UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
UNIDADE ACADÊMICA DO CABO DE SANTO AGOSTINHO
BACHARELADO EM ENGENHARIA CIVIL

JAILSON TEIXEIRA DO NASCIMENTO

Análise numérica de placas finas via método dos elementos finitos

Cabo de Santo Agostinho - PE
2024

JAILSON TEIXEIRA DO NASCIMENTO

Análise numérica de placas finas via método dos elementos finitos

Monografia apresentada ao Curso de Graduação em Engenharia Civil da Unidade Acadêmica do Cabo de Santo Agostinho da Universidade Federal Rural de Pernambuco para obtenção do grau de bacharel em Engenharia Civil.

Área de concentração: Estruturas

Orientador: Prof. Dr. Felipi Pablo Damasceno Fernandes

Coorientador: Prof. Dr. Jordlly Reydson de Barros Silva

Cabo de Santo Agostinho - PE
2024

Dados Internacionais de Catalogação na Publicação
Sistema Integrado de Bibliotecas da UFRPE
Bibliotecário(a): Suely Manzi – CRB-4 809

N244a Nascimento, Jailson Teixeira do.
Análise numérica de placas finas via método dos elementos finitos / Jailson Teixeira do Nascimento. – Cabo de Santo Agostinho, 2024.
104 f.; il.

Orientador(a): Felipi Pablo Damasceno Fernandes.
Co-orientador(a): Jordlly Reydson de Barros Silva.

Trabalho de Conclusão de Curso (Graduação) – Universidade Federal Rural de Pernambuco, Unidade Acadêmica Cabo de Santo Agostinho - UACSA, Bacharelado em Engenharia Civil, Cabo de Santo Agostinho, BR-PE, 2025.

Inclui referências e anexo(s).

1. Método dos elementos finitos . 2. Placas (Engenharia) . 3. Análise numérica. 4. Python (Linguagem de programação de computador) 5. Lajes de concreto. I. Fernandes, Felipi Pablo Damasceno, orient. II. Silva, Jordlly Reydson de Barros, coorient. III. Título

CDD 624

JAILSON TEIXEIRA DO NASCIMENTO

Análise numérica de placas finas via método dos elementos finitos

Monografia apresentada ao Curso de Graduação em Engenharia Civil da Unidade Acadêmica do Cabo de Santo Agostinho da Universidade Federal Rural de Pernambuco para obtenção do grau de bacharel em Engenharia Civil.

Aprovado em: 04/ 10/ 2024

Banca examinadora

____
Prof. Dr. Felipi Pablo Damasceno Fernandes - UFRPE
Orientador

____
Prof. Dr. Jordlly Reydson de Barros Silva - UFPE
Coorientador

____
Prof. Dr. Paulo Marcelo Vieira Ribeiro - UFPE
Examinador 1

____
Prof. Me. Hildo Augusto Santiago Filho - UNICAP
Examinador 2

“(...) ... Teste ideias experimentando e observando. Utilize como base as ideias que passam no teste. Rejeite as que fracassam. Siga as evidências onde quer que elas nos levem e questione tudo. Leve essas regras a sério e o cosmos será seu ... venha comigo!”

Cosmos - Mundos Possíveis, por Neil Degrasse Tyson.

AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus por me conceder a força e resiliência necessária para alcançar meus objetivos.

Aos meus orientadores Dr. Filippe Damasceno e Dr. Jordlly Silva por despertarem em mim o interesse em estudar elementos finitos aplicado à engenharia de estruturas. Isso despertou em mim a beleza e o fascínio por essa ferramenta tão importante e poderosa para a engenharia. Além disso, agradeço também por acreditarem em mim nessa jornada.

Aos companheiros de graduação da UFRPE/UACSA Luiz Fernando e Alisson Barbosa pelas brilhantes ideias compartilhadas para o desenvolvimento deste trabalho.

A minha família, em especial a minha mãe Raimunda que nunca deixou de acreditar em mim, exemplo de personalidade forte e resistente.

RESUMO

Um dos grandes desafios do engenheiro civil em projetos de estruturas de concreto armado, é elaborar um modelo estrutural adequado para a estimar os esforços solicitantes atuantes e, com isso, ter condições de prever o comportamento da estrutura e dimensioná-la para suportar os seus esforços. Em lajes de concreto armado, por exemplo, existem métodos analíticos que podem ser utilizados. Porém, quando se tratam de lajes com geometrias mais gerais, há uma dificuldade de obtenção dos esforços solicitantes de forma precisa. O mesmo se aplica a soluções que levam em consideração o comportamento integrado entre as lajes de um pavimento. Neste trabalho, optou-se pela implementação em *python* de um elemento de placa utilizado para modelar placas finas baseado na teoria de Kirchhoff que se ajustasse a essa diversidade de geometrias. Com isso, foi implementado um elemento inicialmente aplicado em malhas de elementos retangulares. O elemento foi adaptado para ser usado em malhas de elementos retangulares distorcidos, porém foi verificado que, nesse caso, apresenta uma convergência pior dos resultados com o refinamento da malha. No entanto, mesmo com essa adaptação, o elemento obteve um ótimo desempenho na obtenção dos deslocamentos e esforços solicitantes em lajes analisadas separadamente. Além disso, foi estudado, ainda, o desempenho do elemento em pavimentos de edificações, em que a análise é feita de maneira conjunta. Nesse caso, partiu-se de problemas cujos esforços internos já haviam sido estimados pelo método das tabelas e utilizou-se esses resultados como referência das análises por elementos finitos. Sendo assim, observou-se que os esforços internos e deslocamentos obtidos apresentaram resultados razoáveis, se aproximando dos resultados da literatura, evidenciando-se a viabilidade da ferramenta criada.

PALAVRAS-CHAVE: elementos finitos; placas finas; teoria de Kirchhoff; python; lajes de concreto armado.

ABSTRACT

One of the major challenges in civil engineering in reinforced concrete structure projects is to develop a structural model that is suitable for estimating the required forces and, therefore, being able to predict the behavior of the structure and dimension it to withstand these forces. In reinforced concrete plans, for example, there are analytical methods that can be used. However, when dealing with slabs with more general geometries, it is difficult to obtain the required forces more accurately. The same applies to solutions that take into account the integrated behavior between the slabs of a floor. In this work, we chose to implement in Python a plate element used to model thin plates based on Kirchhoff's theory that was suitable for this diversity of geometries. Therefore, an element was implemented that was initially applied to rectangular element meshes. Thus, the element was adapted to be used in distorted rectangular element meshes, and it was found that it did not present convergence of displacements to the analytical reference values. Furthermore, regarding the convergence of internal forces, these behaved in a disturbed manner and, in some cases, did not show clear signs of convergence. However, even with the adaptation for distorted elements, the element obtained an excellent performance in obtaining the forces requested in rectangular slabs observed separately. In addition, the element's performance was also trained on building floors, in which the analysis is performed jointly. In this case, we started from problems of internal forces already done estimated by the table method and used these results as a reference for the finite element analyses. Thus, we demonstrated that some internal and internal forces obtained were close to the predefined values, while others considered very discrepant values.

KEYWORDS: finite elements; thin plates; Kirchhoff theory; python; reinforced concrete slabs.

SUMÁRIO

1 INTRODUÇÃO ... 10

1.1 Objetivos ... 10

1.1.1 Objetivo geral ... 10

1.1.2 Objetivos específicos ... 10

1.2 Justificativa ... 11

2 FUNDAMENTAÇÃO TEÓRICA ... 12

2.1 Breve introdução sobre a teoria da elasticidade ... 12

2.1.1 Estados tridimensional e bidimensional de tensões ... 12

2.1.2 Condições de compatibilidade ... 14

2.1.3 Modelo constitutivo elástico linear isotrópico ... 16

2.2 Teoria clássica de placas finas de Kirchhoff ... 19

2.2.1 Fundamentos da teoria ... 19

2.2.2 Configuração de tensões na placa ... 23

2.2.3 Esforços internos ... 24

2.3 Equação diferencial parcial de placas ... 25

2.3.1 Condições de contorno ... 28

2.3.2 Soluções analíticas da equação diferencial ... 31

2.4 Formulação do elemento finito de placa adotado ... 34

2.4.1 Matriz Jacobiana ... 34

2.4.2 Elemento retangular de placa fina adotado ... 40

2.5 Integração numérica ... 51

2.6 Análise de convergência ... 52

3 METODOLOGIA: IMPLEMENTAÇÃO VIA ELEMENTOS FINITOS ... 54

3.1 Linguagem Python ... 54

3.2 Descrição do código desenvolvido ... 54

3.2.1 Dados de entrada dos problemas -(Item 1 do código em anexo) ... 55

3.2.2 Processamento - (Item 2 do código em anexo) ... 60

3.2.3 Arquivo VTU - (Item 3 do código em anexo) ... 74

3.3 Paraview 74
4 RESULTADOS: VALIDAÇÃO E ANÁLISES 76
4.1 Validação do elemento 76
5 CONCLUSÕES E CONSIDERAÇÕES FINAIS 101
REFERÊNCIAS 102
ANEXO A - CÓDIGO EM PYTHON 104

1 INTRODUÇÃO

Uma fase importante nos projetos de lajes maciças de concreto armado é a estimativa dos esforços solicitantes. Sendo assim, é necessário aplicar um modelo adequado que se simule as condições de contorno do problema real, gerando um quadro de análises detalhadas e, especialmente, que tenha adaptabilidade às diversas geometrias provenientes de projetos de arquitetura arrojados. Essa fase é muito importante para a tomada de decisão do engenheiro, uma vez que serve de referência para um dimensionamento mais otimizado do elemento estrutural.

1.1 Objetivos

Nesta seção são discutidos os objetivos gerais e específicos que fazem parte do escopo do trabalho.

1.1.1 Objetivo geral

Desenvolvimento de um programa computacional em Python para estudar a magnitude dos deslocamentos e esforços internos atuantes em pavimentos de construções via método dos elementos finitos.

1.1.2 Objetivos específicos

- Realizar uma revisão bibliográfica sobre a teoria de flexão de placas finas com ênfase na solução do problema via método dos elementos finitos;
- Implementação da formulação via método dos elementos finitos;
- Análise gráfica dos resultados obtidos usando a ferramenta *Paraview*;
- Validar o código em elementos finitos usando outras soluções teóricas para o estudo de lajes.
- Análise de problemas práticos de flexão de placas em construções.

1.2 Justificativa

O método dos elementos finitos aborda uma grande variabilidade de problemas no dia a dia da engenharia civil. Uma de suas vantagens é a adaptabilidade às diversas geometrias do domínio de estudo oriundas de projetos de arquitetura. No caso de lajes, por exemplo, essa metodologia pode ser empregada em lajes com formas irregulares, contendo furos ou aberturas, contendo ou não, variações de espessura. É uma ferramenta que auxilia na análise detalhada dos esforços internos, facilitando a realização de projetos. Com o avanço da computação, a utilização do método dos elementos finitos se tornou mais acessível e eficiente, permitindo a realização de análises complexas em um tempo curto, facilitando a tomada de decisões.

2 FUNDAMENTAÇÃO TEÓRICA

Neste tópico, é apresentado o referencial teórico e conceitos que embasam o elemento finito implementado neste trabalho.

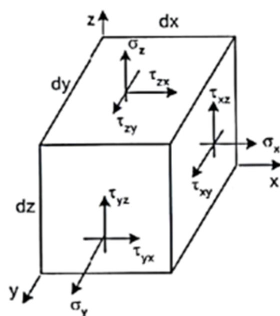
2.1 Breve introdução sobre a teoria da elasticidade

A teoria da elasticidade envolve o estudo de como os materiais se deformam e retornam a sua forma original quando submetidos a ação de forças, sendo importante para compreender seu comportamento e desempenho.

2.1.1 Estados tridimensional e bidimensional de tensões

Para estudar o estado tridimensional de tensão de um material, será considerado um pequeno elemento de dimensões infinitesimais *dx*, *dy* e *dz* com faces paralelas aos planos coordenados (Araújo, 2010). Este elemento é orientado ao longo dos eixos coordenados e retirado de um ponto qualquer do corpo. A representação esquemática desse elemento é apresentada na figura (1).

Figura (1): Estado tridimensional de tensões



Fonte: Araújo, 2010.

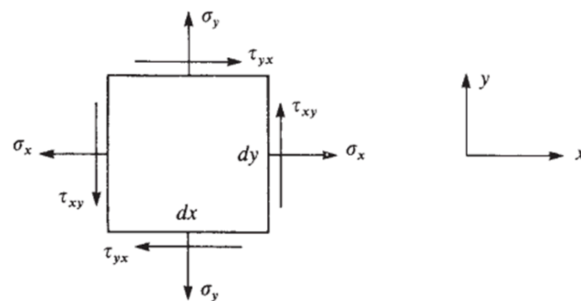
O estado geral de tensões pode ser representado pelas 9 componentes de tensões representado pela matriz do tensor das tensões apresentada a seguir.

$$\sigma = \begin{bmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z \end{bmatrix} \quad (1)$$

Devido à lei de reciprocidade das tensões de cisalhamento (Araújo, 2010), às 9 componentes de tensão são reduzidas a apenas 6 componentes. Sendo assim, temos $\tau_{xy} = \tau_{yx}$, $\tau_{xz} = \tau_{zx}$ e $\tau_{yz} = \tau_{zy}$. Em palavras, essas relações significam que para dois lados opostos perpendiculares entre si de um elemento cúbico, a magnitude das tensões de cisalhamento são iguais (Timoshenko, 1951). Essas considerações fazem com que a matriz do tensor das tensões seja simétrica em relação a sua diagonal principal.

O estado bidimensional de tensões é considerado um caso particular do estado tridimensional. Segundo Vaz (2011), quando a espessura de um corpo é pequena em relação às outras dimensões do mesmo, é comum adotar que as tensões com alguma componente z sejam nulas. A figura (2) apresenta um elemento infinitesimal bidimensional submetido a tensões normais e cisalhantes.

Figura (2): Estado bidimensional de tensões



Fonte: Logan, 2012.

Com isso, a matriz do tensor das tensões apresenta duas componentes normais e duas componentes de cisalhamento.

$$\sigma = \begin{bmatrix} \sigma_x & \tau_{xy} \\ \tau_{yx} & \sigma_y \end{bmatrix} \quad (2)$$

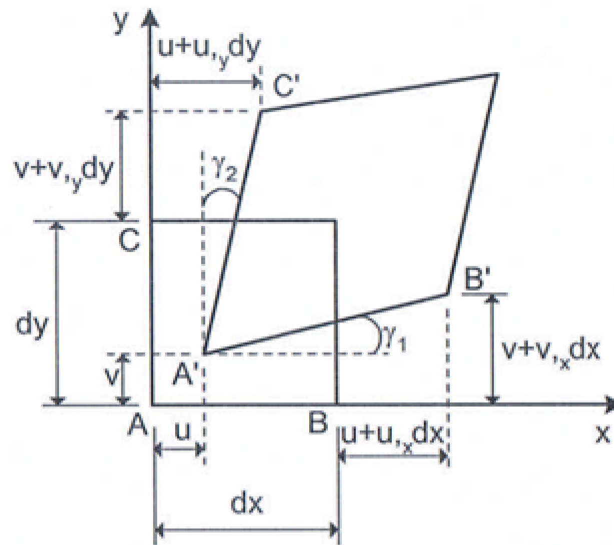
De maneira análoga ao estado tridimensional, vale também a lei de reciprocidade das tensões cisalhantes permitindo a equivalência das tensões cisalhantes $\tau_{xy} = \tau_{yx}$, levando a simetria da matriz.

2.1.2 Condições de compatibilidade

Para relacionar deslocamentos e deformações, considera-se um elemento com dimensões infinitesimais dx, dy e dz como apresentado na figura (3), que apresenta o comportamento do elemento no plano xy quando sujeito a ações de forças externas.

Os deslocamentos de cada ponto do corpo será medido através das funções u, v e w orientadas no sentido dos eixos coordenados x, y e z. O ponto A, sofre um deslocamento u e v como representado na figura (3). Sendo assim, é assumido que essas componentes são pequenas e que variam de maneira contínua ao longo de todo o elemento. Além disso, é assumido que ocorrem apenas pequenas deformações análogas às que ocorrem na engenharia de estruturas (Timoshenko, 1951).

Figura (3): Deformação do elemento



Fonte: Araújo, 2010.

Sendo assim, de acordo com a figura (3), o comprimento da projeção A'B' na direção do eixo x é representado conforme a equação (3).

$$dx' = (1 + \frac{\partial u}{\partial x})dx \tag{3}$$

Na equação (3), o comprimento na direção de x sofreu um incremento de $\frac{\partial u}{\partial x}dx$. Portanto, o termo $\frac{\partial u}{\partial x}$ é deformação unitária do comprimento na direção de x e é representada pela equação (4) a seguir;

$$\varepsilon_x = \frac{\partial u}{\partial x} \tag{4}$$

De maneira análoga, a deformação unitária do comprimento na direção dos eixos y e z são representados pelas equações (5) e (6).

$$\varepsilon_y = \frac{\partial v}{\partial y} \quad (5)$$

$$\varepsilon_z = \frac{\partial w}{\partial z} \quad (6)$$

Devido aos deslocamentos, a face AB do elemento é distorcida de um ângulo γ_1, bem como a face AC do elemento é distorcido de um ângulo γ_2 como mostra a figura (3). Sendo assim, a deformação por corte é dada pela distorção do elemento e é descrita através da equação (7).

$$\gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \quad (7)$$

Analogamente, as distorções nas direções y e z, são representadas pelas equações a seguir:

$$\gamma_{xz} = \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \quad (8)$$

$$\gamma_{zx} = \frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \quad (9)$$

2.1.3 Modelo constitutivo elástico linear isotrópico

Para a maioria dos problemas de engenharia que envolvem modelagem matemática são assumidas premissas e propriedades que simplificam as equações, tornando o problema mais simples de ser solucionado. Em relação a estudos envolvendo o comportamento de materiais estruturais, uma dessas premissas é a de que o material assume um comportamento linear elástico. Na prática, os materiais estruturais possuem certa elasticidade.

Isso significa que, ao ser submetido à ação de forças externas não excedendo certo limite, estas produzem uma deformação no corpo de tal maneira que ao retirar as cargas atuantes, o corpo retorna a sua configuração física original (Timoshenko, 1951).

Segundo Hibbeler (2010), para a maioria dos projetos de engenharia, é assumido que os materiais desenvolvem pequenas deformações. Em muitos casos, isso faz com que o material trabalhe no regime linear elástico, atendendo aos esforços mecânicos para qual foi projetado.

Outra premissa a ser considerada é a de que o material é homogêneo, ou seja, o este é distribuído de maneira contínua ao longo da estrutura de modo que um pequeno elemento de dimensões infinitesimais tenha as mesmas propriedades físicas do material como um todo. Além disso, é assumido que o material é isotrópico. Isso significa que as propriedades são as mesmas em todas as direções do material (Timoshenko, 1951).

Para Timoshenko (1951), para um material elástico-linear, a relação entre tensão e deformação é estabelecida pela lei de Hooke. Sendo assim, para um elemento cúbico infinitesimal submetido à ação de tensões normais uniformemente distribuídas ao longo de cada uma de suas faces, é possível mostrar que as componentes de deformações em cada direção são representadas pelas seguintes equações;

$$\varepsilon_x = \frac{1}{E}\left[\sigma_x - v\left(\sigma_y + \sigma_z\right)\right] \tag{10}$$

$$\varepsilon_y = \frac{1}{E}\left[\sigma_y - v\left(\sigma_x + \sigma_z\right)\right] \tag{11}$$

$$\varepsilon_z = \frac{1}{E}\left[\sigma_z - v\left(\sigma_x + \sigma_z\right)\right] \tag{12}$$

O mesmo raciocínio é aplicado às deformações associadas ao cisalhamento.

$$\gamma_{xy} = \frac{\tau_{xy}}{G} \; ; \gamma_{xz} = \frac{\tau_{xz}}{G} \; ; \gamma_{yz} = \frac{\tau_{yz}}{G} \tag{13}$$

$$G = \frac{E}{2(1+v)} \quad (14)$$

Para o estado plano de tensão, $\tau_{xz} = \tau_{xz} = 0$ (Logan, 1976). Consequentemente, pela equação (13), $\gamma_{xz} = \gamma_{xz} = 0$. Além disso, segundo Vaz (2011), para o estado plano de tensões considera-se $\sigma_z = 0$. Sendo assim, organizando matricialmente as equações (10), (11) e (13), obtém-se:

$$\begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix} = \frac{1}{E} \begin{bmatrix} 1 & -v & 0 \\ -v & 1 & 0 \\ 0 & 0 & 2(1+v) \end{bmatrix} \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} \quad (15a)$$

Ou ainda

$$\varepsilon = D\sigma \quad (15b)$$

Em que D é a matriz de elasticidade que representa o estado plano de tensão, E é o módulo de elasticidade longitudinal do material e v é o coeficiente de Poisson. Usando a lei de Hooke nas equações (10), (11) e (13), e assumindo as condições do estado plano de tensões com $\sigma_z = 0$, podemos escrever as tensões da seguinte maneira;

$$\sigma_x = \frac{E}{1-v^2}(\varepsilon_x + v\varepsilon_y) \quad (16)$$

$$\sigma_y = \frac{E}{1-v^2}(\varepsilon_y + v\varepsilon_x) \quad (17)$$

$$\tau_{xy} = \frac{E}{2(1-v)}\gamma_{xy} \quad (18)$$

Organizando matricialmente, obtém-se:

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = \frac{E}{(1-v^2)} \begin{bmatrix} 1 & v & 0 \\ v & 1 & 0 \\ 0 & 0 & \frac{1-v}{2} \end{bmatrix} \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix} \quad (19a)$$

Ou podemos escrever ainda;

$$\sigma = D^{-1}\varepsilon = C\varepsilon \quad (19b)$$

Em que C é a matriz constitutiva para o estado plano de tensão.

2.2 Teoria clássica de placas finas de Kirchhoff

Este tópico discorre sobre a teoria básica de placas finas envolvendo a teoria de Kirchhoff.

2.2.1 Fundamentos da teoria

A primeira teoria completa para o problema de flexão de placas foi formulada por Gustav R. Kirchhoff (1824 – 1887). Baseando-se nas hipóteses de Bernoulli para vigas, fazendo com que todas as condições de contorno pudessem ser escritas em temos da função de deslocamento e das suas respectivas derivadas em x e y. Para tal, assumiu algumas proposições para a simplificação da modelagem (Szilard, 2004).

Estas proposições estão escritas a seguir:

1. O material da placa é elástico linear, homogêneo e isotrópico;
2. A placa é considerada fina. Isso significa que a sua espessura é pequena quando comparada com as demais dimensões da placa;
3. As deflexões são pequenas quando comparadas a espessura da placa;
4. As rotações da superfície média da placa após a deformação são pequenas;
5. Qualquer linha reta e normal a um ponto da superfície média da placa indeformada permanece reta e normal ao plano tangente à superfície média da placa deformada nesse mesmo ponto;

6. As deflexões da placa ocorrem apenas na direção normal ao plano indeformado inicial;
7. São desconsideradas as tensões normais à superfície média da placa.

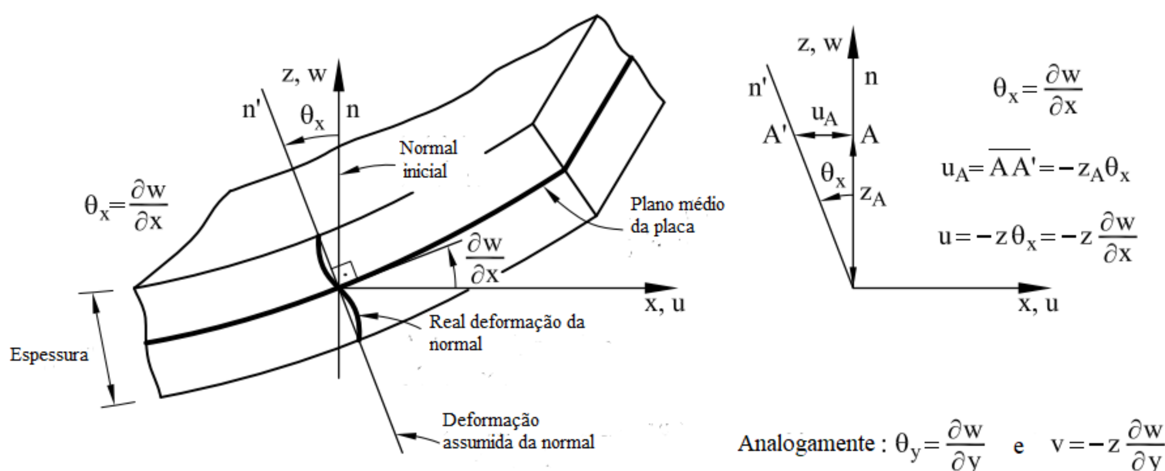
A quinta proposição acima foi inspirada na hipótese das seções planas da teoria de vigas, e consiste em desprezar a deformação por corte (Araújo, 2010). Essa é uma das hipóteses básicas explorada pela NBR 6118/2014 em seu tópico 14.7.1, em que discorre sobre estruturas com elementos de placa.

Um outro critério importante a considerar é a classificação das placas quanto a sua espessura. Uma placa é considerada fina ou delgada quando a razão entre o menor vão da placa e a sua espessura é maior ou igual a 20. Caso contrário, é considerada espessa (Vaz, 2011).

No caso de uma placa espessa, a influência do cisalhamento transversal não é mais desprezível, tornando maior a distorção normal durante a deformação. Sendo assim, retas normais ao plano médio da placa permanecem retas, porém não perpendiculares ao plano tangente da deformada (Carvalho, 2019).

Com intuito de descobrir as relações entre deformações e deslocamentos de uma placa fina consideramos a figura (4) a seguir:

Figura (4): Representação da deformada da placa



Fonte: Oñate, 2013. Adaptado

A figura acima apresenta uma placa deformada em uma seção paralela ao eixo x. O ponto A encontra-se a uma distância z_A do plano médio da placa e sofre um deslocamento transversal de w correspondente ao deslocamento do plano médio.

Além disso, após a deformação, a linha n, inicialmente na vertical, sofre uma rotação de um ângulo θ_x representada pela linha n' .

É importante lembrar que a quinta proposição é apenas uma mera simplificação. Na realidade, as retas normais pós-deformação, são distorcidas e não necessariamente normal a superfície deformada como sugere a figura (4) (Oñate, 2013).

De acordo com a sexta proposição de Kirchhoff, um ponto pertencente ao plano médio da placa movimenta-se apenas na vertical de um valor $w(x, y, z) = w(x, y)$. O ponto A da figura (4) também se desloca com o mesmo valor, porém sofre uma rotação de θ_x correspondente ao giro da seção n. Por ser pequeno, esse giro pode ser aproximado por um deslocamento horizontal AA' representado na figura (4) por u_A(Carvalho, 2019).

Para obter o deslocamento horizontal $u_A(x, y, z)$, baseando-se na figura (4), temos;

$$u_A(x, y, z) = -z_A tan\,(\theta_x) \tag{20}$$

De acordo com a quarta proposição, a placa estará sujeita a pequenas rotações. Sendo assim, a função $tan(\theta_x)$ pode ser substituída pela função θ_x uma vez que ambas são muito próximas para valores de θ_x muito pequenos. A equação (21) ajuda a compreender esse raciocínio.

$$\lim_{\theta_x \to 0} \frac{tan(\theta_x)}{\theta_x} = 1 \tag{21}$$

Isso permite fazer a seguinte simplificação;

$$\tan(\theta_x) \cong \theta_x \tag{22}$$

Portanto, para pequenas rotações, podemos escrever o deslocamento horizontal do ponto A na direção de x da seguinte maneira;

$$u_A(x, y, z) = -z_A\theta_x \tag{23}$$

Analogamente, para a flexão no plano yz , temos;

$$v_A(x, y, z) = -z_A\theta_y \tag{24}$$

Como anteriormente mencionado, na direção de z, o deslocamento é medido através da função w representadas a seguir

$$w(x, y, z) = w(x, y) \tag{25}$$

Com base nisso, de acordo com Oñate (2013) as equações (23), (24) e (25) acima descrevem o campo de deslocamento de um ponto em placas à flexão, em que as funções u, v e w correspondem aos deslocamentos nas direções dos eixos coordenados. Segundo Araújo (2010), as rotações são obtidas pelas derivadas de $w$como descritas a seguir;

$$\theta_x = \frac{\partial w}{\partial x} \tag{26}$$

$$\theta_y = \frac{\partial w}{\partial y} \tag{27}$$

Inserindo as equações (26) e (27) nas expressões do campo de deslocamento, temos;

$$u(x,y,z) = -z\frac{\partial w}{\partial x} \tag{28}$$

$$v(x,y,z) = -z\frac{\partial w}{\partial y} \tag{29}$$

$$w(x,y,z) = w(x,y) \tag{30}$$

2.2.2 Configuração de tensões na placa

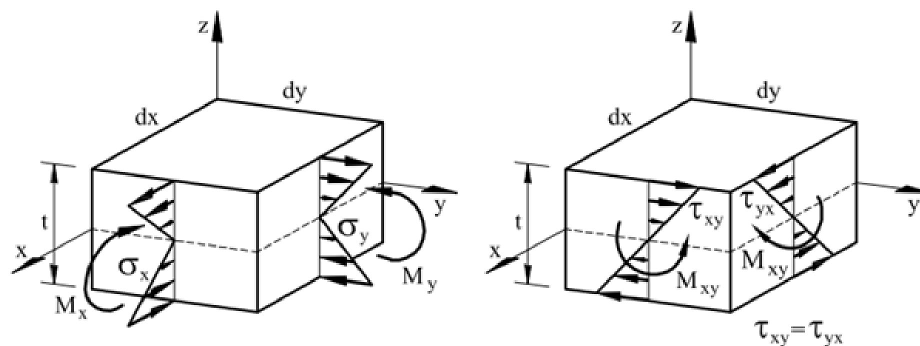
Segundo Araújo (2010), as três componentes de tensão variam linearmente na espessura da placa, anulando-se no plano médio como sugere a figura (5) apresentada por Oñate (2013). Para obter as componentes de tensão da placa substituímos as equações (16), (17) e (18) nas equações (4), (5) e (7) obtendo-se:

$$\sigma_x = \frac{-Ez}{1-v^2}\left(\frac{\partial^2 w}{\partial x^2} + v\frac{\partial^2 w}{\partial y^2}\right) \tag{31}$$

$$\sigma_y = \frac{-Ez}{1-v^2}\left(\frac{\partial^2 w}{\partial y^2} + v\frac{\partial^2 w}{\partial x^2}\right) \tag{32}$$

$$\tau_{xy} = \frac{-Ez}{(1+v)}\left(\frac{\partial^2 w}{\partial x \partial y}\right) \tag{33}$$

Figura (5): Configuração das tensões ao longo da espessura da placa



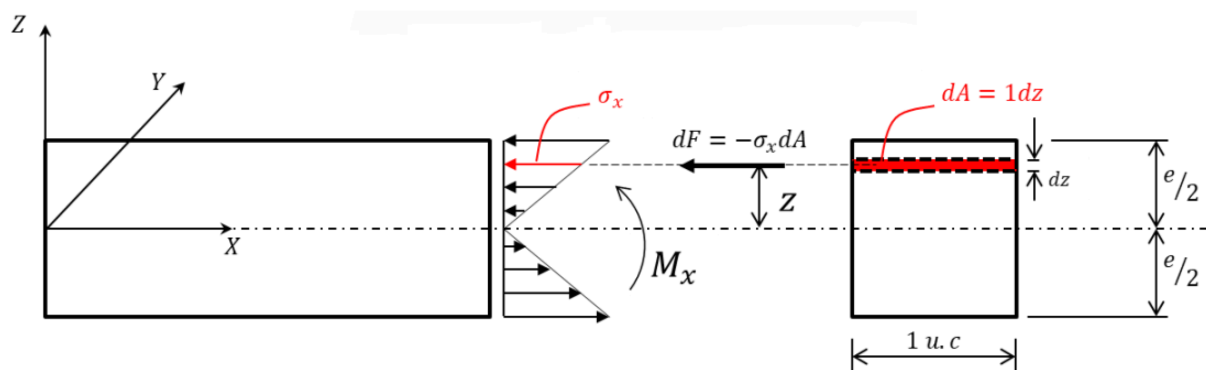
Fonte: Oñate, 2013.

De maneira semelhante ao que acontece nas vigas, às tensões σ_x e σ_y produzem momentos fletores na placa ao passo que as tensões τ_{xy} e τ_{yx} produzem momentos torsores na placa (Araújo, 2010).

2.2.3 Esforços internos

Para a obtenção dos esforços internos, soma-se a tensão atuante em uma área infinitesimal da placa ao longo de toda sua espessura como sugere a figura (6).

Figura (6): Esforços internos por integração



Fonte: Carvalho, 2019.

Da figura (6), retiramos as relações

$$dM_x = zdF \tag{34}$$

$$dM_x = -\sigma_x z dA \tag{35a}$$

Realizando a integração ao longo da espessura temos;

$$M_x = \int_{-e/2}^{e/2} -\sigma_x z dz \tag{35b}$$

De maneira análoga, ocorre com os demais esforços a seguir;

$$M_y = \int_{-e/2}^{e/2} -\sigma_y z dz \tag{36}$$

$$M_{xy} = \int_{-e/2}^{e/2} -\tau_{xy} z dz \tag{37}$$

Efetuando-se a integração, temos;

$$M_x = -D \left[\frac{\partial^2 w}{\partial x^2} + v \frac{\partial^2 w}{\partial y^2} \right] \tag{38}$$

$$M_y = -D \left[\frac{\partial^2 w}{\partial y^2} + v \frac{\partial^2 w}{\partial x^2} \right] \tag{39}$$

$$M_{xy} = -D \left[(1 - v) \frac{\partial^2 w}{\partial x \partial y} \right] \tag{40}$$

Na equação acima, *D* é a rigidez da placa à flexão e é dada pela expressão a seguir:

$$D = \frac{Eh^3}{12 \left(1 - v^2\right)} \tag{41}$$

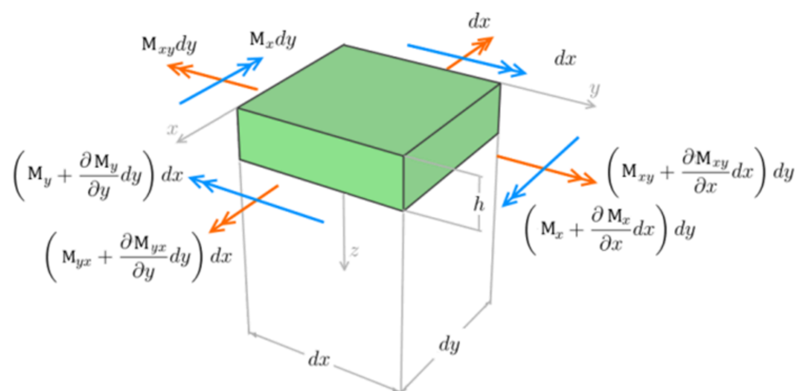
É importante observar que, se tomamos $v = 0$ na equação (41), a rigidez à flexão da placa se assemelha a rigidez à flexão de uma viga de largura unitária. Além disso, as equações (38), (39) e (40) fornecem valores de momento por unidade de comprimento, uma vez que as integrações são tomadas considerando uma faixa unitária da laje (Araújo, 2010).

