UNIVERSIDADE
BEIRA INTERIOR

Trabalho 03 – Placas de Kirchhoff: Soluções Numéricas

José Miguel Correia Barros, M15321

Engenharia Aeronáutica
(2º ciclo de estudos)

Docente: Prof. Doutor Thiago Assis Dutra

24 de Novembro de 2025

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Descrição do Problema

O presente relatório apresenta a proposta de resolução do Trabalho 3 da Unidade Curricular de Placas e Cascas, focando na análise de placas finas ortotrópicas submetidas a carregamentos externos, descritas pela equação de Kirchhoff. Este modelo é fundamental um vez que permite determinar a deflexão das placas sob forças aplicadas. Como soluções analíticas são muitas vezes impraticáveis, recorrem-se a métodos numéricos, sendo o Método dos Elementos Finitos amplamente utilizado para obter resultados precisos em problemas estruturais complexos.

1.1 Tarefa 1

Na Tarefa 1 é analisada uma placa com dimensões a e b, espessura t, composta por um material ototrópico com massa volúmica ρ . Esta placa encontra-se encastrada nas extremidades correspondentes a $x = 0$ e $x = a$, enquanto nas bordas $y = 0$ e $y = b$ está simplesmente apoiada. A configuração geométrica e as condições de fronteira da placa podem ser observadas na figura 1.1,

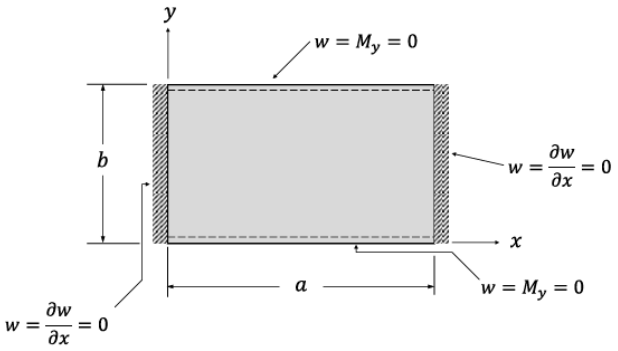


Figura 1.1: Placa ortotrópica referente à Tarefa 1.

Na condição de vibração livre, a equação da energia potencial para a placa é dada por (1.1):

$$\begin{aligned}\Pi = \frac{1}{2}\int\int \Big[& D_{11}\,(w_{,xx})^2 + 2D_{12}\,w_{,xx}w_{,yy} \\ & + D_{22}\,(w_{,yy})^2 + 4D_{66}\,(w_{,xy})^2\Big]\,\mathrm{d}y\,\mathrm{d}x \\ & - \frac{1}{2}\,\omega^2\rho t\int\int w^2\,\mathrm{d}y\,\mathrm{d}x = 0\end{aligned} \quad (1.1)$$

Através do Método de Ritz, vai-se determinar frequência fundamental ω associada à vibração livre da placa ilustrada na figura 1.1, bem como o respetivo modo de vibração. Para tal, considera-se que a forma aproximada da deflexão da placa retangular pode ser expressa conforme apresentado na equação (1.2), de acordo com a formulação do Método de Ritz.

$$w(x,y) \approx W_{mn}(x,y) = \sum_{i=1}^{M}\sum_{j=1}^{N} c_{ij}\,X_i(x)\,Y_j(y) \quad (1.2)$$

Onde c_{ij}, corresponde ao coeficiente de Ritz e $X_i(x)$ e $Y_j(y)$ devem cumprir apenas as condições de fronteira geométricas impostas ao problema.

Em $x = 0$ e $x = a$ a placa da figura 1.1 encontra-se encastradas nessas bordas e por isso $X_i(x)$ é dado pela equação (1.3):

$$X_i(x) = \left(\frac{x}{a}\right)^{i+1} - 2\left(\frac{x}{a}\right)^{i+2} + \left(\frac{x}{a}\right)^{i+3} \quad (1.3)$$

De maneira análoga, em $y = 0$ e $y = b$ a placa encontra-se simplesmente apoiada e por isso $Y_j(y)$ é dada pela equação (1.4):

$$Y_j(y) = \left(\frac{y}{b}\right)^{j} - \left(\frac{y}{b}\right)^{j+1} \quad (1.4)$$

1.2 Tarefa 2

Uma placa fina de dimensões $a \times b$ e espessura t, feita de um material ortotrópico, está sujeita a uma pressão p_0 uniformemente distribuída sobre toda a sua superfície superior (plano xy). As arestas da placa são designadas por Borda 1, Borda 2, Borda 3 e Borda 4 conforme a figura 1.2.

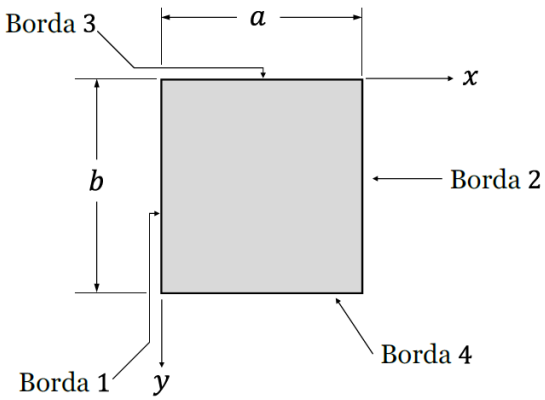


Figura 1.2: Placa ortotrópica referente à Tarefa 2

Na alínea *a)* é solicitada a determinação da expressão da matriz de elasticidade $[D]$, assumindo que os eixos principais de ortotropia do material estão orientados paralelamente aos eixos x e y.

Na alínea *b)*, pretende-se o desenvolvimento de um programa de cálculo baseado no Método dos Elementos Finitos (FEM), capaz de determinar as deflexões $w(x, y)$ da placa no domínio definido por $x \in [0, a]$ e $y \in [0, b]$, bem como de avaliar as forças de reacção nos apoios correspondentes.

Na alínea *c)*, é pedido que se analisem dois casos particulares:

(i) Quando os eixos principais do material não estão alinhados com os eixos x e y;

(ii) Quando o material é constituído por um compósito laminado simétrico, não balanceado e composto por n camadas.

Adicionalmente, são fornecidas algumas notas complementares:

(i) Na discretização da placa, podem ser utilizados elementos triangulares de três nós (com 9 graus de liberdade) ou elementos quadriláteros de quatro nós (com 12 graus de liberdade);

(ii) A malha deve ser progressivamente refinada até se alcançar a convergência dos resultados, devendo a escolha final do número de elementos ser devidamente justificada.

Os parâmetros necessários para o desenvolvimento da Tarefa 1 e para a Tarefa 2 são apresentados na Tabela 1.1.

Caso	a [mm]	b [mm]	t [mm]	E_1 [GPa]	E_2 [GPa]	ν_{12}	G_{12} [GPa]	p_0 [N/m^2]	B1	B2	B3	B4
6	750	600	2	130	10	0.26	5	175	C	C	F	F

Tabela 1.1: Propriedades geométricas, mecânicas e condições de fronteira.

Capítulo 2

Metodologia de Solução

2.1 Tarefa 1

Para determinar a frequência fundamental de vibração ω e o modo de vibração correspondente de uma placa ortotrópica, aplica-se o método de aproximação de Ritz, conforme indicado na equação (1.2). As funções $X_i(x)$ e $Y_j(y)$ são escolhidas de modo a satisfazer as condições de fronteira do problema. A partir desta aproximação, podem obter-se as derivadas de $w(x, y)$, designadamente $w_{,xx}$, $w_{,yy}$ e $w_{,xy}$, que são posteriormente substituídas na equação de equilíbrio correspondente ao caso ortotrópico.

No caso ortotrópico, os momentos fletores e as curvaturas estão relacionados pela matriz de rigidez flexional $[D]$, expressa pela equação (2.1):

$$\begin{bmatrix} M_x \\ M_y \\ M_{xy} \end{bmatrix} = [D] \begin{bmatrix} \kappa_x \\ \kappa_y \\ \kappa_{xy} \end{bmatrix} . \quad (2.1)$$

A matriz $[D]$ representa a rigidez à flexão da placa e é dada por:

$$[D] = \frac{t^3}{12} \begin{bmatrix} Q_{11} & Q_{12} & 0 \\ Q_{12} & Q_{22} & 0 \\ 0 & 0 & Q_{66} \end{bmatrix} . \quad (2.2)$$

Os coeficientes Q_{11}, Q_{22}, Q_{12} e Q_{66} dependem das propriedades elásticas do material e são definidos pelas equações (2.3) a (2.6):

$$Q_{11} = \frac{E_1}{1 - \nu_{12}\nu_{21}}, \quad (2.3)$$

$$Q_{22} = \frac{E_2}{1 - \nu_{12}\nu_{21}}, \quad (2.4)$$

$$Q_{12} = \frac{\nu_{12}E_2}{1 - \nu_{12}\nu_{21}}, \quad (2.5)$$

$$Q_{66} = G_{12}. \quad (2.6)$$

A variação da energia potencial pode ser escrita como:

$$\delta\Pi = \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial \Pi}{\partial c_{ij}}\, \delta c_{ij}, \quad (2.7)$$

sendo c_{ij} os coeficientes da função aproximada $w(x, y)$ utilizada no método de Ritz.

