



# PMR-3401: Mecânica Computacional para Mecatrônica

Exercício-Programa 3

feito por

João Rodrigo Windisch Olenscki 10773224  
Lui Damianci Ferreira 10770579

**Professores:** Prof. Dr. Emilio Carlos Nelli Silva  
Prof. Dr. Flávio Buiochi

29 de julho de 2021



## Lista de Figuras

1	Esquema da ponte a ser analisada (elaborado pelos professores)	6
2	Modos de vibrar encontrados no Python	13
3	Análise transiente com condição inicial não nula	15
4	Análise transiente com condição inicial nula	16
5	Análise da discretização temporal com $v_a = 1 \text{ m/s}$	17
6	Diagrama de resposta em frequência obtido no Python	18
7	Resultado da análise transiente no Ansys	20
8	Diagrama de resposta em frequência obtido no Ansys	21
9	Peça bi-simétrica a ser analisada (desenho elaborado pelos professores)	23
10	Deslocamentos para a estrutura deformada	24
11	Tensões de von Mises nos pontos de interesse para $\Delta x = 1 \text{ mm}$	25
12	Tensões de von Mises nos pontos de interesse para $\Delta x = 0.5 \text{ mm}$	25

## Listas de Tabelas

1	Frequências naturais da estrutura a partir do Python $\Delta x = 1 \text{ m}$	12
2	Frequências naturais da estrutura a partir do Python $\Delta x = 2 \text{ m}$	12
3	Frequências naturais da estrutura a partir do Python $\Delta x = 3 \text{ m}$	12
4	Frequências naturais da estrutura a partir do Python e do ANSYS para $\Delta x = 1 \text{ m}$	19
5	Frequências naturais da estrutura a partir do Python e do ANSYS para $\Delta x = 2 \text{ m}$	19
6	Frequências naturais da estrutura a partir do Python e do ANSYS para $\Delta x = 3 \text{ m}$	19

# Sumário

<b>1</b>	<b>Introdução</b>	<b>5</b>
<b>2</b>	<b>Fundamentos Teóricos</b>	<b>6</b>
2.1	Matrizes de massa e rigidez . . . . .	6
2.2	Análise Modal . . . . .	9
2.3	Análise Harmônica . . . . .	9
2.4	Análise Transiente . . . . .	10
2.4.1	Método da aceleração média constante . . . . .	10
<b>3</b>	<b>Implementação Computacional</b>	<b>10</b>
<b>4</b>	<b>Exercício 1</b>	<b>12</b>
4.1	Item a) . . . . .	12
4.1.1	Sub-item i) . . . . .	12
4.1.2	Sub-item ii) . . . . .	13
4.1.3	Sub-item iii) . . . . .	17
4.2	Item b) . . . . .	18
4.2.1	Sub-item i) . . . . .	18
4.2.2	Sub-item ii) . . . . .	21
<b>5</b>	<b>Exercício 2</b>	<b>23</b>
5.1	Item a) . . . . .	23
5.2	Item b) . . . . .	24
5.3	Item c) . . . . .	26
<b>6</b>	<b>Conclusões</b>	<b>27</b>
<b>Apêndices</b>		<b>28</b>
<b>A</b>	<b><i>Listing do script em Python</i></b>	<b>28</b>

# 1 Introdução

Neste exercício programa foi proposta a aplicação do Método de Elementos Finitos (MEF) para a análise dinâmica, transiente e harmônica de uma ponte através de simulações por *scripts* do Python e também por meio do *software* comercial ANSYS. Através deste tipo de atividade, torna-se possível aos alunos a experimentação da aplicação manual de um dos métodos mais utilizados para a resolução deste tipo de análise em engenharia.

Este relatório visa explicar os fundamentos dos problemas estudados, bem como apresentar resultados das análises realizadas por meio de gráficos e diagramas. Além disso, as soluções realizadas em ANSYS e em Python serão comparadas e analisadas em conjunto, de forma a se identificarem as semelhanças e diferenças nos resultados gerados por cada método.

O relatório foi organizado em Seções: a Seção 2 apresenta os fundamentos teóricos da teoria de vigas e treliças para MEF e, em particular, suas aplicações para o problema proposto; a Seção 3 explica como foi realizada a implementação do problema em Python; a Seção 4 apresenta os resultados e respostas para a Questão 1 do enunciado; a Seção 5, por sua vez, apresenta os resultados da análise estrutural de uma peça 2D por meio do ANSYS; e, por fim, a Seção 6 conclui o relatório retomando seus resultados e explicando as principais conclusões obtidas através do Exercício-Programa.

## 2 Fundamentos Teóricos

O problema proposto no Exercício é a análise estrutural de uma ponte. A Figura 1 mostra o esquema deste problema. Conforme indica o enunciado, a passarela (em azul) e a torre (em vermelho) são descritas como sendo elementos de viga. Estes elementos, no entanto, possuem tensão axial além de transversais, devendo então, serem considerados elementos de pórtico. Os cabos que ligam a passarela à torre (em verde) são descritos como sendo elementos de treliça, que só têm tensão axial.

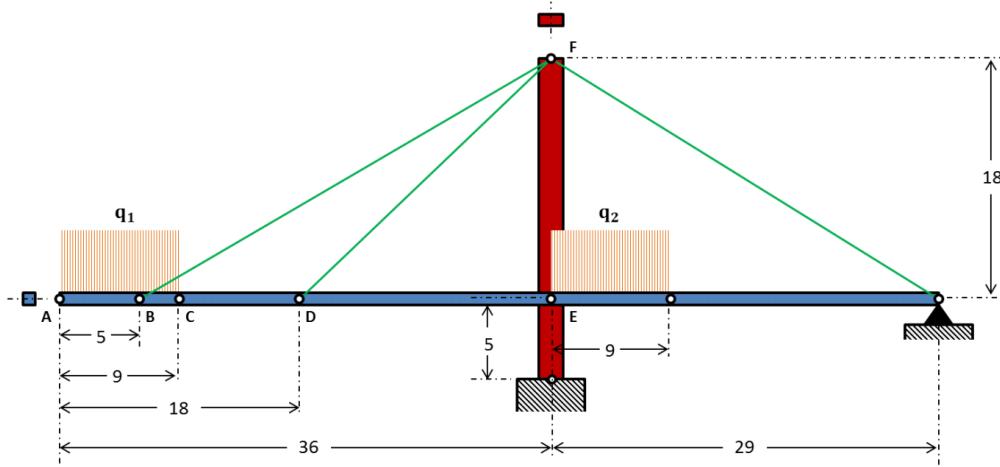


Figura 1: Esquema da ponte a ser analisada (elaborado pelos professores)

A equação dinâmica matricial no Método dos Elementos Finitos é a seguinte:

$$[M]\{\ddot{U}\} + [C]\{\dot{U}\} + [K]\{U\} = \{F\} \quad (1)$$

Onde:

$[M]$  = Matriz de massa;

$[C]$  = Matriz de amortecimento;

$[K]$  = Matriz de rigidez;

$\{F\}$  = Matriz-coluna dos carregamentos; e

$\{U\}$  = Matriz-coluna dos deslocamentos.

### 2.1 Matrizes de massa e rigidez

No MEF, os elementos de treliça possuem a seguinte matriz de rigidez:

$$[K_e]^{\text{treliça}} = \frac{EA}{L} \begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{array}{l} u_1 \\ w_1 \\ \phi_1 \\ u_2 \\ w_2 \\ \phi_2 \end{array}$$

Por outro lado, os elementos de viga possuem a seguinte matriz de rigidez:

$$[K_e]^{\text{viga}} = \frac{EI}{L^3} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 12 & 6L & 0 & -12 & 6L \\ 0 & 6L & 4L^2 & 0 & -6L & 2L^2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -12 & -6L & 12 & 0 & -6L \\ 0 & 6L & 2L^2 & -6L & 0 & 4L^2 \end{bmatrix} \begin{array}{l} u_1 \\ w_1 \\ \phi_1 \\ u_2 \\ w_2 \\ \phi_2 \end{array}$$

Os elementos de pórtico, por sua vez, são elementos de viga que também possuem deslocamento axial, sendo assim, sua matriz de rigidez deve ser o resultado da combinação das matrizes de rigidez de vigas e treliças:

$$[K_e]^{\text{pórtico}} = [K_e]^{\text{treliça}} + [K_e]^{\text{viga}} = \frac{EA}{L} \begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \frac{EI}{L^3} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 12 & 6L & 0 & -12 & 6L \\ 0 & 6L & 4L^2 & 0 & -6L & 2L^2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -12 & -6L & 12 & 0 & -6L \\ 0 & 6L & 2L^2 & -6L & 0 & 4L^2 \end{bmatrix}$$

Por outro lado, a matriz de massa de elementos de treliça pode ser definida da seguinte forma:

$$[M_e]^{\text{treliça}} = \frac{\rho AL}{6} \begin{bmatrix} 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{array}{l} \ddot{u}_1 \\ \ddot{w}_1 \\ \ddot{\phi}_1 \\ \ddot{u}_2 \\ \ddot{w}_2 \\ \ddot{\phi}_2 \end{array}$$

E também, para o elemento de viga:

$$[M_e]^{\text{viga}} = \frac{\rho AL}{420} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 156 & 22L & 0 & 54 & -13L \\ 0 & 22L & 4L^2 & 0 & 13L & -3L^2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 54 & 13L & 0 & 156 & -22L \\ 0 & -13L & -3L^2 & 0 & -22L & 4L^2 \end{bmatrix} \begin{bmatrix} \ddot{u}_1 \\ \ddot{w}_1 \\ \ddot{\phi}_1 \\ \ddot{u}_2 \\ \ddot{w}_2 \\ \ddot{\phi}_2 \end{bmatrix}$$

Analogamente, o elemento de pórtico é definido pela combinação dos dois elementos:

$$[M_e]^{\text{pórtico}} = [M_e]^{\text{treliça}} + [M_e]^{\text{viga}} = \frac{\rho AL}{6} \begin{bmatrix} 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \frac{\rho AL}{420} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 156 & 22L & 0 & 54 & -13L \\ 0 & 22L & 4L^2 & 0 & 13L & -3L^2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 54 & 13L & 0 & 156 & -22L \\ 0 & -13L & -3L^2 & 0 & -22L & 4L^2 \end{bmatrix}$$

Ademais, alguns elementos do problema proposto (treliças de sustentação da passarela e a torre) estão inclinados em relação a horizontal um ângulo  $\theta$ . Faz-se necessário, então, a realização de uma transformação nas matrizes a fim de corrigir este efeito e colocar as variáveis de deslocamento nodal em um sistema global (não solidário ao elemento):

$$[T](\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & 0 & 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ w_1 \\ \phi_1 \\ u_2 \\ w_2 \\ \phi_2 \end{bmatrix}$$

Matrizes quadradas e matrizes-coluna são rotacionadas de maneiras diferentes:

$$[\overline{M}_e] = [T]^t \cdot [M_e] \cdot [T]$$

$$[\overline{F}_e] = [T]^t \cdot [F_e]$$

## 2.2 Análise Modal

Tomando  $[C]$  nulo (ausência de amortecimento) e as cargas nos graus de liberdade não-nulos como sendo nulas, reduz-se a equação dinâmica para a seguinte:

$$(-\omega^2[M] + [K]) \{U\} = \{0\} \quad (2)$$

Esta equação é resolvida através da imposição da condição de determinante nulo. Como tem-se que os deslocamentos nodais são não-nulos (por suposição):

$$\det (-\omega^2[M] + [K]) = 0$$

Multiplicando ambos os termos pela inversa da matriz de massa, obtém-se:

$$\det ([M]^{-1}[K] - \omega^2[I]) = 0 \quad (3)$$

Onde  $[I]$  é a matriz identidade. Observa-se que esta é a equação característica para a obtenção de auto-valores e auto-vetores de uma matriz. Desta forma, conclui-se que os valores de  $\omega^2$  são os auto-valores de  $[M]^{-1}[K]$  e os modos de vibrar da estrutura, seus auto-vetores.

Para tal, basta encontrar os auto-valores e auto-vetores numericamente da matriz  $[M]^{-1}[K]$ . Contudo, com todos os graus de liberdade presentes, deve-se descartar os modos de vibrar de corpo rígido. Uma forma de contornar este problema é restringir os graus e reduzir a matriz antes do cálculo dos auto-valores e auto-vetores, gerando apenas os modos de vibrar que não são de corpo rígido.

