_horizontal,23,time=time)
    #print("x_vector_horizontal =\n",x_points)
    plot_graphics(t_points,x_points,y_label = 'X')
    plot_all_together(t_points,x_points)
    #print_separado(t_points,x_points)
```

Para Resolução dessa tarefa com treliças, foram feita 1 classe e algumas funções para facilitar a construção do script **Assignment_2** que podem ser vistas por completo abaixo:

Classe Beam

A classe Beam foi implementada para representar cada barra como objeto no sistema de treliças descrito. Seus atributos referem-se aos seus nós, assim como o ângulo em relação a horizontal, sua massa, seu comprimento, sua densidade, seu módulo de elasticidade, a área de sua seção transversal e sua matriz de rigidez.

Como métodos, há *make_k_matrix(self) e find_mass(self)*, os quais definem os atributos da matriz de rigidez e da massa, respectivamente, e há os métodos *get_k(self)* e *get_mass(self)*, os quais retornam a matriz de rigidez e a massa.

Definição da classe *Beam*

```
class Beam:
    def __init__(self,no1,no2,ang,lenght):
        self.no1 = no1
        self.no2 = no2
        self.ang = ang*np.pi/180
        self.lenght = lenght
        self.A = A
        self.E = E
        self.dens = DENSIDADE

        self.massa = 0
        self.K = 0

        self.make_k_matrix()
        self.find_mass()

    def __str__(self):
        return 'no1 ={no_1}, no2 ={no_2}, ang ={ang}, lenght
={lenght}'.format(no_1=self.no1,no_2=self.no2,ang=self.ang,lenght=self.lenght)

    def make_k_matrix(self):
        x = (self.A*self.E)/self.lenght
        C = np.cos(self.ang)
        S = np.sin(self.ang)
        matrix = np.array([ [C**2,C*S,-(C**2),-C*S],
                            [C*S,S**2,-C*S,-(S**2)  ],
```

```
                            [-(C**2),-C*S,C**2,C*S],
                            [-C*S,-(S**2),C*S,S**2]],dtype=float)
        self.K = x*matrix

    def find_mass(self):
        self.mass = self.A*self.lenght*self.dens
        pass

    def get_K(self):
        return self.K

    def get_mass(self):
        return self.mass
```

Função *make_total_K_matrix(V):*

Esta função recebe como parâmetro um vetor V de objetos *Beam*. Seu objetivo é construir a matriz de rigidez total do sistema de treliças. Para isso, o vetor V é varrido em um *for loop*, e cada valor de K extraído de V através da função *get_k()*, do objeto *Beam*, é adicionado à matriz de rigidez total. Segue sua implementação:

```
def make_total_k_matrix(V):
    m = len(V)
    k_matrix = np.zeros((m,m),dtype=float)
    for i in range (0,m):
        aux_k = V[i].get_K()
        i_pos = int(2*(V[i].no1) - 1)
        j_pos = int(2*(V[i].no2) - 1)
        k_matrix[(i_pos-1),(i_pos-1)] += aux_k[0,0]
        k_matrix[(i_pos-1),(i_pos)]   += aux_k[0,1]
        k_matrix[(i_pos-1),(j_pos-1)] += aux_k[0,2]
        k_matrix[(i_pos-1),(j_pos)]   += aux_k[0,3]
        k_matrix[(i_pos),  (i_pos-1)] += aux_k[1,0]
        k_matrix[(i_pos),  (i_pos)]   += aux_k[1,1]
        k_matrix[(i_pos),  (j_pos-1)] += aux_k[1,2]
        k_matrix[(i_pos),  (j_pos)]   += aux_k[1,3]
        k_matrix[(j_pos-1),(i_pos-1)] += aux_k[2,0]
        k_matrix[(j_pos-1),(i_pos)]   += aux_k[2,1]
        k_matrix[(j_pos-1),(j_pos-1)] += aux_k[2,2]
        k_matrix[(j_pos-1),(j_pos)]   += aux_k[2,3]
        k_matrix[(j_pos),  (i_pos-1)] += aux_k[3,0]
        k_matrix[(j_pos),  (i_pos)]   += aux_k[3,1]
        k_matrix[(j_pos),  (j_pos-1)] += aux_k[3,2]
        k_matrix[(j_pos),  (j_pos)]   += aux_k[3,3]
    for i in range(m-1,m-5,-1):
        k_matrix = np.delete(k_matrix,i,0)
        k_matrix = np.delete(k_matrix,i,1)
    return k_matrix
```

Função *make_M_matrix(V):*

Esta função recebe como parâmetro um vetor V de objetos *Beam*. Seu objetivo é construir a matriz de massa do sistema de treliças. Para isso, o vetor V é varrido em um *for loop*, e cada valor massa extraído de V através da função *get_mass()*, do objeto *Beam*, é adicionado à matriz M. Segue sua implementação:

```
def make_M_matrix(V):
    n = len(V)
    m_matrix = np.zeros((n,n),dtype=float)
    for i in range(0,n):
        pos1 = int(2*(V[i].no1) - 1)
        pos2 = int(2*(V[i].no2) - 1)
        massa = V[i].get_mass()

        m_matrix[pos1 - 1,pos1 - 1] += massa/2
        m_matrix[pos1,pos1] += massa/2
        m_matrix[pos2 - 1,pos2 - 1] += massa/2
        m_matrix[pos2,pos2] += massa/2

    for i in range(n-1,n-5,-1):
        m_matrix = np.delete(m_matrix,i,0)
        m_matrix = np.delete(m_matrix,i,1)
    return m_matrix
```

Função *inv_diagonal_matrix(matrix):*

Esta função recebe como parâmetro uma matriz diagonal e retorna a sua inversa. Por ser uma matriz diagonal, sua inversa pode ser calculada apenas invertendo os seus da diagonal principal. Segue sua implementação:

```
def inv_diagonal_matrix(matrix):
    diagonal = np.diagonal(matrix)
    if (len(np.where(diagonal == 0)[0]) == 0):
        diagonal = 1/diagonal
        matrix_inv = np.diag(diagonal)
        return matrix_inv
    else:
        print("erro inv_diagonal_matrix")
```

Função *get_x_component(matrix,t,pos,type):*

Esta função recebe como parâmetro a função *matrix*, a qual retorna a posição da barra no tempo, *t* como tempo, *pos* referente a coluna da *matrix* . Seu objetivo é retornar a

componente vertical o horizontal, a depender do parâmetro *type*, de x na matriz *matrix* no tempo *t*

```
def get_x_component(matrix,t,pos,type):
    x0 = matrix(0)[0:12,pos]
    matrix_t = np.zeros(shape = [len(x0),1], dtype = float)
    for i in range(0,len(x0)):
        matrix_t[i] = matrix(t)[(2*i)+type,pos]
    return matrix_t
```

Por fim, obtemos o seguinte os valores para a tarefa 2

```
menores frequencias =
 [ 3.914 22.729 24.004 15.073 14.644]
menores frequencias angulares =
 [ 24.593 142.81  150.822  94.703  92.012]
modo de vibração =
 [[ 2.288e-05  3.496e-03 -3.869e-03 -8.982e-05 -7.789e-04]
 [ 1.623e-05  4.220e-03 -4.214e-03 -9.658e-05 -9.014e-04]
 [ 1.502e-05 -2.707e-03  1.434e-03  2.922e-05  4.664e-04]
 ...
 [ 3.043e-03  6.671e-08 -2.404e-06  1.071e-04 -5.116e-06]
 [-4.903e-03 -6.550e-08  9.890e-08 -1.682e-05  1.856e-05]
 [-2.262e-03  1.017e-07  4.522e-06 -1.665e-04 -5.119e-05]]
```

Análise da tarefa 2:

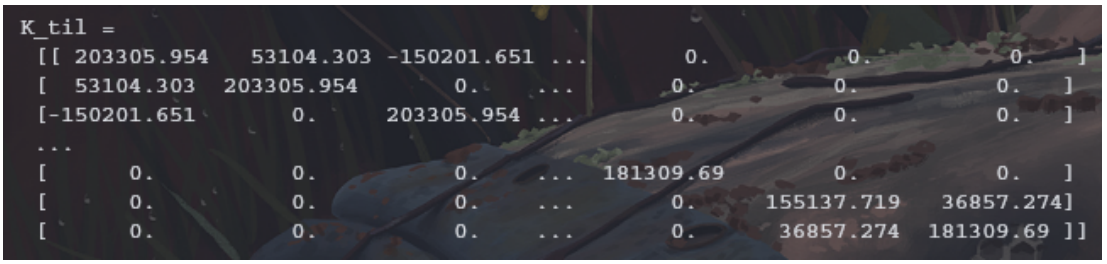
Valor da matriz M obtida:

```
M =
 [[13315.433     0.        0.     ...     0.        0.        0.   ]
 [    0.    13315.433     0.     ...     0.        0.        0.   ]
 [    0.        0.    13315.433 ...     0.        0.        0.   ]
 ...
 [    0.        0.        0.     ... 24706.59     0.        0.   ]
 [    0.        0.        0.     ...     0.    24706.59     0.   ]
 [    0.        0.        0.     ...     0.        0.    24706.59 ]]
```

Valor da matriz K obtida:

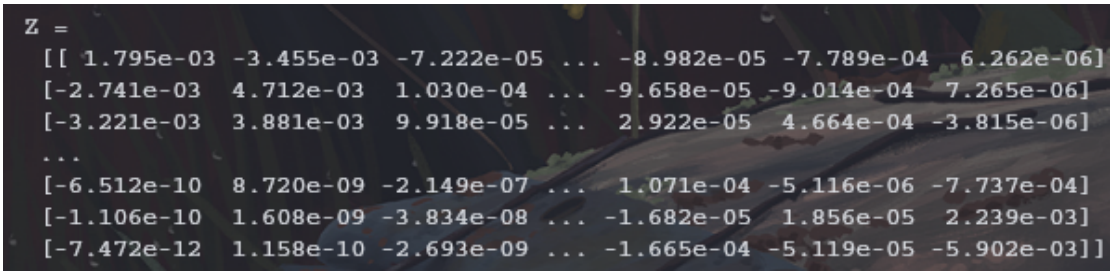
```
K =
 [[ 2.707e+09  7.071e+08 -2.000e+09 ...  0.000e+00  0.000e+00  0.000e+00]
 [ 7.071e+08  2.707e+09  0.000e+00 ...  0.000e+00  0.000e+00  0.000e+00]
 [-2.000e+09  0.000e+00  2.707e+09 ...  0.000e+00  0.000e+00  0.000e+00]
 ...
 [ 0.000e+00  0.000e+00  0.000e+00 ...  4.480e+09  0.000e+00  0.000e+00]
 [ 0.000e+00  0.000e+00  0.000e+00 ...  0.000e+00  3.833e+09  9.106e+08]
 [ 0.000e+00  0.000e+00  0.000e+00 ...  0.000e+00  9.106e+08  4.480e+09]]
```

Valor da matriz K til obtida:



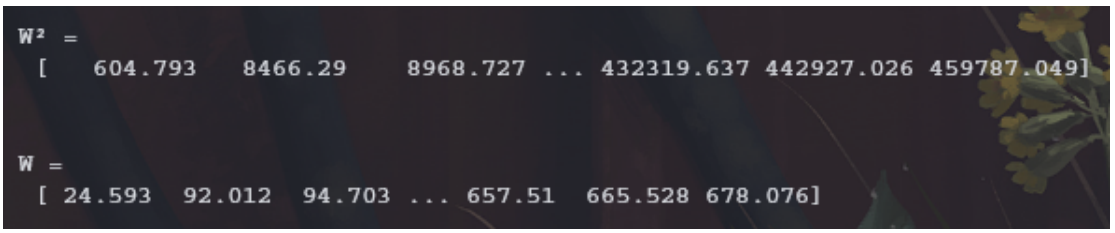
```
K_til =
 [[ 203305.954   53104.303 -150201.651 ...       0.          0.          0.   ]
 [  53104.303  203305.954       0.     ...       0.          0.          0.   ]
 [-150201.651       0.     203305.954 ...       0.          0.          0.   ]
 ...
 [      0.          0.          0.     ...  181309.69        0.          0.   ]
 [      0.          0.          0.     ...       0.     155137.719   36857.274]
 [      0.          0.          0.     ...       0.      36857.274  181309.69 ]]
```

Valor da matriz Z obtida:



```
Z =
 [[ 1.795e-03 -3.455e-03 -7.222e-05 ... -8.982e-05 -7.789e-04  6.262e-06]
 [-2.741e-03  4.712e-03  1.030e-04 ... -9.658e-05 -9.014e-04  7.265e-06]
 [-3.221e-03  3.881e-03  9.918e-05 ...  2.922e-05  4.664e-04 -3.815e-06]
 ...
 [-6.512e-10  8.720e-09 -2.149e-07 ...  1.071e-04 -5.116e-06 -7.737e-04]
 [-1.106e-10  1.608e-09 -3.834e-08 ... -1.682e-05  1.856e-05  2.239e-03]
 [-7.472e-12  1.158e-10 -2.693e-09 ... -1.665e-04 -5.119e-05 -5.902e-03]]
```

Valor de w² e w obtidos :