A partir da equação (2.7), obtêm-se as relações apresentadas nas equações (2.8) a (2.11), que exprimem as variações das derivadas parciais de w relativamente aos coeficientes c_{ij}:

$$\delta w_{,xx} = \frac{\partial w_{,xx}}{\partial c_{ij}}\, \delta c_{ij}, \quad (2.8)$$

$$\delta w_{,yy} = \frac{\partial w_{,yy}}{\partial c_{ij}}\, \delta c_{ij}, \quad (2.9)$$

$$\delta w_{,xy} = \frac{\partial w_{,xy}}{\partial c_{ij}}\, \delta c_{ij}, \quad (2.10)$$

$$\delta w = \frac{\partial w}{\partial c_{ij}}\, \delta c_{ij}. \quad (2.11)$$

Substituindo as equações (2.8)–(2.11) na expressão da energia potencial (2.7), e considerando a matriz $[D]$ definida pela matriz (2.2),obtemos a equação de equilíbrio da placa ortotrópica.

Como as variações δ são arbitrárias, o termo que as acompanha deve anular-se. Assim, ao reorganizar a equação em função dos coeficientes c_{ij}, obtém-se a equação característica da placa, que permite determinar a frequência fundamental de vibração ω.

Com base nesta equação, é possível obter o modo de vibração associado. Para representar graficamente este modo, pode ser utilizado um programa desenvolvido em linguagem *Python*, que gera a superfície $w(x, y)$ considerando valores unitários para a, b e c_{ij}. Esta representação permite visualizar a forma modal correspondente à frequência fundamental da placa ortotrópica.

2.2 Tarefa 2

Nesta tarefa, pretende-se determinar, para uma placa ortotrópica sujeita a um carregamento uniforme, a matriz de elasticidade $[D]$, as deflexões $w(x, y)$, as reações nos apoios e as tensões σ_x, σ_y e τ_{xy}, recorrendo ao Método dos Elementos Finitos (MEF).

2.2.1 Alínea *a)*

A matriz de rigidez flexional do material ortotrópico considerada nesta análise corresponde à obtida na Tarefa 1 e é dada pela equação (2.2). Esta matriz traduz a rigidez à flexão da placa e contém os coeficientes Q_{ij} definidos nas Equações (2.3)–(2.6), os quais dependem diretamente das propriedades elásticas do material (E_1, E_2, ν_{12}, ν_{21} e G_{12}).

2.2.2 Alínea *b)*

Nesta alínea, recorre-se ao Método dos Elementos Finitos (MEF) para determinar as deflexões e as reações na placa ortotrópica. O procedimento geral pode ser descrito pelos seguintes passos:

1. Determinação da matriz de rigidez elementar $[K]^e$ de acordo com as propriedades do material e geometria do elemento;
2. Montagem da matriz global $[K]$ a partir das contribuições de todos os elementos;
3. Cálculo do vetor global de forças nodais $\{Q\}$ associado ao carregamento uniforme p_0;
4. Aplicação das condições de fronteira e resolução do sistema linear de equações:

$$\{\delta\} = [K]^{-1}\{Q\};$$

5. Cálculo das reações nos apoios, obtidas por:

$$\{R\} = [K]_{\text{global}}\{\delta\} - \{Q\}.$$

A matriz de rigidez elementar é obtida pela Equação (2.12):

$$[K]^e = \left([C]^{-1}\right)^T \left(\iint_A [H]^T[D][H]\,dx\,dy\right)[C]^{-1}, \quad \textbf{(2.12)}$$

onde:

- $[H]$ é a matriz das derivadas das funções de forma do elemento;
- $[C]$ relaciona os coeficientes polinomiais a_i com os graus de liberdade nodais.

A matriz $[H]^T$, conforme apresentada em (2.13) é dada por:

$$[H]^T = \begin{bmatrix} 0 & 0 & 0 & -2 & 0 & 0 & -6x & -2y & 0 & 0 & -6xy & 0 \\ 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & -2x & -6y & 0 & -6xy \\ 0 & 0 & 0 & 0 & -2 & 0 & 0 & -4x & -4y & 0 & -12x & -12y \end{bmatrix} \quad (2.13)$$

A equação (2.14) define a função de deslocamento w_e como:

$$w_e = a_1 + a_2x + a_3y + a_4x^2 + a_5xy + a_6y^2 + a_7x^3 + a_8x^2y + a_9xy^2 + a_{10}y^3 + a_{11}x^3y + a_{12}xy^3 \quad (2.14)$$

Desta forma, a matriz $[C]$ é dada por (2.15):

$$[C] = \begin{bmatrix} \frac{\partial w_{ei}}{\partial a_1} & \cdots & \frac{\partial w_{ei}}{\partial a_{12}} \\ \frac{\partial \theta_{xi}}{\partial a_1} & \cdots & \frac{\partial \theta_{xi}}{\partial a_{12}} \\ \frac{\partial \theta_{yi}}{\partial a_1} & \cdots & \frac{\partial \theta_{yi}}{\partial a_{12}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \theta_{yn}}{\partial a_1} & \cdots & \frac{\partial \theta_{yn}}{\partial a_{12}} \end{bmatrix} \quad (2.15)$$

Após a obtenção da matriz $[C]$, procede-se à sua inversão. De acordo com a equação (2.12), é então possível determinar a matriz $[K]_e$ correspondente a cada elemento. O passo seguinte consiste em formar a matriz global $[K]$, a qual é construída através do processo de *meshing* das matrizes elementares $[K]_e$. Esse espalhamento é realizado com base na matriz de conectividade, que assegura a correta correspondência entre os graus de liberdade locais e globais.

Desta forma, obtém-se a matriz global $[K]$. Esta matriz é utilizada para determinar o vetor de deslocamentos $\{\delta\}$, recorrendo à seguinte expressão:

$$\{\delta\} = [K]^{-1} \cdot \{Q\} \quad (2.16)$$

Através da equação (2.16), verifica-se que é necessário calcular a inversa da matriz global $[K]$. No entanto, sabe-se que esta matriz apresenta uma estrutura em banda e, na forma original, o seu determinante é nulo. Consequentemente, não é possível inverter diretamente $[K]$ sem efetuar determinados ajustes. Para que a inversão possa ser realizada,

é necessário aplicar as condições de fronteira do problema, o que implica efetuar cortes apropriados nas linhas e colunas correspondentes.

No caso de estudo em análise, as condições de fronteira fornecidas determinam quais os graus de liberdade que devem ser fixos, permitindo assim obter uma matriz $[K]$ modificada e invertível.

As condições de fronteira da placa são definidas da seguinte forma:

- As bordas 1 e 2, correspondentes a $x = 0$ e $x = a$, estão encastradas. Assim, todos os graus de liberdade são nulos nestes limites, isto é:

$$w = 0, \quad \theta_x = 0, \quad \theta_y = 0$$

- As bordas 3 e 4, correspondentes a $y = 0$ e $y = b$, são livres. Consequentemente, nenhum dos parâmetros é restringido, permanecendo:

$$w \neq 0, \quad \theta_x \neq 0, \quad \theta_y \neq 0$$

O vetor de corte é construído de forma a identificar a posição dos nós pertencentes a cada elemento da placa. Dependendo da localização de cada nó, se ele se encontra numa das bordas encastradas ou nas zonas livres, os valores correspondentes a w, θ_x e θ_y são anulados nas bordas encastradas ou mantêm-se diferentes de zero nas regiões livres.

Além disso, é necessário determinar o vetor de forças nodais $\{Q\}$. Este vetor é calculado com base na sobreposição dos nós, ou seja, dependendo se um nó se encontra isolado, partilhado por dois elementos ou por quatro elementos. Quando um nó é único, por exemplo num canto da placa, aplica-se a força P_1 segundo

$$P_1 = \frac{p_0\, a\, b}{4\, n_{\text{elementos}}}. \quad (2.17)$$

Quando um nó é comum a dois elementos, por exemplo numa borda da placa, a força aplicada é P_2:

$$P_2 = \frac{p_0\, a\, b}{2\, n_{\text{elementos}}}. \quad (2.18)$$

Quando um nó é compartilhado por quatro elementos, por exemplo no centro da placa, aplica-se a força P_3:

$$P_3 = \frac{p_0\, a\, b}{n_{\text{elementos}}}. \quad (2.19)$$

Depois de gerado o vetor de corte, aplicam-se as condições de fronteira ao vetor $\{Q\}$. Desta forma, este vetor passa a possuir o mesmo número de linhas que a matriz quadrada $[K]_{\text{global}}$ reduzida, permitindo efetuar corretamente a multiplicação matricial.

Em seguida, inverte-se a matriz $[K]_{\text{global}}$ reduzida e realiza-se a operação definida na equação (2.16): obtendo-se assim o vetor de deslocamentos $\{\delta\}$, que representa as deflexões em cada nó da placa.

Por fim, para determinar as reações nos apoios, aplica-se a expressão derivada da equação (2.20):

$$\{R\} = [K]_{\text{global}} \cdot \{\delta\} - \{Q\}, \quad (2.20)$$

onde $\{R\}$ representa o vetor das reações de apoio.