2.3 Equação diferencial parcial de placas

A equação diferencial de placas foi estudada por Lagrange inspirado nos trabalhos de Marie-Sophie Germain (1776 - 1831), matemática francesa que escreveu trabalhos sobre teoria de vibração de placas e superfícies elásticas (Szilard, 2004).

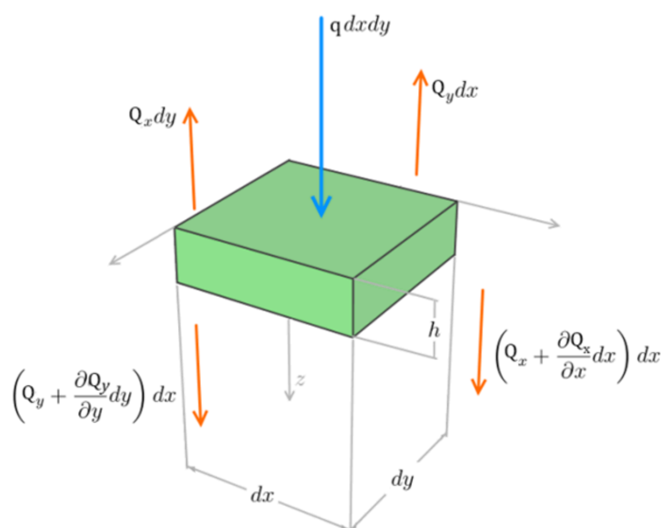
Para a obtenção da equação diferencial parcial de placa é necessário considerar o equilíbrio de um elemento de dimensões infinitesimais tomado em uma posição genérica da placa sujeita a ação de um carregamento genérico $q(x, y)$ (Araújo, 2010). No plano médio da placa são indicados os esforços solicitantes como indicado nas figuras (7) e (8).

Figura (7): Equilíbrio dos momentos



Fonte: Longo, 2018

Figura (8): Equilíbrio das forças cortantes



Fonte: Longo, 2018

Aplicando a condição de equilíbrio dos momentos na placa em torno do eixo x, obtemos;

$$\left(M_x + \frac{\partial M_x}{\partial x}dx\right)dy - M_x dy + \left(M_{xy} + \frac{\partial M_{xy}}{\partial y}dy\right)dx - M_{xy}dx - \left(Q_x + \frac{\partial Q_x}{\partial x}dx\right)dydx = 0 \qquad (42)$$

Eliminando os termos comuns e o termo $\frac{\partial Q_x}{\partial x}(dx)^2$ por se tratar de um infinitésimo de ordem superior (Araújo, 2010), chega-se a seguinte expressão:

$$\frac{\partial M_x}{\partial x} + \frac{\partial M_{xy}}{\partial y} = -Q_x \qquad (42.a)$$

Efetuando o mesmo procedimento em torno do eixo y, e procedendo de maneira análoga ao realizado anteriormente, temos;

$$\frac{\partial M_y}{\partial y} + \frac{\partial M_{xy}}{\partial x} = -Q_y \qquad (43)$$

Fazendo o equilíbrio das componentes na direção de z, temos:

$$\left(Q_x + \frac{\partial Q_x}{\partial x}dx\right)dy - Q_x dy + \left(Q_y + \frac{\partial Q_y}{\partial y}dy\right)dx - Q_y dx + q(x,y)dxdy = 0 \qquad (44a)$$

Organizando os termos obtemos;

$$\frac{\partial Q_x}{\partial x} + \frac{\partial Q_y}{\partial y} = -q \tag{44.b}$$

Derivando a equação (42.a) em relação a x e a equação (43) em relação a y e, em seguida substituindo na equação (44.b), lembrando que $\frac{\partial M_{xy}}{\partial y} = \frac{\partial M_{yx}}{\partial y}$ temos ;

$$\frac{\partial^2 M_x}{\partial x^2} + 2\frac{\partial^2 M_{xy}}{\partial x \partial y} + \frac{\partial^2 M_y}{\partial y^2} = -q \tag{45a}$$

Substituindo as equações (38), (39) e (40) obtidas anteriormente, chegamos a seguinte expressão que modela a deformada de uma placa:

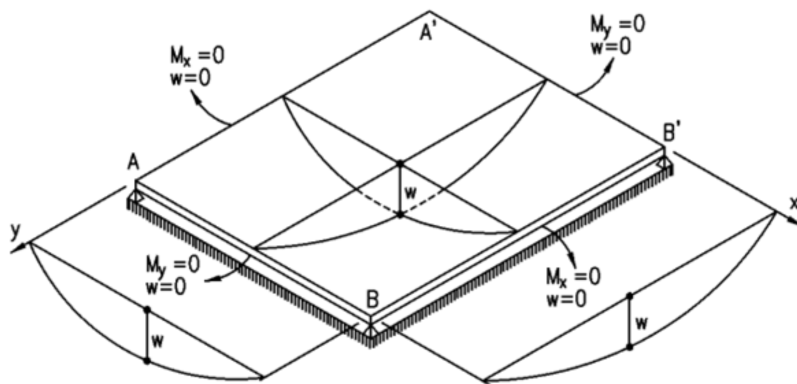
$$\frac{\partial^4 w}{\partial x^4} + 2\frac{\partial^4 w}{\partial x^2 \partial y^2} + \frac{\partial^4 w}{\partial y^4} = -\frac{q}{D} \tag{45b}$$

2.3.1 Condições de contorno

Devido a equação diferencial de placas ser de ordem 4, para obter a solução são necessárias 2 condições de contorno sendo estas de deslocamentos ou de esforços internos em cada bordo da placa. Estas condições de contorno podem ser geométricas, mecânicas ou mistas. As condições de contorno geométricas são impostas em termos de deslocamentos e rotações, enquanto que as condições de contorno mecânicas estão relacionadas a valores prescritos de esforços solicitantes. As condições mistas incluem ambos os tipos. (Araújo, 2010).

Nos bordos simplesmente apoiados, são aplicadas condições de contorno mistas. Se um bordo está apoiado, isso significa que o momento fletor e a flecha w são nulos ao longo desse bordo (DIAS, 2019). A figura (9) a seguir apresenta um esquema de uma placa retangular simplesmente apoiada em todos os bordos.

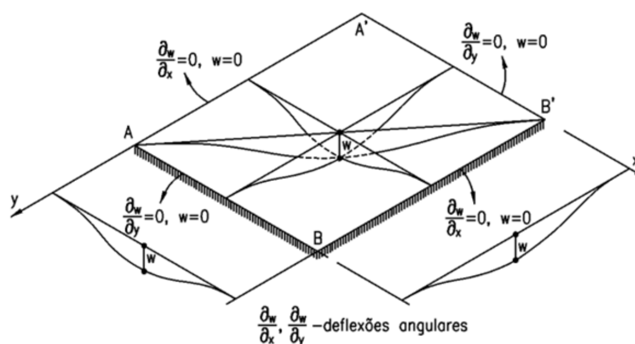
Figura (9): Placa simplesmente apoiada em todos os bordos



Fonte: Dias, 2019.

Em bordos engastados são utilizadas duas condições de contorno geométricas. Se um bordo é perfeitamente engastado, isso significa que a flecha e a rotação ao longo desse bordo são nulos (Araújo, 2010). A placa retangular da figura (10) apresenta a condição de contorno de engaste aplicada em todos os bordos.

Figura (10): Placa retangular engastada em todos os bordos



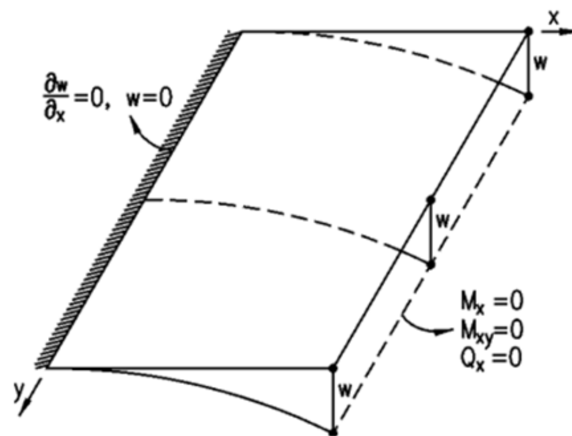
Fonte: Dias, 2019.

Para Araújo (2010), nos bordos livres não carregados são verificadas três condições de contorno mecânicas simultaneamente. Estas estão apresentadas a seguir;

$$M_x = 0\,;\ V_x = 0\,;\ M_{xy} = 0 \tag{46}$$

A figura (11) apresenta uma placa retangular em que apresenta um de seus bordos livres

Figura (11): Placa com bordo livre



Fonte: Dias, 2019.

Contudo, para a solução da equação diferencial de placas é necessário apenas duas condições de contorno, uma vez que a equação é de quarta ordem. Sendo assim, Kirchhoff (1824 – 1887) resolveu este problema simulando uma condição de contorno que envolve o esforço cortante e o momento torçor. A dedução completa para este caso pode ser encontrada na obra de Araújo (2010).

As condições de contorno supra descritas são importantes para obtenção de soluções analíticas da equação (45b). Entretanto, a solução via elementos finitos exige que o domínio de estudo seja discretizado em malhas compostas de unidades menores chamadas de elementos finitos. Toda e qualquer condição de contorno aplicada à malha é diretamente aplicada a cada grau de liberdade associado a cada nó do elemento. Seguindo esse raciocínio, a condição de bordo simplesmente apoiado seria obtido através da restrição do grau de liberdade do nó correspondente à translação.

Caso seja necessário a implantação de um bordo engastado, os nós que compõem o bordo teriam todos os seus três graus de liberdade restringidos de modo a não haver deslocamentos ou rotações em qualquer direção. De maneira análoga, na condição de bordo livre, todos os três graus de liberdade de todos os nós desse bordo estariam isentos de restrição.

Essas condições de contorno aplicadas aos nós da malha serão descritas com mais clareza nos tópicos referentes a formulação do elemento adotado neste trabalho, embasada na metodologia descrita no tópico 3.

2.3.2 Soluções analíticas da equação diferencial

Em termos matemáticos, a equação diferencial de placa é considerada uma equação diferencial parcial linear de ordem 4 de coeficientes constantes (Szilard, 2004). Segundo Araújo (2010), a solução analítica exata da equação diferencial de placas pode ser obtida apenas para poucos casos particulares. Para casos mais gerais de carregamentos e condições de contorno, são encontradas soluções inscritas em termos de séries duplas de Fourier.

As soluções mais conhecidas são a solução de Navier e Lévy. Para tal, é considerado que na condição de $w = 0$, os apoios são indeformáveis, ou seja, que estes são isentos de qualquer deformação provenientes da própria estrutura ou de eventuais recalques da fundação. Essas considerações podem não atender exatamente casos em que as lajes dos edifícios são apoiadas em vigas deformáveis, por exemplo.

Segundo Dias (2019), as soluções analíticas de Navier e Lévy são úteis para o cálculo de deslocamentos e esforços internos em alguns casos, mediante condições de contorno adequadas.

Em 1820, Navier apresentou a solução da equação diferencial de placa à flexão inscrita em termos de séries trigonométricas duplas de Fourier. A equação de Navier é considerada uma solução útil da equação (45b), uma vez que transforma a equação diferencial em uma equação algébrica, facilitando as operações matemáticas necessárias para a obtenção da solução (Szilard, 2004).

Para uma placa retangular como aquela apresentada pela figura (9), de condições de contorno indicadas, submetida a uma carga uniformemente distribuída, a expressão que calcula a flecha é dada pela equação (47).

É preciso lembrar que essa solução diz respeito apenas a placas retangulares simplesmente apoiadas em apoios indeformáveis com carga uniformemente distribuída (Araújo, 2010).

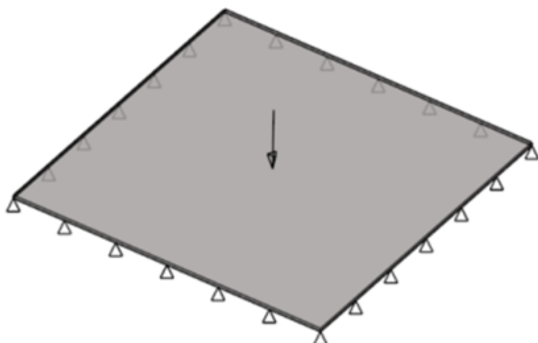
$$w(x,y) = \frac{16p_o}{\pi^6 D} \sum_{m=1}^{\infty} \sum_{n=1}^{\infty} \frac{sin\left(\frac{m\pi x}{a}\right) sin\left(\frac{n\pi y}{b}\right)}{mn\left[\left(\frac{m}{a}\right)^2 + \left(\frac{n}{b}\right)^2\right]^2} \tag{47}$$

Em que p_o representa a carga e os coeficientes a e b são as dimensões da placa e D representa o coeficiente de rigidez à flexão da placa. Substituindo nas equações (38), (39) e (40), é possível obter as expressões para os esforços internos. Araújo (2010) afirma que, uma vez que os esforços internos são obtidos mediante as derivações da equação (49b), ocorre uma perda de precisão nos esforços, sendo necessário, nesse caso, reter um número maior de termos da série para obtenção de uma resposta mais precisa.

A partir das soluções de Navier e Lévy, é possível combinar algumas condições de contorno e carregamento mais variadas e encontrar outras soluções da equação diferencial de placas utilizando-se do princípio da superposição (Araújo, 2010). Carvalho (2019), apresenta algumas expressões analíticas para o cálculo da flecha em placas finas. Essas soluções são oriundas da equação (49b), sendo truncada em uma quantidade finita de termos. Esses procedimentos geram expressões mais enxutas de solução, permitindo uma rápida estimativa do valor da flecha. A figura (12) apresenta a solução analítica particularizada para o caso de placas finas quadradas sob carregamento pontual e uniformemente distribuído.

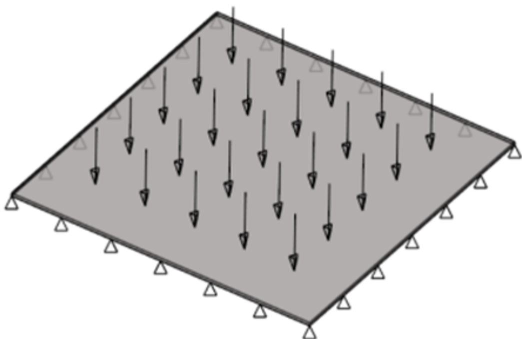
Figura (12): Soluções anaíticas em placas quadradas finas

Configuração de carregamento | **Solução analítica**



$$w_{max} = 0,00406\frac{Pa^2}{D}$$

.



$$w_{max} = 0,00406\frac{qa^4}{D}$$

Fonte: Carvalho, 2019.

Na literatura, existem muitos trabalhos que usam o método das tabelas para o cálculo de placas. Segundo Araújo (2010) a diferença entre elas se dá geralmente na variação do coeficiente de Poisson e também em casos associados ao truncamento das séries de Fourier, algumas das quais podem ser encontradas em Santana (2019), Pinheiro (2010), e também em Araújo (2010).

2.4 Formulação do elemento finito de placa adotado

Essa seção visa apresentar a formulação do elemento finito de placa quadrilateral adotado neste trabalho. São apresentadas as formulações da matriz jacobiana adotada no mapeamento da geometria do elemento, a matriz de funções de forma com suas respectivas funções de interpolação, bem como discorre sobre a formulação do elemento finito usado nas análises deste trabalho. O elemento apresentado é baseado nos trabalhos de Vaz (2011), Logan (2012) e também no elemento MZC apresentado por Oñate (2013).

2.4.1 Matriz Jacobiana

A matriz Jacobiana é utilizada para a transformação entre as coordenadas reais do elemento finito e coordenadas naturais. As coordenadas naturais $\xi\eta$ estão associadas ao elemento com centro na origem como mostra a figura (13). A orientação desse sistema de coordenada é tal que as quatro arestas do elemento sejam delimitadas por -1 e 1. Segundo Logan (2012), esse procedimento é adotado para possibilitar a implementação da integração numérica, conforme será discutido posteriormente.

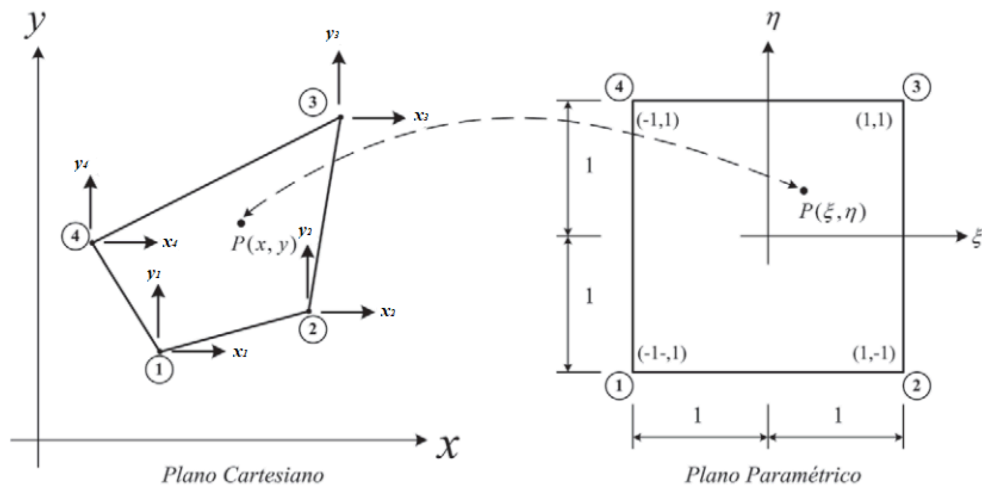
Uma analogia útil para entender a importância da matriz jacobiana é imaginar um mapa utilizado para facilitar a orientação de uma rota, por exemplo. Esse mapeamento facilita a visualização da trajetória, associando cada elemento ou obstáculo presente no percurso real, a uma representação gráfica em papel, porém em uma escala reduzida. Nesse caso, o fator de escala do mapa está associado ao valor absoluto do determinante da matriz jacobiana (Hjelmstad, 2005).

Segundo Vaz (2011), para um elemento quadrilateral, os campos que descrevem as coordenadas naturais são constituídos de polinômios de 4 termos em coordenadas paramétricas para cada ponto.

$$x(\xi, \eta) = a_1 + a_2\xi + a_3\eta + a_4\xi\eta \tag{48}$$

$$y(\xi, \eta) = a_5 + a_6\xi + a_7\eta + a_8\xi\eta \tag{49}$$

Figura (13): Mapeamento da geometria



Fonte: Vaz, 2011. Adaptado.

As equações equações (48) e (49), podem ser representadas de forma matricial ou sucinta, pela equação (50) e equação (50.a) respectivamente.

$$\begin{Bmatrix} x(\xi,\eta) \\ y(\xi,\eta) \end{Bmatrix} = \begin{bmatrix} 1 & \xi & \eta & \xi\eta & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \xi & \eta & \xi\eta \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_8 \end{Bmatrix} \quad (50a)$$

$$u_d(\xi,\eta) = Na(\xi,\eta)a_d \quad (50b)$$

A variável u_d representa o vetor de deslocamentos nodais. Segundo Vaz (2011), o fato dos polinômios paramétricos descritos nas equações (48) e (49) terem 4 termos, totalizando 8 coeficientes incógnitos, é devido as 8 condições de contorno impostas a cada nó do elemento como mostrado a seguir;

$$\begin{aligned} x(-1,-1) &= x_1 & x(1,1) &= x_3 \\ y(-1,-1) &= y_1 & y(1,1) &= y_3 \\ x(1,-1) &= x_2 & x(1,-1) &= x_4 \\ y(1,-1) &= y_2 & y(1,-1) &= y_4 \end{aligned} \tag{51}$$

Usando a equação (50b), podemos replicar as condições de contorno da equação (51) na expressão matricial a seguir;

$$\begin{Bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \\ x_4 \\ y_4 \end{Bmatrix} = \begin{bmatrix} 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{Bmatrix} \tag{52a}$$

Ou

$$d = Aa \tag{52b}$$

Em que d é o vetor que contém os deslocamentos nodais do elemento.

Da equação (52b), tem-se:

$$a = A^{-1}d \tag{52c}$$

Usando a equação (50b) e a equação (52c), chega-se;

$$u_d = Na(\xi, \eta)A^{-1}d \tag{53a}$$

Fazendo $N(\xi, \eta) = Na(\xi, \eta)A^{-1}$, obtém-se;

$$u_d = N(\xi, \eta)d \tag{53b}$$

Em que N é chamada matriz de funções de forma do elemento. Esta, possui o formato apresentado na equação (54).

$$N(\xi, \eta) = \begin{bmatrix} N_1(\xi, \eta) & 0 & N_2(\xi, \eta) & 0 & N_3(\xi, \eta) & 0 & N_4(\xi, \eta) & 0 \\ 0 & N_1(\xi, \eta) & 0 & N_2(\xi, \eta) & 0 & N_3(\xi, \eta) & 0 & N_4(\xi, \eta) \end{bmatrix} \tag{54}$$

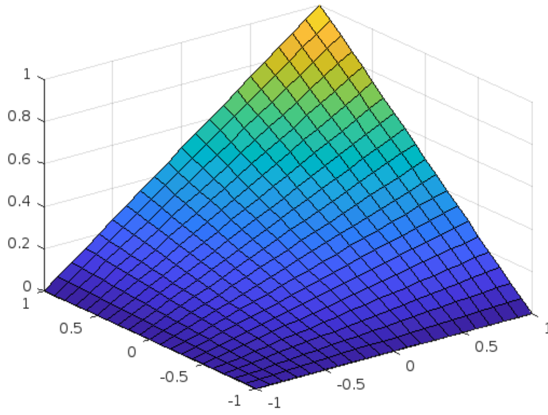
As funções $N_i(\xi, \eta)$ são conhecidas como funções de interpolação. Estas fazem o mapeamento de um ponto no plano paramétrico ξη para um ponto representado no plano cartesiano xy como sugere a figura (13). As funções $N_i(\xi, \eta)$ chamam atenção, pois estas valem 1 no i e 0 nos demais como sugere a figura (14). Combinando as equações (53b) e (54), é possível escrever;

$$x(\xi, \eta) = \sum_{i=1}^{4} N_i(\xi, \eta)x_i$$
$$y(\xi, \eta) = \sum_{i=1}^{4} N_i(\xi, \eta)y_i \tag{55}$$

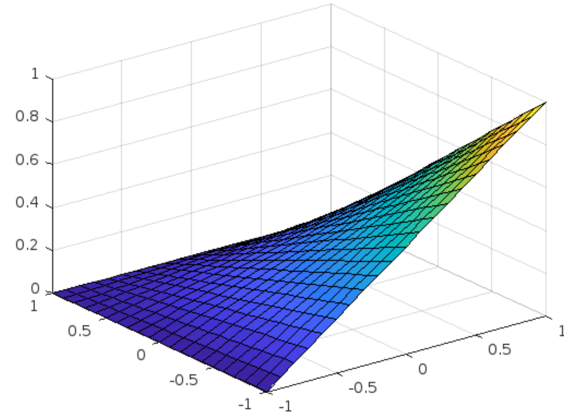
Segundo Silva (2022), a relação entre as coordenadas de um ponto no sistema natural e local é feita através de uma interpolação bilinear com o uso das equações (55), em que as funções $N_i(\xi, \eta)$ são apresentadas na figura (14).

Figura (14): Funções de forma

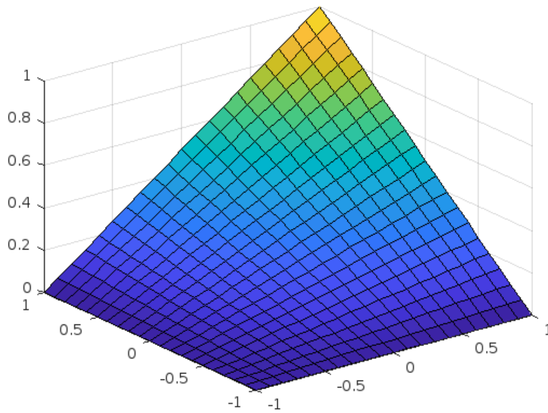
$$N_1(\xi,\eta) = \frac{1}{4}\,(1-\xi)\,(1-\eta)$$



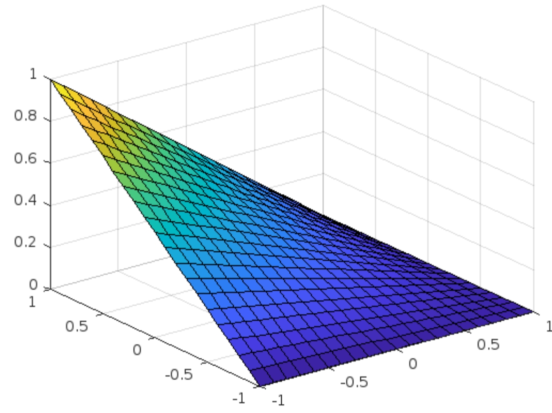
$$N_2(\xi,\eta) = \frac{1}{4}\,(1+\xi)\,(1-\eta)$$



$$N_3(\xi,\eta) = \frac{1}{4}\,(1+\xi)\,(1+\eta)$$



$$N_4(\xi,\eta) = \frac{1}{4}\,(1-\xi)\,(1+\eta)$$



Fonte: Elaborado pelo próprio autor.

Seja uma função φ inscrita inicialmente em termos das variáveis x e y. A equação (56) a seguir refere-se a transformação que relaciona as variáveis xy as variáveis ξη.

$$\varphi(x(\xi,\eta), y(\xi,\eta)) = \varphi(\xi,\eta) \tag{56}$$

Segundo Thomas (2013), a relação entre as derivadas de φ com respeito às coordenadas cartesianas e as coordenadas paramétricas é estabelecida pela regra da cadeia;

$$\frac{\partial \varphi}{\partial \xi} = \frac{\partial \varphi}{\partial x}\frac{\partial x}{\partial \xi} + \frac{\partial \varphi}{\partial y}\frac{\partial y}{\partial \xi}$$
$$\frac{\partial \varphi}{\partial \eta} = \frac{\partial \varphi}{\partial x}\frac{\partial x}{\partial \eta} + \frac{\partial \varphi}{\partial y}\frac{\partial y}{\partial \eta} \quad (57)$$

Matricialmente, tem-se;

$$\begin{Bmatrix} \frac{\partial \varphi}{\partial \xi} \\ \frac{\partial \varphi}{\partial \eta} \end{Bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{Bmatrix} \frac{\partial \varphi}{\partial x} \\ \frac{\partial \varphi}{\partial y} \end{Bmatrix} \quad (58)$$

Com isso, define-se a matriz jacobiana.

$$J(\xi, \eta) = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \quad (59)$$

Usando as equações (55), escreve-se:

$$J(\xi, \eta) = \begin{bmatrix} \sum \frac{\partial N_i(\xi,\eta)}{\partial \xi} x_i & \sum \frac{\partial N_i(\xi,\eta)}{\partial \xi} y_i \\ \sum \frac{\partial N_i(\xi,\eta)}{\partial \eta} x_i & \sum \frac{\partial N_i(\xi,\eta)}{\partial \eta} y_i \end{bmatrix} \quad (60a)$$

Com $i = 1, 2, ... 4$. Ou ainda;

$$J(\xi,\eta) = \begin{bmatrix} \frac{\partial N_1(\xi,\eta)}{\partial \xi} & \frac{\partial N_2(\xi,\eta)}{\partial \xi} & \frac{\partial N_3(\xi,\eta)}{\partial \xi} & \frac{\partial N_4(\xi,\eta)}{\partial \xi} \\ \frac{\partial N_1(\xi,\eta)}{\partial \eta} & \frac{\partial N_2(\xi,\eta)}{\partial \eta} & \frac{\partial N_3(\xi,\eta)}{\partial \eta} & \frac{\partial N_4(\xi,\eta)}{\partial \eta} \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix} \tag{60b}$$

Sucintamente, escreve-se, conforme apresentado por Vaz (2011);

$$J(\xi,\eta) = DNx(\xi,\eta)X \tag{60c}$$

Como pode-se observar na equação (64.c), a matriz $DNx(\xi, \eta)$ contém as derivadas das funções N_i com $i = 1, 2, ... 4$ ao passo em que a matriz X contém as coordenadas nodais do elemento em coordenadas cartesianas.

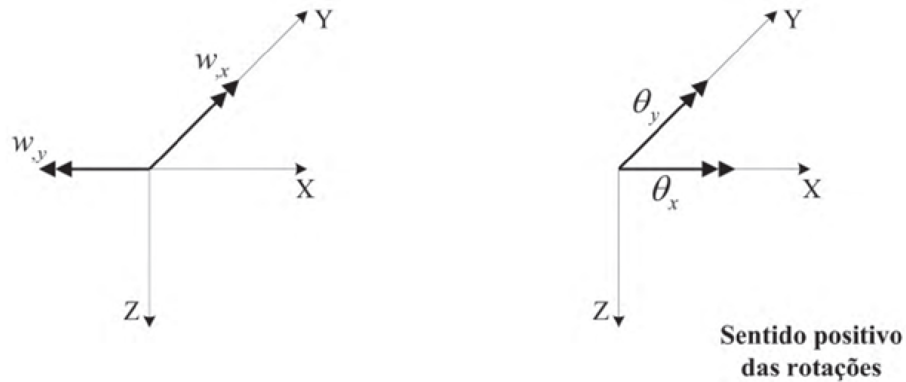
2.4.2 Elemento retangular de placa fina adotado

Para Oñate (2013), uma maneira intuitiva, de satisfazer o requisito de continuidade da função interpoladora de w, é escolher a deflexão e as duas rotações como variáveis nodais. Para a formulação do elemento de placa fina, Vaz (2011) apresenta as seguintes relações entre as rotações e as derivadas de $w(x, y)$.

$$\begin{Bmatrix} \theta_x \\ \theta_y \end{Bmatrix} = \begin{Bmatrix} -w_y \\ w_x \end{Bmatrix} \tag{61}$$

Na equação (61) o sinal negativo é atribuído na rotação em x devido a convenção de sinais das rotações apresentadas pelo autor. Essa convenção de sinais é baseada na figura (15).

Figura (15): Relação entre as derivadas de w e rotações



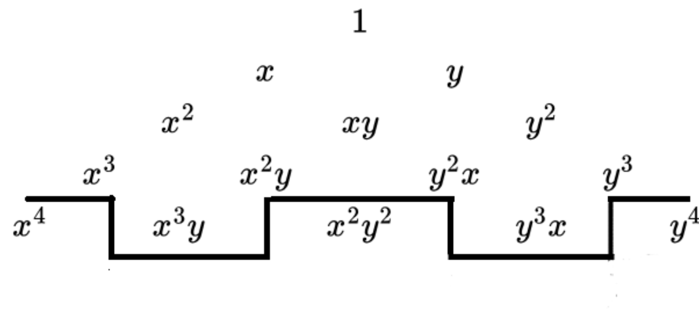
Fonte: Vaz, 2011.

De acordo com Oñate (2013), o elemento de placas finas contém, a princípio, três graus de liberdade por nó. Isso significa que, para um elemento quadrilateral, são contabilizados 12 graus de liberdade no total. O vetor d_i da equação (62) a seguir, apresenta os deslocamentos nodais de cada um dos quatro nós do elemento representados pelo subíndice i na expressão (Vaz, 2011).

$$d_i = \begin{Bmatrix} w_i \\ \theta_{x_i} \\ \theta_{y_i} \end{Bmatrix} = \begin{Bmatrix} w_i \\ -w_{y_i} \\ w_{x_i} \end{Bmatrix} \quad (62)$$

Como o vetor de deslocamentos apresenta 12 graus de liberdade por elemento, a função que aproxima a deflexão do elemento de placa deve possuir 12 termos. Nesse caso, segundo Oñate (2013), é impossível escolher um polinômio completo para descrever w. Sendo assim, três termos do polinômio de ordem 4 serão omitidos. A escolha dos termos é feita pelo triângulo de pascal apresentado na figura (16).

Figura (16): Triângulo de Pascal



Fonte: Oñate, 2013. Adaptado

Com base nisso, o polinômio que aproxima w com seus 12 termos tem a seguinte forma:

$$w\left(x,y\right)=a_1+a_2x+a_3y+a_4x^2+a_5xy+a_6y^2+a_7x^3+a_8x^2y+a_9y^2x+a_{10}y^3+a_{11}x^3y+a_{12}yx^3 \quad (63a)$$

Matricialmente, tem-se:

$$w\left(x,y\right)=\left\{1 \quad x \quad y \quad x^2 \quad xy \quad y^2 \quad x^3 \quad x^2y \quad y^2x \quad y^3 \quad x^3y \quad y^3x\right\}\left\{\begin{matrix} a_1 \\ a_1 \\ \vdots \\ a_{12} \end{matrix}\right\} \quad (63b)$$

Ou sucintamente,

$$w\left(x,y\right)=Na(x,x)a \quad (63c)$$

Segundo Vaz (2011), a opção de adotar os termos x^3y e y^3x deve-se ao fato de estes conterem simultaneamente as variáveis x e y e também por estabelecerem uma relação de simetria, o que não poderia ser obtido fazendo outra combinação com os outros termos restantes.

Além disso, Logan (2012) argumenta que esses termos garantem que haja continuidade nos deslocamentos entre os elementos. Porém, a continuidade das derivadas de w não é garantida.