## 2.3 Análise Harmônica

Para a análise harmônica, se visa encontrar a resposta física da ponte, para forças atuando em uma dada frequência  $\omega$ . Para isto, se analisa a estrutura sem amortecimento. Como as forças atuam em uma frequência específica, pode-se decompor o vetor de forças da seguinte forma:

$$\{F(t)\} = \{F_0\}e^{i\omega t} \quad (4)$$

Com isto, a estrutura estudada também oscila com a mesma frequência  $\omega$ . Assim, pode-se decompor os vetores  $\{U(t)\}$  e  $\{\ddot{U}(t)\}$ , sendo o segundo meramente a segunda derivada do primeiro

$$\begin{aligned} \{U(t)\} &= \{U_0\}e^{i\omega t} \\ \{\ddot{U}(t)\} &= (-\omega^2)\{U_0\}e^{i\omega t} \end{aligned}$$

Então, o sistema recai em um caso de análise estática:

$$[-\omega^2[M] + [K]]\{U_0\} = \{F_0\} \quad (5)$$

Com isto, pode-se obter as respostas dos deslocamentos para uma força oscilatória. Para a análise neste Exercício Programa se escolheu analisar pontos específicos (**A**, **B**, **C** e **F**) e o módulo de seus deslocamentos.

Vale ressaltar que para casos de  $\omega$  muito próximo à frequência natural de ressonância da estrutura, o resultado de  $[-\omega^2[M] + [K]]$  é uma matriz singular, que gera deslocamentos infinitos. A solução da análise harmônica é apenas demonstrar estes picos, já que os valores obtidos por ela dependem da discretização adotada, uma vez que o valor matemático correto seria de infinito.

## 2.4 Análise Transiente

Para a análise transiente do sistema, é importante que a matriz de amortecimento exista e não seja tomada como nula, uma vez que isso acarreta na não dissipação da solução particular do sistema. Além disso, para realizar a análise, utiliza-se o método de Newmark  $\beta$ .

$$[M]\{a_{n+1}\} + [C]\{v_{n+1}\} + [K]\{d_{n+1}\} = \{f_{n+1}\} \quad (6)$$

No qual, tem-se:

$$\begin{aligned} \{d_{n+1}\} &= \{d_n\} + \Delta t \{v_n\} + \frac{(\Delta t)^2}{2} \{(1 - 2\beta)\{a_n\} + 2\beta\{a_{n+1}\}\} \\ \{v_{n+1}\} &= \{v_n\} + \Delta t \{(1 - \gamma)\{a_n\} + \gamma\{a_{n+1}\}\} \end{aligned}$$

Portanto, basta calcular  $\{a_{n+1}\}$  que se obtêm as demais grandezas de interesse. Substituindo as definições de  $\{d_{n+1}\}$  e  $\{v_{n+1}\}$  em 6, pode-se obter uma fórmula para  $\{a_{n+1}\}$ , que depende dos parâmetros  $\beta$  e  $\gamma$ .

Além disso, pode-se calcular o valor de  $\{a_0\}$  com as devidas condições iniciais, a depender do problema. No caso o *script* em Python permite que se possa escolher entre condições iniciais nulas ou para condições iniciais que são a solução do problema estático em  $t = 0$ .

### 2.4.1 Método da aceleração média constante

Conforme descrito anteriormente, o método de Newmark *beta* depende dos parâmetros  $\beta$  e  $\gamma$  para o cálculo do valor de  $\{a_{n+1}\}$  (que acarreta na estabilidade ou não do método). O método da aceleração média constante foi adotado neste exercício programa, tanto para a parte em Python quanto no uso do *software* Ansys. Ele possui os seguintes parâmetros:

$$\rightarrow \beta = \frac{1}{4}$$

$$\rightarrow \gamma = \frac{1}{2}$$

## 3 Implementação Computacional

A implementação do MEF foi feita em Python, por meio de classes que abstraem seus elementos-base, os elementos e o sistema. Os elementos são determinados por um tipo (treliça, viga ou pórtico) com as matrizes de massa e rigidez automaticamente calculadas, conforme a Seção 2.1, e

além disso são compostos por dois nós (pontos) do sistema. A classe sistema abstrai um conjunto de elementos e nós, com as matrizes de massa rigidez globais, as condições de contorno do sistema implementado (engaste, apoio fixo e articulação), as forças concentradas externas e as cargas distribuídas externas.

Com isto, o *script* permite a fácil implementação de qualquer sistema de MEF 2D que seja composto pelos elementos do tipo treliça, viga e pórtico. Além disso, com métodos de análise transiente, harmônica e modal, o sistema permite a visualização por meio da plotagem dos elementos e nós, com opções de um vetor de deformações e uma escala multiplicativa, numeração dos nós e até mesmo numeração dos elementos.

## 4 Exercício 1

### 4.1 Item a)

#### 4.1.1 Sub-item i)

Desenvolva um programa específico de MEF para obter as primeiras seis frequências naturais da estrutura e seus respectivos modos de vibrar. Liste as frequências em Hertz, usando os seguintes comprimentos para os elementos  $\Delta x = 1; 2;$  e  $3\text{ m}$ . Plete os modos de vibrar só para o menor  $\Delta x$ . Se em alguma parte da estrutura não encaixa um elemento, use um elemento de menor tamanho.

Após a implementação do programa em Python para determinar estas frequências, obteve-se os resultados apresentados nas Tabelas 1 a 3

Tabela 1: Frequências naturais da estrutura a partir do Python  $\Delta x = 1\text{ m}$

Harmônico	$f$ (Hz)
1	1.1022
2	3.7231
3	3.9671
4	5.0214
5	10.247
6	12.7425

Tabela 2: Frequências naturais da estrutura a partir do Python  $\Delta x = 2\text{ m}$

Harmônico	$f$ (Hz)
1	1.1022
2	3.7231
3	3.9671
4	5.0214
5	10.2472
6	12.7429

Tabela 3: Frequências naturais da estrutura a partir do Python  $\Delta x = 3\text{ m}$

Harmônico	$f$ (Hz)
1	1.1022
2	3.7231
3	3.9671
4	5.0214
5	10.2482
6	12.7449

Percebe-se que a influência da discretização é pequena para se encontrar os valores de ressonância do sistema. Isso é relevante pois uma discretização maior (ou seja, um  $\Delta x$  mais elevado) demanda menor poder computacional e gera resultados satisfatórios. Isso ocorre pois os valores de frequência devem ser os autovalores da matriz do sistema, que é bem representada pela discretização com  $\Delta x = 3$ .

Com a melhor discretização possível ( $\Delta x = 1$ ), obtiveram-se os seis primeiros modos de vibrar da estrutura analisada. Os modos de vibrar são mostrados na Figura 2

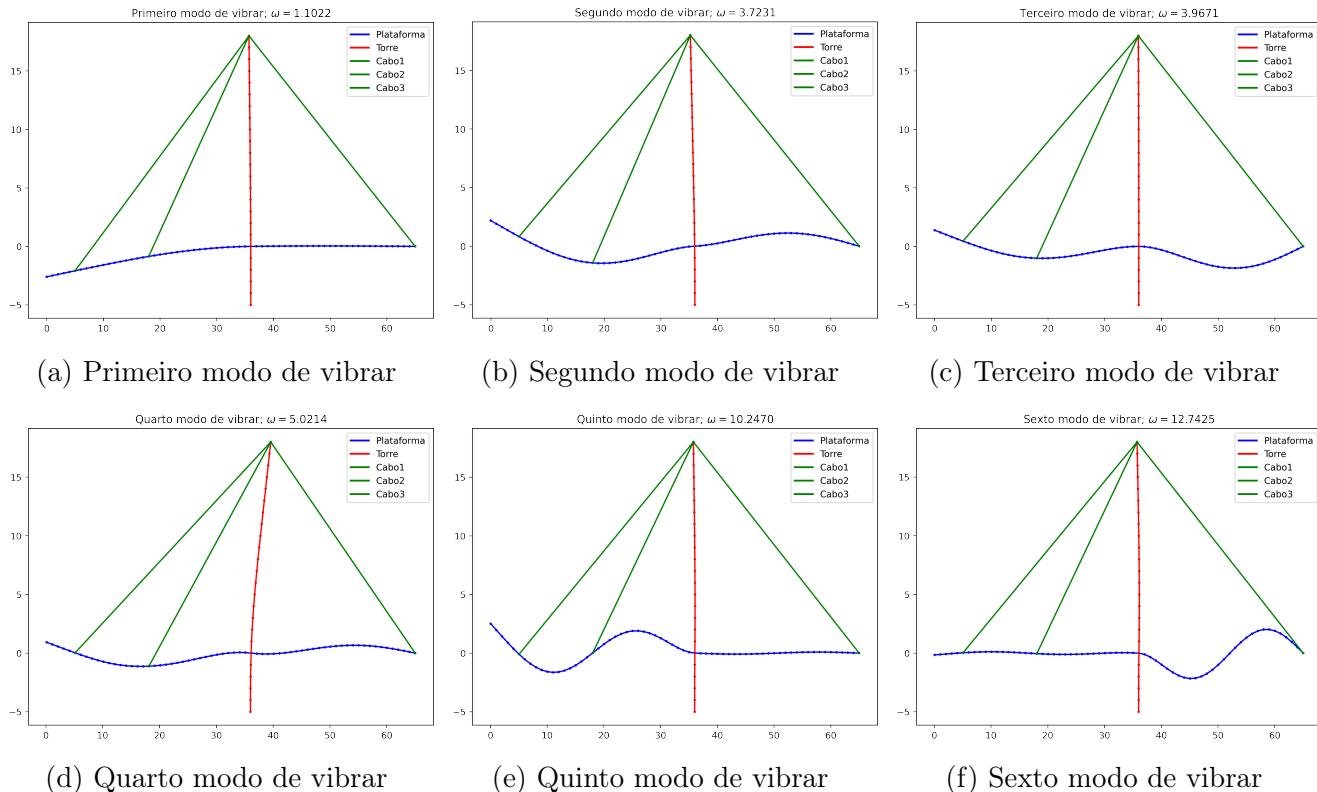


Figura 2: Modos de vibrar encontrados no Python

A partir dos resultados gerados, percebe-se que os modos de vibrar são completamente dependentes do valor de frequência passado, oscilando de formas distintas. Conforme esperado, o primeiro harmônico apresenta apenas uma crista, o segundo 2 cristas e assim em diante. A escala utilizada na plotagem das deformações foi de 8 vezes, para a melhor visualização das deformações na estrutura.

#### 4.1.2 Sub-item ii)

Mediante uma análise transiente, simule o tráfego de pessoas do Grupo 2 para o Grupo 1, se deslocando pela passarela do ponto E ao ponto C. Faça a análise para as velocidades de avanço ( $v_a$ ): 0,5; 1; e 2 m/s, considerando que no tempo zero haja 20 pessoas em cada grupo ( $N_1 = N_2 = 20$ ). Suponha que as pessoas avançam de 1 em 1, separadas entre si por 1 metro. Para facilitar a modelagem, use a discretização  $\Delta x = 1$  m e assim considere que as pessoas

pisam apenas nos nós. A carga vertical ( $V_i$ ) que atua em um ponto da passarela (entre C e E) pode ser modelada como uma variação senoidal entre 0 e 80 Kgf (enquanto as pessoas passam). Os parâmetros  $q_1(N_1)$ ,  $q_2(N_2)$  e  $V_i$  variam no tempo segundo as expressões:

$$N_1 = \begin{cases} 20 & ; \quad t \leq 27/v_a \\ v_a t - 7 & ; \quad 27/v_a < t \leq 47/v_a \\ 40 & ; \quad t > 47/v_a \end{cases}$$

$$N_2 = \begin{cases} 20 - v_a t & ; \quad t \leq 20/v_a \\ 0 & ; \quad t > 20/v_a \end{cases}$$

$$V_i = \begin{cases} 0 & ; \quad t \leq [x(E) - x(i)]/v_a \\ 80 * 9,8 * [1 - \cos(2\pi v_a t)]/2 & ; \quad [x(E) - x(i)]/v_a < t \leq [x(E) + 20 - x(i)]/v_a \\ 0 & ; \quad t > [x(E) + 20 - x(i)]/v_a \end{cases}$$

Com a modelagem com discretização  $\Delta x = 1$  pode-se obter forças nos nós, simulando o caminhar do Grupo 2 e a partir do *script* em Python foi possível gerar e um carregamento distribuído nos elementos que correspondem às posições fixas dos Grupos.