Capítulo 3

Implementação do código

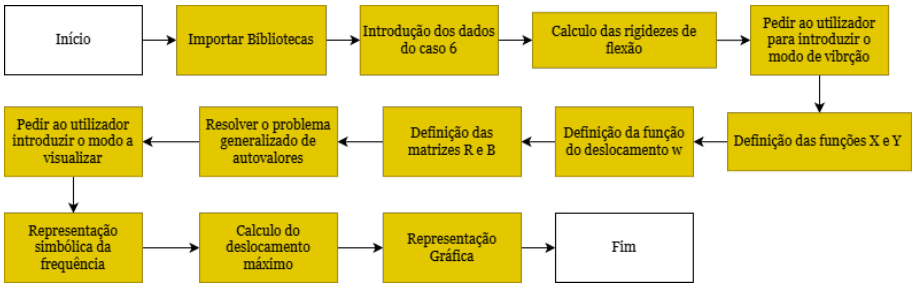


Figura 3.1: Fluxograma referente á Tarefa 1

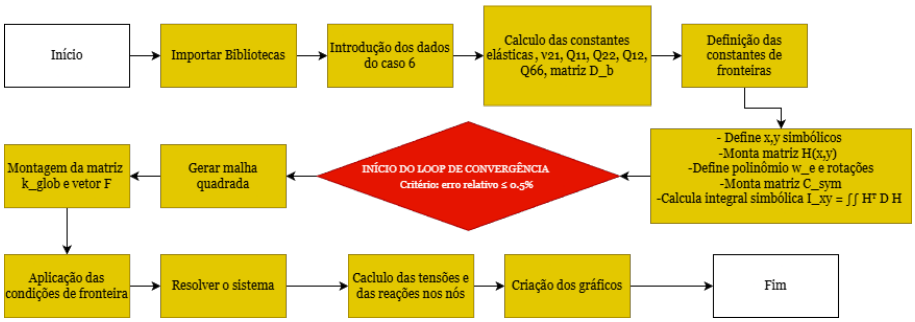


Figura 3.2: Fluxograma referente à Tarefa 2

Na tarefa 1, para resolver computacionalmente o problema proposto, foi utilizada a linguagem de programação *Python*. Inicialmente, são definidas as variáveis simbólicas necessárias para o desenvolvimento da formulação, nomeadamente x, y, a, b, c, D, ν, ω, ρ e t. Posteriormente, é efetuado o cálculo das funções $X_i(x)$ e $Y_j(y)$ de acordo com as expressões indicadas no formulário. Com estas funções, é então possível definir $w(x, y)$, que representa o modo de vibração da placa. Ao derivar e integrar a equação correspondente e reorganizá-la em função de ω, obtém-se a frequência de vibração de forma simbólica. Finalmente, é feita a representação gráfica da função $w(x, y)$, permitindo assim visualizar o comportamento da placa para o modo de vibração pretendido.

A resolução da Tarefa 2 foi igualmente desenvolvida em *Python*. Começa-se por declarar as variáveis simbólicas e os parâmetros indicados no enunciado. De seguida, é feita a leitura da matriz [D], já deduzida na alínea anterior, bem como a definição da matriz [H], de acordo com a equção (2.13), e a construção da matriz [C], conforme a equação (2.15). A matriz de rigidez elementar $[K]_e$ pode ser determinada diretamente, segundo a equação (2.12). O utilizador introduz o número de subdivisões da placa, o que permite gerar uma

matriz [K]$_e$ para cada elemento. Como estas matrizes são inicialmente simbólicas, é aplicado um algoritmo que substitui as variáveis por valores reais, diferentes para cada elemento consoante a sua posição na malha. Esse algoritmo percorre todos os elementos, atribuindo as coordenadas dos nós de acordo com a distância dx (na direção x) e dy (na direção y), obtidas pela divisão das dimensões totais da placa pelo número de elementos por lado.

Com as matrizes [K]$_e$ devidamente atualizadas, procede-se ao seu espalhamento para formar a matriz global [K]$_{glob}$, utilizando uma matriz de conectividade.

Segue-se a aplicação das condições de fronteira:

- Borda 1 ($x = 0$): encastrada;
- Borda 2 ($x = a$): encastrada;
- Borda 3 ($y = 0$): livre;
- Borda 4 ($y = b$): livre.

Estas condições são impostas por meio de um vetor de controlo que identifica os nós localizados nas fronteiras e aplica as restrições correspondentes, eliminando as linhas e colunas associadas na matriz global [K] e no vetor de forças Q.

O vetor Q é construído com base na posição de cada nó da malha. Caso o nó pertença a um canto da placa, aplica-se a força definida pela equação (2.17); se se encontrar numa aresta, utiliza-se a equação (2.18); e, para os nós interiores, aplica-se a força de acordo com a equação (2.19). Após a aplicação das condições de fronteira, o vetor Q é igualmente reduzido, removendo-se as componentes correspondentes aos graus de liberdade restringidos.

Com a matriz $[K]_{glob}$ e o vetor Q devidamente ajustados, resolve-se o sistema linear invertendo a matriz reduzida e multiplicando-a pelo vetor de forças reduzido, obtendo-se o vetor δ das deflexões nodais. Posteriormente, representam-se graficamente as deflexões na placa, permitindo observar a deformação resultante.

Por fim, calculam-se as reações nos apoios através da aplicação da equação (2.20). Os resultados são apresentados graficamente, possibilitando uma visualização clara da distribuição das reações na estrutura e do comportamento global da placa sob as condições de contorno definidas.

Capítulo 4

Resultados e Discussões

Neste capítulo apresentam-se e analisam-se os resultados obtidos nas Tarefas 1 e 2. Os valores calculados e as representações gráficas permite avaliar a precisão das abordagens adotadas e compreender o comportamento estrutural da placa ortotrópica sob análise.

4.1 Resultados da Tarefa 1

Na Tarefa 1, foi determinada a frequência fundamental de vibração ω e o modo de vibração correspondente de uma placa ortotrópica, com base na formulação obtida pela aplicação do método de Ritz (Equações (2.7) e (2.11)).

A equação característica resultante da anulação do termo de variação forneceu os valores próprios do sistema, sendo o menor deles o que corresponde à frequência fundamental de vibração, dada pela equação (4.1).

$$\omega = 2\sqrt{6} \cdot \sqrt{\frac{21D_{11}b^4 + 10D_{12}a^2b^2 + 5D_{22}a^4 + 20D_{66}a^2b^2}{a^4b^4\rho t}} \quad (4.1)$$

A forma associada a esta frequência foi representada graficamente através da função $w(x, y)$, obtida para valores unitários de a, b e c_{ij}. A Figura 4.1 ilustra o modo fundamental de vibração calculado.

Observa-se que as zonas de maior deflexão ocorrem no centro da placa, enquanto as

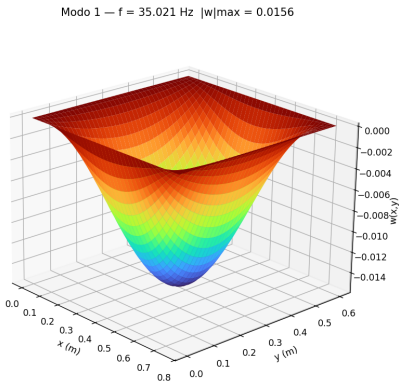


Figura 4.1: Primeiro Modo de Vibração

regiões de fronteira apresentam deslocamentos nulos, conforme as condições de fronteira impostas.

Os resultados obtidos mostram uma boa coerência com a teoria de placas finas ortotrópicas, onde o aumento do módulo de elasticidade na direção principal x (E_1) conduz a uma rigidez superior e, consequentemente, a uma frequência natural mais elevada, tal como previsto pelas relações constitutivas da Equação (2.1).

4.2 Resultados da Tarefa 2

4.2.1 Alinea *a)*

A matriz de elasticidade para o material ortotrópico considerado pode ser escrita como (4.2) :

$$[D] = \frac{E_1\, t^3}{12\,(1-\nu_{12}\nu_{21})} \begin{bmatrix} 1 & \nu_{21} & 0 \\ \nu_{12} & \dfrac{E_2}{E_1} & 0 \\ 0 & 0 & \dfrac{G_{12}\,(1-\nu_{12}\nu_{21})}{E_1} \end{bmatrix} \tag{4.2}$$

Em materiais ortotrópicos, a forma da matriz de elasticidade não coincide com a de materiais isotrópicos, pois depende das direções preferenciais do material. As propriedades E_1, E_2, ν_{12}, ν_{21} e G_{12} traduzem o comportamento elástico ao longo dos eixos principais. Além disso, existe a relação entre os módulos nas duas direções através dos coeficientes de Poisson, podendo escrever-se $E_2 = \dfrac{\nu_{12}}{\nu_{21}} E_1$ quando se considera compatibilidade entre as constantes do material. Esta representação evidencia como a rigidez flexional da placa é afetada pela anisotropia, ficando a resposta dependente da orientação das fibras e das propriedades específicas em cada direção.

4.2.2 Alinea *b)*

Após a aplicação do método descrito no capítulo anterior e a implementação do respetivo código, obtiveram-se os resultados apresentados na Figura 4.2, correspondentes às deflexões da placa para o Caso 6 considerado, utilizando uma malha composta por 411 elementos.

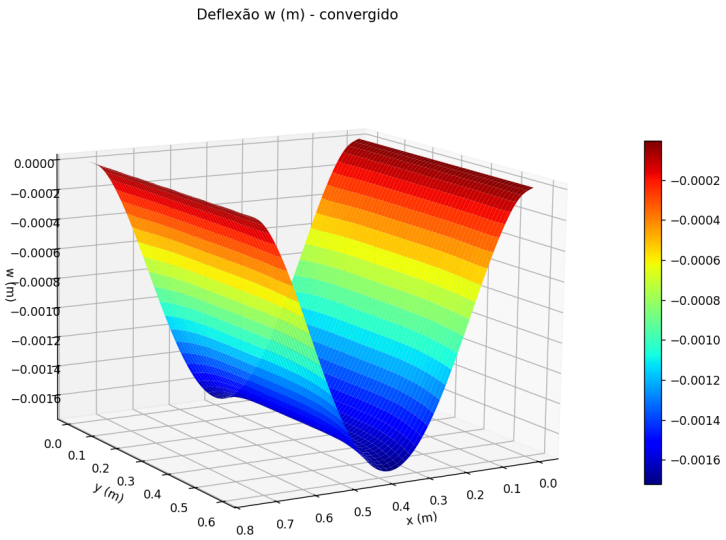
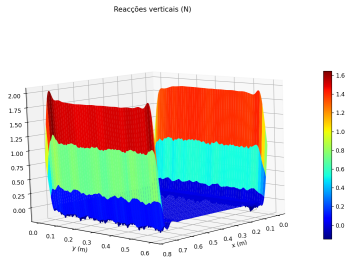


Figura 4.2: Deflexões na placa considerando uma discretização em 411 elementos.