Aplicando os deslocamentos nodais pela expressão (62) na equação (63b), têm-se:

$$d = Aa \tag{64}$$

Em que A é a matriz a seguir.

$$A = \begin{bmatrix} 1 & x_1 & y_1 & x_1^2 & x_1y_1 & y_1^2 & x_1^3 & x_1^2y_1 & y_1^2x_1 & y_1^3 & y_1^3x_1 & x_1^3y_1 \\ 0 & 0 & -1 & 0 & -x_1 & -2y_1 & 0 & -x_1^2 & -2x_1y_1 & -3x_1^2 & -x_1^3 & -3x_1y_1^2 \\ 0 & 1 & 0 & 2x_1 & y_1 & 0 & 3x_1^2 & 2x_1y_1 & y_1^2 & 0 & 3x_1^2y_1 & y_1^3 \\ 1 & x_2 & y_2 & x_2^2 & x_2y_2 & y_2^2 & x_2^3 & x_2^2y_2 & y_2^2x_2 & y_2^3 & y_2^3x_2 & x_2^3y_2 \\ 0 & 0 & -1 & 0 & -x_2 & -2y_2 & 0 & -x_2^2 & -2x_2y_2 & -3x_2^2 & -x_2^3 & -3x_2y_2^2 \\ 0 & 1 & 0 & 2x_2 & y_2 & 0 & 3x_2^2 & 2x_2y_2 & y_2^2 & 0 & 3x_2^2y_2 & y_2^3 \\ 1 & x_3 & y_3 & x_3^2 & x_3y_3 & y_3^2 & x_3^3 & x_3^2y_3 & y_3^2x_3 & y_3^3 & y_3^3x_3 & x_3^3y_3 \\ 0 & 0 & -1 & 0 & -x_3 & -2y_3 & 0 & -x_3^2 & -2x_3y_3 & -3x_3^2 & -x_3^3 & -3x_3y_3^2 \\ 0 & 1 & 0 & 2x_3 & y_3 & 0 & 3x_3^2 & 2x_3y_3 & y_3^2 & 0 & 3x_3^2y_3 & y_3^3 \\ 1 & x_4 & y_4 & x_4^2 & x_4y_4 & y_4^2 & x_4^3 & x_4^2y_4 & y_4^2x_4 & y_4^3 & y_4^3x_4 & x_4^3y_4 \\ 0 & 0 & -1 & 0 & -x_4 & -2y_4 & 0 & -x_4^2 & -2x_4y_4 & -3x_4^2 & -x_4^3 & -3x_4y_4^2 \\ 0 & 1 & 0 & 2x_4 & y_4 & 0 & 3x_4^2 & 2x_4y_4 & y_4^2 & 0 & 3x_4^2y_4 & y_4^3 \end{bmatrix} \tag{65}$$

Reescrevendo a equação (64), tem-se

$$a = A^{-1}d \tag{66}$$

Substituindo a equação (63.c) na equação (66), tem-se:

$$w(x,y) = Na(x,y)A^{-1}d \tag{67a}$$

O produto $Na(x,y)A^{-1}$ pode ser representado por uma única matriz $N(x,y)$ e é chamada matriz de funções de forma (Oñate, 2013). Substituindo na equação (67a), tem-se:

$$w(x,y) = N(x,y)d \tag{67b}$$

Combinando as equações (4), (5) e (7) com as equações (28), (29) e (30), tem-se a seguinte expressão das deformações inscritas em temos da função w:

$$\begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix} = \begin{Bmatrix} u_x \\ v_y \\ u_y + v_x \end{Bmatrix} = -z \begin{Bmatrix} w_x \\ w_y \\ 2w_{xy} \end{Bmatrix} \tag{68a}$$

De forma sucinta, tem-se;

$$\varepsilon(x,y,z) = -zk_k \tag{68b}$$

Na expressão (68b), k_k é o vetor que contém as curvaturas da teoria de Kirchhoff referentes a um ponto genérico no plano médio da placa. Substituindo a equação (63b) na equação (62), obtém-se:

$$\begin{Bmatrix} \varepsilon_x \\ \varepsilon_x \\ \gamma_{xy} \end{Bmatrix} = -z \begin{bmatrix} 0 & 0 & 0 & 2 & 0 & 0 & 6x & 2y & 0 & 0 & 6xy & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 2x & 6y & 0 & 6xy \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4x & 4y & 0 & 6x^2 & 6y^2 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{12} \end{Bmatrix} \tag{69a}$$

Simplificando, tem-se;

$$\varepsilon(x, y, z) = -zQ(x, y)a \tag{69b}$$

Fazendo o uso da equação (66), chega-se a seguinte expressão:

$$\varepsilon(x, y, z) = -zQ(x, y)A^{-1}d \tag{69c}$$

O produto $Q(x, y)A^{-1}$ pode ser apresentado como uma única matriz $B(x, y)$ chamada de matriz de compatibilidade cinemática. Esta relaciona as deformações com o deslocamentos nodais d (Vaz, 2011). Reescrevendo a equação (69.c), tem-se:

$$\varepsilon(x, y, z) = -zB(x, y)d \tag{69d}$$

Para encontrar o vetor das tensões basta multiplicar o vetor das deformações pela matriz constitutiva C da equação (19b). Sendo assim, tem-se:

$$\sigma(x, y, z) = -zCB(x, y)d \tag{70}$$

Para a formulação do problema, aplica-se o princípio dos trabalhos virtuais (Vaz, 2011). Para isso, escreve-se o vetor das deformações virtuais $\delta\varepsilon$ em função do vetor de deslocamentos nodais virtuais δd da seguinte maneira:

$$\delta\varepsilon(x, y, z) = -zB(x, y)\delta d \tag{71}$$

Segundo Vaz (2011), o princípio dos trabalhos virtuais fornece a seguinte relação:

$$\int_v \delta\epsilon^T \sigma dv = \delta d^T f \tag{72}$$

Na equação acima, f é o vetor de cargas nodais. Nesse trabalho, considera-se apenas cargas aplicadas nos nós dos elementos. Substituindo as equações (70) e (71) na equação (72), encontra-se:

$$\delta d^T \int_v z^2 B(x,y)^T C B(x,y) dv d = \delta d^T f \tag{73}$$

Essa equação deve ser válida para todos os campos de deslocamentos virtuais. Explicitando-se a integral da variável z, da equação (73), obtém-se

$$\int_A \int_{-e/2}^{e/2} z^2 B(x,y)^T C B(x,y) dz dA.d = f \tag{74}$$

Em que A é a área do elemento e e é a sua espessura.

Fazendo:

$$K = \int_A \int_{-e/2}^{e/2} z^2 B(x,y)^T C B(x,y) dz dA \tag{75}$$

Reescreve-se a equação (74) da seguinte maneira;

$$Kd = f \tag{76}$$

Em que K é a matriz de rigidez do elemento. Fazendo $B(x,y) = Q(x,y)A^{-1}$ na equação (75), tem-se;

$$K = \int_A \int_{-e/2}^{e/2} z^2 (A^{-1})^T Q(x,y)^T C Q(x,y) A^{-1} dz dA \quad (77)$$

Integrando na variável z, tem-se:

$$K = \int_A \left(A^{-1}\right)^T Q(x,y)^T D Q(x,y) A^{-1} dA \quad (78)$$

Fazendo:

$$K_{op}(x,y) = Q(x,y)^T D Q(x,y) \quad (79)$$

Obtêm-se

$$K = \left(A^{-1}\right)^T K_o(x,y) A^{-1} \quad (80)$$

Em que Fazendo uma mudança de variável como discutido em Thomas (2013), com uso da matriz Jacobiana discutida na seção 2.4.1 e utilizando as equações (55) e (59), a matriz K_o é representada pela equação (82) a seguir;

$$K_o(\xi,\eta) = \int_{-1}^{1} \int_{-1}^{1} K_{op}(x(\xi,\eta), y(\xi,\eta)) \left\| J(x(\xi,\eta), y(\xi,\eta)) \right\| d\xi d\eta \quad (82)$$

Segundo Vaz (2011), a integral acima pode ser resolvida pelo método de Gauss caso a espessura e seja constante. Se o elemento é retangular de dimensões $2a\ x\ 2b$, Melosh (1990) apresentou uma maneira eficaz para resolução numérica da integral da equação (82). Ao invés de utilizar as equações (55), sugere a seguinte substituição como apresentado em Vaz (2011):

$$x(\xi,\eta) = a + a\xi$$
$$y(\xi,\eta) = b + b\eta \tag{83}$$

Para obter as equações (83), Melosh (1990) simplesmente derivou das funções de forma da equação (55). Nesse caso, a matriz jacobiana será então;

$$J(\xi,\eta) = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \tag{84}$$

O determinante da matriz jacobiana gerado pela substituição das equações (83) propostas por Melosh (1990) é, portanto, sempre constante como apresentado a seguir;

$$Det_J(\xi,\eta) = ab \tag{85}$$

Contudo, esse procedimento de Melosh (1990) funciona apenas para elementos retangulares, não permitindo a utilização de elementos distorcidos utilizados em malhas de elementos finitos mais genéricas. Ele foi implementado no trabalho de Vaz (2011), Logan (1976) e também em Oñate (2013).

Neste trabalho, optou-se por adotar a versão completa da matriz jacobina para a transformação apresentada na seção 2.4.1. Isso significa que, para a mudança de variável, ao invés de utilizar as equações (83) como sugerido por Melosh (1990), optou-se pelo uso das equações (55). Assim, o programa desenvolvido foi capaz de analisar problemas com malhas distorcidas, utilizadas para ajustar-se a domínios de estudo mais gerais como discutidos posteriormente.

Para montagem da matriz de rigidez global, utiliza-se o vetor de graus de liberdade de cada elemento GDL. Como cada elemento possui 4 nós e cada nó é atribuído três graus de liberdade, esse vetor pode ser representado como uma matriz coluna com 12 elementos. As três primeiras linhas dessa matriz são os graus de liberdade do nó 1. As próximas três do nó 2 e assim sucessivamente. Esse vetor foi implementado baseando-se no modelo a seguir.

$$GDL(e) = \begin{Bmatrix} 3conec_{e1} - 2 \\ 3conec_{e1} - 1 \\ 3conec_{e1} - 0 \\ 3conec_{e2} - 2 \\ 3conec_{e2} - 1 \\ 3conec_{e2} - 0 \\ 3conec_{e3} - 2 \\ 3conec_{e3} - 1 \\ 3conec_{e3} - 0 \\ 3conec_{e4} - 2 \\ 3conec_{e4} - 1 \\ 3conec_{e4} - 0 \end{Bmatrix} \tag{86}$$

Em que $conec$ é a matriz de conectividade e, portanto, o elemento $conec_{ij}$ representa o nó j do elemento i. No tópico 3 deste trabalho será apresentado detalhes da montagem dessa matriz. Uma vez encontrada a matriz de rigidez global de cada elemento, utiliza-se o vetor de graus de liberdade para montar a matriz de rigidez global K_g da placa. Em seguida, implementa-se o vetor de cargas globais e aplica-se na equação (87a).

$$K_g d_g = f_g \tag{87.a}$$

Em que d_g representa o vetor de deslocamentos globais da placa. Para obtê-lo, basta manipular a equação (87).

$$d_g = K_g^{-1} f_g \tag{87b}$$

Uma vez obtido o vetor de deslocamentos globais, com o vetor GDL é possível calcular o vetor de deslocamentos nodais de cada elemento em particular d_e. É através do vetor $GDL(e)$ que é possível capturar os deslocamentos e rotações no vetor de deslocamentos globais d_g de um elemento em particular.

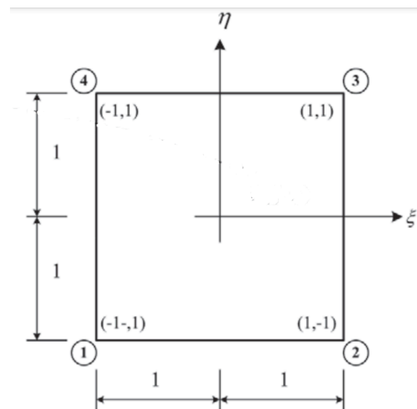
O vetor $GDL(e)$ é, portanto, responsável por localizar os deslocamentos e rotações apenas do elemento e, e, através de uma estrutura de repetição simples, os valores de deslocamento e rotação são armazenados em uma matriz coluna contendo todos os deslocamentos e rotações de cada nó do elemento e. Essa matriz é chamada de matriz $d_e(e)$. A programação referente a montagem do vetor de deslocamentos nodais de cada elemento foi formulada no tópico 2.9 da programação apresentada no anexo A.

Utilizando a matriz da equação (19b) e a matriz de compatibilidade cinemática, vetor dos momentos, por sua vez, pode ser obtido, mediante a seguinte operação;

$$M_k(e,\xi,\eta) = -D(e)B(e,\xi,\eta)d_e(e) \tag{88}$$

Em que $M_k(e, \xi, \eta)$ representa o vetor dos momentos obtidos pela teoria de Kirchhoff . Para a obter o vetor dos momentos em nó específico de um elemento, basta entrar com as coordenada $\xi\eta$ que representa o nó e especificar o elemento. Os pares ordenados (ξ, η) associados a cada nó estão representados na figura (17).

Figura (17): Identificação dos nós dos elementos no plano paramétrico



Fonte: Vaz, 2011. Adaptado

2.5 Integração numérica

Segundo Soriano (2003), as integrações necessárias para a obtenção da matriz de rigidez nos casos em que a geometria dos elementos são obtidas pela interpolação das coordenadas nodais são muito elaboradas e impraticáveis para se realizar analiticamente. Para integração, o método de Gauss-Legendre é um método simples, em que a integração é realizada através de uma soma da aplicação direta de pontos e pesos específicos na função integrando. Para a integração $2x2$, nas variáveis ξ e η, temos os pesos (Equação (89a)) e os pontos de Gauss (Equação (89b)) apresentados a seguir.

$$\xi = \begin{Bmatrix} -\frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{Bmatrix} \qquad \eta = \begin{Bmatrix} -\frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{Bmatrix} \tag{89a}$$

$$w = \begin{Bmatrix} 1 \\ 1 \end{Bmatrix} \tag{89b}$$

Sendo assim, segundo Vaz (2011), a integração é dada pela equação (89a).

$$\int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta)d\xi d\eta = w_1w_1f(\xi_1,\eta_1)+w_1w_2f(\xi_1,\eta_2)+w_2w_1f(\xi_2,\eta_1)+w_2w_2f(\xi_2,\eta_2) \quad \text{(90a)}$$

Ou sucintamente;

$$\int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta)d\xi d\eta = \sum_{i=1}^{2}\sum_{j=1}^{2} w_iw_jf(\xi_i,\eta_j) \quad \text{(90b)}$$

2.6 Análise de convergência

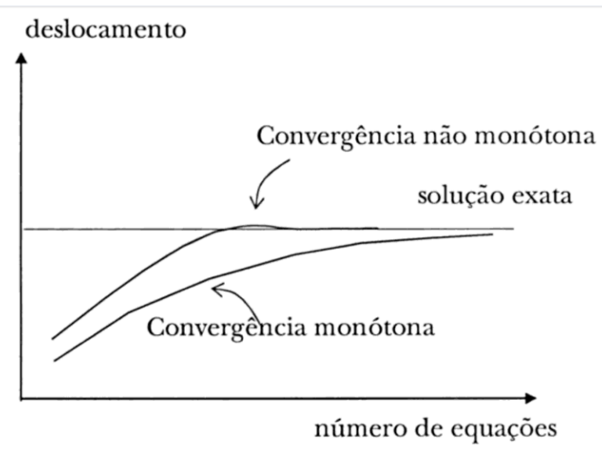
Uma questão relevante a ser considerada no estudo de elementos finitos é identificar se o campo de deslocamento de um elemento converge para a solução exata à medida que se refinam as malhas. De maneira geral, esse refinamento pode acontecer por redução das dimensões dos elementos ou por aumento da ordem do campo de deslocamento dos elementos ou ainda combinando ambos os efeitos (Soriano, 2013).

Cabe ressaltar ainda que a solução exata de um problema por convergência de elementos finitos está associada à minimização da energia potencial total que, segundo Logan (2012), é representado pela equação (90) a seguir.

$$U = \frac{1}{2}\int \left(\sigma_x\varepsilon_x + \sigma_y\varepsilon_y + \tau_{xy}\gamma_{xy}\right) dV \quad \text{(91)}$$

Ainda segundo Soriano (2013), em se tratando de elementos conformes, é de se esperar que haja uma convergência monotônica de deslocamentos para valores inferiores. A figura (18) apresenta duas formas de convergência de um elemento finito.

Figura (18): Convergência mediante ao refino da malha.



Fonte: Soriano, 2013.

3 METODOLOGIA: IMPLEMENTAÇÃO VIA ELEMENTOS FINITOS

Nesta seção é discutida a metodologia utilizada para a implementação do elemento finito de placas finas.

3.1 Linguagem Python

O python é uma linguagem de programação desenvolvida nos anos 1990 por Guido Van Rossum, totalmente gratuita e não precisa passar por processo de compilação uma vez que os comandos e códigos são digitados e executados diretamente. Outra vantagem da linguagem é que esta dispõe de várias ferramentas e bibliotecas que dão suporte a linguagem, sendo oferecidas gratuitamente (Alves, 2021). A sintaxe é rápida e eficiente, ou seja, com poucas linhas de código é possível realizar tarefas complexas.

3.2 Descrição do código desenvolvido

O objetivo do código implementado neste trabalho é a obtenção dos esforços internos e da flecha apresentada em lajes que podem ser modeladas usando a teoria de placas finas apresentadas no tópico 2.2 deste trabalho. O elemento de placa utilizado é baseado no elemento apresentado em Vaz (2011), Logan (2012) e também citado por em Oñate (2013). A diferença básica entre a formulação apresentada pelos autores e a implementada neste trabalho é a mudança de variável discutida em 2.4.2. Dessa maneira, a matriz jacobiana considerada no cálculo é aquela apresentada pela equação (59). O código, apresenta 3 tópicos ou fases básicas, como mostra a figura (19) extraída diretamente da plataforma *Google Colaboratory*, plataforma utilizada para escrever o código .

Figura (19): Página inicial do código inscrito na plataforma *Google Colaboratory*

```
[ ]  1 ### IMPORTAÇÕES
     2 import numpy as np
     3 import math
     4 from scipy.sparse.linalg import spsolve
     5 from scipy.sparse import csc_matrix
```

> **1 - DADOS DE ENTRADA DOS PROBLEMAS**

[ ] ↳ 38 células ocultas

> **2 - PROCESSAMENTO**

[ ] ↳ 36 células ocultas

> **3 - ARQUIVO VTU**

Fonte: Elaborado pelo próprio autor.

Na imagem, destaca-se também todas as importações de bibliotecas utilizadas para a escrita do código. Destaca-se a biblioteca *numpy*, que possui diversos recursos de montagem e operações algébricas com vetores e matrizes. A biblioteca *math* auxilia na escrita e operação de funções e *scipy* é usada exclusivamente para a resolução dos sistemas de equações. Esta última, oferece uma vantagem em termos de tempo de processamento do código, pois a resolução se dá por sistemas de matrizes esparsas. Isso diminui consideravelmente o tempo de processamento dos problemas.

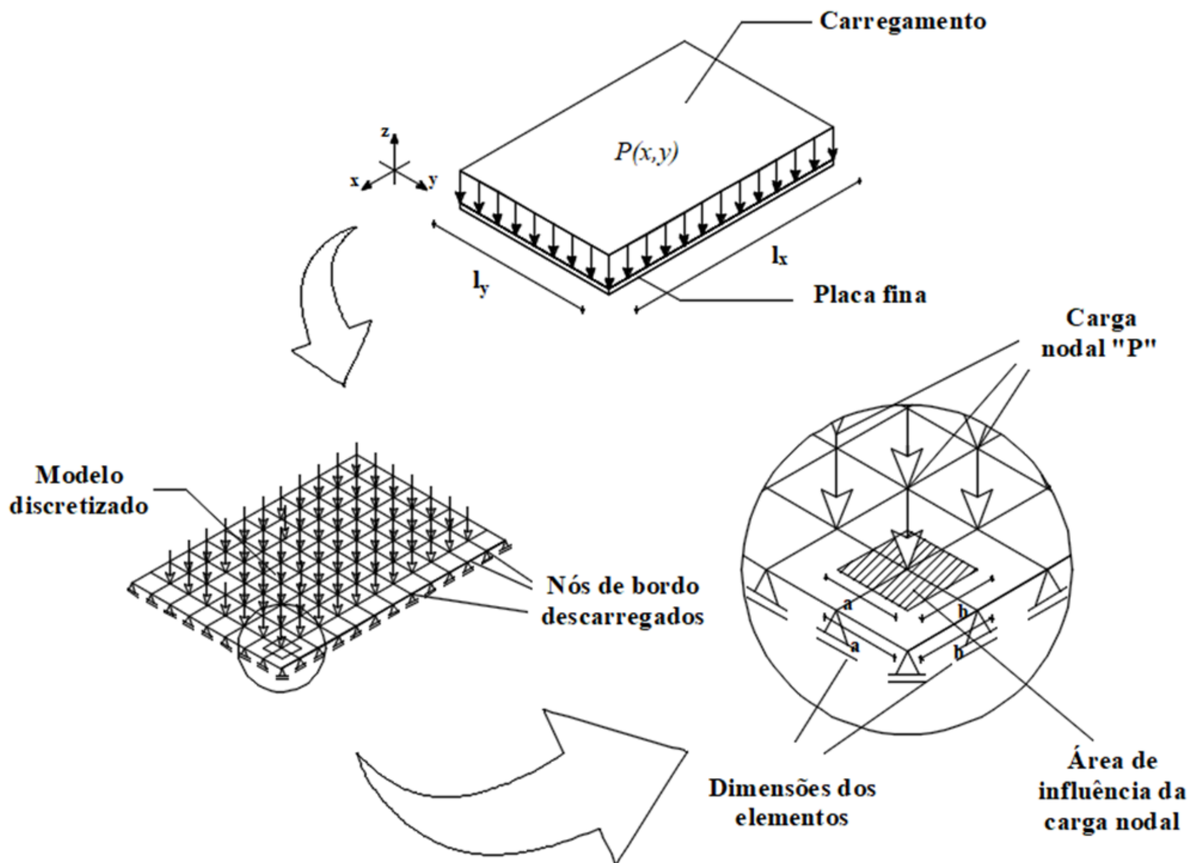
3.2.1 Dados de entrada dos problemas -(Item 1 do código em anexo)

Neste tópico, são inseridos os dados que caracterizam cada problema. São inseridos, por exemplo, a matriz de coordenadas nodais que define geometria do domínio de estudo, bem como quantidade de nós e elementos da malha. Além disso, são inseridas as condições de contorno em cada bordo das lajes através do vetor de graus de liberdade restringidos. Por fim, são computados também, o vetor de cargas externas e a matriz de conectividade.

Modelagem dos problemas

Para a modelagem dos problemas utilizando o elemento de placa,o domínio do problema é particionado em elementos menores como mostra a figura (20).

Figura (20): Elaboração do modelo em elementos finitos



Fonte: Elaborado pelo próprio autor.

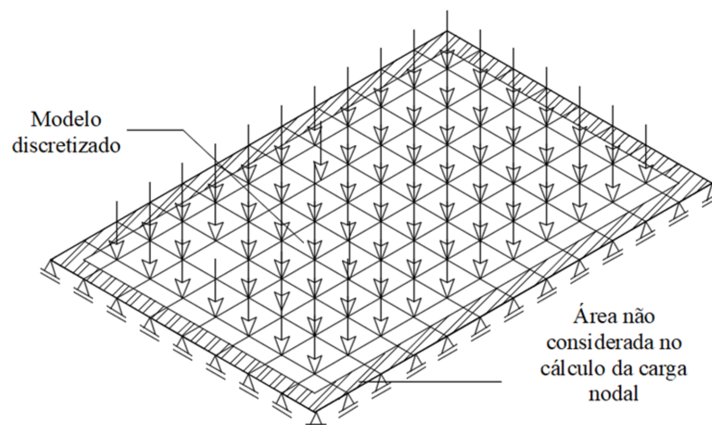
Como observado na figura (20), o problema é tratado como um modelo em elementos finitos, com aplicação do carregamento em forma de carga concentrada nos nós da malha. Para este trabalho, é importante frisar que não são aplicados carregamentos nos bordos quando estes são apoiados ou engastados, uma vez que, teoricamente, adota-se que esses nós não se movimentam na vertical, exceto no caso de borda livre.

Por essa razão, por mais que a malha seja refinada, esta nunca estará sob o carregamento completo, mas sim, cada vez mais próximo dele. Em outras palavras, para efeito de cálculo, apenas contribuem para a deformação da placa as cargas aplicadas aos nós internos da malha. Seguindo esse raciocínio, um caso crítico seria imaginar uma malha análoga àquela da figura (22), porém com apenas 2 elementos. Nesse caso, não haveria nós internos à malha, não sendo possível, nesse caso, a aplicação de carregamentos nodais.

Intuitivamente, é possível esperar os deslocamentos e as rotações, em caso de convergência, que esta seja monótona, ou seja, espera-se que a convergência ocorra por valores inferiores, uma vez que, a medida que a malha é refinada, esta vai ficando mais carregada e portanto, espera-se maiores valores de rotações e flecha.

Neste trabalho, para o cálculo da carga nodal, é aplicada uma metodologia simplificada. Para o caso de uma placa submetida a um carregamento uniformemente distribuído $P(x, y) = p_o$, a carga aplicada em cada nó da malha será, portanto, $P = p_o ab$, em que o produto ab é a área de influência da carga nodal, sendo esta, a área do próprio elemento (Figura 20). Caso a malha seja composta por diferentes elementos, a área de influência da carga nodal é a média das áreas dos elementos. Devido ao uso dessa metodologia, para o cálculo da carga nodal, nunca a área do domínio é utilizada integralmente, restando sempre uma área não utilizada no cálculo como apresentado na figura (21).

Figura (21): Aplicação das cargas nodais



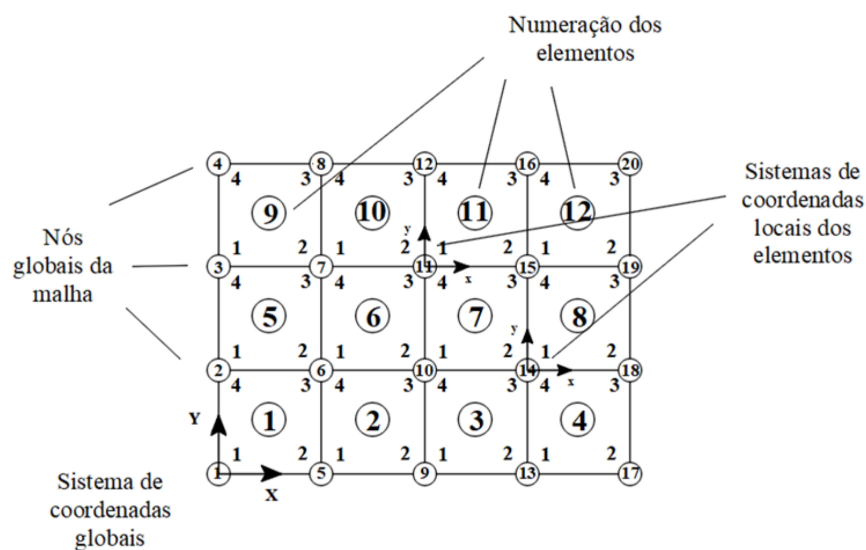
Fonte: Elaborado pelo próprio autor.

Matriz de coordenadas nodais e matriz de conectividade

Em uma malha de elementos finitos, todos os nós da malha tem sua identificação numérica e são armazenados no vetor de coordenadas nodais "*coord*". A depender da geometria da malha, a origem do sistema de coordenadas é posicionada onde for mais conveniente, de modo a facilitar a coleta da abscissa e ordenada de cada nó da malha. Todos os problemas tratados aqui, toda a malha foi posicionada no primeiro quadrante do plano xy, ou seja, as coordenadas nodais são sempre positivas.

Uma vez posicionada a malha no sistema cartesiano, é feita a numeração dos nós globais da malha. Essa numeração pode ser realizada da maneira que for mais conveniente, desde que facilite na identificação e na coleta dos dados. Neste trabalho, a numeração dos nós globais segue um padrão e foi inspirada na numeração utilizada por Vaz (2011). É crescente da esquerda para a direita e de baixo para cima a partir de onde foi adotada a origem. A numeração dos elementos também é crescente da esquerda para a direita e de baixo para cima a partir da escolha do primeiro elemento da malha. A figura (22) apresenta um domínio subdividido em 12 elementos, em que mostra a posição da origem do sistema global XY, bem como a numeração dos nós e elementos.

Figura (22): Numeração dos elementos, nós globais e locais da malha

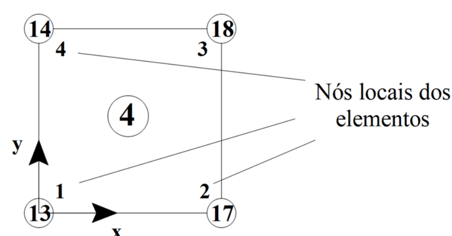


Fonte: Elaborado pelo próprio autor.

A matriz de coordenadas nodais é a matriz que possui a abscissa e a ordenada de cada nó da malha. Esta, portanto, é uma matriz com duas colunas e "*nnos*" linhas. Nesse caso, "*nnos*" representa o número total de nós da malha. Na primeira linha da matriz, está as coordenadas do nó 1, na segunda linha está as coordenadas no nó 2, na terceira as do nó 3 e assim sucessivamente.

Cada elemento da malha é, portanto, representado por 4 nós globais. Cada nó do elemento é atribuída uma numeração local. Segundo Vaz (2011), a numeração local de cada elemento deve respeitar a uma distribuição anti-horária. Isso é importante para que o determinante da matriz jacobiana seja sempre positivo. Com isso, o módulo do determinante da da matriz jacobiana da equação (82) pode ser removido para efeito de cálculo.

Figura (23): Numeração local do elemento



Fonte: Elaborado pelo próprio autor.

A figura (23) apresenta a numeração local de um elemento genérico de uma malha de elementos finitos, em que os nós são numerados no sentido anti-horário. Todos os elementos da malha no código tem sua numeração local conforme indica a figura (23). A matriz de conectividade utiliza essa numeração para associar cada nó local a um nó global correspondente de cada elemento.

Condições de contorno vetor de graus de liberdade restringidos

Para a aplicação das condições de contorno dos problemas é utilizado o vetor de graus de liberdade restringidos GLR. Inicialmente, coleta-se em um vetor coluna com os nós da malha que serão restringidos. Geralmente são nós de bordo, ou nós internos que representam as vigas ou pilares da estrutura. Em seguida, são calculados os graus de liberdade dos nós associados à condição de contorno que será aplicada. Em seguida, é criado um vetor apenas com o grau de liberdade associados às condições de contorno que se deseja aplicar. Esse vetor é chamado de vetor de graus de liberdades restringidos.

A princípio, os graus de liberdades de cada nó de uma malha de elementos finitos são livres, ou seja, o nós transladam verticalmente e rotacionam em ambos os sentidos. Sendo assim, a aplicação das condições de contorno se resume em restringir e esses movimentos. Se a um conjunto de nós se deseja aplicar uma condição de contorno de engaste, por exemplo, todos os graus de liberdade desses nós serão restringidos e, para isso, todos ele devem constar no vetor GLR. Analogamente, para aplicar a um conjunto de nós a restrição de deslocamento, apenas o grau de liberdade de deslocamento deve constar no vetor GLR.

Vetor de cargas externas

O vetor de cargas externas contém os carregamentos e rotações oriundos de forças externas e forças de corpo. É um vetor coluna cuja quantidade de linhas equivale ao número de graus de liberdade da malha. Cada tipo de carregamento assume uma posição no vetor carga relacionado ao grau de liberdade de cada carregamento. Os graus de liberdade sem influência de carregamentos, aparecem no vetor de cargas como valores nulos.

3.2.2 Processamento - (Item 2 do código em anexo)

O tópico de processamento é responsável pela maioria das operações algébricas do programa. É sempre o mesmo para todos os problemas e é responsável, especialmente, por gerar o vetor de deslocamentos nodais da malha, bem como gerar os vetores de esforços internos da estrutura. É, portanto, a parte do código mais carregada em termos de processamento. É subdividido nos subtópicos a seguir;

Vetor dos dados de entrada - (Item 2.1 do código em anexo)

Nesse tópico, através de uma estrutura de repetição, são criados os vetores utilizados durante o processamento. Os vetores criados nesse tópico são, vetor de espessura dos elementos, vetor dos coeficientes de Poisson e o vetor dos módulos de elasticidade de cada elemento. Apesar de ser possível construir elementos com parâmetros variados, todos os elementos das malhas adotadas nos problemas deste trabalho possuem mesma espessura, mesmo módulo de elasticidade e coeficiente de Poisson. Esses parâmetros são fornecidos no tópico de dados de entrada e replicados a todos os elementos.

Coordenada dos nós dos elementos - (Item 2.2 do código em anexo)

Neste tópico, são calculados os vetores de coordenadas globais e locais. Utilizando os vetores de coordenadas globais é feita a construção da matriz "*coorde*", que contém as coordenadas nodais globais de cada elemento. Como dito anteriormente, cada elemento da malha é representado por 4 nós. Cada nó, por sua vez, tem sua coordenada global e sua coordenada local. Além disso, cada elemento tem sua identificação numérica na malha, de modo que seja possível identificá-lo. Tanto as coordenadas globais quanto as locais são montadas na forma de vetor em função de cada elemento no código como mostra a figura (24).

Figura (24): Montagem dos vetores de coordenadas globais

```
1 # VETOR DAS COORDENADAS X GLOBAIS:
2
3 def xg(e):
4   XG = np.array([[coord[int(conec[(e-1),0]-1), 0]],
5                  [coord[int(conec[(e-1),1]-1), 0]],
6                  [coord[int(conec[(e-1),2]-1), 0]],
7                  [coord[int(conec[(e-1),3]-1), 0]]])
8   return XG
9
10 # VETOR DAS COORDENADAS Y GLOBAIS:
11
12 def yg(e):
13   YG = np.array([[coord[int(conec[(e-1),0]-1), 1]],
14                  [coord[int(conec[(e-1),1]-1), 1]],
15                  [coord[int(conec[(e-1),2]-1), 1]],
16                  [coord[int(conec[(e-1),3]-1), 1]]])
17   return YG
```

Fonte: Elaborado pelo próprio autor.

A figura (24), é um recorte do tópico 2.2 do código. O vetor de coordenadas globais é, portanto, um vetor coluna que utiliza a matriz de coordenadas representada por "*coord*" e também pela matriz de conectividade "*conec*". Esta auxilia na captura das coordenadas dentro da matriz "*coord*" do elemento e. Portanto, no código, os vetores de coordenadas são inscritos em termos da variável e. O mesmo raciocínio é aplicado para a obtenção do vetor de coordenadas locais.

A matriz de coordenada dos elementos, por sua vez, é uma matriz com duas colunas e 4 linhas, tendo como entrada, os vetores de coordenada global definidos anteriormente como é representada no tópico código através da figura (25). Esta, é exatamente a matriz X, apresentada na equação (60c), sendo usada diretamente na obtenção da matriz jacobiana de cada elemento da malha.

Figura (25): Matriz de coordenadas nodais dos elementos.

```
36
37 # >>> Matriz de coordenadas dos elementos [coorde]
38
39 def coorde(e):    # Coordenadas globais
40   xG_val = xg(e)
41   yG_val = yg(e)
42   coorde = np.array([[xG_val[1 -1][0], yG_val[1 -1][0]],
43                      [xG_val[2 -1][0], yG_val[2 -1][0]],
44                      [xG_val[3 -1][0], yG_val[3 -1][0]],
45                      [xG_val[4 -1][0], yG_val[4 -1][0]]])
46   return coorde
47
```

Fonte: Elaborado pelo próprio autor.