Mediante as equações de análise transiente descritas na Seção 2.4, basta definir-se de forma adequada as funções de carga e cortante. Desta forma, realizar a análise transiente do problema de engenharia proposto torna-se perfeitamente possível.

Para a análise, observam-se os deslocamentos nodais dos pontos **A**, **B**, **C** e **F**. Além disso, como parâmetro do método de solução transiente do *script* realizaram-se as simulações com condições iniciais nula e não nula. Os resultados obtidos podem ser visualizados nas Figuras 3 e 4

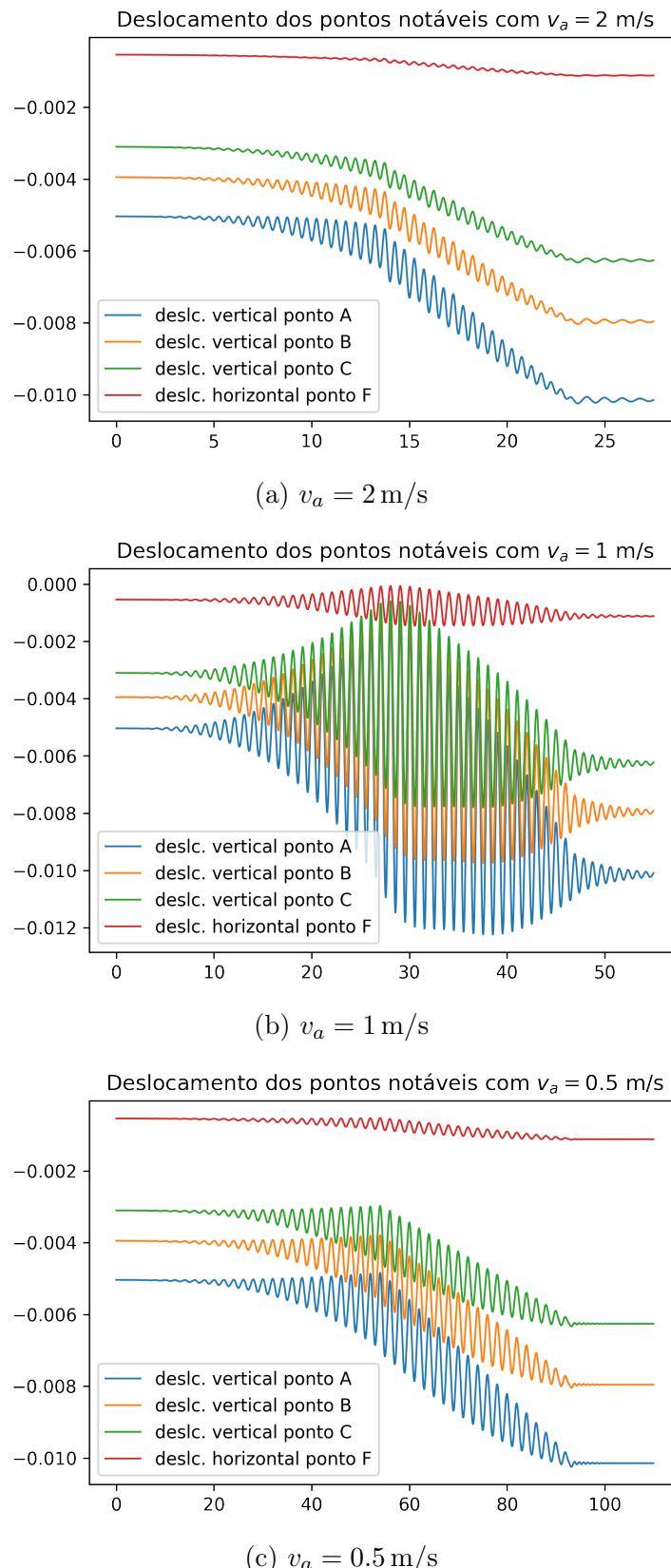
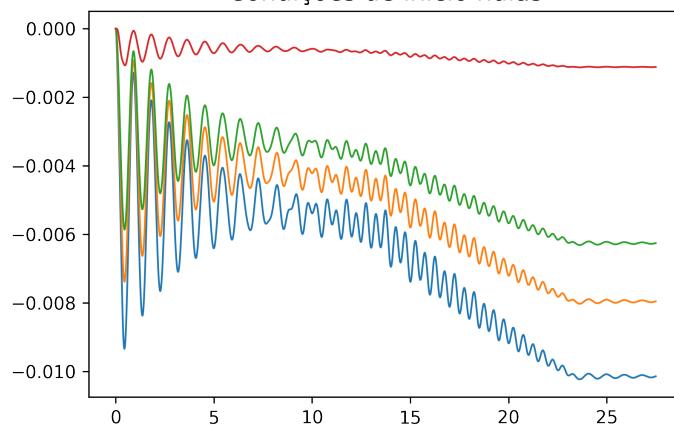
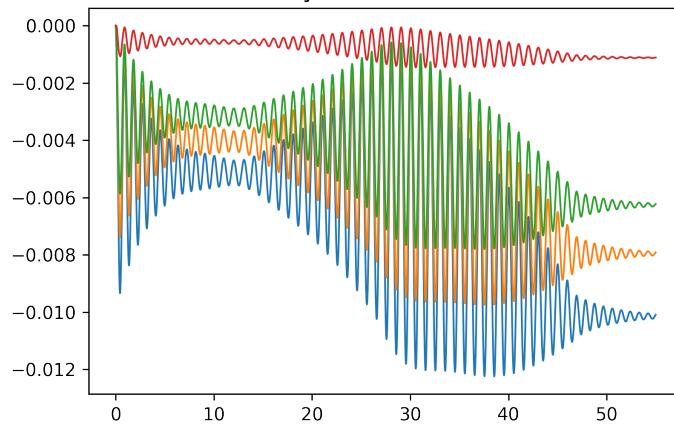


Figura 3: Análise transiente com condição inicial não nula

**Deslocamento dos pontos notáveis com  $v_a = 2 \text{ m/s}$**   
Condições de início nulas

(a)  $v_a = 2 \text{ m/s}$ 

**Deslocamento dos pontos notáveis com  $v_a = 1 \text{ m/s}$**   
Condições de início nulas

(b)  $v_a = 1 \text{ m/s}$ 

**Deslocamento dos pontos notáveis com  $v_a = 0.5 \text{ m/s}$**   
Condições de início nulas

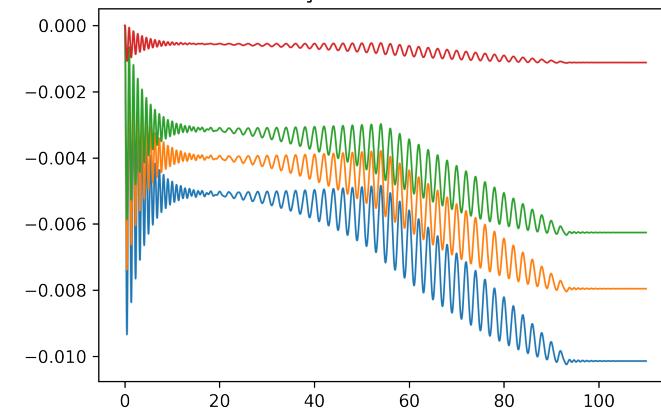
(c)  $v_a = 0.5 \text{ m/s}$ 

Figura 4: Análise transiente com condição inicial nula

Conforme pode-se observar nos gráficos dos pontos notáveis **A**, **B**, **C** e **F**, independente das condições iniciais, tem-se uma maior amplitude das oscilações na estrutura com  $v_a = 1 \text{ m/s}$ . De fato, este valor está mais próximo do primeiro harmônico encontrado na Seção 4.1.1. Então, a estrutura, quando forçada com esta frequência de caminhar das pessoas, possui maiores oscilações, tanto vertical quanto horizontal. A velocidade de avanço  $v_a = 2 \text{ m/s}$  oscila menos do que as demais analisadas pois está mais distante dos valores de frequência natural encontrada. Além disso, entre dois harmônicos têm-se uma mudança de fase, que é possivelmente próxima a esse valor de velocidade de avanço, mitigando seu efeito na deformação da estrutura. O valor de  $v_a = 0.5 \text{ m/s}$  apresenta a resposta oscilatória intermediária, devido aos fatos apresentados e por estar relativamente próxima a uma das frequências de ressonância da estrutura.

Já a análise da discretização do tempo foi realizada estudando apenas o ponto **A**, já que este apresentou os maiores deslocamentos. Para tal, usou-se 3 valores de discretização temporal na velocidade de avanço  $v_a = 1 \text{ m/s}$ . O resultado pode ser visto na Figura 5

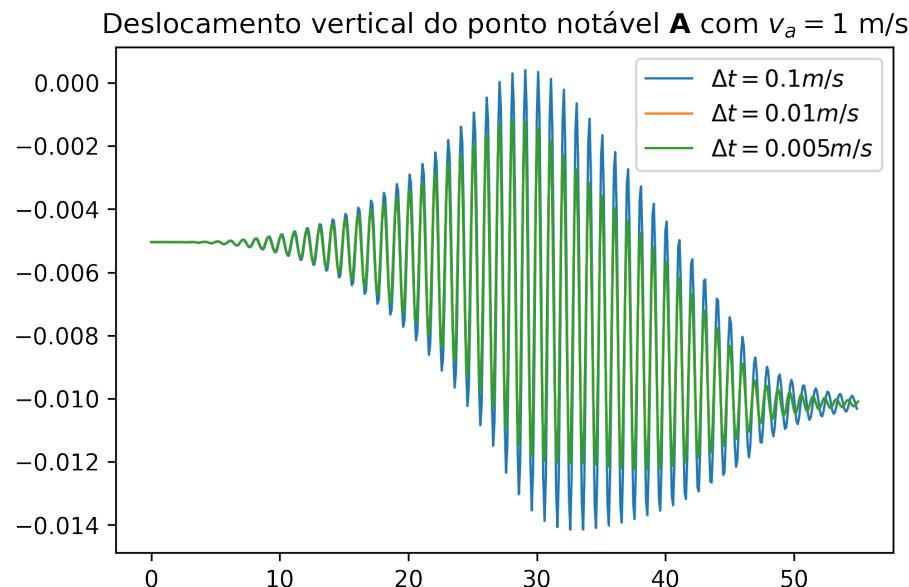


Figura 5: Análise da discretização temporal com  $v_a = 1 \text{ m/s}$

Com a inspeção da resposta obtida, percebe-se que a discretização de  $\Delta t = 0.01 \text{ m/s}$  apresenta as mesmas amplitudes do que  $\Delta t = 0.005 \text{ m/s}$ , indicando que houve uma saturação na precisão obtida com o refinamento do tempo. Além disso, percebe-se que com  $\Delta t = 0.1 \text{ m/s}$  obteve-se maiores amplitudes, já que os valores de força cortante oscilavam com uma frequência mais próxima à uma das frequências naturais.

#### 4.1.3 Sub-item iii)

Obtenha o diagrama de resposta em frequência da norma do deslocamento ( $\|U\|$  vs  $f$ ) para os mesmos pontos solicitados no item anterior (A, B, C e F). Plote as curvas no mesmo gráfico e verifique que sejam observados os picos de ressonância correspondentes às primeiras três frequências obtidas em (i.) (sem amortecimento). Considere, neste caso, apenas carregamen-

tos nodais entre os pontos C e E da estrutura, do tipo  $V_i = 80 \cdot 9.8 \cdot e^{j\omega t}$ . Use uma discretização adequada para a frequência ( $\Delta f$ ). Se necessário, refine o passo  $\Delta f$  próximo aos valores de ressonância.

Através do uso do *script* em Python, ao se gerar as frequências naturais do sistema, pode-se gerar um vetor de frequências que é espaçado de 0.01 Hz entre 0 Hz e 10 Hz que, ao se aproximar dos valores de ressonância encontrados, apresenta um refinamento (por um fator de 10 vezes). Isso acarretou eventualmente em um *warning* que a matriz gerada era singular, pela proximidade do valor com a real frequência do sistema, caso já descrito na determinação de  $\omega$  ao final da Seção 2.3.