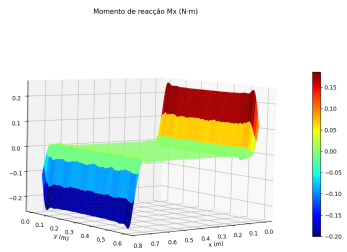
Observa-se que, na região encastrada, as deflexões são nulas, conforme imposto pelas condições de fronteira. À medida que se progride ao longo da placa em direção à extremidade livre, o deslocamento vertical aumenta gradualmente, atingindo o valor máximo de aproximadamente 1.72×10^{-3} m. Este comportamento é típico de uma placa sujeita a carregamento distribuído, onde a flexão é máxima na extremidade livre e nula no encastramento.

A malha de 21×21 elementos revelou-se adequada, assegurando um equilíbrio entre precisão e custo computacional. O refinamento adicional da discretização mostrou variações residuais nos resultados, o que confirma a convergência numérica do método.

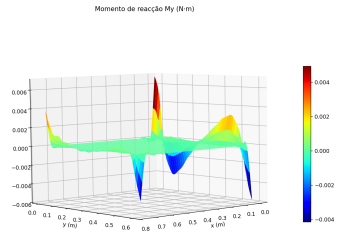
As reações verticais e os momentos concentraram-se na zona encastrada, tal como previsto teoricamente, uma vez que é nessa região que a estrutura resiste integralmente ao carregamento aplicado. As Figuras 4.3a, 4.3b e 4.3c apresentam a distribuição das reações e momentos nos apoios.



(a) Distribuição das reações verticais R_y junto ao encastramento.



(b) Distribuição do momento fletor M_x junto ao encastramento.



(c) Distribuição do momento fletor M_y junto ao encastramento.

Figura 4.3: Distribuição das reações e momentos na zona de encastramento.

Além das deflexões e das reações, foram também analisadas as tensões normais e de corte na superfície superior da placa ($z = +t/2$). As Figuras 4.4, 4.5 e 4.6 apresentam, respetivamente, as distribuições das tensões σ_x, σ_y e τ_{xy}.

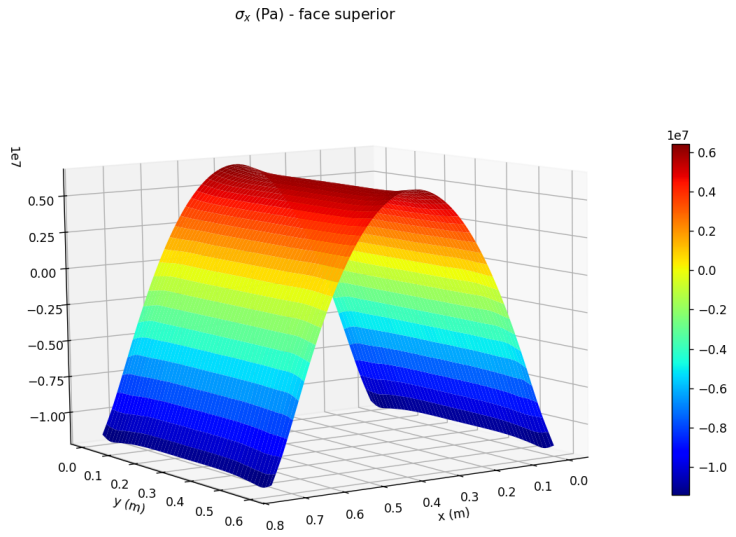


Figura 4.4: Distribuição de tensões σ_x na superfície superior.

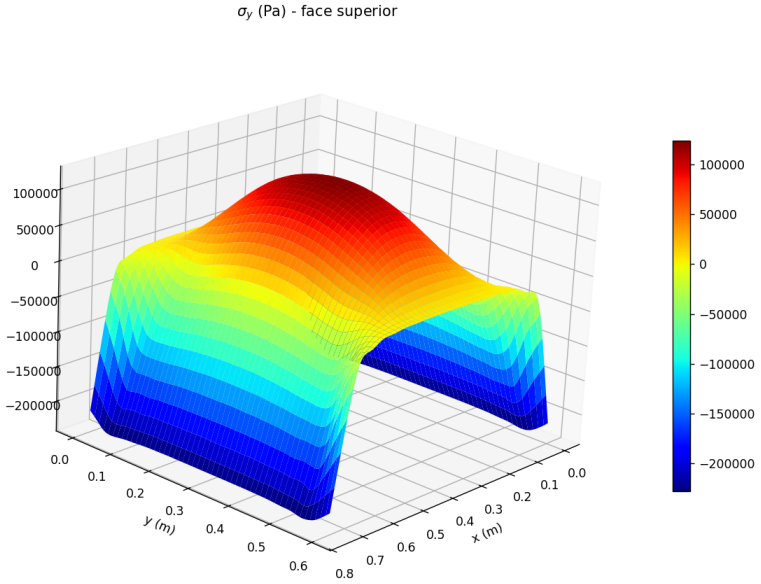


Figura 4.5: Distribuição de tensões σ_y na superfície superior.

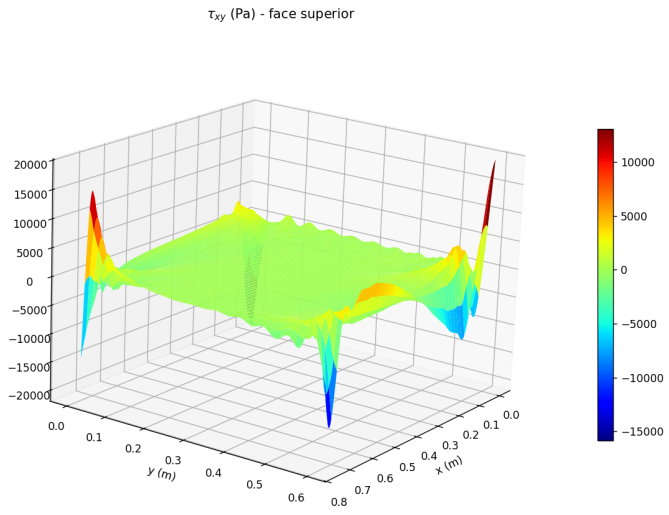


Figura 4.6: Distribuição de tensões de corte τ_{xy} na superfície superior.

Os resultados mostram que:

- A tensão normal σ_x apresenta valores entre 6.42×10^6 N/m^2 e -1.18×10^7 N/m^2, com máximos localizados junto ao encastramento, onde ocorrem os maiores momentos fletores;
- A tensão σ_y varia entre 1.23×10^5 N/m^2 e -2.36×10^5 N/m^2, com valores significativamente inferiores, refletindo a menor influência da flexão nesta direção;
- A tensão de corte τ_{xy} situa-se entre 1.94×10^4 N/m^2 e -2.11×10^4 N/m^2, atingindo os seus valores máximos próximos do encastramento, onde o gradiente de flexão é mais acentuado;
- Reação vertical R_y: entre 8.43×10^{-1} N e 1.99 N;
- Momento em torno do eixo x, M_x: entre -2.42×10^{-1} N·m e 2.41×10^{-1} N·m;
- Momento em torno do eixo y, M_y: entre -5.80×10^{-3} N·m e 6.90×10^{-3} N·m.

Estes resultados confirmam a correta modelação das condições de fronteira, evidenciando picos de reação e momento junto à linha de encastramento, onde ocorre a total absorção dos esforços aplicados.

4.2.3 Alinea *c)*, caso (i)

Quando os eixos principais do material não coincidem com os eixos da placa, é necessário aplicar uma transformação de coordenadas. Neste cenário, as propriedades elásticas como E_1, E_2, v_{12}, v_{21} e G_{12} deixam de ser diretamente associadas aos eixos x e y, exigindo a utilização de uma matriz de rotação que ajusta a orientação das fibras ao novo sistema de

referência. A matriz de rigidez transformada passa então a representar corretamente o comportamento estrutural da placa.

4.2.4 Alinea *c)*, caso (ii)

A simetria relativamente ao plano médio simplifica parte dos cálculos, já que camadas posicionadas acima e abaixo dessa linha apresentam características semelhantes, mas com direções de fibras opostas. Esta configuração contribui para reduzir a complexidade e tornar o comportamento do compósito mais previsível.

No entanto, a ausência de balanceamento implica que as orientações das fibras não estão distribuídas de forma uniforme em torno do centro da estrutura. Esse desequilíbrio pode provocar efeitos indesejados, como torções adicionais, que alteram a distribuição das deflexões na placa.

Por fim, o aumento do número de camadas intensifica a dificuldade da análise, uma vez que cada camada deve ser considerada individualmente na construção da matriz de rigidez. É ainda necessário transformar as propriedades de cada camada para o sistema de coordenadas local, assegurando que o comportamento global da placa seja corretamente descrito.

Capítulo 5

Conclusão

Neste relatório apresenta um estudo numérico e computacional sobre o comportamento vibracional e estrutural de placas ortotrópicas de Kirchhoff, considerando distintas condições de fronteira e propriedades materiais. Através da aplicação de métodos computacionais de solução numérica, foi possível determinar as frequências naturais e os modos de vibração principais, verificando-se a coerência com o comportamento esperado nas zonas de encastramento e de apoio simples.

Os resultados mostraram que, ao dividir a placa em 441 elementos, se alcança um nível de precisão elevado, obtendo-se uma deflexão máxima próxima de 1.72 mm (com um erro relativo de 0.5% .