Matrizes constitutivas do material - (Item 2.3 do código em anexo)

No código, as matrizes constitutivas do material foram inscritas também em função de cada elemento. A figura (26) apresenta a implementação dessas matrizes apresentadas na equação (19b), tópico 2.1.3 deste trabalho que se refere ao modelo constitutivo elástico linear.

Figura (26): Matrizes constitutivas

```
 1 # MATRIZ [D]:
 2 def D(e):
 3   e = e - 1 # Ajuste da numeração do elemento
 4   D = (t[e][0]**3*E[e][0])/(12*(1-v[e][0]**2)) * np.array([[1,       v[e][0], 0],
 5                                                           [v[e][0],  1,       0],
 6                                                           [ 0,       0,       (1-v[e][0])/2]])
 7   return D
 8
 9 #  MATRIZ [C]:
10 def C(e):
11   e = e - 1 # Ajuste da numeração do elemento
12   C = (E[e][0]/((1-v[e][0]**2))) * np.array([[1,       v[e][0], 0],
13                                             [v[e][0],  1,       0],
13                                             [ 0,       0,       (1-v[e][0])/2]])
15   return C
```

Fonte: Elaborado pelo próprio autor.

Formulação das matrizes “A” e “Q” - (Item 2.4 do código em anexo)

A matriz Q é usada para calcular o vetor das deformações como apresenta a equação (69b). Esta é originalmente inscrita em termos das variáveis xy. Porém, devido a transformação de coordenadas, também passa a ser inscrita em termos das variáveis ξη. Também é função de cada elemento, uma vez que o vetor de coordenadas nodais varia em função de cada elemento. A implementação no código está no tópico 2.4 e é apresentado pela figura (27).

Figura (27): Implementação da matriz Q

```
1 # MATRIZ [Q]:
2
3 def Q(e,  ξ , η):
4   x = xe1(e , ξ , η)
5   y = ye1(e , ξ , η)
6   Q = np.array([[ 0, 0, 0, 2, 0, 0, 6*x, 2*y,   0,   0,  6*x*y,      0],
7                 [ 0, 0, 0, 0, 0, 2,   0,   0, 2*x, 6*y,      0,  6*x*y],
8                 [ 0, 0, 0, 0, 2, 0,   0, 4*x, 4*y,   0, 6*x**2, 6*y**2]])
9   return Q
10
```

Fonte: Elaborado pelo próprio autor.

A matriz *A* apresentada pela equação (65), também foi implementada utilizando os vetores de coordenada local dos elementos. Isso significa que também é função de cada elemento.

Figura (28): Matriz A

```
11 # MATRIZ [A]:
12
13 # Matriz A de cada elemento.
14 def A(e):
15   x1, x2, x3, x4 = xl(e)[0][0], xl(e)[1][0], xl(e)[2][0], xl(e)[3][0]
16   y1, y2, y3, y4 = yl(e)[0][0], yl(e)[1][0], yl(e)[2][0], yl(e)[3][0]
17   A = np.array([[ 1, x1,  y1, x1**2, x1*y1, y1**2,   x1**3,  y1*x1**2,  x1*y1**2,    y1**3,   y1*x1**3,    x1*y1**3],
18                 [ 0,  0,  -1,     0,   -x1, -2*y1,       0,    -x1**2,  -2*x1*y1, -3*y1**2,     -x1**3, -3*x1*y1**2],
19                 [ 0,  1,   0,  2*x1,    y1,     0, 3*x1**2,   2*y1*x1,     y1**2,        0, 3*y1*x1**2,       y1**3],
20                 [ 1, x2,  y2, x2**2, x2*y2, y2**2,   x2**3,  y2*x2**2,  x2*y2**2,    y2**3,   y2*x2**3,    x2*y2**3],
21                 [ 0,  0,  -1,     0,   -x2, -2*y2,       0,    -x2**2,  -2*x2*y2, -3*y2**2,     -x2**3, -3*x2*y2**2],
22                 [ 0,  1,   0,  2*x2,    y2,     0, 3*x2**2,   2*y2*x2,     y2**2,        0, 3*y2*x2**2,       y2**3],
23                 [ 1, x3,  y3, x3**2, x3*y3, y3**2,   x3**3,  y3*x3**2,  x3*y3**2,    y3**3,   y3*x3**3,    x3*y3**3],
24                 [ 0,  0,  -1,     0,   -x3, -2*y3,       0,    -x3**2,  -2*x3*y3, -3*y3**2,     -x3**3, -3*x3*y3**2],
25                 [ 0,  1,   0,  2*x3,    y3,     0, 3*x3**2,   2*y3*x3,     y3**2,        0, 3*y3*x3**2,       y3**3],
26                 [ 1, x4,  y4, x4**2, x4*y4, y4**2,   x4**3,  y4*x4**2,  x4*y4**2,    y4**3,   y4*x4**3,    x4*y4**3],
27                 [ 0,  0,  -1,     0,   -x4, -2*y4,       0,    -x4**2,  -2*x4*y4, -3*y4**2,     -x4**3, -3*x4*y4**2],
28                 [ 0,  1,   0,  2*x4,    y4,     0, 3*x4**2,   2*y4*x4,     y4**2,        0, 3*y4*x4**2,       y4**3]])
29   return A
30
```

Fonte: Elaborado pelo próprio autor.

A matriz k_{op} da equação (79) também foi implementada no código. Esta também depende de cada elemento e já é inscrita em termos das variáveis ξη. No código, essa matriz é denominada de k_{op} como mostra a figura (29).

Figura (29): Matriz kop

```
31 # MATRIZ [Kop]:
32
33 def Kop(e, ξ, η):
34   Kop = np.dot(np.transpose(Q(e , ξ , η)), np.dot(D(e),Q(e , ξ , η)))
35   return Kop
36
```

Fonte: Elaborado pelo próprio autor.

Integração via quadratura Gaussiana - (Item 2.5 do código em anexo)

Este subtópico é destinado a inserção de dados e operações referentes a integração numérica pelo método de Gauss-Legendre.

Pontos e pesos de Gauss - (Item 2.5.1 do código em anexo)

Este subtópico é destinado a inserção dos pontos e pesos de Gauss referentes a integração $2x2$. Nele são implementados as equações (89a) e (89b). A figura (30) apresenta os valores inseridos.

Figura (30): Pontos e pesos de Gauss

```
1 # PONTOS DE GAUSS PARAINTEGRAÇÃO 2X2:
2
3 eg = np.array([[-1/np.sqrt(3)],[1/np.sqrt(3)]])
4 ng = np.array([[-1/np.sqrt(3)],[1/np.sqrt(3)]])
5
6 # PESOS DE GAUSS:
7
8 wg = np.array([[1],[1]])
```

Fonte: Elaborado pelo próprio autor.

Funções de forma - (Item 2.5.2 do código em anexo)

Como comentado no tópico 2.4.2 deste trabalho, tendo em vista a utilização de métodos numéricos de resolução das integrais múltiplas para obtenção da matriz de rigidez dos elementos, é feita uma mudança de variável como sugere a equação (82). Para a mudança de variáveis, utilizam-se as funções de forma apresentadas na figura (14). A figura (31) apresenta sua implementação no código.

Figura (31): Implementação das funções de forma

```
1 # FUNÇÕES DE FORMA:
2
3 def N1(ξ , η):
4   N1 = (1/4)*((1 - ξ)*(1 - η))
5   return N1
6
7 def N2(ξ , η):
8   N2 = (1/4)*((1 + ξ)*(1 - η))
9   return N2
10
11 def N3(ξ , η):
12   N3 = (1/4)*((1 + ξ)*(1 + η))
13   return N3
14
15 def N4(ξ , η):
16   N4 = (1/4)*((1 - ξ)*(1 + η))
17   return N4
```

Fonte: Elaborado pelo próprio autor.

Utilizando as funções de forma, é possível escrever as coordenadas xy em função das coordenadas $\xi\eta$ apresentado pela equação (55). A implementação dessa equação no código é feita no tópico 2.5.2 como mostra a figura (32).

Figura (32): Equações de transformação de coordenadas

```
19
20 def xe1(e , ξ , η):  # Coordenada x:
21   xe1 = N1(ξ , η)*xl(e)[0][0] + N2(ξ , η)*xl(e)[1][0] + N3(ξ , η)*xl(e)[2][0] + N4(ξ , η)*xl(e)[3][0]
22   return xe1
23
24 def ye1(e , ξ , η):  #Coordenada y:
25   ye1 = N1(ξ , η)*yl(e)[0][0] + N2(ξ , η)*yl(e)[1][0] + N3(ξ , η)*yl(e)[2][0] + N4(ξ , η)*yl(e)[3][0]
26   return ye1
27
```

Fonte: Elaborado pelo próprio autor.

A diferença básica entre a equação (55) e a sua implementação no código apresentado pela figura (32) é que as variáveis x e y, além de serem inscritas em termos das novas variáveis ξ e η, são inscritas também em função de cada elemento, representado pela variável e. Como podemos observar, a variável e advém do vetor de coordenadas locais de cada elemento, representado no código pelos vetores $xl(e)$ e $yl(e)$, que também dependem de cada elemento.

Matriz Jacobiana - (Item 2.5.3 do código em anexo)

Para obter a matriz de rigidez de cada elemento, é necessário definir a matriz jacobiana de cada elemento e aplicá-la na equação (82). Uma maneira prática de obtê-la é usando a equação (60b). Para tal, basta apenas definir a matriz $DNx(\xi, \eta)$ que contém as derivadas de primeira ordem das funções de forma.

Figura (33): Matriz das derivadas

```
17
18 #  Matriz com as derivadas das funções de interpolação
19
20 def DNx(ξ, η):
21     Nξ_val = Nξ(ξ, η)
22     Nη_val = Nη(ξ, η)
23     DNx = np.array([Nξ_val[:, 0], Nη_val[:, 0]])
24     return DNx
25
```

Fonte: Elaborado pelo próprio autor.

A parte do código representada na figura (33), está inscrita no tópico 2.5.3. Para tal, as derivadas foram inseridas manualmente no código em uma etapa anterior.

Elas são representadas por $N\xi(\xi, \eta)$ e $N\eta(\xi, \eta)$ como mostrado na figura. Uma vez montada a matriz $DNx(\xi, \eta)$, determina-se a matriz jacobiana, bem como o seu determinante. Novamente, a variável e, aparece, indicando que ambas as matrizes dependem de cada elemento.

Figura (34): Matriz Jacobiana e sua inversa

```
26 # >>> Matriz Jacobiana [J]
27
28 def J(e, ξ, η):
29   J = DNx(ξ, η) @ coorde(e)
30   return J
31
32 # >>> Determinante da matriz Jacobiana
33
34 def DetJ(e, ξ, η):
35   DetJ = np.linalg.det(J(e, ξ, η))
36   return DetJ
```

Fonte: Elaborado pelo próprio autor.

Integração - (Item 2.5.4 do código em anexo)

Este tópico é responsável pela realização da integração numérica pelo método de Gauss-Legendre apresentado no tópico 2.5 deste trabalho. A figura (35) apresenta a implementação da equação (90a), para a resolução da integral apresentada pela equação (82), resumindo-se a um procedimento iterativo para obtenção da matriz $Ki(e)$ que representa a matriz K_o na equação (82).

Figura (35): Integração por quadratura Gaussiana

```
[ ]  1 def Ki(e):
     2   Kii = np.zeros((12,12))
     3   for i in range(2):
     4     for j in range(2):
     5       ξa =  eg[i][0]
     6       ηa =  eg[j][0]
     7       Kii = Kii + wg[i][0] * wg[j][0] * Kop(e, ξa, ηa) * DetJ(e, ξa, ηa)
     8   return(Kii)
```

Fonte: Elaborado pelo próprio autor.

Matriz de rigidez dos elementos - (Item 2.6 do código em anexo)

Este tópico no código é responsável por gerar as matrizes de rigidez de cada elemento. Uma vez resolvida a integral, a matriz $Ki(e)$ é utilizada na equação (80), gerando a matriz de rigidez de cada elemento $Ke(e)$ representado na figura (36). É importante lembrar que a matriz $Ke(e)$ tem dimensões $12x12$, ou seja, esta armazena todos os 12 graus de liberdade do elemento.

Figura (36): Obtenção da matriz de rigidez de cada elemento

```
1 def Ke(e):
2   InvA = np.linalg.inv(A(e))
3   Kee = np.dot(np.transpose(InvA),   np.dot(Ki(e),  InvA))
4   return Kee
```

Fonte: Elaborado pelo próprio autor.

Matriz de rigidez Global - (Item 2.7 do código em anexo)

Neste tópico é criado o vetor de graus de liberdade $GDL(e)$ discutido no tópico 2.4.2. Este é crucial para a montagem da matriz de rigidez da estrutura. Para tal, é criada uma matriz K com elementos nulos de dimensão $GLxGL$. Em seguida é realizado o preenchimento dessa matriz utilizando uma estrutura de repetição com $i = 1, 2, 3, \dots 12$. e $j = 1, 2, 3, \dots 12$. baseada na expressão a seguir;

$$Kg_{GDL(e)_i;GDL(e)_j} = Kg_{GDL(e)_i;GDL(e)_j} + Ke(e)_{ij} \tag{92}$$

Dessa maneira, a matriz de rigidez global é montada termo a termo, sendo armazenada no vetor K como mostra a figura (37).

Figura (37): Obtenção da matriz de rigidez da estrutura

```
19 # MATRIZ DE RIGIDEZ GLOBAL:
20
21 K = np.zeros((GL,GL))
22 for e in range(1,(ne+1)):
23   GDL_e = GDL(e)
24   Ke_e = Ke(e)
25   for i in range(12):
26     for j in range(12):
27       K[GDL_e[i][0]-1][GDL_e[j][0]-1] += Ke_e[i][j]
28
```

Fonte: Elaborado pelo próprio autor.

Uma curiosidade sobre a matriz de rigidez global, é que esta é sempre uma matriz quadrada é invertível. Além disso, cada linha da matriz corresponde a uma equação que representa referente a apenas um grau de liberdade do problema. Portanto, as linhas da matriz estão organizadas em ordem crescente de graus de liberdade. As três primeiras linhas da matriz de rigidez global, por exemplo, fazem parte das equações que representam os 3 graus de liberdade do nó 1 da estrutura. Os próximos 3 são representam os graus de liberdade do nó 2 e assim sucessivamente. Portanto, a equação (87b), representa um sistema com GL equações e GL incógnitas, sendo que cada incógnita é um grau de liberdade da estrutura.

Formulação das condições de contorno - (Item 2.8 do código em anexo)

Para obedecer às condições de contorno aplicadas a cada nó da placa, faz-se o uso do vetor de graus de liberdade restringidos, denotado no código por GLR. Existem algumas maneiras de impor condições de contorno a um nó de uma estrutura. Uma estratégia simples é aplicar uma “grande rigidez” ao grau de liberdade correspondente. Por exemplo, para aplicar a condição de contorno deslocamento nulo de um nó, por exemplo, o vetor GLR localiza o grau de liberdade associado ao deslocamento do nó e lhe aplica uma grande rigidez ao deslocamento. Matematicamente, isso é feito multiplicando-se um número grande ao elemento da diagonal principal da matriz de rigidez global correspondente àquele grau de liberdade específico. O mesmo procedimento é aplicado caso seja conveniente restringir quaisquer rotações do nó.

Esse procedimento foi aplicado no código e, como mostra a figura (38), o elemento da diagonal principal localizado pelo vetor de graus de liberdade restringidos é multiplicado por um fator de 10^7.

Figura (38): Condições de contorno

```
1 # Aplicação das condições de contorno:
2
3 for i in range(GLR.shape[0]):
4   K[GLR[i,0]-1,GLR[i,0]-1] = 10**7*K[GLR[i,0]-1,GLR[i,0]-1]
```

Fonte: Elaborado pelo próprio autor.

Solução do sistema de equações - (Item 2.9 do código em anexo)

Neste tópico são calculados os vetores de deslocamento global da estrutura d_g e também o vetor de deslocamentos de cada elemento $d_e(e)$. Como discutido anteriormente, o vetor de deslocamentos globais da estrutura é obtido por meio da equação (93a). Esta é uma das etapas mais requisitadas em termos de tempo de processamento.

Diante de malhas de elementos finitos muito densas, estas geram sistemas lineares de dimensões muito altas e a solução dos problemas ocorre de forma mais demorada. É por esse motivo que optou-se pela resolução mediante a sistemas de matrizes esparsas oferecido pela biblioteca *scipy*, obtendo uma eficiência maior no tempo de processamento quando comparado com a resolução do sistema por meio da biblioteca *numpy*.

Figura (39): Solução do sistema de equações

```
1 # Deslocamentos nodais globais:
2
3 #d = np.linalg.solve(K , f)
4
5 # >>> Vetor dos deslocamentos nodais globais
6 def solve_large_system(K, f):  # Função para resolução so sistema linear
7     K_sparse = csc_matrix(K)   # Converter K para formato esparso CSC
8
9     try:
10        d_sparse = spsolve(K_sparse, f)
11        d = np.array([d_sparse]).T
12        print(f"Solução encontrada com sucesso!")
13        return d
14    except ValueError as e:
15        print(f"Erro ao resolver o sistema: {e}")
16        return None
17    except Exception as e:
18        print(f"Ocorreu um erro inesperado: {e}")
19        return None
20
21 d = solve_large_system(K, f)  # Resolvendo o sistema
22
23 if d is not None:              # Verificando se a solução foi encontrada
24     pass                       # Faça algo com a solução, se necessário
25 else:                          # Trate o caso em que não foi possível resolver o sistema
26     print("Não foi possível encontrar uma solução para o sistema.")
```

Resolução por meio da biblioteca *numpy*.

Fonte: Elaborado pelo próprio autor.

Na figura (39), o vetor d é equivalente ao vetor d_g. Com o auxílio do vetor $GDL(e)$, monta-se o vetor de deslocamentos de cada elemento $d_e(e)$ como mostra a figura (40) a seguir.

Figura (40): Vetor de deslocamento de cada elemento

```
28 # Deslocamentos nodais de cada elemento:
29
30 def de(e):
31   dee = np.zeros((12,1))
32   for i in range (12):
33     dee[i, 0] = d[GDL(e)[i][0]-1][0]
34   return dee
```

Fonte: Elaborado pelo próprio autor.

Matriz de compatibilidade cinemática - (Item 2.10 do código em anexo)

Neste tópico, é calculada a matriz de compatibilidade cinemática $B(x, y)$ discutida no tópico 2.4.2 deste trabalho. Como visto anteriormente, essa matriz advém da expressão $Q(x, y)A^{-1}$. Esta, por sua vez, é descrita também em função de cada elemento.

Efetuando-se a mudança de variáveis, esta é escrita como $B(e, \xi, \eta) = Q(e, \xi, \eta)A(e)^{-1}$. A implementação da matriz foi realizada conforme a figura (41).

Figura (41): Matriz de compatibilidade cinemática

```
1 def B(e, ξ, η):
2   InvA = np.linalg.inv(A(e))
3   B = np.matmul(Q(e, ξ , η), InvA)
4   return B
```

Fonte: Elaborado pelo próprio autor.

Vetor dos momentos - (Item 2.11 do código em anexo)

Neste tópico, é criada a função que calcula o vetor dos momentos contendo os momentos fletores na direção de x, y e o momento torsor. Além disso, são calculados os vetores que contém cada momento em particular. Essa etapa é crucial para a representação gráfica desses esforços no *paraview*.

Função dos momentos - (Item 2.11.1 do código em anexo)

Nesta etapa é implementada a equação (88). A implementação está representada na figura (42). Esta equação fornece os valores dos esforços internos em um nó específico de um elemento da malha. A escolha do nó se dá pela manipulação das variáveis ξη de acordo com a figura (17). Por exemplo, para saber a magnitude dos esforços internos no nó 2 do elemento 1, deve-se fazer $M_k(1, 1, -1)$. O resultado é um vetor coluna com três linhas. A primeira linha com o valor do momento fletor na direção de x, a segunda linha com momento fletor na direção de y e na terceira linha o valor do momento torsor.

Figura (42): Função dos momentos.

```
1 # FUNÇÃO QUE CALCULA O MOMENTO EM FUNÇÃO DOS ELEMENTOS:
2
3 def Mk(e, ξ, η):
4   Mk = np.dot(-D(e), np.dot(B(e, ξ, η), de(e)))
5   return Mk
```

Fonte: Elaborado pelo próprio autor.

Vetor de momentos - (Item 2.11.2 do código em anexo)

Esta etapa do código é destinada a geração dos vetores de esforços internos que serão utilizados no código no tópico denominado *Preparaview* implementado por Silva (2022). Para tal, ao invés de representar os três momentos em um único vetor, como fornecido pela equação dos momentos, estes serão coletados separadamente, ou seja, serão criados três vetores coluna sendo, o primeiro destinado ao armazenamento dos momentos fletores em x, o segundo destinado a armazenar os momentos fletores em y e o terceiro destinado a armazenar apenas os momentos torsores. A dimensão da matriz que representa cada um desses vetores é $nnos\ x\ 1$, ou seja, contemplam todos os nós da malha.

Como abordado anteriormente, os nós de uma malha de elementos finitos podem ser compartilhados por 1, 2, 3 ou 4 elementos. Como explicado em Logan (2012), a expressão do funcional $w(x, y)$ representada pela equação (63a), é inscrita de modo que a continuidade dos deslocamentos entre os elementos seja garantida. Porém, segundo o mesmo autor, essa condição não é garantida no caso das suas derivadas. Como os esforços internos são obtidos fazendo o uso das derivadas de w mediante as equações (38), (39) e (40), é de se esperar que sejam encontrados valores de momento com diferenças para um mesmo nó que seja compartilhado por mais de um elemento.

Para resolver esse problema, optou-se por fazer uma média aritmética entre os valores de momento encontrados em cada elemento que compartilha o mesmo nó. Para tal, neste subtópico do código, foi criada uma maneira de identificar quantos elementos compartilham o mesmo nó. Esse valor é calculado e armazenado em um vetor contador. Em seguida, a função dos momentos apresentada na figura (42), calcula o momento em todos os elementos que compartilham o mesmo nó.

Em seguida, esses valores são somados e armazenados num segundo vetor denominado vetor soma. Sendo assim, é criado um vetor para armazenar os valores da média aritmética a ser efetuada.

Por fim, a média aritmética é calculada fazendo a divisão de cada elemento do vetor soma pelo seu correspondente no vetor contador. Esse procedimento é repetido para os 3 valores de momento, de modo que ao final, obtenha-se três vetores independentes, sendo um contendo a média referente aos momentos fletores em x, outro com a média dos valores dos momentos fletores em y e outro contendo a média dos momentos torsores. No código, esses vetores são denotados por mx, my e mxy respectivamente. A programação dessa parte do código pode ser verificada no anexo A.

3.2.3 Arquivo VTU - (Item 3 do código em anexo)

Após o processamento de cada problema, é gerado um arquivo VTU contendo os dados processados em cada análise. O arquivo *VTU* é utilizado diretamente na plataforma *Paraview* para a visualização gráfica de dados, auxiliando na representação dos dados e processamento de imagens e gráficos e, por isso, é um tipo de arquivo *Paraview VTK Unstructured Grid.* Segundo Silva (2022), a obtenção desse tipo de arquivo é feita através de um módulo *Python* chamado *PyEVTK* desenvolvido por Herrera (2021).

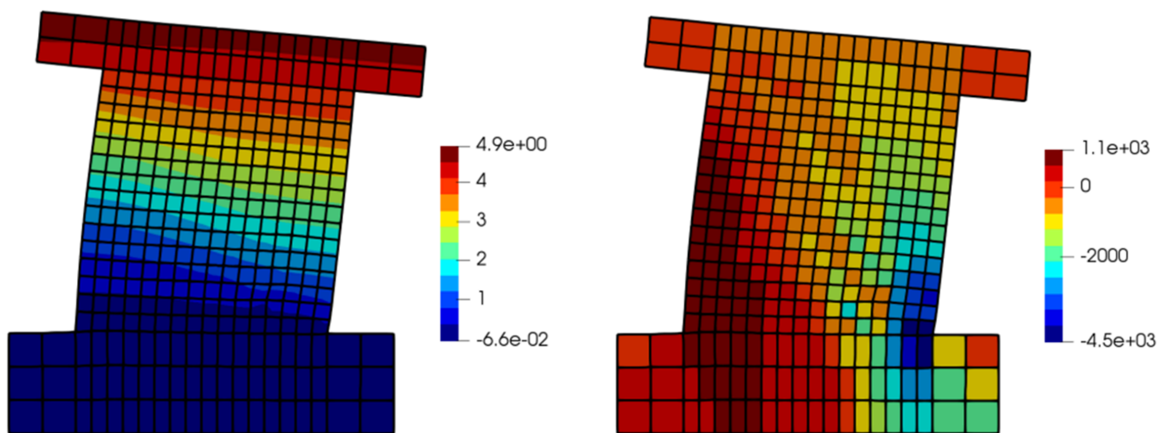
3.3 Paraview

O *Paraview* é um software aberto e gratuito utilizado na visualização e análises de dados com diversos fins científicos. É utilizado por engenheiros, uma ferramenta de pós-processamento de dados onde o usuário é capaz de identificar quaisquer problemas do produto e avaliar o desempenho do mesmo através da simulação com ferramentas gráficas de visualização de dados. Neste trabalho, todas as imagens obtidas para as análises dos esforços solicitantes foram obtidas utilizando o *Paraview*.

Existem algumas formas de representações gráficas utilizando elementos finitos. Em suas análises, Silva (2022) utiliza campos de deslocamento e rotações com distribuição contínua de coloração, ou seja, é possível observar várias paletas de cores em um mesmo elemento (Figura (43a)). Já para representação dos esforços internos utiliza a representação de elementos com paletas de cores discretas, ou seja, cada elemento representa uma cor, que, por sua vez, está associada a um único valor de esforço (Figura (43b)).

Para obter a magnitude dos esforços solicitantes no elemento, Silva (2022) utilizou a média dos pontos de Gauss.

Figura (43): Análise dos deslocamentos e esforços normal em pilar parede



(a) Deslocamento representado em paletas de cores contínuas.

(b) Esforço normal representado em paletas de cores discretas.

Fonte: Silva, 2022.

A magnitude dos esforços internos trabalhados são referentes aos nós dos elementos como esclarecido no tópico 2.2. Como parâmetro de representação, as análises de deslocamentos, rotações e esforços internos deste trabalho serão apresentados inteiramente em campos com paletas de cores contínuas.

4 RESULTADOS: VALIDAÇÃO E ANÁLISES

Essa seção apresenta a aplicação do elemento e o seu desempenho em placas, desde geometrias simples, a geometrias mais complexas contendo os tipos de carregamentos encontrados no campo da engenharia de estruturas. Além disso, são simulados pavimentos de edificações, com foco no estudo dos esforços internos e deslocamentos.

O elemento implementado será submetido a uma série de análises quanto a sua convergência, desempenho, eficiência do código e adaptabilidade às diversas geometrias do domínio de estudo, bem como seu desempenho com relação a malhas com elementos de geometria distorcida.

4.1 Validação do elemento

Para a validação do elemento, será utilizado um exemplo numérico apresentado por Wilson (2002). O exemplo será dividido em 2 análises. A primeira consiste em submeter uma placa quadrada simplesmente apoiada a um carregamento pontual. A segunda consiste na aplicação de um carregamento uniforme na mesma placa usada na primeira análise. Serão estudados também o desempenho da malha dos elementos, sendo esta com elementos retangulares e também com elementos distorcidos. A convergência da malha será estudada à medida que o número de elementos aumenta.

Os parâmetros do problema são definidos por Wilson (2002) e são resumidos na tabela (01) a seguir;

Tabela (1): Dados de entrada do problema

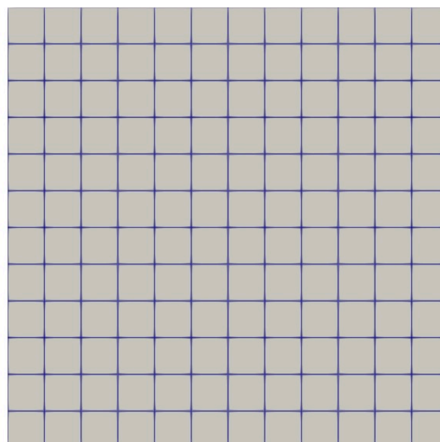
Placa quadrada - Wilson (2002)		
Parâmetro	**Unidade da bibliografia**	**Unidade no SI**
Carga pontual	1 kip	4,4482 kN
Carga uniforme	1 kip/in²*	6894,7573 kN/m²
Espessura da placa	1 in*	0,0254 m
Dimensão	10 in*	2,54 m
Coeficiente de Poisson	0,3	0,3
Módulo de elasticidade	10,92	75290,7496 kPa
*in - Refere-se à polegadas		

Fonte: Elaborado pelo próprio autor.

Na primeira análise, utiliza-se a carga pontual situada no centro da placa. A malha é composta por elementos retangulares retos, ou seja, sem distorções. Foram então testadas 9 malhas, desde a malha 1, $(2x2)$ com 4 elementos até a malha 9 $(64x64)$, contendo 4096 elementos.

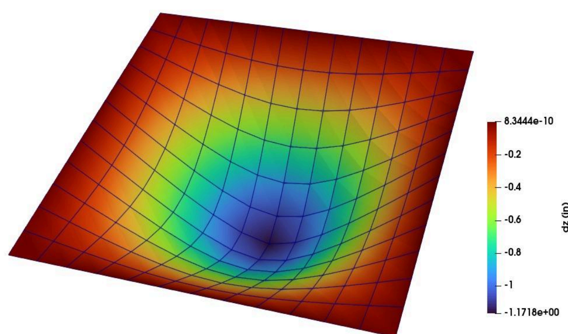
A figura (45) ilustra a representação das análises consideradas no problema para a malha 6 (12x12), contendo 144 elementos. A discretização do domínio de estudo referente à malha 6 é representado pela figura a (44) a seguir.

Figura (44): Elementos retangulares retos - Malha 6

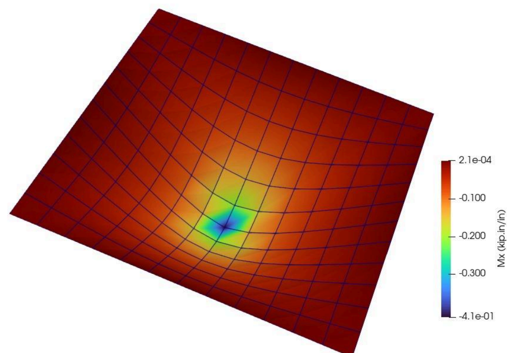


Fonte: Elaborado pelo próprio autor.

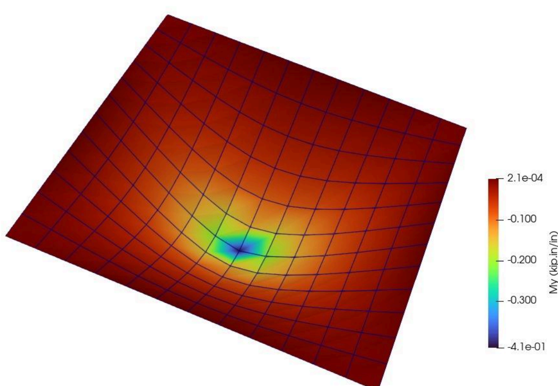
Figura (45): Esforços internos e deslocamentos - Malha 6 (12x12)



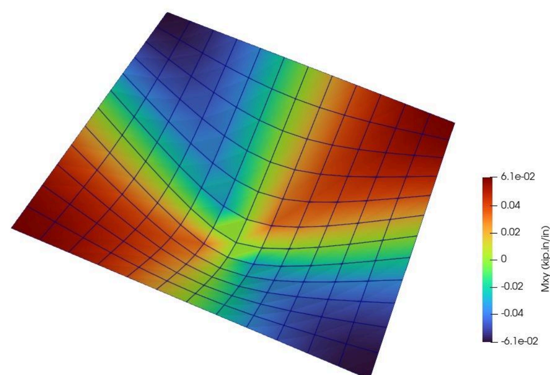
(a) Deslocamento (in)



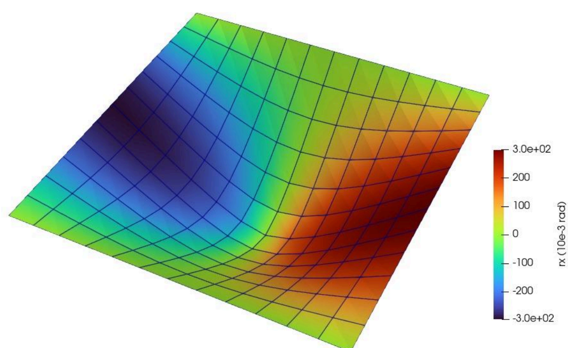
(b) Momento em x (kip.in/in)



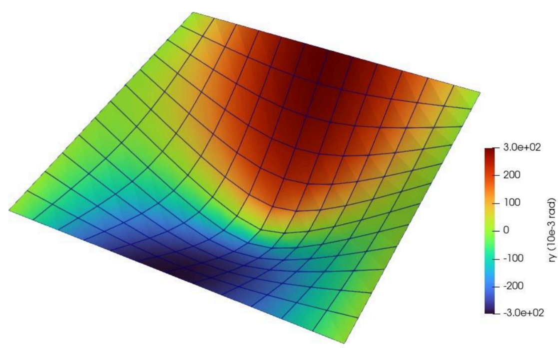
(c) Momento em y (kip.in/in)



(d) Momento torsor (kip.in/in)



(e) Rotações em x (10e-3 rad)



(f) Rotações em y (10e-3 rad)

Fonte: Elaborado pelo próprio autor.

Como esperado, os momentos máximos são iguais e ocorrem no centro da placa devido a sua geometria e o posicionamento da carga pontual. Em contrapartida, o momento torsor máximo ocorre nos cantos simplesmente apoiados como discutido na obra de Araújo (2010). Segundo Wilson (2002), o valor exato do deslocamento da placa fina trabalhada nesse exemplo é de 1,16 polegadas.

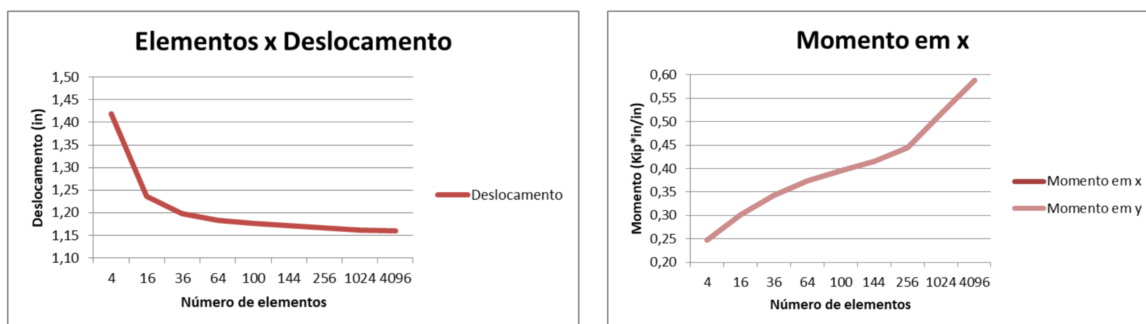
Com intuito de estudar a convergência dos deslocamentos e esforços internos, esse procedimento foi repetido para as diversas malhas cujos resultados estão registrados na tabela (02). Como esperado, à medida que a malha é refinada, há uma convergência dos deslocamentos. Por serem estudados no centro da placa, os momentos na direção de x e y são iguais e máximos nesse ponto. Porém no caso dos esforços internos, de acordo com a figura (46.b), os valores de momento não apresentam sinais de convergência definidos. Já os momentos torsores no centro são cada vez menores à medida que a malha é refinada, convergindo ao valor nulo como era esperado.