```
W² =
 [   604.793   8466.29    8968.727 ... 432319.637 442927.026 459787.049]

W =
 [ 24.593  92.012  94.703 ... 657.51  665.528 678.076]
```

Conclusão

Ao fim da realização do presente Exercício Programa, a dupla aprimorou-se ainda mais na utilização de métodos numéricos para solução de problemas. Junto com o algoritmo QR, desenvolvido no Exercício Programa anterior, ao implementar as transformações de Householder, tornou-se possível a encontrar a solução de ainda mais problemas de engenharia, como o que foi proposto neste exercício.

Além disso, ao implementar o programa desenvolvido, a dupla se superou atentando-se mais à eficiência computacional. Ao substituir algumas operações com matrizes múltiplas da identidade por operações entre vetores, o programa necessitou realizar menos operações para encontrar o mesmo resultado, aumentando assim sua velocidade de execução.

Por fim, após realizar todos testes implementados, foi obtido resultados condizentes e satisfatórios, indicando que foi alcançado êxito na realização da atividade, o que motiva o estudo para poder aplicar tais métodos em futuros desafios os quais serão exigidos na vida profissional.

Referências

- **Fatoração QR - Transformações de Householder.** Disponível em: <https://www.ime.unicamp.br/~marcia/AlgebraLinear/fat_QR%28householder%29.html> Acesso em: 21 de julho de 2021

Apêndice

- Valores disponibilizados no “input_a” fornecido pela disciplina:

```
4
2. 4. 1. 1.
4. 2. 1. 1.
1. 1. 1. 2.
1. 1. 2. 1.
```

- Valores disponibilizados no “input_b” fornecido pela disciplina:

```
20
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 2 1
19 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 2 1
18 18 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 2 1
17 17 17 17 16 15 14 13 12 11 10 9 8 7 6 5 4 2 1
16 16 16 16 16 15 14 13 12 11 10 9 8 7 6 5 4 2 1
15 15 15 15 15 15 14 13 12 11 10 9 8 7 6 5 4 2 1
14 14 14 14 14 14 14 13 12 11 10 9 8 7 6 5 4 2 1
13 13 13 13 13 13 13 13 12 11 10 9 8 7 6 5 4 2 1
12 12 12 12 12 12 12 12 12 11 10 9 8 7 6 5 4 2 1
11 11 11 11 11 11 11 11 11 11 10 9 8 7 6 5 4 2 1
10 10 10 10 10 10 10 10 10 10 10 9 8 7 6 5 4 2 1
 9  9  9  9  9  9  9  9  9  9  9 9 8 7 6 5 4 2 1
 8  8  8  8  8  8  8  8  8  8  8 8 8 7 6 5 4 2 1
 7  7  7  7  7  7  7  7  7  7  7 7 7 7 6 5 4 2 1
 6  6  6  6  6  6  6  6  6  6  6 6 6 6 6 5 4 2 1
 5  5  5  5  5  5  5  5  5  5  5 5 5 5 5 5 4 2 1
 4  4  4  4  4  4  4  4  4  4  4 4 4 4 4 4 4 2 1
 3  3  3  3  3  3  3  3  3  3  3 3 3 3 3 3 3 2 1
 2  2  2  2  2  2  2  2  2  2  2 2 2 2 2 2 2 2 1
 1  1  1  1  1  1  1  1  1  1  1 1 1 1 1 1 1 1 1
```

- Valores disponibilizados no “input_c” fornecido pela disciplina:

```
14 12 28
```

```
7800 0.1 200
1 2 0 10
1 4 90 10
1 5 45 14.14213562373095
2 5 90 10
2 4 135 14.14213562373095
3 4 0 10
3 7 90 10
3 8 45 14.14213562373095
4 5 0 10
4 8 90 10
4 9 45 14.14213562373095
5 6 0 10
5 9 90 10
5 8 135 14.14213562373095
6 9 135 14.14213562373095
6 10 90 10
7 8 0 10
8 9 0 10
8 11 90 10
8 12 45 14.14213562373095
9 10 0 10
9 11 135 14.14213562373095
9 12 90 10
11 12 0 10
11 13 116.5650511770780 11.18033988749895
11 14 33.69006752597979 18.02775637731995
12 13 146.3099324740202 18.02775637731995
12 14 63.43494882292201 11.18033988749895
```