Foram igualmente analisados casos mais avançados, envolvendo materiais compósitos laminados com configurações simétricas e não balanceadas. Constatou-se que a simetria entre as camadas simplifica significativamente o processo de cálculo, enquanto o desequilíbrio entre elas pode originar efeitos indesejáveis, como a torção. Estes aspetos evidenciam a necessidade de considerar cuidadosamente a disposição e as propriedades de cada camada para uma representação rigorosa do comportamento estrutural global.

Em síntese, o trabalho desenvolvido permitiu aprofundar a compreensão dos fatores que afetam a resposta dinâmica e estrutural das placas de Kirchhoff. Ficou demonstrada a importância das características geométrica e da metodologia numérica utilizada, sendo os resultados obtidos consistentes com as previsões teóricas e validando, assim, o modelo proposto.

Bibliografia

Gamboa. P.V., Apontamentos da unidade curricular – Placas e Cascas, 320 acetatos, UBI, 2020.

Gamboa, P. V., Apontamentos da Unidade Curricular – Estruturas em Materiais Compósitos, 320 acetatos, Universidade da Beira Interior, Departamento de Ciências Aeroespaciais, 2025.

Dutra, T., Apontamentos da Unidade Curricular – Placas e Cascas, 250 acetatos, Universidade da Beira Interior, Departamento de Ciências Aeroespaciais, 2025.

Apêndice A

Implementação do código

Tarefa 1

```
#Introduçao de bibliotecas
#Introdução dos parâmetros geométricos
#Calculo das rigidezes de flexão
#Pedir o numero de modos
#Definição da função X e Y
#Definiçao da funçao do deslocamento w
#Definição das matrizes R e B
#Resolver o problema generalizado de autovalores
#Pedir ao utilzador o modo a visualizar
#Reconstruir a funçao w(x,y)
#Reprensentação simbólica da frequência omega
#Cálculo do deslocamento máximo |w|max
#Representação gráfica do modo de vibração

import sympy as sp                  # Biblioteca simbólica (derivadas e
    integrais)
import numpy as np                  # Biblioteca numérica (matrizes, vetores)
from scipy.linalg import eigh       # Resolver problemas generalizados de
    autovalores
import matplotlib.pyplot as plt     # Biblioteca para gráficos 3D interativos


a = 0.750       # [m]
b = 0.600       # [m]
t = 0.002       # [m]
rho = 1600      # [kg/m³]
E1 = 130e9      # [Pa]
E2 = 10e9       # [Pa]
nu12 = 0.26
G12 = 5e9       # [Pa]

nu21 = nu12 * E2 / E1

Q11 = E1 / (1 - nu12 * nu21)
Q22 = E2 / (1 - nu12 * nu21)
Q12 = nu12 * E2 / (1 - nu12 * nu21)
```

```
Q66 = G12

D11 = Q11 * t**3 / 12
D22 = Q22 * t**3 / 12
D12 = Q12 * t**3 / 12
D66 = Q66 * t**3 / 12

try:
    n_modos = int(input("Introduz o número de modos de vibração: "))
except ValueError:
    n_modos = 2

if n_modos < 1:
    n_modos = 2

M = n_modos
N = n_modos
n = M * N   # número total de graus de liberdade
# ============================
x, y = sp.symbols('x y', real=True)
X = []
Y = []

for i in range(1, M + 1):
    X_i = (x / a)**(i + 1) - 2 * (x / a)**(i + 2) + (x / a)**(i + 3)
    X.append(X_i)

for j in range(1, N + 1):
    Y_j = (y / b)**j - (y / b)**(j + 1)
    Y.append(Y_j)

c = sp.symbols(f'c0:{n}', real=True)
w = 0
k = 0

for i in range(M):
    for j in range(N):
        w += c[k] * X[i] * Y[j]
        k += 1

R = np.zeros((n, n), dtype=float)
B = np.zeros((n, n), dtype=float)

for p in range(n):
    wp = sp.diff(w, c[p])
    wp_xx = sp.diff(wp, x, 2)
    wp_yy = sp.diff(wp, y, 2)
```

```
82      wp_xy = sp.diff(sp.diff(wp, x), y)
83
84      for q in range(n):
85          wq = sp.diff(w, c[q])
86          wq_xx = sp.diff(wq, x, 2)
87          wq_yy = sp.diff(wq, y, 2)
88          wq_xy = sp.diff(sp.diff(wq, x), y)
89
90          expr_R = (D11 * wp_xx * wq_xx +
91                    2 * D12 * wp_xx * wq_yy +
92                    D22 * wp_yy * wq_yy +
93                    4 * D66 * wp_xy * wq_xy)
94
95          expr_B = rho * t * wp * wq
96
97          R[p, q] = float(sp.integrate(sp.integrate(expr_R, (x, 0, a)), (y, 0, b)
                ))
98          B[p, q] = float(sp.integrate(sp.integrate(expr_B, (x, 0, a)), (y, 0, b)
                ))
99
100
101 omega2, vec = eigh(R, B)
102 omega2 = np.real(omega2[omega2 > 1e-8])
103 omega = np.sqrt(omega2)
104 freq = omega / (2 * np.pi)
105
106 print("\n=== FREQUÊNCIAS NATURAIS ===")
107 for i, f in enumerate(freq, start=1):
108     print(f"Modo {i}: f = {f:.3f} Hz")
109
110 try:
111     modo = int(input("\nIntroduz o número do modo a visualizar (ex: 1): "))
112 except ValueError:
113     modo = 1
114
115 if modo < 1 or modo > len(freq):
116     modo = 1
117
118 phi = vec[:, modo - 1]
119 phi = phi / np.max(np.abs(phi))
120
121
122 Nx, Ny = 80, 80
123 x_vals = np.linspace(0, a, Nx)
124 y_vals = np.linspace(0, b, Ny)
125 Xgrid, Ygrid = np.meshgrid(x_vals, y_vals)
126 W = np.zeros_like(Xgrid, dtype=float)
```

```
127
128 k = 0
129 for i in range(M):
130     for j in range(N):
131         X_fun = sp.lambdify(x, X[i], "numpy")
132         Y_fun = sp.lambdify(y, Y[j], "numpy")
133         W += phi[k] * X_fun(Xgrid) * Y_fun(Ygrid)
134         k += 1
135
136
137 x, y, c1, a_s, b_s, t_s, rho_s, D11_s, D22_s, D12_s, D66_s = sp.symbols(
138     'x y c1 a b t rho D11 D22 D12 D66', real=True)
139
140 X1 = (x / a_s)**2 - 2 * (x / a_s)**3 + (x / a_s)**4
141 Y1 = (y / b_s) - (y / b_s)**2
142 w_sym = c1 * X1 * Y1
143
144 w_xx = sp.diff(w_sym, x, 2)
145 w_yy = sp.diff(w_sym, y, 2)
146 w_xy = sp.diff(sp.diff(w_sym, x), y)
147
148 U = sp.integrate(sp.integrate(D11_s*w_xx**2 + 2*D12_s*w_xx*w_yy +
149                               D22_s*w_yy**2 + 4*D66_s*w_xy**2, (y, 0, b_s)), (x
                                  , 0, a_s))
150 T = sp.integrate(sp.integrate(rho_s * t_s * w_sym**2, (y, 0, b_s)), (x, 0, a_s)
    )
151
152 omega_symbolic = sp.simplify(sp.sqrt(U / T))
153
154 print("\n=== EXPRESSÃO SIMBÓLICA DA FREQUÊNCIA ω ===")
155 sp.pretty_print(omega_symbolic)
156 print("\n=== EXPRESSÃO EM LATEX ===")
157 print(sp.latex(omega_symbolic))
158
159 w_max = np.max(np.abs(W))
160 print(f"\nValor máximo do deslocamento relativo |w|max = {w_max:.4f}")
161
162
163
164 fig = plt.figure("Modo de vibração", figsize=(8, 6))
165 ax = fig.add_subplot(111, projection='3d')
166
167
168 ax.plot_surface(Xgrid, Ygrid, -W, cmap='turbo', edgecolor='none')
169
170 ax.set_xlabel('x (m)', color='white')
171 ax.set_ylabel('y (m)', color='white')
```

```
ax.set_zlabel('w(x,y)', color='white')
ax.set_title(f"Modo {modo} — f = {freq[modo-1]:.3f} Hz  |w|max = {w_max:.4f}",
             color='white')

ax.set_facecolor('black')
fig.patch.set_facecolor('black')
ax.xaxis.label.set_color('white')
ax.yaxis.label.set_color('white')
ax.zaxis.label.set_color('white')
ax.tick_params(colors='white')
plt.show()
```