Tabela (2): Análise com carga pontual

Carga Pontual em Placa Quadrada Simplesmente Apoiada - Wilson (2002)						
Malha	Dim. Malha	Nº de elementos	Deslocamento (in)	Mx (Kip*in/in)	My (Kip*in/in)	Mxy (Kip*in/in)
1	2 x 2	4	1,418	0,248	0,248	0,040
2	4 x 4	16	1,237	0,302	0,302	0,027
3	6 x 6	36	1,199	0,344	0,344	0,024
4	8 x 8	64	1,184	0,373	0,373	0,023
5	10 x 10	100	1,176	0,396	0,396	0,023
6	12 x 12	144	1,172	0,415	0,415	0,023
7	16 x 16	256	1,167	0,445	0,445	0,023
8	32 x 32	1024	1,162	0,516	0,516	0,022
9	64 x 64	4096	1,161	0,588	0,588	0,022

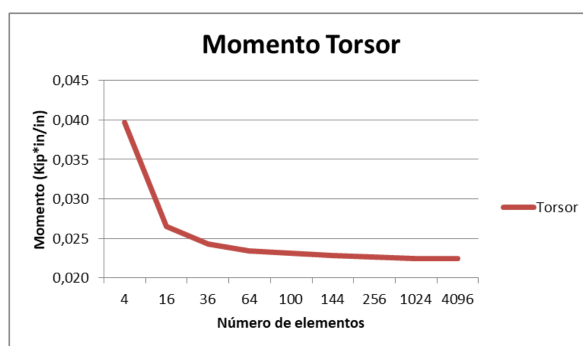
Fonte: Elaborado pelo próprio autor.

Figura (46): Esforços internos e deslocamentos - Malha 6 (12x12)



(a) Deslocamento (in)

(b) Momento em x e y (kip.in/in)



(c) Momento torsor (kip.in/in)

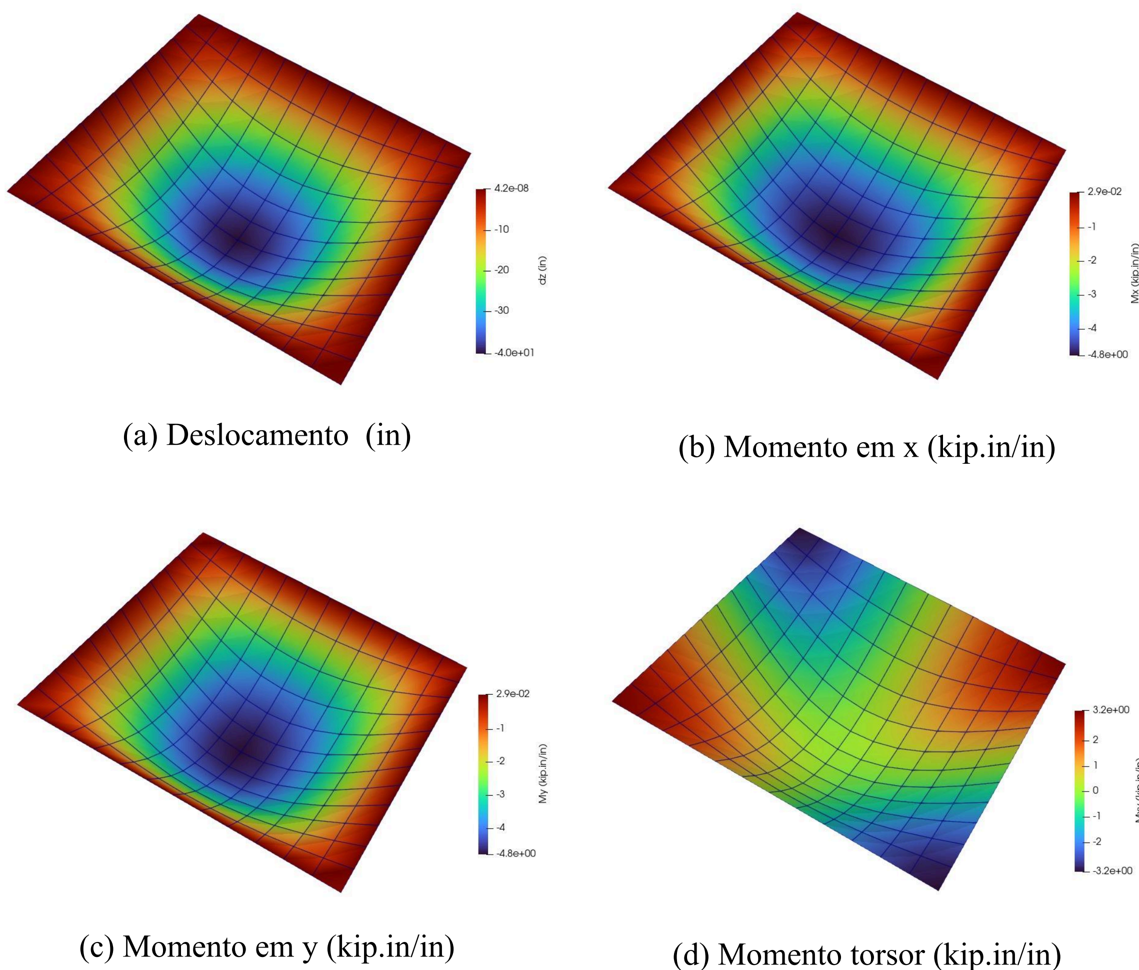
Fonte: Elaborado pelo próprio autor.

De acordo com o gráfico de deslocamentos (Figura 46.a), a convergência ocorre de maneira decrescente.

Segundo Soriano (2003), esta é uma convergência não monótona. Além disso, chama atenção o comportamento dos momentos nas direções de x e y. Apesar de estar próximos da solução teórica, estes não apresentam sinais de convergência. Segundo Wilson (2002), esse fato está relacionado à singularidade associada à concentração de tensão causada pela aplicação da carga pontual. Por outro lado, o momento torsor assume um comportamento esperado, diminuindo à medida que a malha é refinada.

Por outro lado, a solicitação por carga uniformemente distribuída na placa apresentou resultados mais satisfatórios em termos de convergência de deslocamentos e esforços internos. A análise leva em consideração uma carga distribuída de 1 kip/in², na mesma malha usada anteriormente submetida a carga pontual. O comportamento da placa está apresentado na **f**igura (47).

Figura (47): Esforços internos e deslocamentos - Malha 6 (12x12)

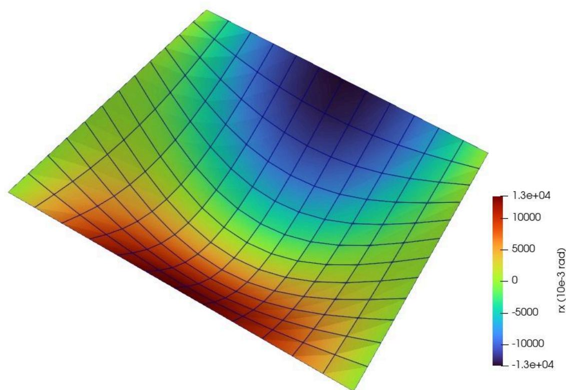


(a) Deslocamento (in)

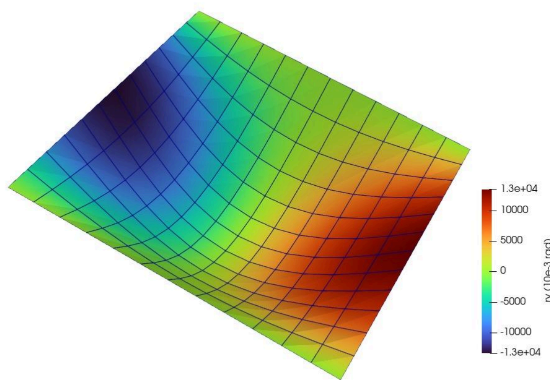
(b) Momento em x (kip.in/in)

(c) Momento em y (kip.in/in)

(d) Momento torsor (kip.in/in)



(e) Rotação em x (10-3 rad)



(f) Rotação em y (10e-3 rad)

Fonte: Elaborado pelo próprio autor.

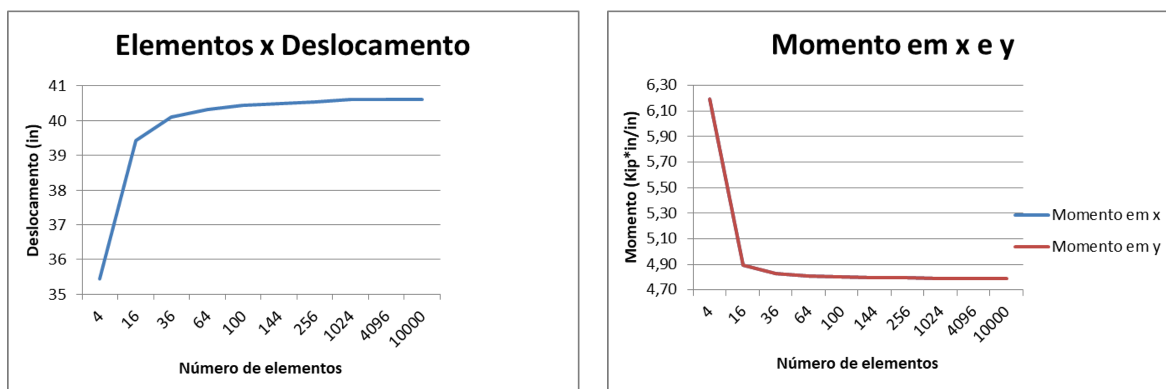
De maneira análoga, o problema foi submetido às 9 malhas ensaiadas anteriormente juntamente com a malha 10, contendo 10000 elementos. Os resultados foram coletados na tabela (03).

Tabela (3): Análise com carga uniformemente distribuída

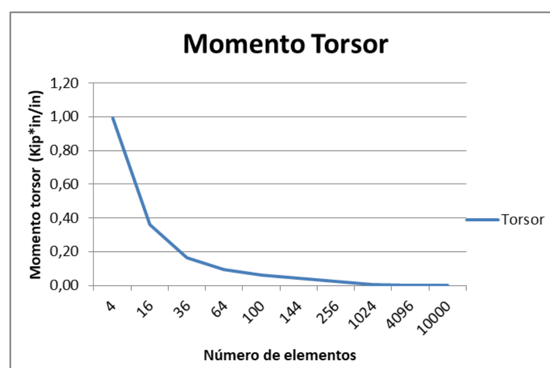
Carga Uniformemente Distribuída em Placa Quadrada Simplesmente Apoiada - Wilson (2002)						
Malha	Dim. Malha	Nº de elementos	Deslocamentos (in)	Mx (Kip*in/in)	My (Kip*in/in)	Mxy (Kip*in/in)
1	2 x 2	4	35,454	6,195	6,195	0,993
2	4 x 4	16	39,419	4,896	4,896	0,360
3	6 x 6	36	40,099	4,827	4,827	0,165
4	8 x 8	64	40,331	4,808	4,808	0,094
5	10 x 10	100	40,437	4,801	4,801	0,060
6	12 x 12	144	40,494	4,797	4,797	0,042
7	16 x 16	256	40,551	4,793	4,793	0,024
8	32 x 32	1024	40,605	4,790	4,790	0,006
9	64 x 64	4096	40,619	4,789	4,789	0,001
10	100 x 100	10000	40,622	4,789	4,789	0,001

Fonte: Elaborado pelo próprio autor.

Figura (48): Esforços internos e deslocamentos - Malha 6 (12x12)



(a) Deslocamento (in) (b) Momento em x e y (kip.in/in)



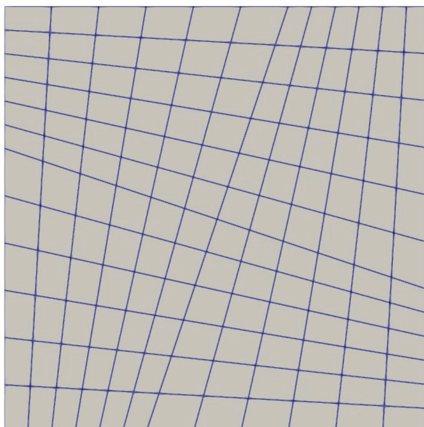
(c) Momento torsor (kip.in/in)

Fonte: Elaborado pelo próprio autor.

Com relação a convergência dos deslocamentos (Figura 48.a), observa-se claramente uma convergência do tipo monótona como discutido na obra de Soriano (2013), convergindo para a solução correta do problema por valores inferiores. Quanto aos esforços internos, estes também convergem para valores bem definidos por valores superiores. Para este problema, o elemento obteve um bom desempenho na obtenção dos deslocamentos e esforços internos.

A mesma placa ensaiada anteriormente foi discretizada em elementos distorcidos como representa a figura (49). Para este problema, a mudança básica é apenas na matriz de coordenadas nodais, em que os nós da malha foram deslocados de modo a distorcer a forma de cada elemento.

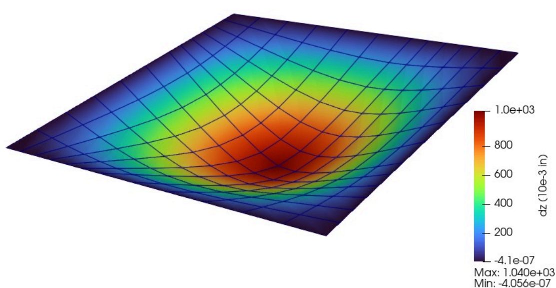
Figura (49): Elementos distorcidos - Malha 6



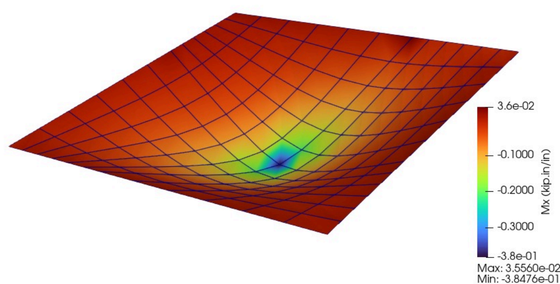
Fonte: Elaborado pelo próprio autor.

A malha da figura (49) foi submetida ao carregamento pontual da tabela (1). O comportamento da malha é representado pela figura (50).

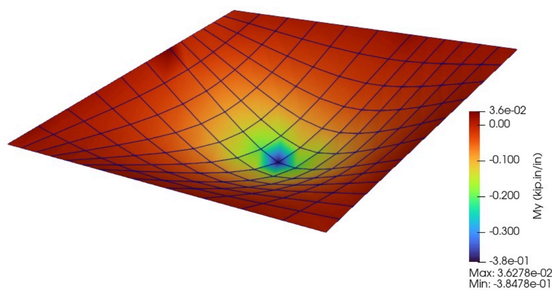
Figura (50): Esforços internos e deslocamentos - Malha 6 (12x12)



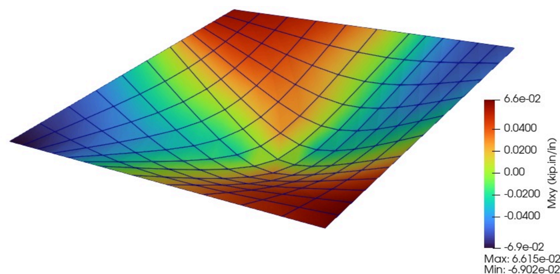
(a) Deslocamento (in)



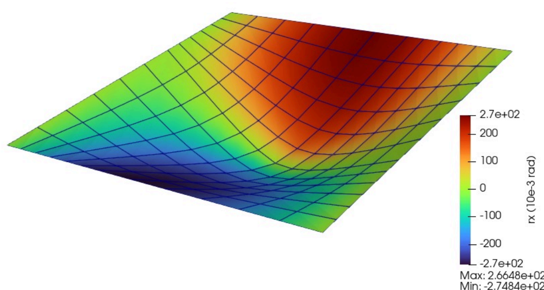
(b) Momento em x (kip.in/in)



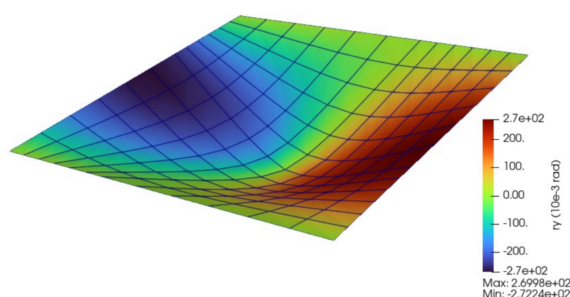
(c) Momento em y (kip.in/in)



(d) Momento torsor (kip.in/in)



(e) Rotação em x (10-3 rad)



(f) Rotação em y (10e-3 rad)

Fonte: Elaborado pelo próprio autor.

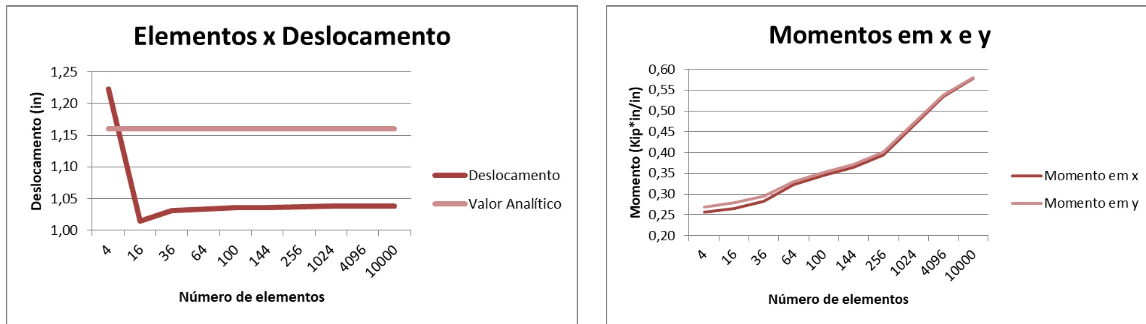
Os resultados numéricos obtidos nas análises foram registrados na tabela (4) a seguir.

Tabela (4): Análise com carga pontual com malha distorcida - Malha 6

Carga Pontual em Placa Quadrada em Malha Distorcida						
Malha	**Dim. Malha**	**Nº de elementos**	**Deslocamento (in)**	**Mx (Kip*in/in)**	**My (Kip*in/in)**	**Mxy (Kip*in/in)**
1	2 x 2	4	1,223	0,256	0,268	0,019
2	4 x 4	16	1,015	0,265	0,279	0,040
3	6 x 6	36	1,031	0,283	0,294	0,015
4	8 x 8	64	1,033	0,323	0,330	0,017
5	10 x 10	100	1,035	0,344	0,352	0,015
6	12 x 12	144	1,036	0,364	0,372	0,016
7	16 x 16	256	1,037	0,394	0,401	0,016
8	32 x 32	1024	1,038	0,466	0,470	0,017
9	64 x 64	4096	1,038	0,535	0,538	0,018
10	100 x 100	10000	1,038	0,579	0,581	0,019

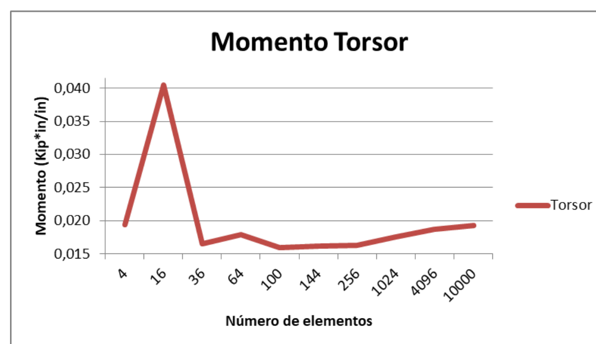
Fonte: Elaborado pelo próprio autor.

Figura (51): Esforços internos e deslocamentos - Malha 6 (12x12)



(a) Deslocamento (in)

(b) Momento em x e y (kip.in/in)



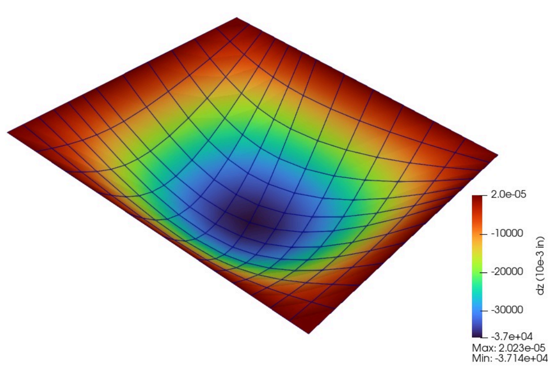
(c) Momento torsor (kip.in/in)

Fonte: Elaborado pelo próprio autor.

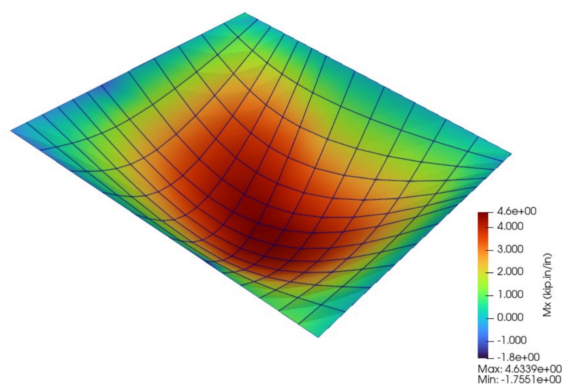
Em relação aos deslocamentos (Figura (51a)), observa-se que o elemento apresenta sinais de convergência “por cima”, ou seja, parte de valores superiores ao analítico. Porém, apesar de convergir, este não apresentou convergência para o valor analítico, mas sim para um valor inferior a este como é observado na tabela (4). Quanto aos momentos fletores na direção de x e y, inicialmente, chamou a atenção a assimetria entre os momentos máximos nas primeiras malhas (Figura (51b)), além de não apresentar sinais de convergência definidos semelhante ao caso da malha retangular. Além disso, os valores de momento apresentam valores bem inferiores ao valor analítico considerado, à medida que a malha é refinada. O gráfico do momento torsor (Figura (51c)), por sua vez, apresentou oscilações em seu comportamento, não apresentando sinais de estabilização definidos.

O mesmo procedimento foi aplicado para a análise do desempenho da malha distorcida sob carregamento uniformemente distribuído. O comportamento da placa para essa situação é representado pela figura (52).

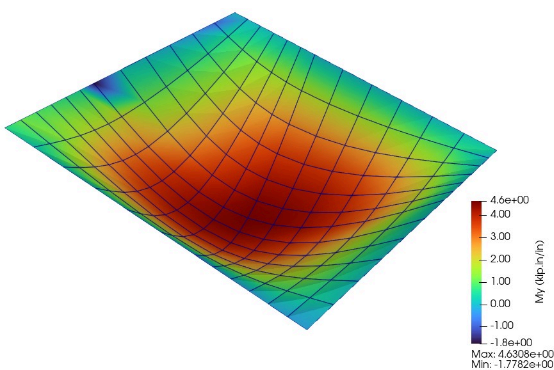
Figura (52): Esforços internos e deslocamentos - Malha 6 (12x12)



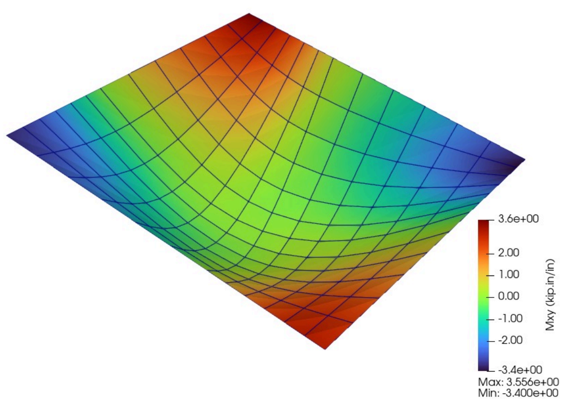
(a) Deslocamento (in)



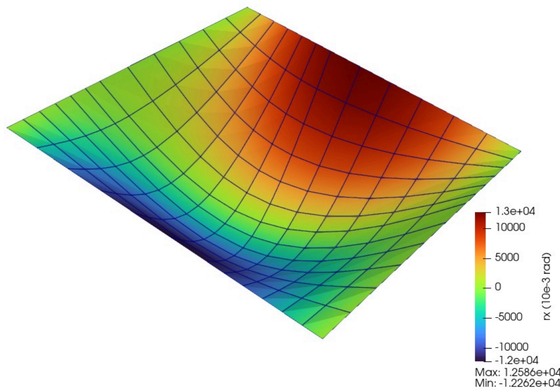
(b) Momento em x (kip.in/in)



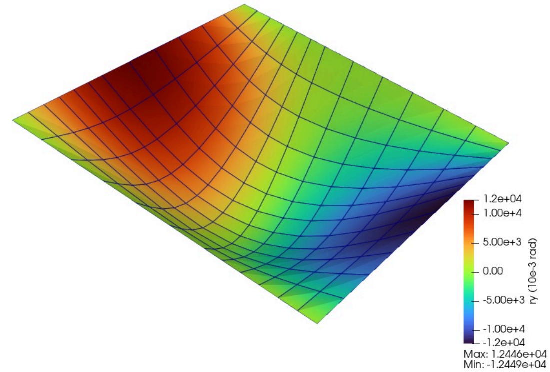
(c) Momento em y (kip.in/in)



(d) Momento torsor (kip.in/in)



(e) Rotação em x (10-3 rad)



(f) Rotação em y (10e-3 rad)

Fonte: Elaborado pelo próprio autor.

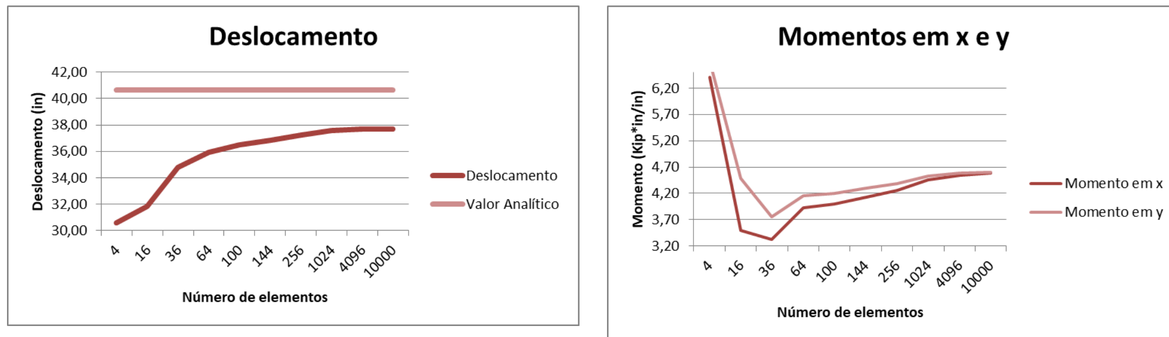
Analogamente, o mesmo problema foi ensaiado nas 10 malhas e os resultados numéricos foram registrados na tabela (5).

Tabela (5): Análise com carga pontual com malha distorcida - Malha 6

Carga Uniforme em Placa Quadrada em Malha Distorcida						
Malha	**Dim. Malha**	**Nº de elementos**	**Deslocamento (in)**	**Mx (Kip*in/in)**	**My (Kip*in/in)**	**Mxy (Kip*in/in)**
1	**2 x 2**	**4**	30,569	6,404	6,705	0,472
2	**4 x 4**	**16**	31,832	3,489	4,478	0,660
3	**6 x 6**	**36**	34,776	3,326	3,758	0,180
4	**8 x 8**	**64**	35,910	3,925	4,154	0,081
5	**10 x 10**	**100**	36,518	4,002	4,203	0,146
6	**12 x 12**	**144**	36,862	4,130	4,293	0,130
7	**16 x 16**	**256**	37,221	4,256	4,385	0,119
8	**32 x 32**	**1024**	37,580	4,451	4,524	0,072
9	**64 x 64**	**4096**	37,664	4,547	4,586	0,038
10	**100 x 100**	**10000**	37,676	4,580	4,606	0,025

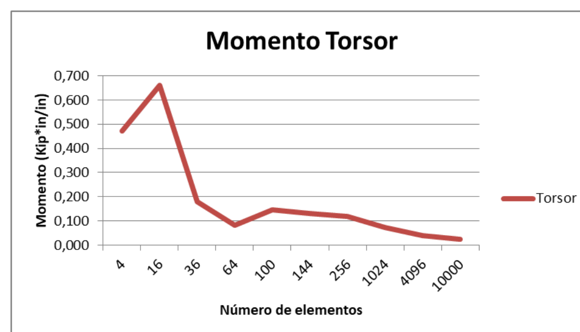
Fonte: Elaborado pelo próprio autor.

Figura (53): Esforços internos e deslocamentos - Malha 6 (12x12)



(a) Deslocamento (in)

(b) Momento em x e y (kip.in/in)



(c) Momento torsor (kip.in/in)

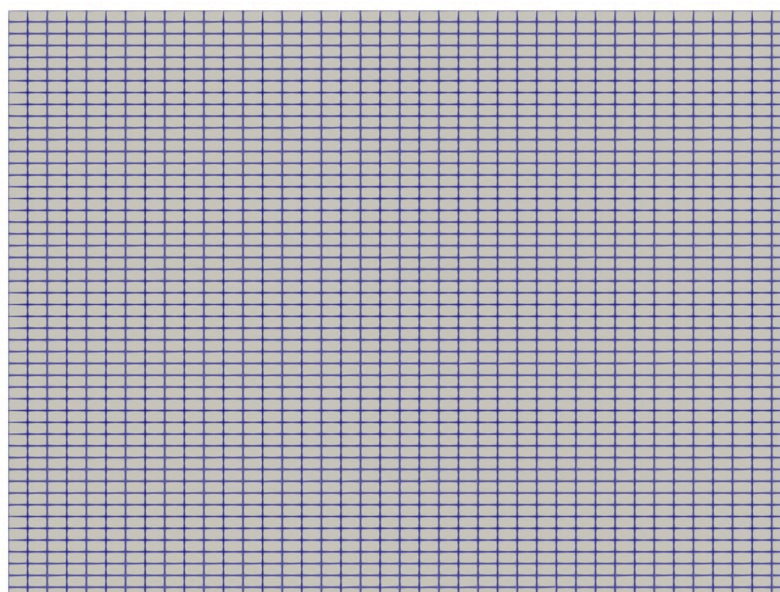
Fonte: Elaborado pelo próprio autor.

Em relação aos deslocamentos (Figura (53a)), observou-se uma convergência monótona partindo de valores inferiores, ou seja, assumiu um comportamento semelhante ao problema com malha retangular. Entretanto, apesar de apresentar sinais de convergência, este não se aproxima do valor analítico de referência. Em relação aos momentos fletores nas direções de x e y, de acordo com a figura (53b), observa-se a assimetria dos esforços, estabilizando-se à medida que a malha é refinada. O momento torsor, (Figura (53c)), por sua vez, apresentou um comportamento oscilatório, aproximando-se de zero medida que a malha é refinada.

Esses resultados corroboram com Oñate (2013), em que afirma que a convergência desse elemento não é garantida para formas quadrilaterais arbitrárias. Esse fato, de certa maneira, limita o uso do elemento a domínios que podem ser discretizados apenas em elementos de placa retangulares. No entanto, nesses casos, o autor afirma que é um elemento preciso como verificado nos exemplos deste tópico.

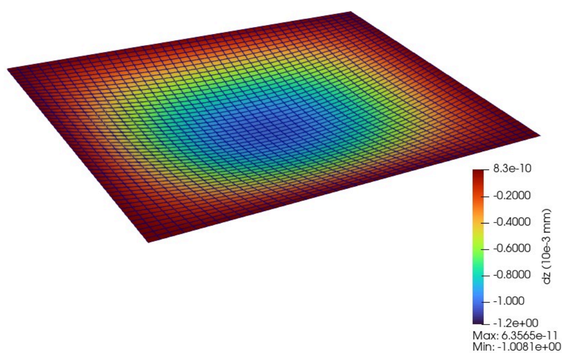
Para verificar a acurácia do elemento, será modelado um problema apresentado na obra de Araújo (2010), em uma laje de dimensões 4 m x 3 m, com espessura de 10 cm e carregamento uniformemente distribuído de 5 kN/m². É adotado 0,2 para o coeficiente de Poisson e adotado um concreto C30. Para tal, o domínio de estudo foi discretizado em uma malha com 2000 elementos, totalizando 2091 nós e, portanto, 6273 graus de liberdade. A malha discretizada é apresentada na figura (54). O resultado da análise é apresentado na figura (55).

Figura (54): Domínio discretizado.

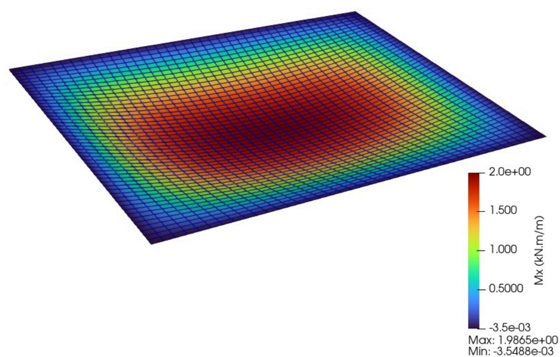


Fonte: Elaborado pelo próprio autor.