A resposta, nos mesmos pontos notáveis A, B, C e F pode ser vista na Figura 6

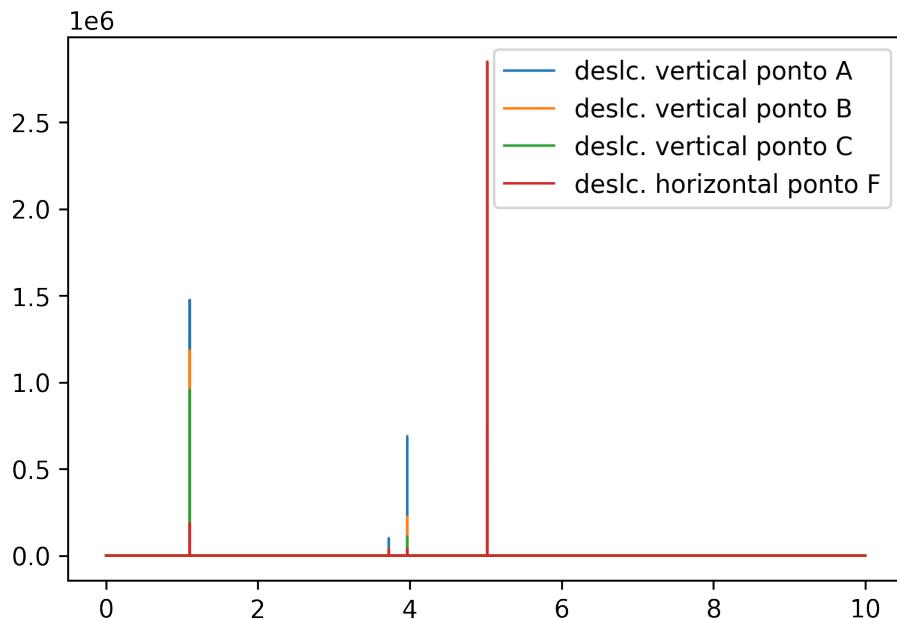


Figura 6: Diagrama de resposta em frequência obtido no Python

Vale ressaltar que os valores obtidos não correspondem aos valores de deslocamentos reais, que, idealmente, teriam valor infinito. Com tais frequências é evidente que a estrutura iria a colapso. Pode-se notar que os valores de frequência encontrados correspondem aos da análise modal na Seção 4.1.1. Além disso, comparando com os modos de vibrar, percebe-se que o quarto valor de ressonância encontrado é o que apresenta maior deslocamento horizontal do ponto F, enquanto que o primeiro é o que apresenta o maior deslocamento vertical do ponto A.

## 4.2 Item b)

### 4.2.1 Sub-item i)

Resolva os itens (a.i), (a.ii) e (a.iii). Para a representação dos modos de vibrar, evite os gráficos de fundo preto, e, para a análise harmônica, resolva sem amortecimento.

Os valores de frequência natural obtidos no Ansys para as diferentes discretizações são vistas nas Tabelas 4 a 6. Além disso, já se comparou com os valores obtidos no Python.

Tabela 4: Frequências naturais da estrutura a partir do Python e do ANSYS para  $\Delta x = 1 \text{ m}$

Harmônico	$f$ (Hz)	
	ANSYS	Python
1	1.0959	1.1022
2	3.6995	3.7231
3	3.9592	3.9671
4	4.9690	5.0214
5	10.210	10.247
6	12.695	12.7425

Tabela 5: Frequências naturais da estrutura a partir do Python e do ANSYS para  $\Delta x = 2 \text{ m}$

Harmônico	$f$ (Hz)	
	ANSYS	Pyhton
1	1.0964	1.1022
2	3.7159	3.7231
3	3.9815	3.9671
4	4.9769	5.0214
5	10.347	10.2472
6	12.936	12.7429

Tabela 6: Frequências naturais da estrutura a partir do Python e do ANSYS para  $\Delta x = 3 \text{ m}$

Harmônico	$f$ (Hz)	
	ANSYS	Pyhton
1	1.0974	1.1022
2	3.7434	3.7231
3	4.0196	3.9671
4	4.9900	5.0214
5	10.565	10.2482
6	13.362	12.7449

Para a análise transitória, o resultado obtido pode ser visto na Figura 7. Nela a simulação com a velocidade de avanço  $v_a = 0.5 \text{ m/s}$  foi realizada com condições inciais não nulas e as demais com condições inciais nulas.

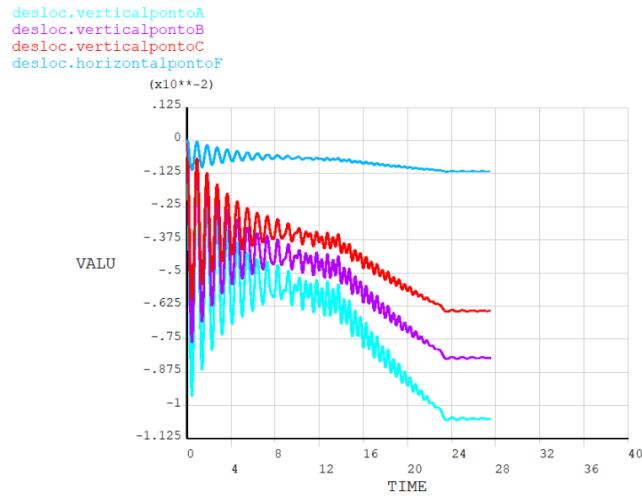
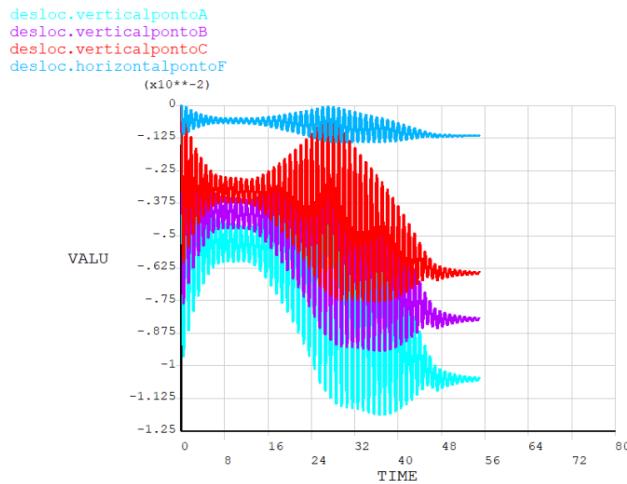
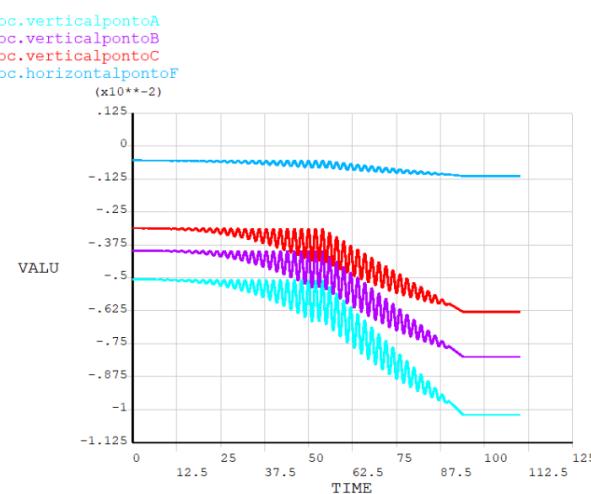
(a)  $v_a = 2 \text{ m/s}$ (b)  $v_a = 1 \text{ m/s}$ (c)  $v_a = 0.5 \text{ m/s}$ 

Figura 7: Resultado da análise transiente no Ansys

Finalmente, a análise harmônica realizada no *software* Ansys pode ser vista na Figura 8

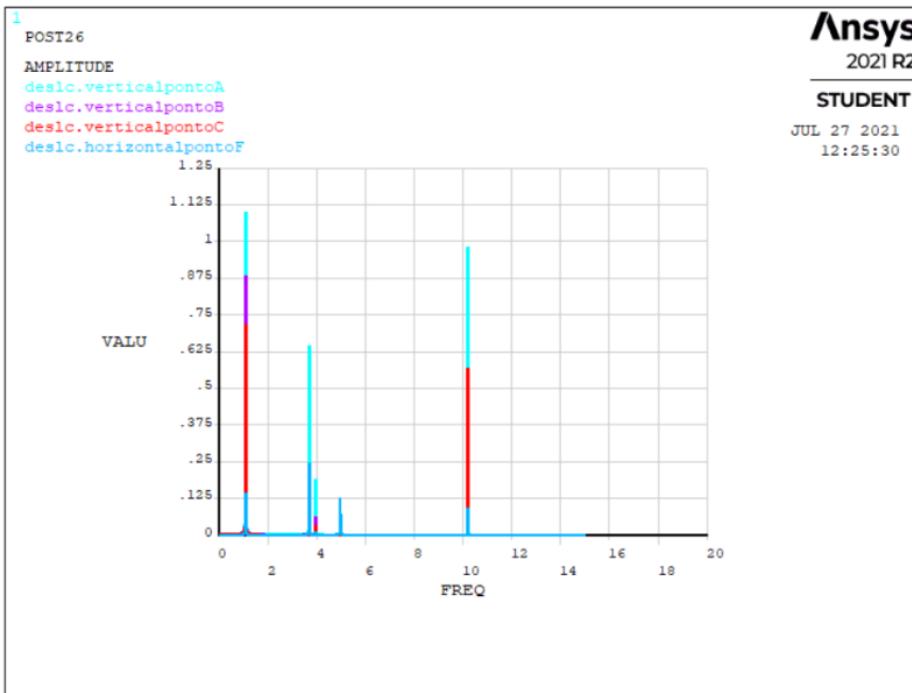


Figura 8: Diagrama de resposta em frequência obtido no Ansys

#### 4.2.2 Sub-item ii)

Compare os resultados do Ansys (ou similar) com os do seu programa, plotando no mesmo gráfico: o deslocamento vertical dos pontos A, B e C, e o deslocamento horizontal do ponto F em função do tempo para cada velocidade de avanço, usando  $\Delta t = 0.01$  s (a.ii); e o diagrama de resposta em frequência (a.iii).

Com relação aos valores de frequência natural obtidos, percebe-se que independe da discretização tanto o Python quanto o Ansys obtiveram resultados semelhantes, como pode ser visto nas Tabelas 4 a 6. As frequências no geral não oscilam muito com relação à discretização da malha e foram bem semelhantes nas duas respostas comparadas. É de se esperar que com frequências maiores se obtenha uma diferença maior. Além disso como estas frequências dependem das matrizes de massa e rigidez do sistema, das quais não se tem informações no Ansys, é provável que a diferença advenha desta formulação.

Quanto às respostas transientes, pode-se afirmar com segurança que as respostas obtidas foram semelhantes. Isso se dá pois os parâmetros utilizados para o método de Newmark  $\beta$  terem sido os mesmos. Percebe-se que novamente a velocidade  $v_a = 1$  m/s apresenta maiores amplitudes enquanto que  $v_a = 2$  m/s as segundas maiores amplitudes. Para comparar a questão da condição de início, plotou-se a velocidade  $v_a = 0.5$  m/s com a condição de início não nula e se obteve os mesmos resultados do *script* de Python.

Para a comparação das análises harmônicas, comparou-se os diagramas em frequência obtidos nos dois programas. Percebe-se que os valores obtidos no Python são bem maiores, o

que é, por sua vez, fruto da discretização mais refinada nas frequências próximas aos valores de ressonância obtidos. Como o Ansys apenas seleciona os valores máximo e mínimo e a discretização, não existe refinamento próximo aos valores de ressonância e então os valores apresentados são bem menores. Apesar disso, pode-se averiguar que os valores de frequência de ressonância correspondem aos picos de deslocamento. Pode-se, ainda, afirmar novamente que os resultados apresentados pelo *script* feito pelos alunos é coerente com os obtidos via *software* comercial.

## 5 Exercício 2

O exercício 2 proposto no Exercício-Programa foi a análise estrutural de uma peça bi-simétrica, mostrada na Figura 9. Devido a sua bi-simetria, apenas um quarto da peça foi construído. Esta prática torna a análise mais eficiente uma vez que diminui a carga de processamento que existe no processamento do MEF. Para substituir o vínculo físico da peça com suas outras partes (lateral e inferior), restringiram-se os lados solidários aos eixos de simetria.