Listagem A.1: Código em Python para o caso 6

A.1 Tarefa 2

```
#Importar bibliotecas
#Definir as porpriedades do material
#Definir a matriz D para o caso ortotrópico
#Definir o tipo de borda em cada lado
#Definição simbolica das variaveis
    #Definir a matriz H simbolica e inicializaçao em 0
    #Definir o polinomio w_e (a1...a12)
    #Definir as rotações theta_x e theta_y (derivadas)
    #construir a matriz simbolica C e inicializaçao em 0
    #Preencher a matriz C simbolica com ([1,4,7,10] -> w; [2,5,8,11] -> theta_x
        ; [3,6,9,12] -> theta_y)
    #Integrar simbolicamente I_xy = ∫∫ H' * D * H dx dy
#Iniciar o metodo de convergencia
    #Definir o erro relativo de convergencia
    #Iniciar a malha com 1 quadrado por lado
    #iniciar o loop de convergencia
        #Incrementar o numero de quadrados por lado
        #Gerar a malha de nos (cada no tem 3 graus de liberdade) e a
            conectividade
        #Montar a matriz K e o vetor F e inicializa-los em 0
        # Loop pelos elementos para calcular Ke e Fe e espalhar para K, F
        # Obter coordenadas do elemento
        # Avaliar integral simbolica I_xy para o elemento
        # Construir c_num
        # Substituir em cada no e preencher as linhas correspondentes
        # Verificar o condicionamento de C_num e calcular a inversa
        # Calcular a matriz elementar Ke e o vetor de cargas elementar Fe
        #Aplicar as condições de fronteira
```

```
27          #Definir a funçao para aplicar as condiçoes de fronteira (seja que tipo
                de fronteira for)
28          #Resolver o sistema de equações lineares
29          #Montar o vetor dos deslocamentos nodais e inicializa em 0
30          #Calcular os deslocamentos nodais w
31          #Calcular o erro relativo face ao passo anterior (verificar a
                convergencia)
32  #Usar a solução armnazenada
33  #Calcular as tensões nodais
34  #Resultados numéricos e impressões finais
35  #Represenentação gráfica dos resultados
36
37
38  import warnings                                     # para emitir avisos
        controlados
39  import numpy as np                                  # operações numéricas
40  import sympy as sp                                  # cálculo simbólico
41  import scipy.sparse as sp_sparse                    # matrizes esparsas
42  import scipy.sparse.linalg as spla                  # resolução de sistemas
        esparsos
43  from scipy.interpolate import griddata              # interpolação para
        grelha (griddata)
44  import matplotlib.pyplot as plt                     # visualização / gráficos
45  # ----------------------------
46  a = 0.750           # [m]
47  b = 0.600           # [m]
48  t = 0.002           # [m]
49  E1 = 130e9          # [Pa]
50  E2 = 10e9           # [Pa]
51  nu12 = 0.26
52  G12 = 5e9           # [Pa]
53  p0 = -175.0         # [N/m^2] (negativa = para baixo)
54  # ----------------------------
55  nu21 = (E2 / E1) * nu12
56  Q11 = E1 / (1 - nu12 * nu21)
57  Q22 = E2 / (1 - nu12 * nu21)
58  Q12 = nu21 * Q11
59  Q66 = G12
60  Q_mat = np.array([[Q11, Q12, 0.0],
61                    [Q12, Q22, 0.0],
62                    [0.0, 0.0, Q66]])
63  D_b = (t**3) / 12.0 * Q_mat
64  print("\nMatriz D_b (flexão) [Pa·m³]:")
65  print(D_b)
66  # ----------------------------
67  B1 = 'C'    # borda x=0. encastrada
68  B2 = 'C'    # borda x=a. encastrada
```

```
69 B3 = 'F'   # borda y=0. livre
70 B4 = 'F'   # borda y=b. livre
71 # ---------------------------
72 x, y, x1, x2, y1, y2 = sp.symbols('x y x1 x2 y1 y2', real=True)
73 D_sym = sp.Matrix(D_b)
74 # ---------------------------
75 H = sp.Matrix(sp.zeros(3, 12))
76 H[0, :] = sp.Matrix([0, 0, 0, -2, 0, 0, -6*x, -2*y, 0, 0, -6*x*y, 0]).T
77 H[1, :] = sp.Matrix([0, 0, 0, 0, 0, -2, 0, 0, -2*x, -6*y, 0, -6*x*y]).T
78 H[2, :] = sp.Matrix([0, 0, 0, 0, -2, 0, 0, -4*x, -4*y, 0, -6*x*2, -6*y*2]).T
79 #---------------------------
80 a_syms = sp.symbols('a1:13', real=True)
81 w_e = (
82     a_syms[0]
83     + a_syms[1]*x
84     + a_syms[2]*y
85     + a_syms[3]*x**2
86     + a_syms[4]*x*y
87     + a_syms[5]*y**2
88     + a_syms[6]*x**3
89     + a_syms[7]*x**2*y
90     + a_syms[8]*x*y**2
91     + a_syms[9]*y**3
92     + a_syms[10]*x**3*y
93     + a_syms[11]*x*y**3
94 )
95 #---------------------------
96 theta_x = sp.diff(w_e, x)                                    # theta_x = ∂w
       ∂/x
97 theta_y = sp.diff(w_e, y)                                    # theta_y = ∂w
       ∂/y
98 #---------------------------
99 C_sym = sp.Matrix(sp.zeros(12, 12))
100 #---------------------------
101 for row in range(12):
102     for col in range(12):
103         if row in [0, 3, 6, 9]:
104             C_sym[row, col] = sp.diff(w_e, a_syms[col])
105         elif row in [1, 4, 7, 10]:
106             C_sym[row, col] = sp.diff(theta_x, a_syms[col])
107         else:
108             C_sym[row, col] = sp.diff(theta_y, a_syms[col])
109 #---------------------------
110 print('A integrar simbolicamente H\'*D*H (apenas uma vez) ...')
111 Integrand = H.T * D_sym * H
112 I_xy = sp.integrate(sp.integrate(Integrand, (x, x1, x2)), (y, y1, y2))
```

```
113 I_xy = sp.simplify(I_xy)                                    # simplificar
    expressão simbólica resultante
114 # ----------------------------
115 tol_rel_pct = 0.5                                           # tolerância
    relativa em percentagem (0.5%)
116 w_prev = np.nan                                            # valor
    anterior w_max (para comparar)
117 converged = False                                          # flag de
    convergência
118 n_side = 1                                                 # inicia com 1
    quadrado por lado (malha 1x1)
119 print(f'\nIniciando sequência de refinamento automática (critério: erro <= {
    tol_rel_pct:.2f}%)\n')
120 while not converged:
121     n_side += 1                                           # incrementa
        número de quadrados por lado
122     n_quadrados = n_side * n_side                         # número total
        de quadrados
123     nnx = n_side + 1                                      # nós em x
124     nny = n_side + 1                                      # nós em y
125     n_nos = nnx * nny                                     # número total
        de nós
126     dx = a / n_side                                       # Incremento em
        x
127     dy = b / n_side                                       # Incremento em
        y
128
129     coords = np.zeros((n_nos, 2))                         # inicializa
        array de coordenadas
130     idx = 0                                               # contador de
        nós
131     for j in range(0, n_side + 1):                        # varre linhas y
132         for i in range(0, n_side + 1):                    # varre colunas
            x
133             coords[idx, :] = [i * dx, j * dy]             # posição do nó
                (x, y)
134             idx += 1                                      # incremento do
                índice
135
136     conn = np.zeros((n_quadrados, 4), dtype=int)          # inicializa
        conectividade
137     ecount = 0                                            # contador de
        elementos
138     for j in range(1, n_side + 1):                        # varre
        elementos por linha
139         for i in range(1, n_side + 1):                    # varre
            elementos por coluna
```

```
140             ecount += 1                                     # índice do
                    elemento
141             n1 = (j - 1) * (n_side + 1) + i                 # índice do nó
                    inferior-esquerdo (BL)
142             conn[ecount - 1, :] = [n1, n1 + 1, n1 + n_side + 2, n1 + n_side +
                    1]  # BL BR TR TL
143     #---------------------------
144     dofs = 3 * n_nos                                        # 3 graus de
            liberdade por nó (w, theta_x, theta_y)
145     K = sp_sparse.lil_matrix((dofs, dofs), dtype=float)
146     F = np.zeros((dofs,), dtype=float)                      # inicializa
            vetor de forças globais
147     #---------------------------
148     for e in range(n_quadrados):
149         nodes = conn[e, :]                                  # lista de 4 nós
                do elemento (ainda 1-based)
150         # obter coordenadas do elemento para x e y dos 4 nós (presume-se ordem
                BL BR TR TL)
151         Xe = coords[nodes - 1, 0]                           # x coords dos
                nós do elemento
152         Ye = coords[nodes - 1, 1]                           # y coords dos
                nós do elemento
153         x1v = Xe[0]                                         # x1 do elemento
                (BL.x)
154         x2v = Xe[1]                                         # x2 do elemento
                (BR.x)
155         y1v = Ye[0]                                         # y1 do elemento
                (BL.y)
156         y2v = Ye[3]                                         # y2 do elemento
                (TL.y)
157
158         I_num_mat = np.array(sp.N(I_xy.subs({x1: x1v, x2: x2v, y1: y1v, y2: y2v
                })), dtype=float)
159         C_num = np.zeros((12, 12), dtype=float)             # inicializa
                C_num
160         C_sub = C_sym.subs({x: x1v, y: y1v})                # valor simbólico
                em (x1,y1)  --> BL
161         C_num[0:3, :] = np.array(C_sub[0:3, :].evalf(), dtype=float)
162         C_sub = C_sym.subs({x: x2v, y: y1v})                # valor simbólico