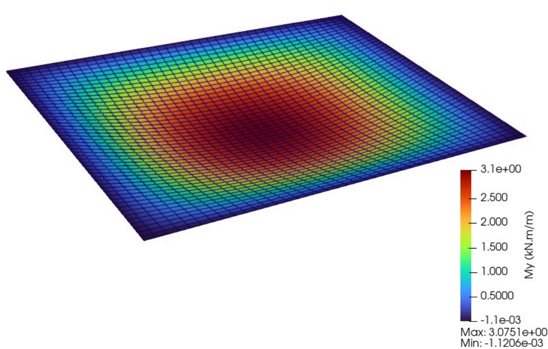
Figura (55): Esforços internos e deslocamentos - Malha 6 (12x12)



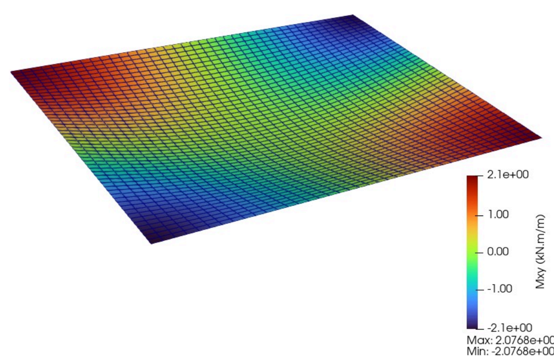
(a) Deslocamento (mm)



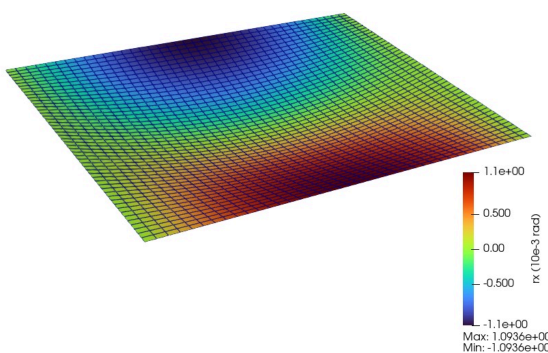
(b) Momento em x (kN.m/m)



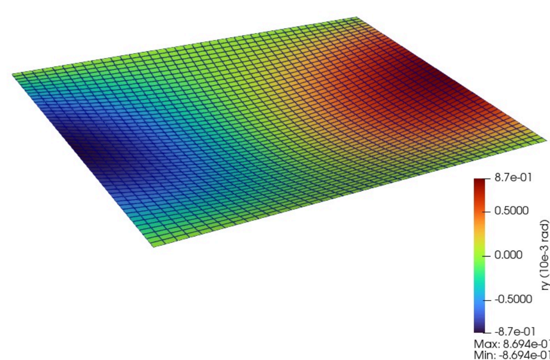
(c) Momento em y (kN.m/m)



(d) Momento torsor (kN.m/m)



(e) Rotação em x (10-3 rad)



(f) Rotação em y (10e-3 rad)

Fonte: Elaborado pelo próprio autor.

Os valores numéricos de esforços internos e deslocamentos foram coletados na tabela (6) abaixo.

Tabela (6): Resultados das análises.

RESULTADOS				
MOMENTOS FLETORES POSITIVOS E DESLOCAMENTOS				
	Deslocamento (mm)	Mx (kN.m/m)	My (kN.m/m)	Mxy (kN.m/m)
Araújo (2010)	1,000	1,990	3,070	2,080
Nascimento (2024)	1,008	1,986	3,075	2,076

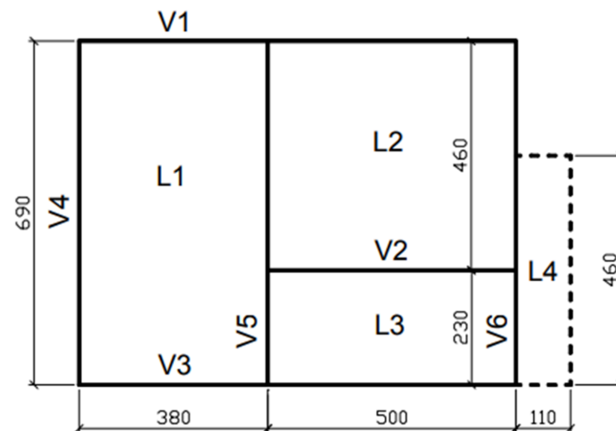
Fonte: Elaborado pelo próprio autor.

Os exemplos apresentados até aqui, levam em consideração placas retangulares analisadas em particular. Para esses casos, a NBR 6118/2014 afirma que, no caso de predominância de carregamentos permanentes, as lajes vizinhas podem ser consideradas isoladas, sendo realizada a compatibilização dos momentos de forma aproximada. Em contrapartida, o método dos elementos finitos permite uma análise conjunta, levando em consideração o comportamento simultâneo entre as lajes, de modo a simular um comportamento mais próximo da realidade.

Nesse sentido, NBR 6118/2014, argumenta que, o método dos elementos finitos pode ser usado para a obtenção dos esforços solicitantes e que as análises estruturais devem ser feitas com modelo estrutural adequado, bem como a discretização da estrutura deve ser suficiente para que não haja erros significativos nas análises. Além disso, o modelo estrutural deve representar a geometria dos elementos estruturais, características dos materiais, carregamentos atuantes e as condições de contorno. Sendo assim, a partir de então, será verificado o desempenho do elemento em lajes de pavimentos de edificações, em que todas as lajes serão analisadas em conjunto.

Para tal, serão utilizados como exemplo os pavimentos apresentados nos trabalhos de Pinheiro (2010) e Porto (2015). Inicialmente, será modelado o pavimento apresentado por Pinheiro (2010), cuja geometria é apresentada na figura (56).

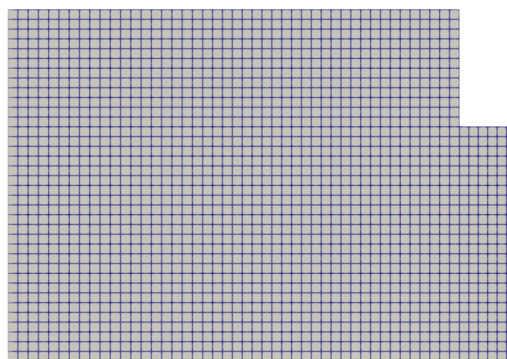
Figura (56): Domínio discretizado (Medidas em centímetros).



Fonte: Pinheiro, 2010.

Para este exemplo, o domínio como um todo foi discretizado em uma malha contendo 1764 elementos retangulares, totalizando 1852 nós e 5556 graus de liberdade. A malha discretizada é apresentada na figura (57).

Figura (57): Domínio discretizado.



Fonte: Elaborado pelo próprio autor.

Os carregamentos foram inseridos conforme utilizados pelo autor (Tabela (7)). Para a análise, foi considerada uma laje maciça de 10 cm de espessura utilizando um concreto de classe C25. Como sugere o item 14.7.3 da NBR 6118/2014, foi utilizado 0,2 para o coeficiente de Poisson.

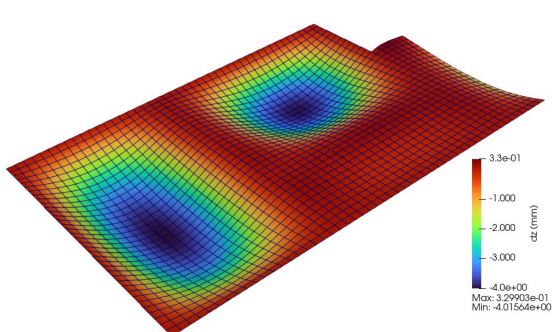
Tabela (7): Carregamentos.

CARREGAMENTOS NAS LAJES				
	Laje 01	Laje 02	Laje 03	Laje 04
Ação (kN/m²)	7,5	7,5	7,5	6,5

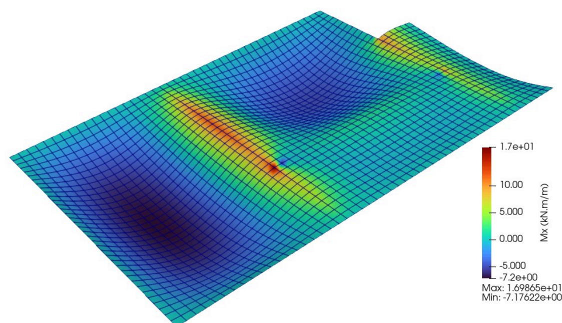
Fonte: Elaborado pelo próprio autor.

Para a laje 4, o autor considera uma carga linear de 4,09 kN/m na extremidade da laje, simulando o carregamento devido ao guarda corpo. O resultado das análises está representado na figura (58).

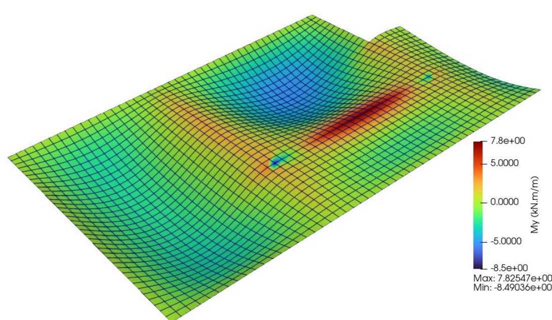
Figura (58): Esforços internos e deslocamentos - Malha 6 (12x12)



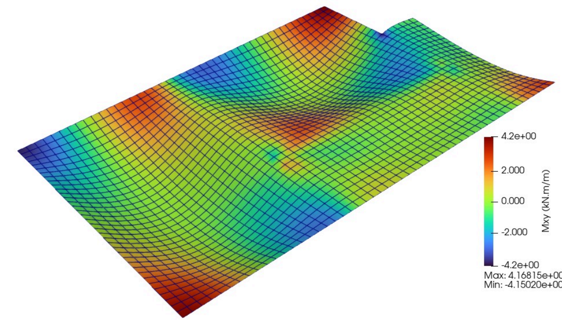
(a) Deslocamento (mm)



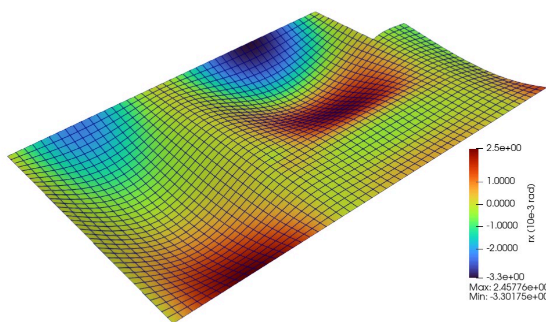
(b) Momento em x (kN.m/m)



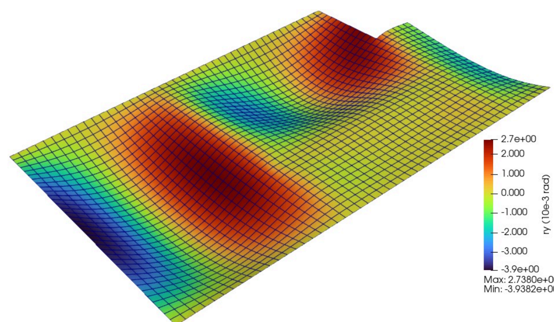
(c) Momento em y (kN.m/m)



(d) Momento torsor (kN.m/m)



(e) Rotação em x (10-3 rad)



(f) Rotação em y (10e-3 rad)

Fonte: Elaborado pelo próprio autor.

Os resultados obtidos nas análises foram coletados na tabela (8). Alguns resultados foram encontrados pelo autor utilizando o método das tabelas e foram elencados na tabela (9) e servem como referência para as análises.

Tabela (8): Resultados das análises.

RESULTADOS - Nascimento (2024)							
MOMENTOS FLETORES POSITIVOS E DESLOCAMENTOS					MOMENTOS FLETORES NEGATIVOS		
	Deslocamento (mm)	Mx (kN.m/m)	My (kN.m/m)	Mxy (kN.m/m)	Bordo	Mxe (kN.m/m)	Mye (kN.m/m)
LAJE 01	4,01	7,17	2,94	3,72/-3,86	L1 - L2	11,89	-
LAJE 02	3,96	4,92	5,15	3,06/-4,09	L1 - L3	11,07	-
LAJE 03	0,207	0,704	1,84	0,72/-1,11	L3 - L2	-	2,42
LAJE 04	1,44	-	-	-	L2 - L4	9,61	-

Fonte: Elaborado pelo próprio autor.

Tabela (9): Flecha e esforços internos compatibilizados do autor - Pinheiro (2010)

RESULTADOS - Pinheiro (2010)							
MOMENTOS FLETORES POSITIVOS E DESLOCAMENTOS					MOMENTOS FLETORES NEGATIVOS		
	Deslocamento (mm)	Mx (kN.m/m)	My (kN.m/m)	Mxy (kN.m/m)	Bordo	Mxe (kN.m/m)	Mye (kN.m/m)
LAJE 01	-	6,26	1,80	-	L1 - L2	13,73	-
LAJE 02	4,1	6,36	5,73	-	L1 - L3	3,25	-
LAJE 03	-	0,63	2,79	-	L3 - L2	-	4,96
LAJE 04	-	-	-	-	L2 - L4	8,43	-

Fonte: Elaborado pelo próprio autor.

Os deslocamentos e esforços internos vazios na tabela (9), não foram calculados pelo autor e, portanto, não são utilizados na comparação. Quanto aos deslocamentos, o valor da flecha na laje 2 do modelo se aproximou daquela calculada pelo autor, com uma diferença de 0,14 mm. Quanto aos momentos fletores em x e y, estes apresentaram algumas diferenças em relação ao modelo da autor.

A princípio, a metodologia de obtenção dos esforços internos são diferentes, e, portanto, é plausível a diferença entres os valores. O método das tabelas utilizado por Pinheiro (2010) oferece mais praticidade, sendo uma metodologia mais simplificada e útil em projetos preliminares. É o caso do exemplo do Araújo (2010), analisado anteriormente.

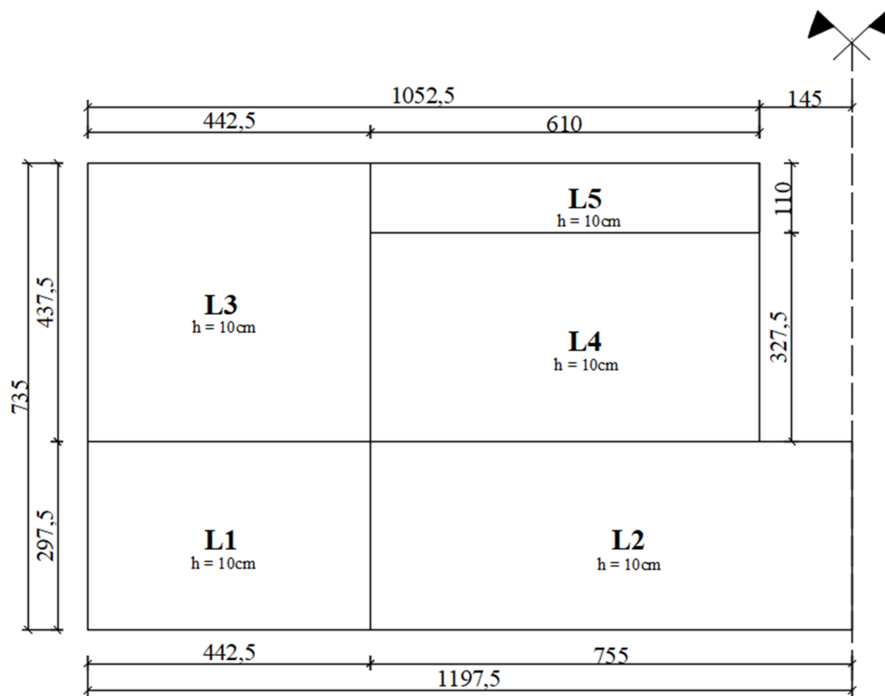
Uma observação importante em relação ao método dos elementos finitos, é que este fornece uma análise mais detalhada e precisa especialmente em geometrias mais complexas. A exemplo disso, pode-se utilizar o modelo dos momentos fletores em x da figura (58b).

Neste modelo, observa-se, por exemplo, que a distribuição dos momentos negativos ao longo da viga 5 não é constante, com valores máximos próximos da região central. Essa informação é útil para o engenheiro, uma vez que é possível detectar a região mais apropriada para aplicar um reforço caso seja conveniente, podendo trazer economia na execução do projeto.

Nas figuras (58b) e (58c) são apresentados picos de momento fletores positivos e negativos no modelo. Para esse fato, cabe uma análise mais criteriosa e está fora do escopo deste trabalho. Em relação aos momentos torsores representados pela figura (58d), o modelo está de acordo com Araújo (2010), em que afirma que os momentos torsores ocorrem nos cantos apoiados. O modelo também apresenta rotações coerentes, com magnitudes máximas próximas aos apoios como apresentado pela figura (58c).

O segundo exemplo é apresentado na obra de Porto (2015). Este problema refere-se a análise de um pavimento de uma edificação residencial com 3 pavimentos tipo com 2 apartamentos por pavimento e garagem no andar térreo. O pavimento tipo possui 5 lajes cuja geometria é apresentada na figura (59).

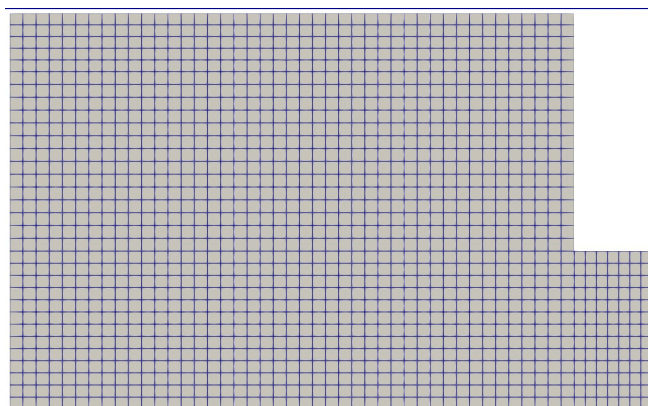
Figura (59): Lajes de um edifício residencial



Fonte: Porto, 2015. Adaptado. (Medidas em centímetros)

Devido a simetria do problema, o domínio de estudo será apenas um dos lados da edificação apresentada pela figura (59). Uma vez definido o domínio de estudo, este é discretizado em uma malha contendo 1467 elementos retangulares, totalizando 1550 nós e, portanto, 4650 graus de liberdade. A discretização do domínio é apresentada pela figura (60).

Figura (60): Edifício residencial apresentado - Porto (2015).



Fonte: Elaborado pelo próprio autor.

Os carregamentos foram inseridos conforme utilizados pelo autor (Tabela (10)). Para a análise, foi considerada uma laje maciça de 10 cm de espessura utilizando um concreto de classe C30. Como sugere o item 14.7.3 da NBR 6118/2014, também foi utilizado 0,2 para o coeficiente de Poisson. O resultado das análises está representado na figura (61).

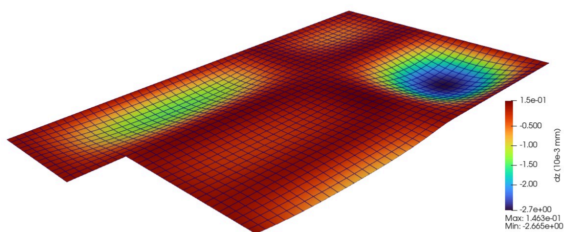
Tabela (10): Carregamentos das Lajes - Porto (2015).

CARREGAMENTOS NAS LAJES					
	Laje 01	Laje 02	Laje 03	Laje 04	Laje 05
Ação (kN/m²)	6,7	8,18	6,57	5,0	5,0

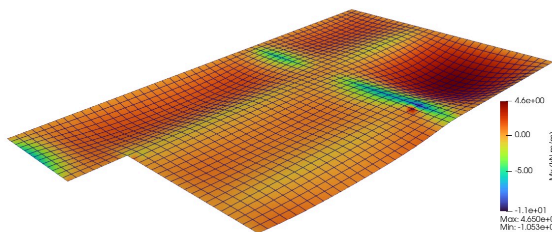
Fonte: Elaborado pelo próprio autor.

Na extremidade da laje 05, o autor considera uma carga linear de 2 kN/m e um momento também aplicado linearmente de 0,88 kN.m/m, simulando os efeitos causados por um guarda corpo. A aplicação desses carregamentos foi implantada conforme exposto no tópico 3 deste trabalho.

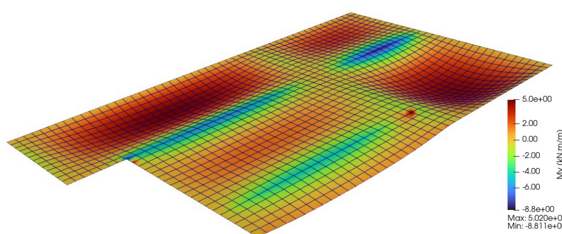
Figura (61): Esforços internos e deslocamentos - Malha 6 (12x12)



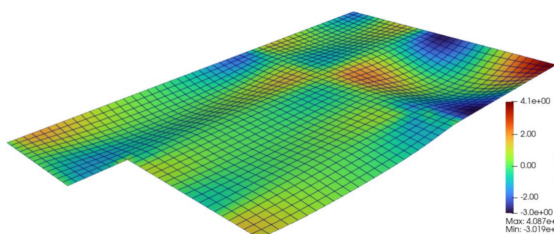
(a) Deslocamento (mm)



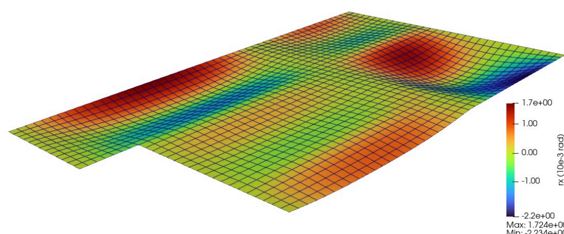
(b) Momento em x (kN.m/m)



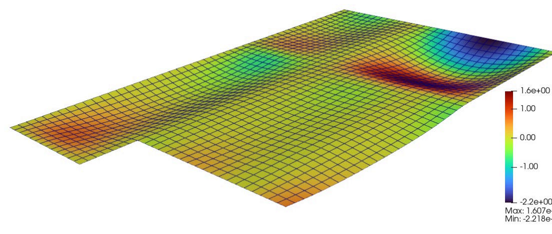
(c) Momento em y (kN.m/m)



(d) Momento torsor (kN.m/m)



(e) Rotação em x (10-3 rad)



(f) Rotação em y (10e-3 rad)

Fonte: Elaborado pelo próprio autor.

Os resultados obtidos nas análises foram coletados na tabela (11). Alguns resultados foram encontrados pelo autor utilizando o método das tabelas e foram elencados na tabela (12) e servem como referência para as análises.

Tabela (11): Resultados das análises.

RESULTADOS - Nascimento (2024)							
MOMENTOS FLETORES POSITIVOS E DESLOCAMENTOS					MOMENTOS FLETORES NEGATIVOS		
	Deslocamento (mm)	Mx (kN.m/m)	My (kN.m/m)	Mxy (kN.m/m)	Bordo	Mxe (kN.m/m)	Mye (kN.m/m)
LAJE 01	0,599	1,396	2,839	1,13/-1,53	L1 - L2	5,032	-
LAJE 02	1,419	1,655	5,020	1,68/-1,82	L1 - L3	-	8,203
LAJE 03	2,664	4,649	4,635	3,85/-2,93	L3 - L4	6,696	-
LAJE 04	0,282	0,719	1,716	0,79/-0,86	L2 - L4	-	6,304
LAJE 05	0,808	-	-	-	L4 - L5	-	4,643
LAJE 02*	-	-	-	-	L2 - L2	6,171	-
*Momento negativo na interface de simetria							

Fonte: Elaborado pelo próprio autor.

A coluna referente aos momentos torsores apresenta a máxima e a mínima magnitude desse esforço em cada laje respectivamente. O autor não trata dos esforços de momento torsor em suas análises.

Tabela (12): Flecha e esforços internos compatibilizados do autor - Porto (2015)

RESULTADOS - Porto (2015)							
MOMENTOS FLETORES POSITIVOS E DESLOCAMENTOS					MOMENTOS FLETORES NEGATIVOS		
	Deslocamento (mm)	Mx (kN.m/m)	My (kN.m/m)	Mxy (kN.m/m)	Bordo	Mxe (kN.m/m)	Mye (kN.m/m)
LAJE 01	0,740	1,34	2,81	-	L1 - L2	5,56	-
LAJE 02	1,280	1,25	5,46	-	L1 - L3	-	7,48
LAJE 03	1,880	3,91	3,77	-	L3 - L4	7,03	-
LAJE 04	0,950	1,02	3,08	-	L2 - L4	-	7,80
LAJE 05	0,323	-	-	-	L4 - L5	-	6,11
LAJE 02*	-	-	-	-	L2 - L2	6,34	-
*Momento negativo na interface de simetria							

Fonte: Elaborado pelo próprio autor.

5 CONCLUSÕES E CONSIDERAÇÕES FINAIS

Com base nas análises através dos exemplos trabalhados, observou-se que o elemento finito de placa implementado neste trabalho funciona bem e pode ser usado para obtenção dos esforços solicitantes em lajes finas de concreto armado. Na tentativa de usar esse elemento em domínios mais gerais, as equações (83) utilizadas por Melosh (1990) para a substituição das coordenadas cartesianas foram substituídas pelas equações (55) apresentadas por Vaz (2011). Com isso, o determinante da matriz jacobiana dos elementos não é mais constante, possibilitando o uso de elementos quadrangulares de geometria distorcida. Com essa nova formulação, o elemento ganhou adaptabilidade, podendo ser utilizado em elementos com geometria distorcida.

Porém, ao ser submetido a malhas com elementos distorcidos, o elemento não apresentou um desempenho satisfatório, apresentando deslocamentos com sinais de convergência para valores diferentes do valor analítico de referência. Além disso, no caso da malha distorcida, os esforços internos apresentaram sinais de assimetria e comportamento perturbado, não apresentando sinais de convergência definidos. Apesar disso, segundo Oñate (2013), isso não invalida o elemento, uma vez que funciona bem em domínios que podem ser discretizados em elementos retangulares. Nesse caso, o elemento foi testado na prática em pavimentos de edificações, obtendo-se resultados razoáveis.

É importante frisar que, apesar da mudança nas equações de substituição, não houve sinais de perda de desempenho do elemento. Cabe ressaltar ainda, a eficiência da solução do sistema de equações por meio de matrizes esparsas através da biblioteca *scipy* em relação ao modo convencional de resolução oferecido pela biblioteca *numpy*. Mediante as análises, foi observado que o tempo de processamento dos problemas reduziu consideravelmente. Portanto, essa ferramenta se mostrou útil, uma vez que agiliza a solução de sistemas de equações de ordens altas, geradas por malhas muito densas. Cabe ressaltar ainda, as vantagens obtidas no uso de métodos computacionais de engenharia como a utilização do *python* através da plataforma *Google Colaboratory* e *paraview*.

REFERÊNCIAS

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 6118**: Projeto de Estruturas de Concreto - Procedimento. Rio de Janeiro, 2014.

ARAÚJO, José Milton de. **Curso de Concreto Armado.** 3.ed. Rio Grande: Dunas, 2010.

ALVES, William Pereira. **Programação Python**: Aprenda de Forma Rápida. São Paulo, Expressa, 2021.

CARVALHO, Jeferson da Silva. **Estudo sobre Elementos Finitos de placas Baseados na Teoria de Reissner-Mindlin.** 2019. Trabalho monográfico (Especialização em estruturas) - Universidade Federal de Minas Gerais, Belo Horizonte, 2019. Disponível em: https://repositorio.ufmg.br/bitstream/1843/48513/1/TrabalhoFinalEspecializacao-JeffersonCarvalho-2019a.pdf. Acesso em: 23 set. 2024.

DIAS, N. L. **A Teoria da Flexão de placas Envolvendo a Equação Diferencial de Lagrange.** 2019. Trabalho de conclusão de Curso (Bacharelado em Matemática Aplicada) - Universidade Federal do Rio Grande, Rio Grande do Sul, 2019. Disponível em: https://imef.furg.br/images/documentos/matematica-aplicada/monografias/2019-Nickolas_Leitao_Dias.pdf . Acesso em: 23 set. 2024.

HJELMSTAD, K. D. **Fundamentals of Structural Mechanics**. 2. ed. [S.l]: Springer, 2005.

HIBBELER, R. C. **Resistência dos Materiais.** 7. ed. São Paulo: Pearson, 2010.

HERRERA, P. **Pyevtk 1.6.0**. Disponível em: https://pypi.org/project/pyevtk/. Acesso em: 29 jul. 2024.

LONGO, L. F. Análise de placas pela Teoria da Elasticidade. **AltoQi**, 2018. Disponível em: https://suporte.altoqi.com.br/hc/pt-br/articles/360003053334. Acesso em: 22 set. 2024.

LOGAN, Daryl L. **A first course in the finite element method.** 5 ed. Platteville: Cengage Learning, 1976.

MELOSH, R. J. **Structural Engineering Analysis by Finite Elements**. New Jersey: Prentice Hall, 1990.

SORIANO, H. L. **Método de elementos finitos em análise de estruturas**. 1. ed. São Paulo: EDUSP, 2003. Disponível em: https://books.google.com.br/books?id=ZQrMIb0Q9gYC&printsec=frontcover#v=onepage&q&f=false . Acesso em: 22 set. 2024.

SILVA, J. R. de B. **Análise de pilares-parede de concreto armado via método dos elementos finitos**. 2022. Tese (Doutorado em Estruturas) - Universidade Federal de Pernambuco, Recife, 2022. Disponível em: https://repositorio.ufpe.br/bitstream/123456789/44595/1/TESE%20Jordlly%20Reydson%20de%20Barros%20Silva.pdf . Acesso em: 23 set. 2024.

SZILARD, R. **Theories and Applications of Plate Analysis.** New Jersey: John Wiley & Sons, Inc, 2004.

SANTANA, Victor Araújo de. **Análise Estrutural de Placas Retangulares Submetidas a Carregamentos Estáticos Trapezoidais.** 2019. Trabalho de conclusão de curso (Bacharelado em Engenharia Civil) - Universidade Federal Rural de Pernambuco, Cabo de Santo Agostinho, 2019. Disponível em: https://repository.ufrpe.br/bitstream/123456789/1622/1/tcc_vitorcarneirosantana.pdf . Acesso em: 23 set. 2024.

THOMAS, George Brinton. **Cálculo.** 12. ed. São Paulo: Pearson Education do Brasil, 2013.

TIMOSHENKO, S.; GOODIER, J. N. **Theory of Elasticity.** New York: McGraw-Hill Book Company, 1951.

OÑATE, Eugênio. **Structural Analysis with the finite element method**: Linear statics. Barcelona: Springer, 2013.

PORTO, T. B. ; FERNANDES, D. S. G. **Curso Básico de Concreto Armado**: Conforme a NBR 6118:2014. São Paulo: Oficina de Textos, 2015.

PINHEIRO, L. M. **Fundamentos do Concreto e Projeto de Edifícios**. São Paulo: Departamento de engenharia de estruturas - Escola de engenharia de São Carlos, 2010. Disponível em: https://arquitetonica.wordpress.com/wp-content/uploads/2011/11/fundamentos-do-concreto-eesc-usp.pdf . Acesso em: 22 set. 2024.

VAZ, Luiz Eloy. **Método dos elementos finitos em análises de estruturas.** Rio de janeiro: Elsevier, 2011.

WILSON, Edward L. **Three-Dimensional Static and Dinamic Analysis of Structures**. 3. ed. California: Computers and Structures, Inc. 2002. Disponível em: http://civilwares.free.fr/ForLinh/3D%20Static%20And%20Dynamic%20Analysis%20Of%20Structures%20-%20E%20Wilson.pdf . Acesso em: 22 set. 2024.

ANEXO A - CÓDIGO EM PYTHON

ELEMENTO FINITO DE PLACA - TEORIA DE KIRCHHOFF

Resumo: O foco deste trabalho se refere ao comportamento de placas à flexão quando submetidas a carregamentos aplicados perpendicularmente a estas. Segundo VAZ (2011), as placas são elementos bidimensionais cuja espessura é muito menor que as outras dimensões. Para a modelagem do problema de placas, considera-se que esta se situa no plano médio da laje. Na Engenharia Civil, os elementos estruturais que podem ser modelados usando elemento finito de placa são lajes de concreto armado, tabuleiros de pontes, radeiers entre outros. Podemos citar para descrever o comportamento de lajes, a teoria de Kirchhof e a de Mindlin. A teoria abordada nesse trabalho é a teoria de Kirchhoff, basicamente atribuída para a análise de placas finas. A teoria de Mindlin é aplicada em casos de placas espessas e não será objeto deste exercício.

Arquivo: PLACAS À FLEXÃO

Objetivo: O código a seguir apresenta a formulação de um elemento finito de placa, presente em Vaz (2011), baseando-se na teoria de Kirchhoff aplicada à teoria de flexão de placas finas.

Desenvolvido por: Jailson Nascimento.