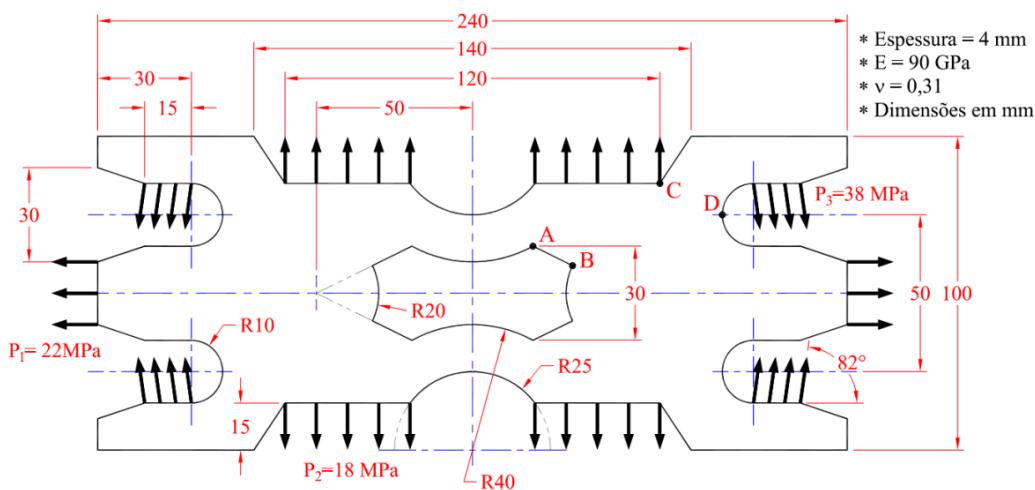


Figura 9: Peça bi-simétrica a ser analisada (desenho elaborado pelos professores)

### 5.1 Item a)

Plote a estrutura deformada e identifique o máximo valor de deslocamento e onde ocorre.

Após a construção da peça solicitada e a aplicação dos esforços requeridos, plotou-se a estrutura deformada, que pode ser observada na Figura 10. Percebe-se pela figura que o valor máximo de deslocamento observado é de 0.387 mm e ocorre na parte superior direita da placa. Este deslocamento está, provavelmente, ligado as cargas causadas por  $P_3$ . Observa-se ainda que a presença do corte semi-circular abaixo deste destacamento faz com que a peça tenha para onde deformar sem sofrer reações estruturais, o que facilita uma maior deformação.

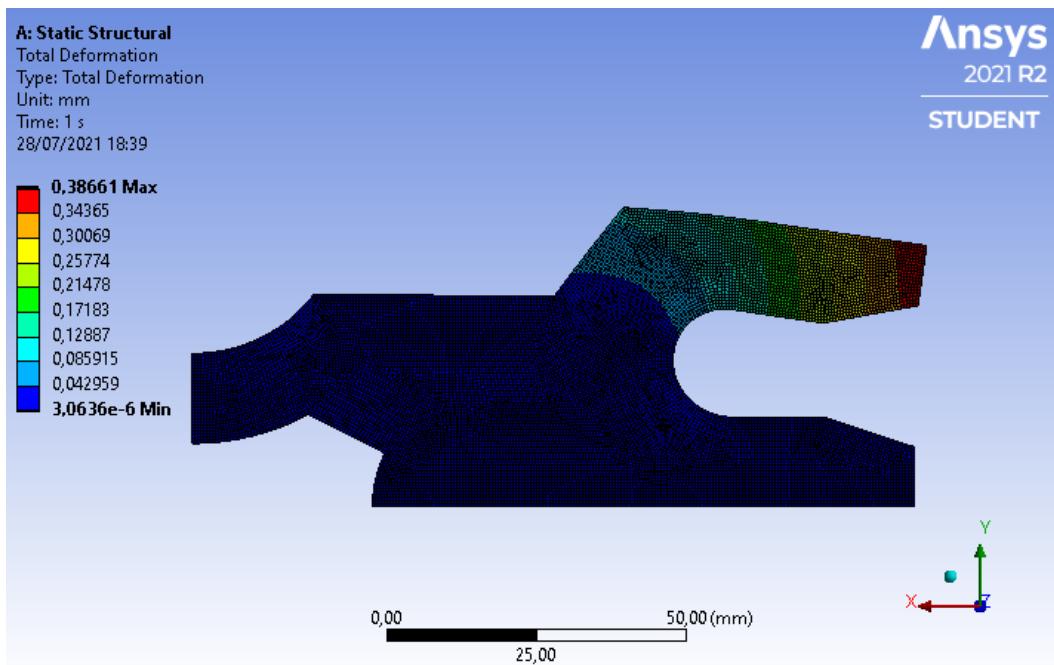


Figura 10: Deslocamentos para a estrutura deformada

## 5.2 Item b)

Plote as tensões mecânicas de von Mises na estrutura e obtenha os valores de tensão nos pontos A, B, C, e D. Verifique a influência da discretização da malha nos resultados, escolha uma discretização inicial  $\Delta x_1$  e logo refine-a da forma  $\Delta x_2 = \Delta x_1/2$ .

Tomando, inicialmente,  $\Delta x = 1 \text{ mm}$ , obtiveram-se as tensões de von Mises apresentadas na Figura 11.

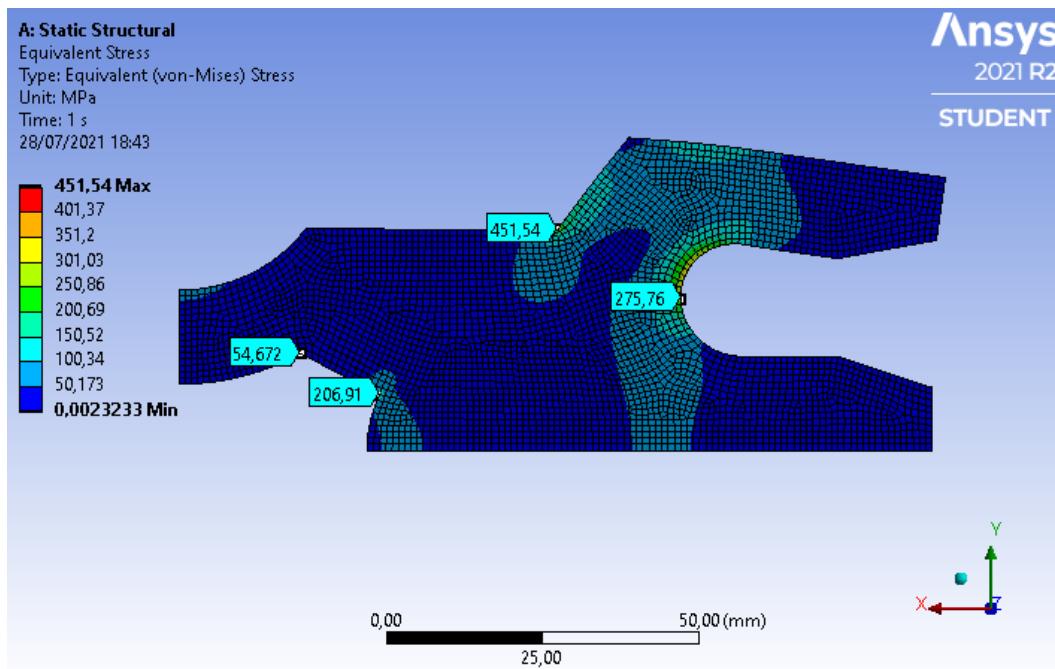


Figura 11: Tensões de von Mises nos pontos de interesse para  $\Delta x = 1 \text{ mm}$

Após o refinamento da malha, tem-se  $\Delta x = 0.5 \text{ mm}$ . Com esta discretização, obtiveram-se as tensões de von Mises apresentadas na Figura 12.

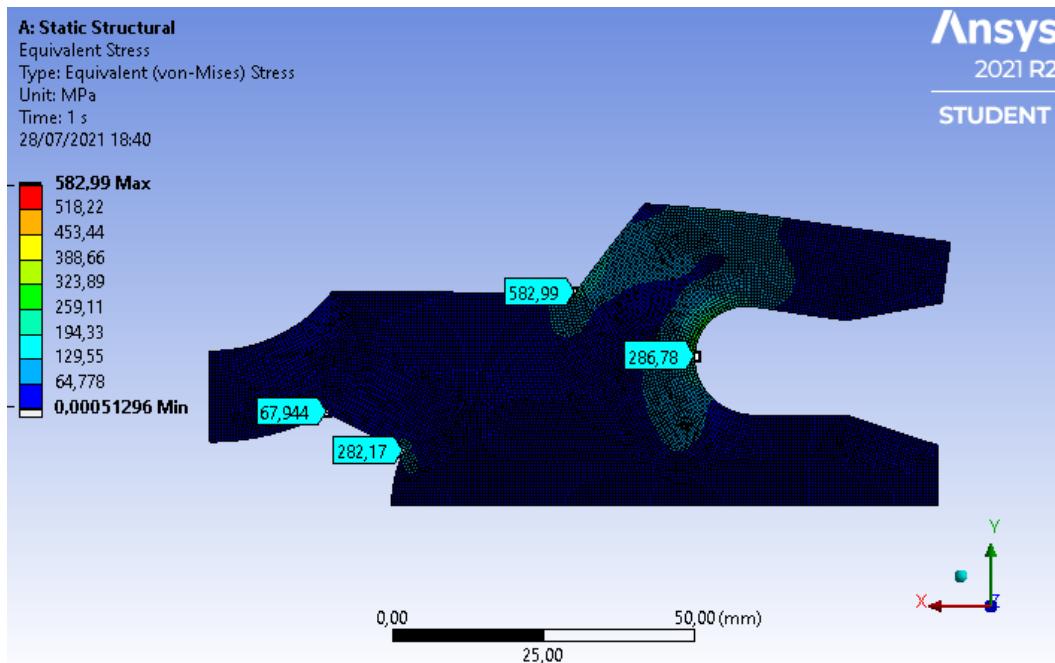


Figura 12: Tensões de von Mises nos pontos de interesse para  $\Delta x = 0.5 \text{ mm}$

Observa-se que o refinamento da malha aumenta os picos da tensão de von Mises. Este comportamento era, de fato, esperado, uma vez que a diminuição do tamanho dos elementos da malha causam uma concentração de tensão muito grande nos elementos individuais.

### 5.3 Item c)

Identifique o máximo valor de tensão de von Mises e onde ocorre, bem como os demais pontos onde ocorrem concentração de tensões na estrutura.

O valor máximo da tensão de von Mises identificado foi de 582.99 MPa e ocorre no ponto C. Observa-se que as tensões se concentram nas extremidades da peça, principalmente nas que apresentam cantos-vivos próximos às regiões de aplicação dos carregamentos. O ponto C, em particular, é um canto-vivo de ângulo bastante “fechado” e encontra-se próximo de duas regiões de aplicação de carregamentos:  $P_2$  e  $P_3$ .

Estas concentrações podem ser evitadas através da eliminação dos cantos-vivos ou de sua substituição por curvas de circunferência ou *splines*. Caso isto não seja possível, deve-se visar a construção dos ângulos o mais próximo de 180° possível.

## 6 Conclusões

Através do desenvolvimento teórico e da experimentação prática observada no decorrer deste Exercício-Programa, obtiveram-se resultados e aprendizados muito interessantes em relação ao Método de Elementos Finitos. A possibilidade de se modelar o mesmo problema através do Python, um *software* convencional de programação e também através do ANSYS, um *software* comercial de engenharia permitiram aos alunos o amplo entendimento da implementação do método, bem como a observação de suas limitações.

Além disso, as variações de parâmetros propostas neste Exercício permitiram que os alunos adquirissem boa noção dos efeitos da discretização da malha nos resultados finais em um projeto de grande complexidade, algo bastante distinto do que se analisava durante as aulas teóricas, onde os problemas eram de complexidade bastante simples.

Por fim, este exercício se mostrou extremamente interessante ao apresentar o MEF como uma ferramenta eficiente para a resolução de problemas de engenharia complexos em estática. Através dele pode-se aplicar e fixar diversos conceitos aprendidos em aula e de extrema importância para os algoritmos de mecânica computacional.