                em (x2,y1)  --> BR
163         C_num[3:6, :] = np.array(C_sub[3:6, :].evalf(), dtype=float)
164         C_sub = C_sym.subs({x: x2v, y: y2v})                # valor simbólico
                em (x2,y2)  --> TR
165         C_num[6:9, :] = np.array(C_sub[6:9, :].evalf(), dtype=float)
166         C_sub = C_sym.subs({x: x1v, y: y2v})                # valor simbólico
                em (x1,y2)  --> TL
167         C_num[9:12, :] = np.array(C_sub[9:12, :].evalf(), dtype=float)
```

```
168
169         try:
170             cond_C = np.linalg.cond(C_num)                    # número de
                    condição
171         except np.linalg.LinAlgError:
172             cond_C = np.inf
173         if cond_C > 1e12:
174             warnings.warn(f'C_num mal condicionado no elemento {e+1} (cond={
                    cond_C:.3e}).')
175         try:
176             Cinv = np.linalg.inv(C_num)                       # inversa de
                    C_num
177         except np.linalg.LinAlgError:
178             warnings.warn(f'Erro a inverter C_num no elemento {e+1}; será usado
                     pseudo-inversa.')
179             Cinv = np.linalg.pinv(C_num)                      #matriz inversa
                    generalizada
180
181         Ke = Cinv.T.dot(I_num_mat).dot(Cinv)                  # montagem de Ke
182         Ae = (x2v - x1v) * (y2v - y1v)                        # área do elemento
183         Fe = np.zeros((12,), dtype=float)                     # vetor de cargas
                elementar
184         for ni in range(4):
185             Fe[ni*3 + 0] = p0 * Ae / 4.0                      # aplicar carga p0
                     distribuída igualmente em w-dof
186
187         # espalhar Ke e Fe para K e F globais (localizar os graus de liberdade
                do elemento)
188         dofs_e = np.zeros((12,), dtype=int)                   # índices globais
                dos graus de liberade do elemento
189         for iN in range(4):
190
191             node_idx = nodes[iN] - 1                          # converter 1-
                    based -> 0-based
192             dofs_e[iN*3 + 0] = node_idx * 3 + 0               # w DOF
193             dofs_e[iN*3 + 1] = node_idx * 3 + 1               # theta_x DOF
194             dofs_e[iN*3 + 2] = node_idx * 3 + 2               # theta_y DOF
195
196         for r in range(12):
197             for c_idx in range(12):
198                 K[dofs_e[r], dofs_e[c_idx]] += Ke[r, c_idx]
199         for r in range(12):
200             F[dofs_e[r]] += Fe[r]
201
202     def apply_borda_fun(bord, coords_local, n_nos_local, dofs_local, tol_local,
            fixed_in):
```

```
203         fixed_out = fixed_in.copy()                                 # copia do vetor
                de fixos
204         typ = bord['type']                                          # tipo de borda
205         coord = bord['coord']                                       # coordenada
                associada ('x' ou 'y')
206         val = bord['value']                                         # valor do borda
                (o ou a ou b)
207         for ni in range(n_nos_local):                               # corre os nós
208             x_i = coords_local[ni, 0]                               # coordenada x do
                    nó
209             y_i = coords_local[ni, 1]                               # coordenada y do
                    nó
210             on_borda = False                                        # flag se nó está
                    na borda
211             if coord == 'x':                                        # se bordo
                    vertical (x fixo)
212                 if abs(x_i - val) < tol_local:
213                     on_borda = True
214             else:                                                   # bordo horizontal
                     (y fixo)
215                 if abs(y_i - val) < tol_local:
216                     on_borda = True
217             if on_borda:
218                 idx_base = ni * 3                                   # base dos graus
                        de liberdade do nó no vetor global
219                 if typ == 'C':                                      # encastrado: w,
                        theta_x, theta_y fixos
220                     fixed_out[idx_base + 0] = True                  # fixar w
221                     fixed_out[idx_base + 1] = True                  # fixar theta_x
222                     fixed_out[idx_base + 2] = True                  # fixar theta_y
223                 elif typ == 'S':                                    # simplesmente
                        apoiado
224                     if coord == 'y':                                # borda horizontal
                            -> w=0 e theta_x=0
225                         fixed_out[idx_base + 0] = True              # w
226                         fixed_out[idx_base + 1] = True              # theta_x
227                     else:                                           # borda vertical
                            -> w=0 e theta_y=0
228                         fixed_out[idx_base + 0] = True              # w
229                         fixed_out[idx_base + 2] = True              # theta_y
230                 elif typ == 'F':                                    # livre -> nada a
                        fixar
231                     pass
232                 else:
233                     raise ValueError(f"Tipo de borda desconhecido: {typ} (
                            esperado C, S ou F)")
234         return fixed_out
```

```
235
236     borda_map = {
237         'B1': {'type': B1, 'coord': 'x', 'value': 0.0},     # B1: x=0
238         'B2': {'type': B2, 'coord': 'x', 'value': a},       # B2: x=a
239         'B3': {'type': B3, 'coord': 'y', 'value': 0.0},     # B3: y=0
240         'B4': {'type': B4, 'coord': 'y', 'value': b}        # B4: y=b
241     }
242
243     tol = 1e-9                                              # tolerância para
            identificar nós na borda
244     fixed = np.zeros((dofs,), dtype=bool)
245     # aplicar cada borda ao vetor fixed
246     fixed = apply_borda_fun(borda_map['B1'], coords, n_nos, dofs, tol, fixed)
247     fixed = apply_borda_fun(borda_map['B2'], coords, n_nos, dofs, tol, fixed)
248     fixed = apply_borda_fun(borda_map['B3'], coords, n_nos, dofs, tol, fixed)
249     fixed = apply_borda_fun(borda_map['B4'], coords, n_nos, dofs, tol, fixed)
250     free = np.where(~fixed)[0]                              # índices dos DOFs
            livres
251
252     K_csr = K.tocsr()                                       # converte para CSR
253     Kff = K_csr[free, :][:, free]                           # submatriz livre-
            livre (esparsa)
254     Ff = F[free]                                            # vetor de forças
            livres
255
256     try:
257         U = np.zeros((dofs,), dtype=float)                  # inicializar vetor
                de deslocamentos
258         Uf = spla.spsolve(Kff.tocsc(), Ff)                  # resolve os graus de
                liberdade livres
259         U[free] = Uf                                        # monta o vetor
                completo U
260     except Exception as ex:
261         warnings.warn(f'Falha ao resolver sistema linear esparso: {ex};
                tentando solução densa.')
262         Kff_dense = Kff.toarray()
263         Uf = np.linalg.solve(Kff_dense, Ff)
264         U[free] = Uf
265
266     # calcular deslocamentos nodais Wnod (w DOFs, cada 3º valor começando em 0)
267     Wnod = U[0::3]                                          # extrair w do vetor
            U
268     w_max = np.max(np.abs(Wnod))                            # valor máximo
            absoluto da deflexão nodal
269
270     # calcular erro relativo face ao passo anterior (em %)
271     if not np.isnan(w_prev):
```

```
272             erro_rel = abs(w_max - w_prev) / w_prev * 100.0
273         else:
274             erro_rel = np.inf
275 
276         print(f'Iteração: malha {n_side}x{n_side} -> max|w| = {w_max:.6e} m  (erro
                rel = {erro_rel:.3f}%)')
277 
278         if (not np.isnan(w_prev)) and (erro_rel <= tol_rel_pct):
279             converged = True
280             print(f'\nConvergência atingida! erro relativo = {erro_rel:.3f}%\n')
281             # guardar solução convergida para pós-processamento
282             U_converged = U.copy()
283             coords_converged = coords.copy()
284             conn_converged = conn.copy()
285             n_nos_converged = n_nos
286             K_converged = K_csr.copy()
287             F_converged = F.copy()
288             Reac_converged = (K_csr.dot(U) - F)              # reações nodais
289             break
290 
291 
292         w_prev = w_max
293 # ----------------------------
294 U = U_converged                                              # vetor de
        deslocamentos convergido
295 coords = coords_converged                                    # coordenadas
        convergidas
296 conn = conn_converged                                        # conectividade
        convergida
297 n_nos = n_nos_converged                                      # número de nós
        convergido
298 dofs = 3 * n_nos                                            # graus de liberdade
         totais
299 K_full = K_converged                                        # matriz K
        convergida
300 F_full = F_converged                                        # vetor de forças
        convergido