Criado em: 18/03/2024

Modificado em: 29/09/2024

REFERÊNCIAS BIBLIOGRÁFICAS

- ARAÚJO, J. M. de;**Curso de Concreto Armado** V.2. Rua Tiradentes, 105 - Cidade Nova - Rio Grande: Dunas, 2010.
- VAZ, LUIZ ELOY; **Método dos elementos finitos em análises de estruturas**. Rio de janeiro, Elsevier, 2011.
- WILSON, EDWARD L.; **Three-Dimensional Static and Dinamic Analysis of Structures**. California, Computers and Structures, Inc. 1998.
- HERRERA, P. **Evtk - export vtk**. Disponível em: <https:// pypi.org/ project/ pyevtk/ >, Acesso em: 19/06/2024.
- SILVA, J. R. de B.; **Análises de Pilares-Parede de Concreto Armado Via Método dos Elementos Finitos**. Recife, UFPE, 2022.
- OÑATE, EUGENIO. **Structural Analysis with the finite element method**. Linear statics. Volume 2. Beams, plates and shells. 1. ed. Barcelona: SPRINGER, 2013.
- PORTO, T. B.;FERNANDES, D. S. G.; **Curso Básico de concreto Armado: Conforme a NBR 6118:2014**.São Paulo, Oficina de Textos, 2015.
- PINHEIRO, L. M.; **Fundamentos do Concreto e Projeto de Edifícios**. São Paulo, Departamento de engenharia de estruturas - Escola de engenharia de São Carlos, 2010.


```
1 ### IMPORTAÇÕES
2 import numpy as np
3 import math
4 from scipy.sparse.linalg import spsolve
5 from scipy.sparse import csc_matrix
```

1 - DADOS DE ENTRADA DOS PROBLEMAS

1.1 - PRIMEIRO PROBLEMA: Exemplo de VAZ (2011) com 1 elemento

```
1 ##########################################################################
2 # DIMENSÕES DA PLACA (GEOMETRIA RETANGULAR):
3
4 H = 10 # Altura
5 L = 10 # Comprimento
6
7 # NUMERO DE ELEMENTOS NA HORIZONTAL E VERTICAL DA MALHA:
8
9 nL = 1     # Número de elementos ao longo do comprimento
10 nH = 1     # Número de elementos ao longo da altura
11
12 # DIMENSÕES DO ELEMENTO:
13
14 altura = H/nH     # Altura do elemento
15 base = L/nL       # Base do elemento
16
17 # DADOS GERAIS:
18
19 ne = nL*nH                # Número de elementos
20 nnos = (nL+1)*(nH+1)      # Número de nós da malha
21 GL = nnos*3               # Graus de liberdade
22 esp = 2                   # Espessura da placa
23 P = 1                     # Carga aplicada
24
```

```
25 # PROPRIEDADES DO MATERIAl:
26
27 vp = 0                   # Constante de poisson
28 El = 1000                # Módulo de elasticidade
29
30
31 # MATRIZ DE INCIDÊNCIA:
32
33 conec = np.array([[1, 3, 4, 2]])
34
35 # MATRIZ DE COORDENADAS NODAIS:
36
37 coord =np.array([[0,  0],
38                  [0, 10],
39                  [10, 0],
40                  [10,10]])
41
42 # NÓS GLOBAIS EM QUE SERÃO RESTRINGIDOS OS GRAUS DE LIBERDADE:
43
44 b5 = np.array([[1], [2]])   # Nós dos extremos
45
46 rrx5 = 3 * b5      #rotação em x
47 rry5 = rrx5 - 1    #rotação em y
48 rw5  = rrx5 - 2    #Deslocamento
49
50 # VETOR DE GRAUS DE LIBERDADE RESTRIGIDOS:
51
52 GLR = np.concatenate((rw5, rry5, rrx5))
53
54 # VETOR DE CARGAS EXTERNAS:
55
56 f = np.zeros((GL,1))
57
58 f[7-1][0]  = P          # Carga aplicada no nó 5 dirigida para cima
59 f[10-1][0] = P          # Carga aplicada no nó 6 dirigida para cima
60
61 # DADOS PARAVIEW:
62 nome = 'Primeiro Problema'
63 FatorEscala = 0.08
```

1.2 - SEGUNDO PROBLEMA: Exemplo de VAZ (2011) com 2 elementos

```
1 ##########################################################################
2 # DIMENSÕES DA PLACA (GEOMETRIA RETANGULAR):
3
4 H = 10      # Altura
5 L = 10      # Comprimento
6
7 # NÚMERO DE ELEMENTOS NA HORIZONTAL E VERTICAL DA MALHA:
8
9 nL = 2     # Número de elementos ao longo do comprimento
10 nH = 1     # Número de elementos ao longo da altura
11
12 # DIMENSÕES DO ELEMENTO:
13
14 altura = H/nH     # Altura do elemento
15 base = L/nL       # Base do elemento
16
17 # DADOS GERAIS:
18
19 ne = 2                    # Número de elementos
20 nnos = 6                  # Número de nós da malha
21 GL = nnos*3               # Graus de liberdade
22 esp = 2                   # Espessura da placa
23 P = 1                     # Carga aplicada
24 El = 1000                 # Módulo de elasticidade
25 vp = 0                  # Coeficiente de Poisson
26
27 # MATRIZ DE INCIDÊNCIA:
28
29 conec = np.array([[1, 3, 4, 2],
30                   [3, 5, 6, 4]])
31
32 # MATRIZ DE COORDENADAS NODAIS:
33 coord =np.array([[ 0, 0],
34                  [ 0,10],
35                  [ 5, 0],
36                  [ 5,10],
37                  [10, 0],
38                  [10,10]])
```

```
39
40 # VETOR DE GRAUS DE LIBERDADE RESTRIGIDOS:
41
42 GLR = np.array(([1],
43                 [2],
44                 [3],
45                 [4],
46                 [5],
47                 [6]))
48
49 # VETOR DE CARGAS EXTERNAS:
50
51 f = np.zeros((GL,1))
52 f[13-1][0] = P          # Carga aplicada no nó 5 dirigida para cima
53 f[16-1][0] = P          # Carga aplicada no nó 6 dirigida para cima
54
55 # DADOS PARAVIEW:
56 nome = 'Segundo Problema'
57 FatorEscala = 0.08
```

Clique duas vezes (ou pressione "Enter") para editar

1.3 - TERCEIRO PROBLEMA: Exemplo 8.9.1 One Element Beam. Wilson (1998)

```
1 # DADOS GERAIS:
2
3 ne = 1                    # Número de elementos
4 nnos = 4                  # Número de nós da malha
5 GL = nnos*3               # Graus de liberdade
6 esp = 2                   # Espessura da placa (m)
7 P = 0.5                   # Carga aplicada (kN)
8 El = 10000                # Módulo de Elasticidade
9 vp = 0                    # Coeficiente de Poisson
10
11
12 # DIMENSÕES DOS ELEMENTOS:
13
14 base = 6   # Base do elemento (m)
15 altura = 0.2  # Altura do elemento (m)
16
17
18 # MATRIZ DE INCIDÊNCIA:
19
20 conec = np.array([[1, 3, 4, 2]])
21
22 # MATRIZ DE COORDENADAS NODAIS:
23
24 coord =np.array([[0,   0],
25                  [0, altura],
26                  [base,   0],
27                  [base, altura]])
28
29 # PROPRIEDADES DOS MATERIAIS:
30
31 #E = np.array([[10000]]) # Módulo de elasticidade dos elemento
32
33 #v = np.array([[0]])     # Coeficiente de Poisson
34
35
36 # VETOR DE GRAUS DE LIBERDADE RESTRIGIDOS:
37
38 GLR = np.array(([1],
39                 [2],
40                 [3],
41                 [4],
42                 [5],
43                 [6]))
44
45 # VETOR DE CARGAS EXTERNAS:
46
47 f = np.zeros((GL,1))
48 f[7-1][0] = P          # Carga aplicada no nó 5 dirigida para cima
49 f[10-1][0] = P         # Carga aplicada no nó 6 dirigida para cima
50
51 # DADOS PARAVIEW:
52 nome = 'Terceiro Problema'
53 FatorEscala = 0.08
```

1.4 - QUARTO PROBLEMA: Pavimento com múltiplas lajes. Pinheiro (2010)

1.4.1 - Dados gerais e apresentação do problema

```
1 # DADOS GERAIS:
2
3 ne = 1764                         # Número de elementos
4 nnos = 1852                        # Número de nós da malha
5 GL = nnos*3                       # Graus de liberdade
6 esp = 0.1                         # Espessura da placa
7 El = 0.85*1000*5600*(25)**(1/2)        # Módulo de elasticidade
8 vp = 0.2                          # Coeficiente de poisson
```

1.4.2 - Matriz de conectividade

```
1 # MATRIZ DE CONECTIVIDADE:
2
3 AA = np.array([1, 38, 39, 2])
4 BB = np.array([1703, 1728, 1729, 1704])
5
6 CC = np.zeros((46,4))
7 for i in range(46):
8   CC[i] = AA +37*i
9 #print(CC)
10 DD = np.zeros((5,4))
11 for i in range(5):
12   DD[i] = BB + 25*i
13 #print(DD)
14 EE = np.concatenate((CC,DD), axis=0)
15
16 #print(EE)   # MATRIZ DE CONECTIIIDADE REFERENTE A PRIMEIRA LINHA DE ELEMENTOS DA MALHA
17
18 Matriz = []
19 for i in range(24):
20   FF = EE + 1*i
21   Matriz.append(FF)
22   #print(FF)
23
24 # Juntando todas as matrizes em uma única matriz:
25
26 GG = np.concatenate(Matriz)
27 #print(GG)
28
29 ## PARTE DE CIMA DA MALHA (fora do balanço)
30
31 HH = np.array([25, 62, 63, 26])
32
33 JJ = np.zeros((45,4))
34 for i in range(45):
35   JJ[i] = HH +37*i
36 #print(JJ)
37
38 Matriz2 = []
39 for i in range(12):
40   KK = JJ + 1*i
41   Matriz2.append(KK)
42   #print(KK)
43
44 # Juntando todas as matrizes em uma única matriz:
45
46 LL = np.concatenate(Matriz2)
47 #print(LL)
48
49 conec = np.concatenate((GG,LL), axis=0)
50 print("Dimensão da matriz de conectividade:",conec.shape)
```

```
Dimensão da matriz de conectividade: (1764, 4)
```

1.4.3 - Matriz de coordenadas nodais

```
1 # PRIMEIRA MATRIZ DE COORDENADAS NODAIS
2
3 coord_1 = np.array([[0,   0*230/1200],
4                     [0,   1*230/1200],
5                     [0,   2*230/1200],
```

```
 6                     [0,   3*230/1200],
 7                     [0,   4*230/1200],
 8                     [0,   5*230/1200],
 9                     [0,   6*230/1200],
10                     [0,   7*230/1200],
11                     [0,   8*230/1200],
12                     [0,   9*230/1200],
13                     [0,  10*230/1200],
14                     [0,  11*230/1200],
15                     [0,  12*230/1200],
16
17                     [0,  2.30+1*230/1200],
18                     [0,  2.30+2*230/1200],
19                     [0,  2.30+3*230/1200],
20                     [0,  2.30+4*230/1200],
21                     [0,  2.30+5*230/1200],
22                     [0,  2.30+6*230/1200],
23                     [0,  2.30+7*230/1200],
24                     [0,  2.30+8*230/1200],
25                     [0,  2.30+9*230/1200],
26                     [0,  2.30+10*230/1200],
27                     [0,  2.30+11*230/1200],
28                     [0,  2.30+12*230/1200],
29
30                     [0,  4.60+1*230/1200],
31                     [0,  4.60+2*230/1200],
32                     [0,  4.60+3*230/1200],
33                     [0,  4.60+4*230/1200],
34                     [0,  4.60+5*230/1200],
35                     [0,  4.60+6*230/1200],
36                     [0,  4.60+7*230/1200],
37                     [0,  4.60+8*230/1200],
38                     [0,  4.60+9*230/1200],
39                     [0,  4.60+10*230/1200],
40                     [0,  4.60+11*230/1200],
41                     [0,  4.60+12*230/1200]])
42 #print(coord_1.shape)
43 # Montando as demais matrizes de coordenadas com a primeira parte da malha:
44
45 coord_2 = np.zeros((20,2))
46 for i in range(20):
47   coord_1[:,0] = (380/1900)*i
48   coord_2 = np.vstack((coord_2,coord_1))
49 coord_3 = coord_2[20:]
50
51 #print(coord_3)
52
53 #Matriz que contempla a segunda (Abcissa 2) divisão da linha de vigas verticais:
54
55 coord_5 = np.array([[(3.80+(500/2600)),    0*230/1200],
56                     [(3.80+(500/2600)),    1*230/1200],
57                     [(3.80+(500/2600)),    2*230/1200],
58                     [(3.80+(500/2600)),    3*230/1200],
59                     [(3.80+(500/2600)),    4*230/1200],
60                     [(3.80+(500/2600)),    5*230/1200],
61                     [(3.80+(500/2600)),    6*230/1200],
62                     [(3.80+(500/2600)),    7*230/1200],
63                     [(3.80+(500/2600)),    8*230/1200],
64                     [(3.80+(500/2600)),    9*230/1200],
65                     [(3.80+(500/2600)),   10*230/1200],
66                     [(3.80+(500/2600)),   11*230/1200],
67                     [(3.80+(500/2600)),   12*230/1200],
68
69                     [(3.80+(500/2600)),    2.30+1*230/1200],
70                     [(3.80+(500/2600)),    2.30+2*230/1200],
71                     [(3.80+(500/2600)),    2.30+3*230/1200],
72                     [(3.80+(500/2600)),    2.30+4*230/1200],
73                     [(3.80+(500/2600)),    2.30+5*230/1200],
74                     [(3.80+(500/2600)),    2.30+6*230/1200],
75                     [(3.80+(500/2600)),    2.30+7*230/1200],
76                     [(3.80+(500/2600)),    2.30+8*230/1200],
77                     [(3.80+(500/2600)),    2.30+9*230/1200],
78                     [(3.80+(500/2600)),    2.30+10*230/1200],
79                     [(3.80+(500/2600)),    2.30+11*230/1200],
80                     [(3.80+(500/2600)),    2.30+12*230/1200],
81
82                     [(3.80+(500/2600)),    4.60+1*230/1200],
83                     [(3.80+(500/2600)),    4.60+2*230/1200],
84                     [(3.80+(500/2600)),    4.60+3*230/1200],
85                     [(3.80+(500/2600)),    4.60+4*230/1200],
86                     [(3.80+(500/2600)),    4.60+5*230/1200],
87                     [(3.80+(500/2600)),    4.60+6*230/1200],
```

```
 88                     [(3.80+(500/2600)),    4.60+7*230/1200],
 89                     [(3.80+(500/2600)),    4.60+8*230/1200],
 90                     [(3.80+(500/2600)),    4.60+9*230/1200],
 91                     [(3.80+(500/2600)),    4.60+10*230/1200],
 92                     [(3.80+(500/2600)),    4.60+11*230/1200],
 93                     [(3.80+(500/2600)),    4.60+12*230/1200]])
 94 #print(coord_5)
 95 # Montando as demais matrizes de coordenadas com a primeira parte da malha:
 96
 97 coord_4 = np.zeros((27,2))
 98 for i in range(1,27):
 99   coord_5[:,0] = 3.80+(500/2600)*i
100   coord_4 = np.vstack((coord_4,coord_5))
101 coord_6 = coord_4[27:]
102
103 #print(coord_6)
104
105 #Matriz que contempla a segunda (Abcissa 3) divisão da linha de vigas verticais:
106
107 coord_8 = np.array([[(8.80+(110/600)),     0*460/2400],
108                     [(8.80+(110/600)),     1*460/2400],
109                     [(8.80+(110/600)),     2*460/2400],
110                     [(8.80+(110/600)),     3*460/2400],
111                     [(8.80+(110/600)),     4*460/2400],
112                     [(8.80+(110/600)),     5*460/2400],
113                     [(8.80+(110/600)),     6*460/2400],
114                     [(8.80+(110/600)),     7*460/2400],
115                     [(8.80+(110/600)),     8*460/2400],
116                     [(8.80+(110/600)),     9*460/2400],
117                     [(8.80+(110/600)),    10*460/2400],
118                     [(8.80+(110/600)),    11*460/2400],
119                     [(8.80+(110/600)),    12*460/2400],
120                     [(8.80+(110/600)),    13*460/2400],
121                     [(8.80+(110/600)),    14*460/2400],
122                     [(8.80+(110/600)),    15*460/2400],
123                     [(8.80+(110/600)),    16*460/2400],
124                     [(8.80+(110/600)),    17*460/2400],
125                     [(8.80+(110/600)),    18*460/2400],
126                     [(8.80+(110/600)),    19*460/2400],
127                     [(8.80+(110/600)),    20*460/2400],
128                     [(8.80+(110/600)),    21*460/2400],
129                     [(8.80+(110/600)),    22*460/2400],
130                     [(8.80+(110/600)),    23*460/2400],
131                     [(8.80+(110/600)),    24*460/2400]])
132
133 #print(coord_8)
134
135 # Montando as demais matrizes de coordenadas com a primeira parte da malha:
136
137 coord_7 = np.zeros((7,2))
138 for i in range(1,7):
139   coord_8[:,0] = 8.80+(1.10/6)*i
140   coord_7 = np.vstack((coord_7,coord_8))
141 coord_9 = coord_7[7:]
142
143 #print(coord_9)
144
145 ## JUNTANDO AS MATRIZES DE COORDENADAS NODAIS:
146
147 coord = np.concatenate((coord_3, coord_6, coord_9))
148
149 print("Tamanho do vetor de coordenadas:",coord.shape )
```

```
Tamanho do vetor de coordenadas: (1852, 2)
```

1.4.4 - Nós que serão restrigidos

```
 1 # Bordo 05:  (Nós dos extremos)
 2
 3 b6 = np.array([[1], [37], [1702], [1690], [1852], [1828]])
 4
 5 # Bordo 01(Esquerda):
 6
 7 b1 = np.zeros((35,1))
 8 for i in range(35):
 9   b1[i]= 2 + 1*i
10
11 # Bordo 02 (Baixo):
12
13 b2 = np.zeros((45,1))
```

```
14 for i in range(1,46):
15   b2[i-1]= 1 + 37*i
16
17 # Bordo 03 (Cima):
18
19 b3 = np.zeros((44,1))
20 for i in range(44):
21   b3[i]= 74 + 37*i
22
23 # Bordo 04 (Direita):
24
25 b4 = np.zeros((35,1))
26 for i in range(35):
27   b4[i]= 1667 + 1*i
28
29 # Viga 05:
30
31 v5 = np.zeros((25,1))
32 for i in range(25):
33   v5[i]= 753 + 37*i
34
35 # Viga 02:
36
37 v2 = np.zeros((35,1))
38 for i in range(35):
39   v2[i]= 705 + 1*i
40
41 #print("Bordo 01: ===========")
42 #print(b1)
43 #print("Bordo 02: ===========")
44 #print(b2)
45 #print("Bordo 03: ===========")
46 #print(b3)
47 #print("Bordo 04: ===========")
48 #print(b4)
49 #print("Viga 05: ===========")
50 #print(v5)
51 #print("Viga 02: ===========")
52 #print(v2)
53 #print("===========")
54
55 # LOCALIZAÇÃO DOS GRAUS DE LIBERDADE:
56
57 #Bordo 01
58
59 rrx1 = 3 * b1  - 0     #rotação em x
60 rry1 =   rrx1  - 1     #rotação em y
61 rw1  =   rrx1  - 2     #Deslocamento
62
63 #Bordo 02
64
65 rrx2 = 3 * b2  - 0     #rotação em x
66 rry2 =   rrx2  - 1     #rotação em y
67 rw2  =   rrx2  - 2     #Deslocamento
68
69 #Bordo 03
70
71 rrx3 = 3 * b3  - 0     #rotação em x
72 rry3 =   rrx3  - 1     #rotação em y
73 rw3  =   rrx3  - 2     #Deslocamento
74
75 #Bordo 04
76
77 rrx4 = 3 * b4  - 0     #rotação em x
78 rry4 =   rrx4  - 1     #rotação em y
79 rw4  =   rrx4  - 2     #Deslocamento
80
81 #Bordo dos extremos:
82
83 rrx6 = 3 * b6  - 0     #rotação em x
84 rry6 =   rrx6  - 1     #rotação em y
85 rw6  =   rrx6  - 2     #Deslocamento
86
87 # Viga 05:
88
89 vvx5 = 3 * v5  - 0     #rotação em x
90 vvy5 =   vvx5  - 1     #rotação em y
91 vw5  =   vvx5  - 2     #Deslocamento
92
93 # Viga 02:
94
95 vvx2 = 3 * v2  - 0     #rotação em x
```

```
96 vvy2 =   vvx2  - 1     #rotação em y
97 vw2  =   vvx2  - 2     #Deslocamento
98
99 # VETOR DE GRAUS DE LIBERDADE RESTRIGIDOS:
100
101 GLR1 = np.concatenate((rw1, rry1,
102                        rw2, rrx2,
103                        rw3, rrx3,
104                        rw4, rry4,
105                        rw6,
106                        vw5, vvx5,
107                        vw2, vvy2))
108
109 GLR = GLR1.astype(int)    # Transformação dos elementos do vetor em elementos inteiros!!!
```

1.4.5 - Nós internos de cada laje

```
1 ##################################
2
3 # NÓS DA LAJE 01:
4
5 q1 = np.zeros((18,1))
6 for i in range(18):
7   q1[i]= 39 + 37*i
8
9 #print(q1)
10
11 Matrixx = []
12 for i in range(35):
13   A = q1+1*i
14   Matrixx.append(A)
15   #print(A)
16 Q1 = np.concatenate(Matrixx)
17 #print(Q1)
18
19 #################################
20
21 # NÓS DA LAJE 02:
22
23 q2 = np.zeros((25,1))
24 for i in range(25):
25   q2[i]= 754 + 37*i
26
27 #print(q2)
28
29 Matrixx2 = []
30 for i in range(24):
31   A = q2+1*i
32   Matrixx2.append(A)
33   #print(A)
34 Q2 = np.concatenate(Matrixx2)
35 #print(Q2)
36
37 #################################
38
39 # NÓS DA LAJE 03:
40
41 q3 = np.zeros((25,1))
42 for i in range(25):
43   q3[i]= 742 + 37*i
44 #print(q3)
45
46 Matrixx3 = []
47 for i in range(11):
48   A3 = q3+1*i
49   Matrixx3.append(A3)
50   #print(A3)
51 Q3 = np.concatenate(Matrixx3)
52 #print(Q3)
53
54 #################################
55
56 # NÓS INTERNOS DA LAJE 04:
57
58 q4 = np.zeros((5,1))
59 for i in range(5):
60   q4[i]= 1704 + 25*i
61 #print(q4)
62
63 Matrixx4 = []
```

```
64 for i in range(23):
65   A4 = q4+1*i
66   Matrixx4.append(A4)
67   #print(A4)
68 Q4 = np.concatenate(Matrixx4)
69 #print(Q4)
70
71 #################################
72
73 # NÓS DO BORDO DA LAJE 04:
74
75 Q5 = np.zeros((25,1))
76 for i in range(25):
77   Q5[i]= 1828 + 1*i
78
79 #print(Q5)
80
81 ######################################
82
83 # NÓS ENTRE L1-L3:
84
85 NL13 = np.zeros((11,1))
86 for i in range(11):
87   NL13[i]= 705 + 1*i
88
89 ######################################
90
91 # NÓS ENTRE L1-L2:
92
93 NL12 = np.zeros((23,1))
94 for i in range(23):
95   NL12[i]= 717 + 1*i
96
97 ######################################
98
99 # NÓS ENTRE L2-L3:
100
101 NL23 = np.zeros((25,1))
102 for i in range(25):
103   NL23[i]= 753 + 1*i
104
105 ######################################
106
107 # NÓS ENTRE L2-L4:
108
109 NL24 = np.zeros((24,1))
110 for i in range(24):
111   NL24[i]= 1667 + 1*i
112
```

1.4.6 - Vetor de cargas externas

```
1 ########################################################################
2 # VETOR DE CARGAS EXTERNAS:
3
4 # CARGA DA LAJE 01:
5
6 P1 = 7.5*((380/1900)*(230/1200))
7
8 # CARGA DA LAJE 02:
9
10 P2 = 7.5*((500/2600)*(230/1200))
11
12 # CARGA DA LAJE 03:
13
14 P3 = 7.5*((500/2600)*(230/1200))
15
16 # CARGA DA LAJE 04:
17
18 P4 = 6.5*((110/600)*(230/1200))
19
20 # CARGA DO GUARDA CORPO LAJE 04:
21
22 P5 = 4.09*(46/240)
23
24 ################################################################################
25
26 nQ1 = 3 * Q1 - 2           # Grau de liberdade de deslocamento
27 nQ2 = 3 * Q2 - 2           # Grau de liberdade de deslocamento
28 nQ3 = 3 * Q3 - 2           # Grau de liberdade de deslocamento
```

```
29 nQ4 = 3 * Q4 - 2           # Grau de liberdade de deslocamento
30 nQ5 = 3 * Q5 - 2           # Grau de liberdade de deslocamento
31
32 nQ11 = nQ1.astype(int)     # Transformando a matriz em numeros inteiros
33 nQ22 = nQ2.astype(int)     # Transformando a matriz em numeros inteiros
34 nQ33 = nQ3.astype(int)     # Transformando a matriz em numeros inteiros
35 nQ44 = nQ4.astype(int)     # Transformando a matriz em numeros inteiros
36 nQ55 = nQ5.astype(int)     # Transformando a matriz em numeros inteiros
37
38 f = np.zeros((GL,1))
39 f[nQ11-1] = - P1        # Carga da laje 01 aplicada nos nós  dirigida para baixo!
40 f[nQ22-1] = - P2        # Carga da laje 02 aplicada nos nós  dirigida para baixo!
41 f[nQ33-1] = - P3        # Carga da laje 03 aplicada nos nós  dirigida para baixo!
42 f[nQ44-1] = - P4        # Carga da laje 04 aplicada nos nós  dirigida para baixo!
43 f[nQ55-1] = - P5        # Carga da laje 05 aplicada nos nós  dirigida para baixo!
44
45 # DADOS PARAVIEW:
46
47 nome = 'SextoProblema (Pinheiro)'
48 FatorEscala = 200
```

1.5 - QUINTO PROBLEMA: Exemplo 8.9.3 Simply Supported Square Plate. Wilson (1998) - (MALHA RETANGULAR)

```
 1 # DIMENSÕES DA PLACA (GEOMETRIA RETANGULAR):
 2
 3 H = 10  # Altura
 4 L = 10  # Comprimento
 5
 6
 7 # NUMERO DE ELEMENTOS NA HORIZONTAL E VERTICAL DA MALHA:
 8
 9 M = 12  #Dimensão da malha quadrada!!!!
10 nL = 12     # Número de elementos ao longo do comprimento
11 nH = 12     # Número de elementos ao longo da altura
12
13 # DIMENSÕES DO ELEMENTO:
14
15 altura = H/nH
16 base = H/nL
17
18 # DADOS GERAIS:
19
20 ne = nL * nH                    # Número de elementos
21 nnos = (nL+1)*(nH+1)            # Número de nós da malha
22 GL = nnos*3                     # Graus de liberdade
23 esp = 1                         # Espessura da placa
24 #P = (10/M)**2                   # Carga Aplicada
25 El = 10.92                      # Módulo de elasticidade
26 vp = 0.3                        # Coeficiente de poisson
27
28
29 # MATRIZ DE CONECTIVIDADE:
30
31 # Montagem da primeira matriz de conectividade referente a primeira linha de elementos da malha:
32
33 conec1 = np.zeros((nH,4))
34 for i in range(nH):
35   conec1[i,0] = i*(nL+1) + 1
36   conec1[i,1] = nL+2 + i*(nL+1)
37   conec1[i,2] = nL+3 + i*(nL+1)
38   conec1[i,3] =i*(nL+1) + 2
39
40 # Montando as demais matrizes de conectividade:
41
42 Matriz = []
43 for i in range(nL):
44   A = conec1+1*i
45   Matriz.append(A)
46
47 # Juntando todas as matrizes em uma única matriz:
48
49 conec = np.concatenate(Matriz)
50
51 print("## MATRIZ DE CONECTIVIDADE: ========================================")
52 print("Nº de linhas da matriz:",conec.shape[0])
53 print("Nº de colunas da matriz:",conec.shape[1])
54 print("Nº de elementos da matriz:",conec.size)
55
56 # MATRIZ DE COORDENADAS NODAIS:
57
```

```
 58 # Montagem da primeira matriz de coordenadas:
 59
 60 coord1 = np.zeros((nL+1,2))
 61 for i in range(nL+1):
 62   coord1[i,1] = base*i
 63
 64 # Montando as demais matrizes de coordenadas:
 65
 66 coord2 = np.zeros((nH+1,2))
 67 for i in range(nH+1):
 68   coord1[:,0] = base*i
 69   coord2 = np.vstack((coord2,coord1))
 70 coord = coord2[nH+1:]
 71
 72 print("## MATRIZ DE COORDENADAS: ==========================================")
 73 print("Nº de linhas da matriz de coordenadas:",coord.shape[0])
 74 print("Nº de colunas da matriz de coordenadas:",coord.shape[1])
 75 print("Nº de elementos da matriz de coordenadas:",coord.size)
 76
 77 # NÓS EM QUE SERÃO RESTRINGIDOS OS GRAUS DE LIBERDADE SEPARADOS POR BORDO DA LAJE:
 78
 79 # Bordo 05:  (Nós dos extremos)
 80
 81 o = M-1
 82 b5 = np.array([[1], [M+1], [(M+1)**2], [M*(M+1)+1]])
 83
 84 # Bordo 01(Esquerda):
 85
 86 b1 = np.zeros((o,1))
 87 for i in range(M):
 88   b1[i-1]=b5[0]+1*i
 89
 90 # Bordo 02 (Baixo):
 91
 92 b2 = np.zeros((o,1))
 93 for i in range(M):
 94   b2[i-1]=b5[0]+(M+1)*i
 95
 96 # Bordo 03 (Cima):
 97
 98 b3 = np.zeros((o,1))
 99 for i in range(M):
100   b3[i-1]=b5[1]+(M+1)*i
101
102 # Bordo 04 (Direita):
103
104 b4 = np.zeros((o,1))
105 for i in range(M):
106   b4[i-1]=b5[3]+1*i
107
108 # LOCALIZAÇÃO DOS GRAUS DE LIBERDADE:
109
110 #Bordo 01
111
112 rrx1 = 3 * b1  - 0     #rotação em x
113 rry1 =   rrx1  - 1     #rotação em y
114 rw1  =   rrx1  - 2     #Deslocamento
115
116 #Bordo 02
117
118 rrx2 = 3 * b2  - 0     #rotação em x
119 rry2 =   rrx2  - 1     #rotação em y
120 rw2  =   rrx2  - 2     #Deslocamento
121
122 #Bordo 03
123
124 rrx3 = 3 * b3  - 0     #rotação em x
125 rry3 =   rrx3  - 1     #rotação em y
126 rw3  =   rrx3  - 2     #Deslocamento
127
128 #Bordo 04
129
130 rrx4 = 3 * b4  - 0     #rotação em x
131 rry4 =   rrx4  - 1     #rotação em y
132 rw4  =   rrx4  - 2     #Deslocamento
133
134 #Bordo dos extremos:
135
136 rrx5 = 3 * b5  - 0     #rotação em x
137 rry5 =   rrx5  - 1     #rotação em y
138 rw5  =   rrx5  - 2     #Deslocamento
139
```

```
140 # VETOR DE GRAUS DE LIBERDADE RESTRIGIDOS:
141
142 GLR1 = np.concatenate((rw1,rry1, rrx1,
143                        rw2, rrx2, rry2,
144                        rw3, rrx3, rry3,
145                        rw4, rry4, rrx4,
146                        rw5))
147
148 GLR = GLR1.astype(int)    # Transformação dos elementos do vetor em elementos inteiros!!!
149
150
151 # VETOR DE CARGAS EXTERNAS:
152
153 ##############################################################################
154 #                  CARREGAMENTO PONTUAL NO CENTRO DA PLACA
155 ##############################################################################
156 #nn1 = ((M/2)+1)+((M/2)*(M+1)) # Nó central da placa!!!!
157
158
159 #Nq = np.array([[nn1]])   # Nó em que será aplicdo a carga.
160
161 #nq1 = 3 * Nq - 2         # Grau de liberdade
162
163 #nq = nq1.astype(int)   # Transformando em números inteiros!!!!
164
165 #f = np.zeros((GL,1))
166 #f[nq-1] = -P        # Carga aplicada no nó 41 dirigida para baixo!
167
168 ################################################################################
169 #                  CARREGAMENTO UNIFORMEMENTE DISTRIBUÍDO
170 ################################################################################
171 P = (10/nH)**2
172 # Montagem da primeira matriz de aplicalção das cargas:
173
174 NNq = np.zeros((nH-1,1))
175 for i in range(nH-1):
176   NNq[i] = nH + 3 + i
177
178 # Montando as demais matrizes de de aplicação das cargas:
179
180 Matriz2 = []
181 for i in range(nH-1):
182   C = NNq + (nH+1)*i
183   Matriz2.append(C)
184
185 # Juntando todas as matrizes em uma única matriz:
186
187 Nq = np.concatenate(Matriz2)
188
189 nnq = 3 * Nq - 2               # Grau de liberdade de deslocamento
190
191 nq = nnq.astype(int)           # Transformando a matriz em numero inteiros
192
193 f = np.zeros((GL,1))
194 f[nq-1] = -P        # Carga aplicada no nó 41 dirigida para baixo!
195
196 ################################################################################
197
198 # DADOS PARAVIEW:
199 nome = 'QuintoProblema 12X12'
200 FatorEscala = 0.08
```

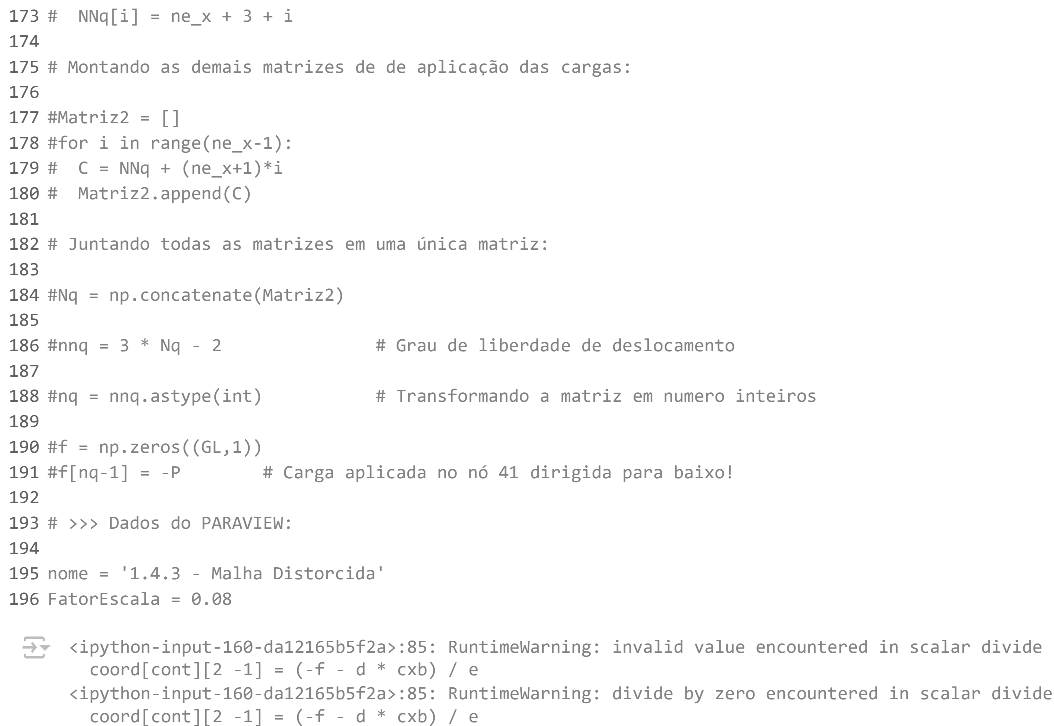
```
## MATRIZ DE CONECTIVIDADE: ========================================
Nº de linhas da matriz: 144
Nº de colunas da matriz: 4
Nº de elementos da matriz: 576
## MATRIZ DE COORDENADAS: ==========================================
Nº de linhas da matriz de coordenadas: 169
Nº de colunas da matriz de coordenadas: 2
Nº de elementos da matriz de coordenadas: 338
```