# Apêndices

## A Listing do script em Python

Listagem 1: Código fonte

---

```

import numpy as np
np.set_printoptions(precision=4)
np.set_printoptions(suppress=True)
# import sys
#np.set_printoptions(threshold=sys.maxsize)
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

### definição das funções auxiliares ###
def angle(u, v=np.array([1, 0])):
    """
    Função que calcula o ângulo entre dois vetores

    Parameters
    -----
    u: np.ndarray, dim=(2, )
    v: np.ndarray (default=np.array([1, 0])), dim=(2, )
        vetor horizontal para direita

    Returns
    -----
    angle: float
        valor, em radianos, do ângulo entre os vetores
    """
    u_norm = u/np.linalg.norm(u)
    v_norm = v/np.linalg.norm(v)
    dot = np.dot(u_norm, v_norm)
    angle = np.arccos(dot)

    return angle

def calculate_I(b, h=None):
    """
    Função que calcula o momento de inércia

    Parameters
    -----
    b: float
        largura da seção transversal (ou diâmetro)
    h: float (default=None)
        altura da seção transversal, se None considera que trata-se
        de uma seção transversal circular
    """
    if h is None:
        return np.pi*((b/2)**4)/2
    return (b**3)*h/12

def globalize_K(Ke, total_size, idx):
    """
    Função que transforma uma matriz Ke (local) em uma matriz K (global)

    Parameters

```

```

-----
Ke: np.array (dim=len(idx)xlen(idx))
    Matriz de rigidez local
total_size: int
    tamanho total do sistema (número de variáveis independentes)
idx: list (or array-like)
    índice das variáveis que Ke se refere a
'''
Keg = np.zeros((total_size, total_size))
Keg[np.ix_(idx, idx)] = Ke

return Keg

def globalize_F(Fe, total_size, idx):
    '''
    Função que transforma uma matriz Fe (local) em uma matriz Fe (global)

    Parameters
    -----
    Fe: np.array (dim=len(idx)x1)
        Matriz de forças distribuídas
    total_size: int
        tamanho total do sistema (número de variáveis independentes)
    idx: list (or array-like)
        índice das variáveis que Fe se refere a
    '''
    Feg = np.zeros(total_size)
    Feg[np.ix_(idx)] = Fe

    return Feg

def anula_cond_contorno(A, idx):
    '''
    Anula as linhas e colunas em que se sabe que o movimento/rotação
    é nulo, deixando 1 no lugar

    Parameters
    -----
    A: np.ndarray
        matriz global
    idx: list
        lista dos índices (inteiros, 0-indexados)
    '''
    A_cc = A.copy()
    for var in idx:
        A_cc[var, :] = 0
        A_cc[:, var] = 0
        A_cc[var, var] = 1
    return A_cc

def reduz_matriz(A, idx, eliminate=False):
    '''
    Reduz a matriz global com condições de contorno para
    uma matriz apenas com as linhas e colunas dos elementos
    não triviais (não nulos) do sistema

    Parameters
    -----
    A: np.ndarray
        matriz global
    idx: list

```

```

        lista dos índices (inteiros, 0-indexados)
    """
A_red      = A.copy()
dimension = A.ndim
if eliminate == True:
    all_indexes = np.arange(0, A.shape[0])
    difference = set(all_indexes) - set(idx)
    idx = list(difference)
if dimension > 1:
    A_red = A[np.ix_(idx, idx)]
else:
    A_red = A[np.ix_(idx)]
return A_red

def discretiza(ponto1, ponto2, dx, dx_list=[1, 2, 3]):
    """
Função que discretiza um elemento, dado dois pontos a ele pertencentes. Além disso, respeita a imposição de que se não houver como gerar o elemento usa um dx menor, e apenas um, da lista de opções.
    """
#lista de opções
dx_list = np.array(dx_list)
dx_menores = dx_list[dx_list <= dx]

#defineição de onde é a discretização
if ponto1[1] == ponto2[1]:
    #tem coordenadas y iguais
    #a discretização é em x
    pto1 = ponto1[0]
    pto2 = ponto2[0]
    orientacao = 'x'
elif ponto1[0] == ponto2[0]:
    #tem coordenadas x iguais
    #a discretização é em y
    pto1 = ponto1[1]
    pto2 = ponto2[1]
    orientacao = 'y'
else:
    #tem coordenadas x e y distintas
    print("discretização não horizontal/vertical")
    return 0

#discretizando em dx até o valor que der
discretizacao = np.arange(pto1, pto2, dx)

#pegando o quanto falta para a discretização
diff = pto2 - discretizacao[-1]

if diff != 0:
    #se a discretização não foi perfeita
    if diff in dx_menores:
        #se tem um valor dx que encaixe no espaço que 'sobra'
        discretizacao = np.append(
            discretizacao, discretizacao[-1] + diff
        )
    else:
        # há um ponto que não encaixa e não temos um dx
        # adequado para ele
        print("Erro na discretização!")

```

```

# colocando o vetor com as duas coordenadas
if orientacao == 'x':
    discretizacao = np.array(
        [np.array([x, ponto1[1]]) for x in discretizacao]
    )
elif orientacao == 'y':
    discretizacao = np.array(
        [np.array([ponto1[0], y]) for y in discretizacao]
    )

return discretizacao

def discretiza_elemento(point_list, dx):
    elemento = []
    for i, ponto in enumerate(point_list[:-1]):
        discretizacao = discretiza(ponto, point_list[i+1], dx)
        if i+2 != len(point_list):
            # se não for o último elemento
            elemento.extend(discretizacao[:-1])
            # não se coloca o último ponto para evitar repetição
        else:
            # se for o último elemento
            elemento.extend(discretizacao)
    return elemento

def deleta_phis(deslocamentos, n=3, ):
    """
    Função que acha as deformações em x e em y a partir de um vetor
    resultante do modo de vibrar, ignorando o grau de liberdade phi
    dos pontos.
    """
    num_gdl = deslocamentos.size
    #deslocamentos[n-1:num_gdl:n] = 0
    desloc = np.delete(deslocamentos, np.arange(n-1, num_gdl, n))
    return desloc

def find_index_displacement(system, point, tipo):
    index = system._find_point_index(point)
    if tipo == 'horizontal':
        return 3*index
    elif tipo == 'vertical':
        return 3*index+1

# Funções de Hermite
H1 = lambda x, L: 1 - 3*x**2/L**2 + 2*x**3/L**3
dH1 = lambda x, L: -6*x/L**2 + 6*x**2/L**3
d2H1 = lambda x, L: -6/L**2 + 12*x/L**3
H2 = lambda x, L: x - 2*x**2/L + x**3/L**2
dH2 = lambda x, L: 1 - 4*x/L + 3*x**2/L**2
d2H2 = lambda x, L: -4/L + 6*x/L**2
H3 = lambda x, L: 3*x**2/L**2 - 2*x**3/L**3
dH3 = lambda x, L: 6*x/L**2 - 6*x**2/L**3
d2H3 = lambda x, L: 6/L**2 - 12*x/L**3
H4 = lambda x, L: -x**2/L + x**3/L**2
dH4 = lambda x, L: -2*x/L + 3*x**2/L**2
d2H4 = lambda x, L: -2/L + 6*x/L**2

### definição das classes utilizadas ###
class Elemento:
    ''

```

```

Classe para um Elemento, podendo ele ser de viga/pórtico ou de
treliça

    """
def __init__(self, ini, fim, tipo, params, numero=None):
    """
    Parameters
    -----
    ini: np.ndarray
        coordenada do início do elemento
    fim: np.ndarray
        coordenada do fim do elemento
    tipo: str {'viga', 'trelica'}
        tipo de elemento a ser estudado
    params: dict
        dicionário com os seguintes parâmetros (no SI):
        - A: área transversal do elemento
        - E: módulo de Young do elemento
        - rho: densidade do elemento
        - I: momento de inércia do elemento
        outros parâmetros relevantes (como tamanho L e o ângulo de
        inclinação theta são determinados através de contas com os
        parâmetros já dados)
    q: float
        valor da carga distribuída no elemento
    numero: int
        número do elemento na estrutura, caso se queira saber de
        um dado elemento
    """
    self.ini      = ini
    self.fim     = fim
    self.tipo    = tipo
    self.L       = self.calculate_L()
    self.theta   = self.calculate_theta()
    (self.E ,
     self.A ,
     self.rho,
     self.I ,) = params.values()
    #self.q      = q
    self.Ke     = self.build_Ke_matrix()
    self.Me     = self.build_Me_matrix()
    #inicializa-se como nulo
    self.Fe     = None
    #self.Fe     = self.build_q_array()
    self.Numero = numero

def calculate_L(self, ):
    """
    Método que calcula o tamanho L do elemento
    """
    return np.linalg.norm(self.ini - self.fim)

def calculate_theta(self, ):
    """
    Método que calcula a inclinação do elemento através das
    coordenadas de seu início e seu fim
    """
    u = self.fim - self.ini
    return angle(u)

def rotate_matrix(self,):
    """


```

Método que cria uma matriz de rotação para o elemento criado a partir do ângulo theta

Notes

-----

A matriz rotacionada é obtida a partir da seguinte expressão:

$$\dots \text{math}:: [M]_{\text{rot}} = [T]^{\text{T}} \cdot [M] \cdot [T]$$

Com  $[M]$  a matriz original (em seu próprio eixo coordenado) e  $[M]_{\text{rot}}$  a matriz rotacionada, sendo  $[M]$  uma matriz quadrada. No caso que queremos rotacionar uma matriz-coluna (carga), deve-se aplicar a seguinte equação:

$$\dots \text{math}:: [C]_{\text{rot}} = [T]^{\text{T}} \cdot [C]$$
  
'''

```
c = np.cos(self.theta)
s = np.sin(self.theta)
T = np.array([
    [ c, s, 0, 0, 0], # u1
    [-s, c, 0, 0, 0], # v1
    [ 0, 0, 1, 0, 0], # phi1
    [ 0, 0, 0, c, s, 0], # u2
    [ 0, 0, 0, -s, c, 0], # v2
    [ 0, 0, 0, 0, 0, 1], # phi2
])
return T
```

def build\_Ke\_matrix(self,):

'''

Método que constrói a matriz de elasticidade local do elemento

'''

E = self.E

A = self.A

L = self.L

I = self.I

if self.tipo == 'trelica':

```
    Ke = E*A/L*np.array([
        [ 1, 0, 0, -1, 0, 0], # u1
        [ 0, 0, 0, 0, 0, 0], # v1
        [ 0, 0, 0, 0, 0, 0], # phi1
        [-1, 0, 0, 1, 0, 0], # u2
        [ 0, 0, 0, 0, 0, 0], # v2
        [ 0, 0, 0, 0, 0, 0], # phi2
    ])
else:
```

```
    Ke = E*A/L*np.array([
        [ 1, 0, 0, -1, 0, 0], # u1
        [ 0, 0, 0, 0, 0, 0], # v1
        [ 0, 0, 0, 0, 0, 0], # phi1
        [-1, 0, 0, 1, 0, 0], # u2
        [ 0, 0, 0, 0, 0, 0], # v2
        [ 0, 0, 0, 0, 0, 0], # phi2
    ])
    Ke += ((E*I)/L**3)*np.array([

```

```
        [0, 0, 0, 0, 0, 0], # u1
        [0, 12, 6*L, 0, -12, 6*L], # v1
        [0, 6*L, 4*L**2, 0, -6*L, 2*L**2], # phi1
        [0, 0, 0, 0, 0, 0], # u2
        [0, -12, -6*L, 0, 12, -6*L], # v2
        [0, 6*L, 2*L**2, 0, -6*L, 4*L**2], # phi2
    ])

```

```
T = self.rotate_matrix()
Ke_ = T.T @ Ke @ T
```

```

    return Ke_

def build_Me_matrix(self,):
    '''
    Método que constrói a matriz de massas local do elemento
    '''

    E = self.E
    A = self.A
    L = self.L
    I = self.I
    rho = self.rho
    if self.tipo == 'trelica':
        Me = rho*A*L/6*np.array([
            [2, 0, 0, 1, 0, 0], # u1
            [0, 0, 0, 0, 0, 0], # v1
            [0, 0, 0, 0, 0, 0], # phi1
            [1, 0, 0, 2, 0, 0], # u2
            [0, 0, 0, 0, 0, 0], # v2
            [0, 0, 0, 0, 0, 0], # phi2
        ])
    else:
        Me = (rho*A*L/6*np.array([
            [2, 0, 0, 1, 0, 0], # u1
            [0, 0, 0, 0, 0, 0], # v1
            [0, 0, 0, 0, 0, 0], # phi1
            [1, 0, 0, 2, 0, 0], # u2
            [0, 0, 0, 0, 0, 0], # v2
            [0, 0, 0, 0, 0, 0], # phi2
        ]) +
        rho*A*L/420*np.array([
            [0, 0, 0, 0, 0, 0], # u1
            [0, 156, 22*L, 0, 54, -13*L], # v1
            [0, 22*L, 4*L**2, 0, 13*L, -3*L**2], # phi1
            [0, 0, 0, 0, 0, 0], # u2
            [0, 54, 13*L, 0, 156, -22*L], # v2
            [0, -13*L, -3*L**2, 0, -22*L, 4*L**2], # phi2
        ]))
    )
    T = self.rotate_matrix()
    Me_ = T.T @ Me @ T

    return Me_

def build_q_array(self,q):
    '''
    Método que constrói a matriz-coluna de forças distribuídas
    local do elemento, para uma função q que depende do tempo
    '''

    L = self.L

    Fe = q*L/12*np.array(
        [0, #u1
         6, #v1
         L, #phi1
         0, #u2
         6, #v2
         -L #phi2
    ]
)