301 Reac = Reac_converged                                       # vetor de reações (
        K*U - F)
302 Wnod = U[0::3]                                             # deflexões nodais w
303 
304 Reac_y = Reac[0::3]       # forças verticais (reacções de w)
305 Reac_tx = Reac[1::3]      # momentos em torno de x (reacções de theta_x)
306 Reac_ty = Reac[2::3]      # momentos em torno de y (reacções de theta_y)
307 
308 # Identificar nós encastrados (x=0 e x=a)
309 tol = 1e-9
```

```
n_enc = np.where(
    (np.abs(coords[:,0] - 0.0) < tol) | (np.abs(coords[:,0] - a) < tol)
)[0]

# Extrair apenas as reacções nos nós encastrados
Ry_enc = Reac_y[n_enc]
Mx_enc = Reac_tx[n_enc]
My_enc = Reac_ty[n_enc]

print('\n--- REACÇÕES NOS ENCASTRAMENTOS ---')
print(f'Nº de nós encastrados: {len(n_enc)}')
print(f'Força vertical Ry: max = {np.max(Ry_enc):.3e} N | min = {np.min(Ry_enc):.3e} N')
print(f'Momento Mx: max = {np.max(Mx_enc):.3e} N·m | min = {np.min(Mx_enc):.3e} N·m')
print(f'Momento My: max = {np.max(My_enc):.3e} N·m | min = {np.min(My_enc):.3e} N·m')

# Usar a relção constitutiva para calcular tensões nodais
Qmat = Q_mat
sigma_x_node = np.zeros((n_nos,), dtype=float)              # sigma_x por nó
sigma_y_node = np.zeros((n_nos,), dtype=float)              # sigma_y por nó
tau_xy_node = np.zeros((n_nos,), dtype=float)               # tau_xy por nó
cont = np.zeros((n_nos,), dtype=int)                        # contador de contribuições por nó
gp = np.array([-1.0/np.sqrt(3.0), 1.0/np.sqrt(3.0)])

# loop por elementos para calcular tensões nodais (média das contribuições)
for e in range(conn.shape[0]):
    nodes = conn[e, :]                                      # nós do elemento (1-based)
    Xe = coords[nodes - 1, 0]                               # x coords dos 4 nós
    Ye = coords[nodes - 1, 1]                               # y coords dos 4 nós
    # deslocamentos nodais do elemento (12x1)
    Ue = np.zeros((12,), dtype=float)
    for iN in range(4):
        node_idx = nodes[iN] - 1
        Ue[iN*3:(iN*3 + 3)] = U[node_idx*3:(node_idx*3 + 3)]
    # montar C_num novamente (como antes)
    x1v = Xe[0]; x2v = Xe[1]; y1v = Ye[0]; y2v = Ye[3]
    C_num = np.zeros((12, 12), dtype=float)
    C_sub = C_sym.subs({x: x1v, y: y1v}); C_num[0:3, :] = np.array(C_sub[0:3, :].evalf(), dtype=float)
    C_sub = C_sym.subs({x: x2v, y: y1v}); C_num[3:6, :] = np.array(C_sub[3:6, :].evalf(), dtype=float)
```

```
348     C_sub = C_sym.subs({x: x2v, y: y2v}); C_num[6:9, :] = np.array(C_sub[6:9,
            :].evalf(), dtype=float)
349     C_sub = C_sym.subs({x: x1v, y: y2v}); C_num[9:12, :] = np.array(C_sub[9:12,
             :].evalf(), dtype=float)
350     # inversa para obter coeficientes a_elem
351     try:
352         Cinv = np.linalg.inv(C_num)
353     except np.linalg.LinAlgError:
354         Cinv = np.linalg.pinv(C_num)
355     a_elem = Cinv.dot(Ue)                                   # coeficientes do
            polinómio do elemento
356
357     # integração 2x2 Gauss para avaliar curvaturas e tensões e acumular em nós
358     for i in range(2):
359         xi = gp[i]
360         for j in range(2):
361             eta = gp[j]
362             # mapear xi,eta para coordenadas físicas x_p,y_p (assumindo
                    mapeamento bilinear simplificado)
363             x_p = 0.5 * ((1 - xi) * x1v + (1 + xi) * x2v)  # mapeamento linear
                    em x (elementos regulares)
364             y_p = 0.5 * ((1 - eta) * y1v + (1 + eta) * y2v)  # mapeamento
                    linear em y
365             H_eval = np.array(H.subs({x: x_p, y: y_p}).evalf(), dtype=float)  #
                     avalia H no ponto (x_p,y_p)
366             kappa = H_eval.dot(a_elem)                      # curvaturas (3x1)
367             z = t / 2.0                                     # posição z para
                    cálculo de tensões (face superior = +t/2)
368             sigma_vec = -z * (Qmat.dot(kappa))              # vetor [sigma_x;
                    sigma_y; tau_xy] = -z * Q * kappa
369             # encontrar nó mais próximo para agregar contributo (simplificação)
370             dists = np.sum((coords - np.array([x_p, y_p]))**2, axis=1)
371             n_i = np.argmin(dists)                          # índice do nó mais
                     próximo (0-based)
372             sigma_x_node[n_i] += sigma_vec[0]               # acumular sigma_x
373             sigma_y_node[n_i] += sigma_vec[1]               # acumular sigma_y
374             tau_xy_node[n_i] += sigma_vec[2]                # acumular tau_xy
375             cont[n_i] += 1                                  # incrementar
                    contador de contributos
376
377 # média das contribuições em cada nó (somente onde cont>0)
378 mask = cont > 0
379 sigma_x_node[mask] = sigma_x_node[mask] / cont[mask]
380 sigma_y_node[mask] = sigma_y_node[mask] / cont[mask]
381 tau_xy_node[mask] = tau_xy_node[mask] / cont[mask]
382 # ----------------------------
383 print(f'--- RESULTADOS FINAIS (malha {n_side}x{n_side}) ---')
```

```
384 print(f'Deflexão máxima (|w|) = {np.max(np.abs(Wnod)):.6e} m')
385 ReacZ = Reac[0:dofs:3]                                    # reações na
        componente vertical (cada 3º)
386 print(f'Reacção vertical: max = {np.max(ReacZ):.3e} N | min = {np.min(ReacZ):.3
        e} N')
387 if np.any(mask):
388     print('Tensões (z = +t/2) (Pa):')
389     print(f' sigma_x: max = {np.max(sigma_x_node[mask]):.3e} , min = {np.min(
            sigma_x_node[mask]):.3e}')
390     print(f' sigma_y: max = {np.max(sigma_y_node[mask]):.3e} , min = {np.min(
            sigma_y_node[mask]):.3e}')
391     print(f' tau_xy : max = {np.max(tau_xy_node[mask]):.3e} , min = {np.min(
            tau_xy_node[mask]):.3e}')
392 else:
393     print('Nenhuma contribuição de tensão calculada (verifica malha).')
394
395 # ----------------------------
396 Xg_lin = np.linspace(0, a, 200)                           # 200 pontos em x
397 Yg_lin = np.linspace(0, b, 200)                           # 200 pontos em y
398 Xg, Yg = np.meshgrid(Xg_lin, Yg_lin)                      # grelha 2D
399 Wgrid = griddata(points=coords, values=Wnod, xi=(Xg, Yg), method='cubic')
400
401 #Gráfico da deflexão
402 fig = plt.figure()
403 ax = fig.add_subplot(111, projection='3d')
404 surf = ax.plot_surface(Xg, Yg, Wgrid, cmap='jet', edgecolor='none')        #
        superfície
405 fig.colorbar(surf, ax=ax, shrink=0.6)                                     #
        barra de cor
406 ax.set_title('Deflexão w (m) - convergido')
407 ax.set_xlabel('x (m)'); ax.set_ylabel('y (m)'); ax.set_zlabel('w (m)')
408 ax.view_init(elev=45, azim=30)
409 plt.tight_layout()
410
411 #Grafico sigma_x
412 Sxgrid = griddata(points=coords, values=sigma_x_node, xi=(Xg, Yg), method='
        cubic')
413 fig2 = plt.figure()
414 ax2 = fig2.add_subplot(111, projection='3d')
415 surf2 = ax2.plot_surface(Xg, Yg, Sxgrid, cmap='jet', edgecolor='none')
416 fig2.colorbar(surf2, ax=ax2, shrink=0.6)
417 ax2.set_title(r'$\sigma_x$ (Pa) - face superior'); ax2.set_xlabel('x (m)'); ax2
        .set_ylabel('y (m)')
418 ax2.view_init(elev=45, azim=30)
419 plt.tight_layout()
420
421 #Grafico sigma_y
```

```
422 Sygrid = griddata(points=coords, values=sigma_y_node, xi=(Xg, Yg), method='
        cubic')
423 fig3 = plt.figure()
424 ax3 = fig3.add_subplot(111, projection='3d')
425 surf3 = ax3.plot_surface(Xg, Yg, Sygrid, cmap='jet', edgecolor='none')
426 fig3.colorbar(surf3, ax=ax3, shrink=0.6)
427 ax3.set_title(r'$\sigma_y$ (Pa) - face superior'); ax3.set_xlabel('x (m)'); ax3
        .set_ylabel('y (m)')
428 ax3.view_init(elev=45, azim=30)
429 plt.tight_layout()
430
431 #Gráfico tau_xy
432 Txygrid = griddata(points=coords, values=tau_xy_node, xi=(Xg, Yg), method='
        cubic')
433 fig4 = plt.figure()
434 ax4 = fig4.add_subplot(111, projection='3d')
435 surf4 = ax4.plot_surface(Xg, Yg, Txygrid, cmap='jet', edgecolor='none')
436 fig4.colorbar(surf4, ax=ax4, shrink=0.6)
437 ax4.set_title(r'$\tau_{xy}$ (Pa) - face superior'); ax4.set_xlabel('x (m)');
        ax4.set_ylabel('y (m)')
438 ax4.view_init(elev=45, azim=30)
439 plt.tight_layout()
440
441 #Gráfico Reacções verticais
442 Rgrid = griddata(points=coords, values=ReacZ, xi=(Xg, Yg), method='cubic')
443 fig5 = plt.figure()
444 ax5 = fig5.add_subplot(111, projection='3d')
445 surf5 = ax5.plot_surface(Xg, Yg, Rgrid, cmap='jet', edgecolor='none')
446 fig5.colorbar(surf5, ax=ax5, shrink=0.6)
447 ax5.set_title('Reacções verticais (N)'); ax5.set_xlabel('x (m)'); ax5.
        set_ylabel('y (m)')
448 ax5.view_init(elev=45, azim=30)
449 plt.tight_layout()
450
451 #Gráficos dos momentos de reacção
452 Mxgrid = griddata(points=coords, values=Reac_tx, xi=(Xg, Yg), method='cubic')
453 Mygrid = griddata(points=coords, values=Reac_ty, xi=(Xg, Yg), method='cubic')
454
455 fig6 = plt.figure()
456 ax6 = fig6.add_subplot(111, projection='3d')
457 surf6 = ax6.plot_surface(Xg, Yg, Mxgrid, cmap='jet', edgecolor='none')
458 fig6.colorbar(surf6, ax=ax6, shrink=0.6)
459 ax6.set_title('Momento de reacção Mx (N·m)')
460 ax6.set_xlabel('x (m)'); ax6.set_ylabel('y (m)')
461 ax6.view_init(elev=45, azim=30)
462 plt.tight_layout()
463
```

```
464 fig7 = plt.figure()
465 ax7 = fig7.add_subplot(111, projection='3d')
466 surf7 = ax7.plot_surface(Xg, Yg, Mygrid, cmap='jet', edgecolor='none')
467 fig7.colorbar(surf7, ax=ax7, shrink=0.6)
468 ax7.set_title('Momento de reacção My (N·m)')
469 ax7.set_xlabel('x (m)'); ax7.set_ylabel('y (m)')
470 ax7.view_init(elev=45, azim=30)
471 plt.tight_layout()
472
473 plt.show()
```

Listagem A.2: Código em Python para o caso 6