```

```

T      = self.rotate_matrix()
Fe_   = T.T @ Fe
self.Fe = Fe_
pass

def is_contained(self, inicio_dominio, final_dominio):
    """
    Método que determina se um elemento é pertence ao domínio
    determinado por dois nós.
    """
    contained_ini = np.all(self.ini >= inicio_dominio)
    contained_fim = np.all(self.fim <= final_dominio)
    return contained_ini and contained_fim

class Sistema:
    """
    Classe que abstrai a criação de um sistema a ser resolvido
    """
    def __init__(self,
                 struct_dict,
                 dict_cc,
                 dict_cargas_distribuidas,
                 list_cargas_pontuais,
                 alpha      = 0.1217,
                 beta       = 0.0087,
                 gamma     = 1/2,
                 beta_newmark = 1/4,
                 dx=1
                 ):
        self.struct_dict      = struct_dict
        self.cc               = dict_cc
        self.cargas           = dict_cargas_distribuidas
        self.forcas           = list_cargas_pontuais
        self.dx               = dx
        (self.point_list,
         self.elements,)     = self._gera_elementos()
        self.struct_ids       = self._gera_ids()
        self.num_nodes        = len(self.point_list)
        self.num_elements     = len(self.elements)
        self.num_gdl          = 3*len(self.point_list)
        #para cada nó, temos u, v e phi

        #parametros Rayleigh
        self.alpha = alpha
        self.beta  = beta

        #cria-se as matrizes vazias
        self.Keg = self._create_global_matrix()
        self.Meg = self._create_global_matrix()
        self.Feg = self._create_global_vector()
        self.Ceg = self._create_global_vector()

        #a partir dai, se preenche as matrizes
        self._fill_global_rigidity_matrix() #matriz de rigidez
        self._fill_global_mass_matrix()      #matriz de massa
        if self.dx == 1:
            #só usados em análises com dx=1
            self.Feg = self._fill_Fe_matrix()    #vetor de cargas inicial
            self.Ceg = self._fill_Ce_matrix()    #vetor de forças inicial

```

```

#aplica-se as condições de contorno na matriz de rigidez
self._apply_cc()

def _gera_elementos(self,):
    #pontos (x, y)/ nós do sistema
    point_list = []

    #elementos do sistema, que ligam dois nós distintos
    elements = []

    #propriedades que são passadas aos elementos
    keys_elemento = ['E', 'A', 'rho', 'I']

    #para cada uma das estruturas
    num_element=0
    for struct in self.struct_dict:
        #pegamos os pontos que as compõe
        points = self.struct_dict[struct]['points']
        #adicionamos estes nós a lista de todos os nós
        point_list += points

        #com eles, pode-se criar cada elemento dessa estrutura
        for i in range(len(points)-1):
            #criação dos elementos
            element = Elemento(
                points[i], points[i+1],
                self.struct_dict[struct]['tipo'],
                {key: self.struct_dict[struct][key]
                 for key in keys_elemento},
                numero=num_element
            )
            elements.append(element)
            num_element += 1

    #pega os nós unicos apenas
    point_list = np.unique(point_list, axis=0)

    return point_list, elements

def _gera_ids(self,):
    #criar uma lista dos ids na lista de pontos que correspondem aos
    #elementos da estrutura, assim pode-se plotar a deformação deles
    struct_ids = {}
    #dicionario com os elementos e os ids dos pontos
    for struct in self.struct_dict:
        #para cada estrutura se pega os pontos dela
        points = self.struct_dict[struct]['points']
        points_ids_element = []
        for point in points:
            #para cada ponto se encontra os ids correspondentes
            id_ponto = self._find_point_index(point)
            points_ids_element.append(id_ponto)
        #adiciona-se os ids do elemento
        struct_ids[struct] = points_ids_element
    return struct_ids

def _create_global_matrix(self,):
    Keg = np.zeros((self.num_gdl, self.num_gdl))

```

```

        return Keg

    def _create_global_vector(self,):
        Feg = np.zeros((self.num_gdl, 1))
        return Feg

    def _find_point_index(self, point):
        points = self.point_list
        mask = np.all(points == point, axis=1)

        index = int(np.where(mask)[0])
        return index

    def _find_element_indexes(self, element):
        '''
        Função que dado um elemento da estrutura, encontra o índice
        dos nós que este elemento liga
        '''
        ini, fim = element.ini, element.fim

        ini = self._find_point_index(ini)
        fim = self._find_point_index(fim)

        return ini, fim

    def _find_element_by_indexes(self, ini, fim):
        #acha um elemento dentre todos por seus índices
        for el in self.elements:
            el_id_ini, el_id_fim = self._find_element_indexes(el)
            if el_id_ini == ini and el_id_fim == fim:
                return el
        #se não achar nenhum elemento
        return None

    def _find_element_by_number(self, number):
        #acha um elemento dentre todos por seu número
        for el in self.elements:
            num = el.Numero
            if number == num:
                return el
        #se não achar nenhum elemento
        return None

    def _fill_global_rigidity_matrix(self,):
        #para cada elemento, se adiciona as suas componentes à
        #matriz global
        for i, el in enumerate(self.elements):
            ini_index, fim_index = self._find_element_indexes(el)
            #pega-se os nós do elemento

            Ke_local = el.Ke
            #pega-se sua matriz de rigidez local

            indexes_ini = np.arange(3*ini_index, 3*(ini_index+1))
            indexes_fim = np.arange(3*fim_index, 3*(fim_index+1))
            indexes = np.concatenate((indexes_ini, indexes_fim), axis=0)
            #com os indices, pode-se saber em quais posições colocar na
            # matriz global

            Ke_globalizado = globalize_K(
                Ke_local, self.num_gdl, indexes

```

```

        )
    self.Keg += Ke_globalizado
pass

def _fill_global_mass_matrix(self,):
    #para cada elemento, se adiciona para a matriz de massa global
    for el in self.elements:
        ini_index, fim_index = self._find_element_indexes(el)
        #pega-se os nós do elemento

        Me_local = el.Me
        #pega-se sua matriz de rigidez local

        indexes_ini = np.arange(3*ini_index, 3*(ini_index+1))
        indexes_fim = np.arange(3*fim_index, 3*(fim_index+1))
        indexes = np.concatenate((indexes_ini, indexes_fim), axis=0)
        #com os indices, pode-se saber em quais posições colocar na
        # matriz global

        #se globaliza a matriz Me
        Me_globalizado = globalize_K(Me_local, self.num_gdl, indexes)
        self.Meg += Me_globalizado
    pass

def _fill_Fe_matrix(self,t=0):
    #cria-se a matriz de cargas distribuídas vazia
    Feg = np.zeros(self.num_gdl)

    for grupo in self.cargas:
        #q_func depende de t e da posição
        inicio, final, q_func = self.cargas[grupo]
        for elemento in self.elements:
            tem_carga = elemento.is_contained(inicio, final)
            if tem_carga:
                elemento.build_q_array(q_func(t))
                Fe_local = elemento.Fe
                ponto_inicial = elemento.ini
                ponto_final = elemento.fim

                ind          = self._find_point_index(ponto_inicial)
                idx          = [3*ind, 3*ind+1, 3*ind+2]
                ind          = self._find_point_index(ponto_final)
                idx          += [3*ind, 3*ind+1, 3*ind+2]

                Fe_globalizada = globalize_F(Fe_local,
                                              self.num_gdl,
                                              idx)
                Feg += Fe_globalizada
    return Feg

def _fill_Ce_matrix(self,t=0):
    #cria-se a matriz de forças vazia
    Ceg = np.zeros(self.num_gdl)

    for forca in self.forcas:
        #F_func depende do tempo e da posição
        pontos_aplicacao, F_func, graus_atuantes = forca

        for ponto in pontos_aplicacao:
            modulo_F = F_func(t, ponto[0])

```

```

        indice = self._find_point_index(ponto)
        indices = []

        forcas = []
        if 'u' in graus_atuantes:
            forcas.append(modulo_F)
        else:
            forcas.append(0)
        if 'v' in graus_atuantes:
            forcas.append(modulo_F)
        else:
            forcas.append(0)
        if 'phi' in graus_atuantes:
            forcas.append(modulo_F)
        else:
            forcas.append(0)

        #indices
        indices.append(3*indice)
        indices.append(3*indice+1)
        indices.append(3*indice+2)

        Ce_local      = np.array(forcas)
        Ce_globalizada = globalize_F(
            Ce_local,
            self.num_gdl,
            indices
        )

        Ceg += Ce_globalizada
    return Ceg

def _apply_cc(self,):
    #precisa-se de uma forma de pegar os pontos com condição de
    #contorno e quais movimentos estão restritos.
    indices_restritos_global = []
    for tipo_cc in self.cc.keys():
        #cada tipo de condição tem pontos de aplicação e os
        #movimentos que restringe
        lista_aplicacoes = self.cc[tipo_cc]
        for tupla in lista_aplicacoes:
            ponto_aplicacao, restricoes = tupla
            indice = self._find_point_index(ponto_aplicacao)

            if 'u' in restricoes:
                indices_restritos_global.append(3*indice)
            if 'v' in restricoes:
                indices_restritos_global.append(3*indice+1)
            if 'phi' in restricoes:
                indices_restritos_global.append(3*indice+2)
    self.Meg = anula_cond_contorno(self.Meg,
        indices_restritos_global)
    self.Keg = anula_cond_contorno(self.Keg,
        indices_restritos_global)
    pass

def reduz_graus_redundantes(self,):
    indices_restritos_global = []
    for tipo_cc in self.cc.keys():
        lista_aplicacoes = self.cc[tipo_cc]

```

```

        for tupla in lista_aplicacoes:
            ponto_aplicacao, restricoes = tupla
            indice = self._find_point_index(ponto_aplicacao)

            if 'u' in restricoes:
                indices_restritos_global.append(3*indice)
            if 'v' in restricoes:
                indices_restritos_global.append(3*indice+1)
            if 'phi' in restricoes:
                indices_restritos_global.append(3*indice+2)
        Meg = reduz_matriz(self.Meg,
                            indices_restritos_global, eliminate=True)
        Keg = reduz_matriz(self.Keg,
                            indices_restritos_global, eliminate=True)
        if self.dx == 1:
            Ceg = reduz_matriz(self.Ceg,
                                indices_restritos_global, eliminate=True)
            Feg = reduz_matriz(self.Feg,
                                indices_restritos_global, eliminate=True)
        else:
            Ceg, Feg = self.Ceg, self.Feg
        return Meg, Keg, Ceg, Feg, indices_restritos_global

    def solve_system(self,):
        """
        Função que resolve o sistema sendo ddot{U} nula.
        Com isto, pode-se encontrar a condição inicial do sistema
        para uma análise transiente, caso o sistema inicie com
        forças/cargas.
        """
        lado_direito = self.Ceg + self.Feg
        lado_esquerdo = self.Keg
        Ue = np.linalg.solve(lado_esquerdo, lado_direito)
        return Ue

    def acha_modos_de_vibrar(self,n=6):
        Meg, Keg, Ceg, Feg, restritos = self.reduz_graus_redundantes()
        A = np.linalg.inv(Meg) @ Keg
        values, vectors = np.linalg.eig(A)
        values = np.sqrt(values)/(2*np.pi)

        idx = values.argsort() #[::-1]
        values = values[idx]
        vectors = vectors[:,idx].T

        #para voltar ao sistema orginal, vamos pegar os indices livres
        graus_totais = np.arange(self.num_gdl)
        #os restritos já temos da função que restringe
        livres = np.array(list(set(graus_totais) - set(restritos)))
        deformacoes = []
        for i in range(n):
            #cria-se uma matriz de deslocamentos vazia
            u = np.zeros(self.Keg.shape[0])
            u[np.ix_(livres)] = vectors[i]
            deformacao = deleta_phis(u)
            deformacao = np.array(deformacao)

            deformacao = deformacao.reshape((-1,2))
            deformacoes.append(deformacao)
        #valores de frequencia natural
        #deformacoes em x e y para cada nó do sistema

```

```

        return values[:n], deformacoes

    def plot_struct(self,
                    escala=1,
                    deformacoes=None,
                    node_names=False,
                    elements_names=False,
                    fontsize=4,
                    legend=None):
        plt.figure(figsize=(8, 6), dpi=400)
        lines, names = [], []
        for struct in self.struct_dict:
            color = self.struct_dict[struct]['color']
            lines.append(Line2D([0], [0], color=color, lw=2))
            names.append(struct)
            ids = self.struct_ids[struct]
            points = self.point_list[np.ix_(ids)]
            x, y = zip(*points)
            if deformacoes is None:
                #não passamos deformacoes
                x_def = np.zeros_like(x)
                y_def = np.zeros_like(y)
            else:
                points_deformados = deformacoes[np.ix_(ids)]
                x_def, y_def = zip(*points_deformados)
                x_new = [original+escala*deformacao for original,
                         deformacao in zip(x, x_def)]
                y_new = [original+escala*deformacao for original,
                         deformacao in zip(y, y_def)]
            #plota-se os nós
            plt.scatter(x_new, y_new, facecolors='none',
                        s=3, edgecolors=color)
            if node_names == True:
                from adjustText import adjust_text
                ax = plt.gca()
                texts = [plt.text(x_new[i], y_new[i], '%s' %ide,
                                  fontsize=fontsize)
                         for i, ide in enumerate(ids)]
                adjust_text(texts)

            #coloca-se a linha ligando os pontos
            texts_elements = []
            for i in range(0, len(x)-1):
                plt.plot(x_new[i:i+2], y_new[i:i+2], color+'-')
                if elements_names == True:
                    #acha-se o elemento
                    elemento = self._find_element_by_indexes(
                        ids[i], ids[i+1])
                    num = elemento.Numero
                    text = plt.text((x_new[i]+x_new[i+1])/2,
                                    (y_new[i]+y_new[i+1])/2,
                                    '%s' %num,
                                    fontsize=fontsize)
                    texts_elements.append(text)
            ax = plt.gca()
            ax.legend(lines, names)
            if legend != None:
                plt.title(legend)
            pass

    def _gera_omegas(self, wi=0, wf=10, dw=0.01, fator=10):

```

```

omegas, _ = self.acha_modos_de_vibrar()
omegas = [omega for omega in omegas if omega<wf]
omegas = [wi-dw] + omegas
omegas = omegas + [wf-dw]
omega_atual = omegas[0]
omega_prox = omegas[1]
omega_vector = np.array([])
for i in range(1, len(omegas)):
    antes = np.arange(omega_atual+dw, omega_prox-dw, dw)
    proximo = np.arange(omega_prox-dw, omega_prox+dw, dw/fator)
    omega = np.concatenate((antes, proximo))
    #depois = np.arange(omega_prox+dw, wf, dw)
    omega_atual = omegas[i]
    if i != len(omegas)-1:
        omega_prox = omegas[i+1]
    omega_vector = np.concatenate((omega_vector, omega))
return omega_vector

def analise_harmonica(self,):
    omega_vector = self._gera_omegas()

    freq_vector = (2*np.pi)*omega_vector

    F = self._f(0)

    U_total = np.zeros((self.num_gdl, len(omega_vector)))
    for i, omega in enumerate(freq_vector):
        K_chapeu = (-omega**2 * self.Meg + self.Keg)
        try:
            U = np.linalg.solve(K_chapeu, F)
        except:
            pass
        U_total[:, i] = U
    return np.abs(U_total), omega_vector

def _C(self, t):
    return self._fill_Ce_matrix(t)

def _F(self, t):
    return self._fill_Fe_matrix(t)

def _f(self, t):
    f = self._C(t) + self._F(t)
    return f

def sistema_transiente(self, gamma, beta_newmark,
                      tf, dt, cc_nula=False):
    Meg, Keg, Ceg, Feg, restritos = self.reduz_graus_redundantes()
    #para voltar ao sistema orginal, vamos pegar os indices livres
    graus_totais = np.arange(self.num_gdl)
    #os restritos já temos da função que restringe
    livres = np.array(list(set(graus_totais) - set(restritos)))

    #matriz de amortecimento
    Cgm = self.alpha * Meg + self.beta * Keg

    #cria-se o vetor de tempos
    time_vector = np.arange(0, tf, dt)

    #coloca-se as condições iniciais
    if not cc_nula:

```

```

        U_atual = self.solve_system()[np.ix_(livres)]
    else:
        U_atual = np.zeros_like(livres)
    V_atual = np.zeros_like(livres)
    A_atual = np.linalg.inv(Meg) @ (
        self._f(0)[np.ix_(livres)] - Cgm@V_atual - Keg@U_atual
    )

    M_chapeu = Meg + dt*gamma*Cgm+(dt**2)*beta_newmark*Keg

    U_transiente = np.zeros((self.num_gdl, len(time_vector)))

    for i, t in enumerate(time_vector):
        #U_atual, V_atual, A_atual
        U_transiente[:, i][np.ix_(livres)] = U_atual

        F_fut = self._f(t+dt)[np.ix_(livres)]
        F_chapeu = (F_fut
                    - Cgm@(V_atual + dt*(1-gamma)*A_atual)
                    - Keg@(U_atual +
                           + dt*V_atual+((dt**2)/2)*(1-2*beta_newmark)*A_atual)
        )

        A_fut = np.linalg.solve(M_chapeu, F_chapeu)
        U_fut = (U_atual
                  + dt*V_atual
                  + ((dt**2)/2)*((1-2*beta_newmark)*A_atual +
                  + 2*beta_newmark*A_fut)
        )
        V_fut = (V_atual
                  + dt* ((1-gamma)*A_atual + gamma*A_fut)
        )

        #atualiza-se os vetores
        U_atual = U_fut
        V_atual = V_fut
        A_atual = A_fut

    return U_transiente, time_vector

### definição da geometria ###

# Pontos notáveis ([x, y])
A = np.array([0, 0])
B = np.array([5, 0])
C = np.array([9, 0])
D = np.array([18, 0])
E = np.array([36, 0])
F = np.array([36, 18])
G = np.array([36, -5]) # ponto criado artificialmente
H = np.array([45, 0]) # ponto criado artificialmente
I = np.array([65, 0]) # ponto criado artificialmente

# Dimensões vigas/treliças
A1 = (1.8, 0.9) # torre
A2 = (0.9, 0.9) # ponte
D3 = (0.05,) # treliças

# Propriedades
Young_modulus = 210e9 #Pa
rho = 7600 #Kg/m^3

```

```

#valores de dx e valor escolhido
dx_list = [1, 2, 3]
dx = 1

# Plataforma (viga)
# A-B-C-D-E-H-I
plataforma_points = [A, B, C, D, E, H, I]
plataforma_points = discretiza_elemento(plataforma_points, dx)

# Torre (viga)
# G-E-F
torre_points = [G, E, F]
torre_points = discretiza_elemento(torre_points, dx)

# Cabos (treliça)
# 1: B-F
cab01_points = [B, F]
# 2: D-F
cab02_points = [D, F]
# 3: I-F
cab03_points = [I, F]

## parâmetros da análise transiente ##

#valores de va e valor escolhido
va_list = [0.5, 1, 2]
va= 0.5

#discretização no tempo
deltat = 0.1

# Valores iniciais de N1 e N2, número de pessoas, em função do tempo
def n1(t, va=1):
    if t < 27/va:
        return 20
    elif 27/va <= t and t <= 47/va:
        return va*t-7
    else:# t > 47/va
        return 40

def n2(t, va=1):
    if t<= 20/va:
        return 20 - va*t
    else: #t > 20/va
        return 0

N1 = lambda t: n1(t, va=va)
N2 = lambda t: n2(t, va=va)

# Lambda functions para o valor da carga distribuída
# L é parâmetro desta fórmula para o caso em que queremos dividir
# os elementos com carga concentrada em sub-elementos, os quais
# terão suas próprias cargas distribuídas
q = lambda t, N, L: -N(t)*80*9.8/L
#importante ressaltar o sinal negativo (-) já que a força
#aponta para baixo!
q1 = lambda t: q(t, N1, 9)
q2 = lambda t: q(t, N2, 9)

```

```

#função para definir a força cortante
def v_i(t, x_ponto, va=1):
    if t <= (E[0] - x_ponto)/va:
        return 0
    elif t > (E[0] - x_ponto)/va and t <= (E[0] + 20 - x_ponto)/va:
        return -80*9.8*(1 - np.cos(2*np.pi*va*t))/2
    else:
        return 0
Vi = lambda t, x: v_i(t, x, va=va)

## montagem do sistema ##

sistema_dict = {
    'Plataforma': {
        'points': plataforma_points,
        'A': A2[0]*A2[1],
        'tipo': 'viga/portico',
        'E': Young_modulus,
        'color': 'b',
        'I': calculate_I(A2[0], A2[1]),
        'rho': rho,
    },
    'Torre': {
        'points': torre_points,
        'A': A1[0]*A1[1],
        'tipo': 'viga/portico',
        'E': Young_modulus,
        'color': 'r',
        'I': calculate_I(A1[0], A1[1]),
        'rho': rho,
    },
    'Cabo1': {
        'points': cabo1_points,
        'A': (np.pi*D3[0]**2)/4,
        'tipo': 'trelica',
        'E': Young_modulus,
        'color': 'g',
        'I': calculate_I(D3[0]),
        'rho': rho,
    },
    'Cabo2': {
        'points': cabo2_points,
        'A': (np.pi*D3[0]**2)/4,
        'tipo': 'trelica',
        'E': Young_modulus,
        'color': 'g',
        'I': calculate_I(D3[0]),
        'rho': rho,
    },
    'Cabo3': {
        'points': cabo3_points,
        'A': (np.pi*D3[0]**2)/4,
        'tipo': 'trelica',
        'E': Young_modulus,
        'color': 'g',
        'I': calculate_I(D3[0]),
        'rho': rho,
    },
}

```

```

condicoes_contorno_dict = {
    #tipo_cc: [ponto_aplicação, [graus_restrigidos]],
    'engaste': [(G, ['u', 'v', 'phi'])],
    'apoio fixo': [(I, ['u', 'v'])],
}
}

#N1 e N2 são duas funções do tempo
dict_cargas_distribuidas = {
    #carga : [inicio, fim, q(t, posicao)],
    "grupo1": [A, C, q1],
    "grupo2": [E, H, q2],
}
}

lista_forcas = [
    #[[pontos_aplicação], módulo_F(t, posicao), [graus_atuantes]],
    [discretiza_elemento([C, E], 1), Vi, ['v']]
]

sistema = Sistema(
    sistema_dict,
    condicoes_contorno_dict,
    dict_cargas_distribuidas,
    lista_forcas,
    dx=dx
)

#a.i
values, vectors = sistema.acha_modos_de_vibrar()

#a.ii
U_transiente, time_vector = sistema.sistema_transiente(
    #gamma e betas do método da aceleração média constante
    gamma= 1/2,
    beta = 1/4,
    #tempo final de 55/va
    tf = 55/va,
    #delta t para simulação
    deltat = deltat
)

```

#define-se novas funções

```

def v_i_transiente(t=0, x=0):
    return -80*9.8
Vi_harmonica = lambda t, x: v_i_transiente(t)

lista_forcas_harmonica = [
    #[[pontos_aplicação], módulo_F(t, posicao), [graus_atuantes]],
    [discretiza_elemento([C, E], 1), Vi_harmonica, ['v']]
]

```

```

sistema_harmonico = Sistema(
    sistema_dict,
    condicoes_contorno_dict,
    dict_cargas_distribuidas,
    lista_forcas_harmonica
)

#a.iii
U_harmonico, omegas = sistema_harmonico.analise_harmonica()

```

```
# pontos solicitados no enunciado
indice_A = find_index_dispacement(sistema, A, 'vertical')
indice_B = find_index_dispacement(sistema, B, 'vertical')
indice_C = find_index_dispacement(sistema, C, 'vertical')
indice_F = find_index_dispacement(sistema, F, 'horizontal')
```

---