1.6 - QUINTO PROBLEMA: Exemplo 8.9.3 Simply Supported Square Plate. Wilson (1998) - (MALHA DISTORCIDA)

```
1 """ >>> MALHA DISTORCIDA <<< """  # d_máx = 1.1956 in (0.3279 in)  ;   d_máx_4° = 1.1937 in
2
3 Lx = 10                            # Comprimento em X da laje
4 Ly = 10                            # Comprimento em Y da laje
5 ne_x = 12                        # Número de elementos na direção de X
6 ne_y = 12                         # Número de elementos na direção de Y
7 ne = ne_x * ne_y                   # Número de elementos
8 nnos = (ne_x + 1) * (ne_y + 1)     # Número de nós
```

```
 9 GL = 3*nnos                        # Número de graus de liberdade
10 El = 10.92                          # Módulo de elasticidade básico para os elementos (ksi)
11 #E = np.full((ne, 1), Eb)           # Módulo de elasticidade
12 vp = 0.3                           # Coeficiente de Poisson básico para os elementos
13 #v = np.full((ne, 1), vb)           # Coeficiente de Poisson
14 esp = 1                            # Espessura do elemento básico para os elementos (in)
15 #t = np.full((ne, 1), tb)           # Espessura do elemento
16
17
18 # >>> Determinando a equação geral das retas
19 def equacao_geral_da_reta(x1, y1, x2, y2):
20     if x1 == x2:                          # Verificar se a reta é vertical (x1 == x2)
21         return 0, x1, 0                   # Retornar uma tupla indicando que a reta é vertical
22     elif y1 == y2:                        # Verificar se a reta é horizontal (y1 == y2)
23         return 0, 0, y1                   # Retornar uma tupla indicando que a reta é horizontal
24     else:
25         m = (y2 - y1) / (x2 - x1)         # Calcular a inclinação da reta (coeficiente angular)
26         c = y1 - m * x1                   # Calcular o termo independente (coeficiente linear)
27         return m, -1, c                   # Retornar os coeficientes da equação geral da reta no formato ax + by + c = 0
28
29
30 # >>> Determinando as coordenadas do bordo X
31 div_1x = 2 / 3                              # Fator de divizão das arestas x
32 div_2x = 1 - div_1x                         # Fator de divizão das arestas x complementar
33
34 coord_x_baixo = np.zeros((ne_x + 1, 1))     # Coordenadas do bordo x na parte de baixo
35 for i in range(1, (ne_x + 1) + 1):
36   if i <= (ne_x / 2 + 1):
37     coord_x_baixo[i -1][0] = (i - 1) * (div_1x * Lx / (ne_x / 2))
38   else:
39     coord_x_baixo[i -1][0] = (ne_x / 2) * (div_1x * Lx / (ne_x / 2)) + (i - ((ne_x / 2) + 1)) * (div_2x * Lx / (ne_x / 2))
40
41
42 coord_x_cima = np.zeros((ne_x + 1, 1))     # Coordenadas do bordo x na parte de cima
43 for i in range(1, (ne_x + 1) + 1):
44   if i <= (ne_x / 2 + 1):
45     coord_x_cima[i -1][0] = (i - 1) * (div_2x * Lx / (ne_x / 2))
46   else:
47     coord_x_cima[i -1][0] = (ne_x / 2) * (div_2x * Lx / (ne_x / 2)) + (i - ((ne_x / 2) + 1)) * (div_1x * Lx / (ne_x / 2))
48
49
50 # >>> Determinando as coordenadas do bordo Y
51 div_1y = 1 / 3                              # Fator de divizão das arestas y
52 div_2y = 1 - div_1y                         # Fator de divizão das arestas y complementar
53
54 coord_y_direita = np.zeros((ne_y + 1, 1))   # Coordenadas do bordo y a direita
55 for j in range(1, (ne_y + 1) + 1):
56   if j <= (ne_y / 2 + 1):
57     coord_y_direita[j -1][0] = (j - 1) * (div_1y * Ly / (ne_y / 2))
58   else:
59     coord_y_direita[j -1][0] = (ne_y / 2) * (div_1y * Ly / (ne_y / 2)) + (j - ((ne_y / 2) + 1)) * (div_2y * Ly / (ne_y / 2))
60
61
62 coord_y_esquerda = np.zeros((ne_y + 1, 1))  # Coordenadas do bordo y a esquerda
63 for j in range(1, (ne_y + 1) + 1):
64   if j <= (ne_y / 2 + 1):
65     coord_y_esquerda[j -1][0] = (j - 1) * (div_2y * Ly / (ne_y / 2))
66   else:
67     coord_y_esquerda[j -1][0] = (ne_y / 2) * (div_2y * Ly / (ne_y / 2)) + (j - ((ne_y / 2) + 1)) * (div_1y * Ly / (ne_y / 2))
68
69
70 # >>> Coordenadas nodais
71 coord = np.zeros((nnos, 2))
72 cont = -1
73 for i in range(1, (ne_x + 1) + 1):
74   for j in range(1, (ne_y + 1) + 1):
75     cont += 1
76     cxb = coord_x_baixo[i -1][0]
77     cxc = coord_x_cima[i -1][0]
78     cyd = coord_y_direita[j -1][0]
79     cye = coord_y_esquerda[j -1][0]
80
81     if cxb == cxc:
82       a, b, c = equacao_geral_da_reta(cxb,  0 , cxc,  10)
83       d, e, f = equacao_geral_da_reta( 0 , cyd,  10, cye)
84       coord[cont][1 -1] = cxb
85       coord[cont][2 -1] = (-f - d * cxb) / e
86
87     elif cyd == cye:
88       a, b, c = equacao_geral_da_reta(cxb,  0 , cxc,  10)
89       d, e, f = equacao_geral_da_reta( 0 , cyd,  10, cye)
90       coord[cont][1 -1] = (-c - b * cyd) / a
```

```
 91       coord[cont][2 -1] = cyd
 92
 93     else:
 94       a, b, c = equacao_geral_da_reta(cxb,  0 , cxc,  10)
 95       d, e, f = equacao_geral_da_reta( 0 , cyd,  10, cye)
 96       A = np.array([[a, b],        # Matriz de coeficientes
 97                     [d, e]])
 98
 99       b = np.array([[-c],          # Vetor de constantes
100                     [-f]])
101
102       xy = np.linalg.solve(A, b)   # Resolver o sistema de equações
103
104       coord[cont][1 -1] = xy[1 -1][0]
105       coord[cont][2 -1] = xy[2 -1][0]
106
107 coord[1 -1][2 -1] = 0
108 coord[(ne_y + 1) -1][2 -1] = 10
109 coord[(nnos - ne_y) -1][2 -1] = 0
110 coord[nnos -1][2 -1] = 10
111
112
113 # >>> Matriz conectividade (incidência)
114 conec = np.zeros((ne, 4), dtype=int)
115 cont3 = -1
116 for i in range(1, ne_x + 1):
117   for j in range(1, ne_y + 1):
118     cont3 += 1
119     conec[cont3][1 -1] = j + (i - 1) * (ne_y + 1)
120     conec[cont3][2 -1] = conec[cont3][1 -1] + (ne_y + 1)
121     conec[cont3][3 -1] = conec[cont3][2 -1] + 1
122     conec[cont3][4 -1] = conec[cont3][3 -1] - (ne_y + 1)
123
124
125 # >>> Direções restringidas
126 nós_bordo_esquerda = np.zeros((1, ne_y - 1))
127 nós_bordo_direita = np.zeros((1, ne_y - 1))
128 nós_bordo_cima =  np.zeros((1, ne_x - 1))
129 nós_bordo_baixo = np.zeros((1, ne_x - 1))
130
131 for i in range(0, ne_y -1):      # Indentificando os graus de liberdade a serem restringidos nos bordos a esquerda e direita
132   nós_bordo_esquerda[1 -1][i] = 2 + i
133   nós_bordo_direita[1 -1][i] = ((ne_x + 0) * (ne_y + 1) + 2) + i
134
135 for j in range(0, ne_x -1):      # Indentificando os graus de liberdade a serem restringidos nos bordos a superior e inferior
136   nós_bordo_cima[1 -1][j] = (2 + j) * (ne_y + 1)
137   nós_bordo_baixo[1 -1][j] = (ne_y + 2) + j * (ne_y + 1)
138
139 nós_quina = np.array([[1, (ne_y + 1), (ne_x + 1) * (ne_y + 1) - ne_y, (ne_x + 1) * (ne_y + 1)]])  # Indentificando os graus de liber
140
141 GLR = np.array([np.concatenate((nós_bordo_esquerda[0] * 3 - 2,
142                                 nós_bordo_esquerda[0] * 3 - 1,
143                                 nós_bordo_direita[0] * 3 - 2,
144                                 nós_bordo_direita[0] * 3 - 1,
145                                 nós_bordo_cima[0] * 3 - 2,
146                                 nós_bordo_cima[0] * 3 - 0,
147                                 nós_bordo_baixo[0] * 3 - 2,
148                                 nós_bordo_baixo[0] * 3 - 0,
149                                 nós_quina[0] * 3 - 2))]).astype(int).T
150
151 ###############################################################################
152 #             CARREGAMENTO PONTUAL APLICADA NO CENTRO DA PLACA
153 ###############################################################################
154
155 # Vetor de carga pontual
156 P = np.zeros((GL, 1))         # Vetor das cargas pontuais (Kip)
157 P[int(((ne_y + 1) * (ne_x / 2 + 1) - ne_y / 2) * 3 - 2) -1] = 1
158 Fb = np.zeros((GL, 1))        # Vetor de força de corpo
159 Fs = np.zeros((GL, 1))        # Vetor de força de superfície
160 f = P + Fb + Fs               # Vetor global de forças
161
162 ###############################################################################
163 #                CARREGAMENTO UNIFORMEMENTO DISTRIBUÍDO
164 ###############################################################################
165
166 # VETOR DE CARGAS EXTERNAS:
167
168 #P = (10/ne_x)**2
169 # Montagem da primeira matriz de aplicalção das cargas:
170
171 #NNq = np.zeros((ne_x-1,1))
172 #for i in range(ne_x-1):
```



```
173 #  NNq[i] = ne_x + 3 + i
174
175 # Montando as demais matrizes de de aplicação das cargas:
176
177 #Matriz2 = []
178 #for i in range(ne_x-1):
179 #  C = NNq + (ne_x+1)*i
180 #  Matriz2.append(C)
181
182 # Juntando todas as matrizes em uma única matriz:
183
184 #Nq = np.concatenate(Matriz2)
185
186 #nnq = 3 * Nq - 2               # Grau de liberdade de deslocamento
187
188 #nq = nnq.astype(int)           # Transformando a matriz em numero inteiros
189
190 #f = np.zeros((GL,1))
191 #f[nq-1] = -P        # Carga aplicada no nó 41 dirigida para baixo!
192
193 # >>> Dados do PARAVIEW:
194
195 nome = '1.4.3 - Malha Distorcida'
196 FatorEscala = 0.08
```

```
<ipython-input-160-da12165b5f2a>:85: RuntimeWarning: invalid value encountered in scalar divide
  coord[cont][2 -1] = (-f - d * cxb) / e
<ipython-input-160-da12165b5f2a>:85: RuntimeWarning: divide by zero encountered in scalar divide
  coord[cont][2 -1] = (-f - d * cxb) / e
```

Clique duas vezes (ou pressione "Enter") para editar

1.7 - SÉTIMO PROBLEMA: Pavimento com múltiplas lajes. Porto (2015)

1.7.1 - Dados gerais e apresentação do problema

```
1 # DADOS GERAIS:
2
3 ne = 1467                    # Número de elementos
4 nnos = 1550                  # Número de nós da malha
5 GL = nnos*3                  # Graus de liberdade
6 esp = 0.1                    # Espessura da placa
7 El = 1000*5600*(30)**(1/2)   # Módulo de elasticidade
8 vp = 0.2                     # Coeficiente de poisson
```

1.7.2 - Matriz de conectividade

```
1 # MATRIZ DE CONECTIVIDADE:
2
3 AA = np.array([1, 34, 35, 2])
4 BB = np.array([1453, 1467, 1468, 1454])
5
6 CC = np.zeros((44,4))
7 for i in range(44):
8   CC[i] = AA +33*i
9
10 DD = np.zeros((6,4))
11 for i in range(6):
12   DD[i] = BB + 14*i
13
14 EE = np.concatenate((CC,DD), axis=0)
15
16 #print(DD)   # MATRIZ DE CONECTIIIDADE REFERENTE A PRIMEIRA LINHA DE ELEMENTOS DA MALHA
17
18 Matriz = []
19 for i in range(13):
20   FF = EE + 1*i
21   Matriz.append(FF)
22   #print(FF)
23
24 # Juntando todas as matrizes em uma única matriz:
25
26 GG = np.concatenate(Matriz)
27 #print(GG)
28
29 ## PARTE DE CIMA DA MALHA (Descotinuidade na geometria)
```

```
30
31 HH = np.array([14, 47, 48, 15])
32
33 JJ = np.zeros((43,4))
34 for i in range(43):
35   JJ[i] = HH +33*i
36 #print(JJ)
37
38 Matriz2 = []
39 for i in range(19):
40   KK = JJ + 1*i
41   Matriz2.append(KK)
42   #print(KK)
43
44 # Juntando todas as matrizes em uma única matriz:
45
46 LL = np.concatenate(Matriz2)
47 #print(LL)
48
49 conec = np.concatenate((GG,LL), axis=0)
50 print("Dimensão da matriz de conectividade:",conec.shape)
```

```
Dimensão da matriz de conectividade: (1467, 4)
```

1.7.3 - Matriz de coordenadas nodais

```
1 # PRIMEIRA MATRIZ DE COORDENADAS NODAIS
2
3 #Matriz que contempla a primeira divisão da linha de vigas verticais:
4
5 coord_1 = np.array([[0,    0*2975/13000],
6                     [0,    1*2975/13000],
7                     [0,    2*2975/13000],
8                     [0,    3*2975/13000],
9                     [0,    4*2975/13000],
10                    [0,    5*2975/13000],
11                    [0,    6*2975/13000],
12                    [0,    7*2975/13000],
13                    [0,    8*2975/13000],
14                    [0,    9*2975/13000],
15                    [0,   10*2975/13000],
16                    [0,   11*2975/13000],
17                    [0,   12*2975/13000],
18                    [0,   13*2975/13000],
19
20                    [0,   2.975+1*(3375/14000)],
21                    [0,   2.975+2*(3375/14000)],
22                    [0,   2.975+3*(3375/14000)],
23                    [0,   2.975+4*(3375/14000)],
24                    [0,   2.975+5*(3375/14000)],
25                    [0,   2.975+6*(3375/14000)],
26                    [0,   2.975+7*(3375/14000)],
27                    [0,   2.975+8*(3375/14000)],
28                    [0,   2.975+9*(3375/14000)],
29                    [0,   2.975+10*(3375/14000)],
30                    [0,   2.975+11*(3375/14000)],
31                    [0,   2.975+12*(3375/14000)],
32                    [0,   2.975+13*(3375/14000)],
33                    [0,   2.975+14*(3375/14000)],
34
35                    [0,   6.35+1*(11/50)],
36                    [0,   6.35+2*(11/50)],
37                    [0,   6.35+3*(11/50)],
38                    [0,   6.35+4*(11/50)],
39                    [0,   6.35+5*(11/50)]])
40
41 # Montando as demais matrizes de coordenadas com a primeira parte da malha:
42
43 coord_2 = np.zeros((19,2))
44 for i in range(19):
45   coord_1[:,0] = (4425/18000)*i
46   coord_2 = np.vstack((coord_2,coord_1))
47 coord_3 = coord_2[19:]
48
49 #print(coord_3)
50
51 #Matriz que contempla a segunda (Abcissa 2) divisão da linha de vigas verticais:
52
53 coord_5 = np.array([[(4.425+(6.10/25)),    0*2975/13000],
54                     [(4.425+(6.10/25)),    1*2975/13000],
```

```
55                     [(4.425+(6.10/25)),    2*2975/13000],
56                     [(4.425+(6.10/25)),    3*2975/13000],
57                     [(4.425+(6.10/25)),    4*2975/13000],
58                     [(4.425+(6.10/25)),    5*2975/13000],
59                     [(4.425+(6.10/25)),    6*2975/13000],
60                     [(4.425+(6.10/25)),    7*2975/13000],
61                     [(4.425+(6.10/25)),    8*2975/13000],
62                     [(4.425+(6.10/25)),    9*2975/13000],
63                     [(4.425+(6.10/25)),   10*2975/13000],
64                     [(4.425+(6.10/25)),   11*2975/13000],
65                     [(4.425+(6.10/25)),   12*2975/13000],
66                     [(4.425+(6.10/25)),   13*2975/13000],
67
68                     [(4.425+(6.10/25)),   2.975+1*(3375/14000)],
69                     [(4.425+(6.10/25)),   2.975+2*(3375/14000)],
70                     [(4.425+(6.10/25)),   2.975+3*(3375/14000)],
71                     [(4.425+(6.10/25)),   2.975+4*(3375/14000)],
72                     [(4.425+(6.10/25)),   2.975+5*(3375/14000)],
73                     [(4.425+(6.10/25)),   2.975+6*(3375/14000)],
74                     [(4.425+(6.10/25)),   2.975+7*(3375/14000)],
75                     [(4.425+(6.10/25)),   2.975+8*(3375/14000)],
76                     [(4.425+(6.10/25)),   2.975+9*(3375/14000)],
77                     [(4.425+(6.10/25)),   2.975+10*(3375/14000)],
78                     [(4.425+(6.10/25)),   2.975+11*(3375/14000)],
79                     [(4.425+(6.10/25)),   2.975+12*(3375/14000)],
80                     [(4.425+(6.10/25)),   2.975+13*(3375/14000)],
81                     [(4.425+(6.10/25)),   2.975+14*(3375/14000)],
82
83                     [(4.425+(6.10/25)),   6.35+1*(11/50)],
84                     [(4.425+(6.10/25)),   6.35+2*(11/50)],
85                     [(4.425+(6.10/25)),   6.35+3*(11/50)],
86                     [(4.425+(6.10/25)),   6.35+4*(11/50)],
87                     [(4.425+(6.10/25)),   6.35+5*(11/50)]])
88
89 # Montando as demais matrizes de coordenadas com a primeira parte da malha:
90
91 coord_4 = np.zeros((26,2))
92 for i in range(1,26):
93   coord_5[:,0] = 4.425+(610/2500)*i
94   coord_4 = np.vstack((coord_4,coord_5))
95 coord_6 = coord_4[26:]
96
97 #print(coord_6)
98
99 #Matriz que contempla a segunda (Abcissa 3) divisão da linha de vigas verticais:
100
101 coord_8 = np.array([[(10.525+(145/700)),    0*2975/13000],
102                     [(10.525+(145/700)),    1*2975/13000],
103                     [(10.525+(145/700)),    2*2975/13000],
104                     [(10.525+(145/700)),    3*2975/13000],
105                     [(10.525+(145/700)),    4*2975/13000],
106                     [(10.525+(145/700)),    5*2975/13000],
107                     [(10.525+(145/700)),    6*2975/13000],
108                     [(10.525+(145/700)),    7*2975/13000],
109                     [(10.525+(145/700)),    8*2975/13000],
110                     [(10.525+(145/700)),    9*2975/13000],
111                     [(10.525+(145/700)),   10*2975/13000],
112                     [(10.525+(145/700)),   11*2975/13000],
113                     [(10.525+(145/700)),   12*2975/13000],
114                     [(10.525+(145/700)),   13*2975/13000]])
115
116 #print(coord_8)
117
118 # Montando as demais matrizes de coordenadas com a primeira parte da malha:
119
120 coord_7 = np.zeros((8,2))
121 for i in range(1,8):
122   coord_8[:,0] = 10.525+(1.45/7)*i
123   coord_7 = np.vstack((coord_7,coord_8))
124 coord_9 = coord_7[8:]
125
126 #print(coord_9)
127
128 ## JUNTANDO AS MATRIZES DE COORDENADAS NODAIS:
129
130 coord = np.concatenate((coord_3, coord_6, coord_9))
131
132 print("Tamanho do vetor de coordenadas:",coord.shape )
```

```
Tamanho do vetor de coordenadas: (1550, 2)
```

1.7.4 - Nós que serão restrigidos

```
1 # Bordo 06:  (Nós dos extremos)
2
3 b6 = np.array([[1], [33], [1452], [1433], [1550], [1537]])
4
5 # Bordo 01(Esquerda):
6
7 b1 = np.zeros((31,1))
8 for i in range(31):
9   b1[i]= 2 + 1*i
10
11 # Bordo 02 (Baixo):
12
13 b2_1 = np.zeros((44,1))
14 for i in range(1,45):
15   b2_1[i-1]= 1 + 33*i
16
17 b2_2 = np.zeros((5,1))
18 for i in range(1,6):
19   b2_2[i-1]= 1453 + 14*i
20
21 b2 = np.concatenate((b2_1,b2_2))
22
23 # Bordo 03 (Cima):
24
25 b3 = np.zeros((18,1))
26 for i in range(18):
27   b3[i]= 66 + 33*i
28
29 # Bordo 04 (Direita):
30
31 b4 = np.zeros((18,1))
32 for i in range(18):
33   b4[i]= 1434 + 1*i
34
35 # Bordo 05 (Direita):
36
37 b5 = np.zeros((12,1))
38 for i in range(12):
39   b5[i]= 1538 + 1*i
40
41 # Viga 05:
42
43 v5_1 = np.zeros((42,1))
44 for i in range(42):
45   v5_1[i]= 47 + 33*i
46
47 v5_2 = np.zeros((6,1))
48 for i in range(6):
49   v5_2[i]= 1466 + 14*i
50
51 v5 = np.concatenate((v5_1,v5_2))
52
53 # Viga 02:
54
55 v2_1 = np.zeros((12,1))
56 for i in range(12):
57   v2_1[i]= 596 + 1*i
58
59 v2_2 = np.zeros((18,1))
60 for i in range(18):
61   v2_2[i]= 609 + 1*i
62
63 v2 = np.concatenate((v2_1,v2_2))
64
65 # Viga 03:
66
67 v3 = np.zeros((24,1))
68 for i in range(24):
69   v3[i]= 655 + 33*i
70
71 # LOCALIZAÇÃO DOS GRAUS DE LIBERDADE:
72
73 #Bordo 01
74
75 rrx1 = 3 * b1  - 0     #rotação em x
76 rry1 =   rrx1  - 1     #rotação em y
77 rw1  =   rrx1  - 2     #Deslocamento
78
```

```
79 #Bordo 02
80
81 rrx2 = 3 * b2  - 0     #rotação em x
82 rry2 =   rrx2  - 1     #rotação em y
83 rw2  =   rrx2  - 2     #Deslocamento
84
85 #Bordo 03
86
87 rrx3 = 3 * b3  - 0     #rotação em x
88 rry3 =   rrx3  - 1     #rotação em y
89 rw3  =   rrx3  - 2     #Deslocamento
90
91 #Bordo 04
92
93 rrx4 = 3 * b4  - 0     #rotação em x
94 rry4 =   rrx4  - 1     #rotação em y
95 rw4  =   rrx4  - 2     #Deslocamento
96
97 #Bordo 05
98
99 rrx5 = 3 * b5  - 0     #rotação em x
100 rry5 =   rrx5  - 1     #rotação em y
101 rw5  =   rrx5  - 2     #Deslocamento
102
103 #Bordo dos extremos:
104
105 rrx6 = 3 * b6  - 0     #rotação em x
106 rry6 =   rrx6  - 1     #rotação em y
107 rw6  =   rrx6  - 2     #Deslocamento
108
109 # Viga 05:
110
111 vvx5 = 3 * v5  - 0     #rotação em x
112 vvy5 =   vvx5  - 1     #rotação em y
113 vw5  =   vvx5  - 2     #Deslocamento
114
115 # Viga 02:
116
117 vvx2 = 3 * v2  - 0     #rotação em x
118 vvy2 =   vvx2  - 1     #rotação em y
119 vw2  =   vvx2  - 2     #Deslocamento
120
121 # Viga 03:
122
123 vvx3 = 3 * v3  - 0     #rotação em x
124 vvy3 =   vvx3  - 1     #rotação em y
125 vw3  =   vvx3  - 2     #Deslocamento
126
127 # VETOR DE GRAUS DE LIBERDADE RESTRIGIDOS:
128
129 GLR1 = np.concatenate((rw1, rry1,
130                        rw2, rrx2,
131                        rw3, rrx3,
132                        rw4, rry4,
133                        rw5, rry5, rrx5,
134                        rw6,
135                        vw5, vvx5,
136                        vw2, vvy2,
137                        vw3, vvx3))
138
139 GLR = GLR1.astype(int)    # Transformação dos elementos do vetor em elementos inteiros!!!
```

1.7.5 - Vetor de cargas externas

```
1 ########################################################################
2 # VETOR DE CARGAS EXTERNAS:
3
4 # CARGA DA LAJE 01:
5
6 P1 = 6.7*((4425/18000)*(2975/13000))
7 q1 = np.zeros((17,1))
8 for i in range(17):
9   q1[i]= 35 + 33*i
10 #print(q1)
11
12 Matrixx = []
13 for i in range(12):
14   A = q1+1*i
15   Matrixx.append(A)
16   #print(A)
```

```
17 Q1 = np.concatenate(Matrixx)
18 #print(Q1)
19
20 # CARGA DA LAJE 02:
21
22 # Como a laje 02 é composta por 2 tipos de elementos, o valor da carga nodal será calculado em relação a média das áreas desses eleme
23
24 BL2_elem1 = (610/2500)  # Comprimento do elemento 1 da laje 02 (A Altura é comum a ambos os elementos)
25 BL2_elem2 = (145/700)   # Comprimento do elemento 2 da Laje 02 (A Altura é comum a ambos os elementos)
26
27 BL2_med_elem = (BL2_elem1 + BL2_elem2)/2
28
29 P2 = 8.18*((BL2_med_elem)*(2975/13000))
30
31 q2 = np.zeros((30,1))
32 for i in range(30):
33   if i <=23:
34     q2[i]= 629 + 33*i
35   else:
36     q2[i]= 1454 + 14*(i-24)
37
38 #print(q2)
39
40 Matrixx2 = []
41 for i in range(12):
42   A = q2+1*i
43   Matrixx2.append(A)
44   #print(A)
45 Q2 = np.concatenate(Matrixx2)
46 #print(Q2)
47
48 # CARGA DA LAJE 03:
49
50 # Como a laje 03 é composta por 2 tipos de elemento, o valor da carga nodal será calculado em relação a média das áreas desses elemer
51
52 AL3_elem1 = (4375/14000)  # Altura do elemento 1 da laje 03 (O comprimento é comum a ambos os elementos)
53 AL3_elem2 = (11/50)       # Altura do elemento 2 da Laje 03 (O comprimento é comum a ambos os elementos)
54
55 AL3_med_elem = (AL3_elem1 + BL2_elem2)/2
56
57 P3 = 6.57*(AL3_med_elem)*(4425/18000)
58
59 q3 = np.zeros((17,1))
60 for i in range(17):
61   q3[i]= 48 + 33*i
62 #print(q3)
63
64 Matrixx3 = []
65 for i in range(18):
66   A3 = q3+1*i
67   Matrixx3.append(A3)
68   #print(A3)
69 Q3 = np.concatenate(Matrixx3)
70 #print(Q3)
71
72 # CARGA DA LAJE 04:
73
74 P4 = 5.0*(3375/14000)*(610/2500)
75
76 q4 = np.zeros((24,1))
77 for i in range(24):
78   q4[i]= 642 + 33*i
79 #print(q4)
80
81 Matrixx4 = []
82 for i in range(13):
83   A4 = q4+1*i
84   Matrixx4.append(A4)
85   #print(A3)
86 Q4 = np.concatenate(Matrixx4)
87 #print(Q4)
88
89 # CARGA DA LAJE 05:
90
91 P5 = 5.0*(610/2500)*(11/50) # Carga nodal
92 M = 0.88*(610/2500)         # Momento Nodal
93
94 q5 = np.zeros((24,1))
95 for i in range(24):
96   q5[i]= 656 + 33*i
97 #print(q5)
98
```

```
 99 Matrixx5 = []
100 for i in range(4):
101   A5 = q5+1*i
102   Matrixx5.append(A5)
103   #print(A5)
104 Q5 = np.concatenate(Matrixx5)
105 #print(Q5)
106 
107 # CARGA DO GUARDA CORPO LAJE 05:
108 
109 P6 = 2.0*(610/2500)
110 
111 Q6 = np.zeros((24,1))
112 for i in range(24):
113   Q6[i]= 660 + 33*i
114 
115 #print(Q6)
116 
117 # NÓS ENTRE L1-L3:
118 
119 NL13 = np.zeros((17,1))
120 for i in range(17):
121   NL13[i]= 47 + 33*i
122 
123 # NÓS ENTRE L1-L2:
124 
125 NL12 = np.zeros((12,1))
126 for i in range(12):
127   NL12[i]= 596 + 1*i
128 
129 # NÓS ENTRE L3-L4:
130 
131 NL34 = np.zeros((12,1))
132 for i in range(12):
133   NL34[i]= 609 + 1*i
134 
135  # NÓS ENTRE L2-L4:
136 
137 NL24 = np.zeros((19,1))
138 for i in range(19):
139   NL24[i]= 740 + 33*i
140 
141 # NÓS ENTRE L4-L5:
142 
143 NL45 = np.zeros((20,1))
144 for i in range(20):
145   NL45[i]= 721 + 33*i
146 
147 ################################################################################
148 
149 nQ1 = 3 * Q1 - 2           # Grau de liberdade de deslocamento
150 nQ2 = 3 * Q2 - 2           # Grau de liberdade de deslocamento
151 nQ3 = 3 * Q3 - 2           # Grau de liberdade de deslocamento
152 nQ4 = 3 * Q4 - 2           # Grau de liberdade de deslocamento
153 nQ5 = 3 * Q5 - 2           # Grau de liberdade de deslocamento
154 nQ6 = 3 * Q6 - 2           # Grau de liberdade de deslocamento
155 
156 mQ6 = 3*Q6-1                 # Grau de liberdade de rotação em y.
157 
158 nQ11 = nQ1.astype(int)     # Transformando a matriz em numeros inteiros
159 nQ22 = nQ2.astype(int)     # Transformando a matriz em numeros inteiros
160 nQ33 = nQ3.astype(int)     # Transformando a matriz em numeros inteiros
161 nQ44 = nQ4.astype(int)     # Transformando a matriz em numeros inteiros
162 nQ55 = nQ5.astype(int)     # Transformando a matriz em numeros inteiros
163 nQ66 = nQ6.astype(int)     # Transformando a matriz em numeros inteiros
164 
165 mQ66 = mQ6.astype(int)     # Transformando a matriz em numeros inteiros
166 
167 f = np.zeros((GL,1))
168 f[nQ11-1] = - P1        # Carga da laje 01 aplicada nos nós  dirigida para baixo!
169 f[nQ22-1] = - P2        # Carga da laje 02 aplicada nos nós  dirigida para baixo!
170 f[nQ33-1] = - P3        # Carga da laje 03 aplicada nos nós  dirigida para baixo!
171 f[nQ44-1] = - P4        # Carga da laje 04 aplicada nos nós  dirigida para baixo!
172 f[nQ55-1] = - P5        # Carga da laje 05 aplicada nos nós  dirigida para baixo!
173 f[nQ66-1] = - (P5+P6)   # Carga da laje 05 aplicada nos nós  dirigida para baixo!
174 f[mQ66-1] = - M          # Carga da laje 05 aplicada nos nós  dirigida para baixo!
175 
176 # DADOS PARAVIEW:
177 
178 nome = 'SetimoProblema (Porto)'
179 FatorEscala = 200
```

1.8 - OITAVO PROBLEMA: Exemplo 2.6 Placa retangular simplesmente apoiada. Araújo (2010).

1.8.1 - Dados gerais e apresentação do problema

```
1 # DIMENSÕES DA PLACA (GEOMETRIA RETANGULAR):
2
3 L = 3 # Altura
4 H = 4 # Comprimento
5
6 # NUMERO DE ELEMENTOS NA HORIZONTAL E VERTICAL DA MALHA:
7
8 nH = 40 # Nº de elementos na horizontal
9 nL = 50 # Nº de elementos na vertical
10
11 # DIMENSÕES DO ELEMENTO:
12
13 altura = L/nL
14 base = H/nH
15
16 # DADOS GERAIS:
17
18 ne = nL * nH                    # Número de elementos
19 nnos = (nL+1)*(nH+1)            # Número de nós da malha
20 GL = nnos*3                     # Graus de liberdade
21 esp = 0.1                         # Espessura da placa
22 P = 5*(altura*base)             # Carga Aplicada
23 El = 1000*5600*(30)**(1/2)      # Módulo de elasticidade
24 vp = 0.2                        # Coeficiente de poisson
25
26
27 # MATRIZ DE CONECTIVIDADE:
28
29 # Montagem da primeira matriz de conectividade referente a primeira linha de elementos da malha:
30
31 conec1 = np.zeros((nH,4))
32 for i in range(nH):
33   conec1[i,0] = i*(nL+1) + 1
34   conec1[i,1] = nL+2 + i*(nL+1)
35   conec1[i,2] = nL+3 + i*(nL+1)
36   conec1[i,3] =i*(nL+1) + 2
37
38 #print(conec1)
39 # Montando as demais matrizes de conectividade:
40
41 Matriz = []
42 for i in range(nL):
43   A = conec1+1*i
44   Matriz.append(A)
45
46 # Juntando todas as matrizes em uma única matriz:
47
48 conec = np.concatenate(Matriz)
49
50 print("## MATRIZ DE CONECTIVIDADE: ========================================")
51 print("Nº de linhas da matriz:",conec.shape[0])
52 print("Nº de colunas da matriz:",conec.shape[1])
53 print("Nº de elementos da matriz:",conec.size)
54
55 # Montagem da primeira matriz de coordenadas:
56
57 coord1 = np.zeros((nL+1,2))
58 for i in range(nL+1):
59   coord1[i,1] = altura*i
60
61 #print(coord1)
62 # Montando as demais matrizes de coordenadas:
63
64 coord2 = np.zeros((nH+1,2))
65 for i in range(nH+1):
66   coord1[:,0] = base*i
67   coord2 = np.vstack((coord2,coord1))
68 coord = coord2[nH+1:]
69
70 print("## MATRIZ DE COORDENADAS: ==========================================")
71 print("Nº de linhas da matriz de coordenadas:",coord.shape[0])
72 print("Nº de colunas da matriz de coordenadas:",coord.shape[1])
73 print("Nº de elementos da matriz de coordenadas:",coord.size)
74
```

```
75 #print(coord)
76
```

```
## MATRIZ DE CONECTIVIDADE: ========================================
Nº de linhas da matriz: 2000
Nº de colunas da matriz: 4
Nº de elementos da matriz: 8000
## MATRIZ DE COORDENADAS: ==========================================
Nº de linhas da matriz de coordenadas: 2091
Nº de colunas da matriz de coordenadas: 2
Nº de elementos da matriz de coordenadas: 4182
```

1.8.2 - Nós que serão restrigidos

```
1 # Bordo 05:  (Nós dos extremos)
2
3 b5 = np.array([[1], [1+nL], [1+nH*(nL+1)], [(1+nH*(nL+1))+nL]])
4
5 # Bordo 01(Esquerda):
6
7 b1 = np.zeros((nL-1,1))
8 for i in range(nL-1):
9   b1[i]= 2 + 1*i
10
11 # Bordo 02 (Baixo):
12
13 b2 = np.zeros((nH-1,1))
14 for i in range(1,nH):
15   b2[i-1]= 1 + (1+nL)*i
16
17 # Bordo 03 (Cima):
18
19 b3 = np.zeros((nH-1,1))
20 for i in range(1,nH):
21   b3[i-1]= (1+nL) + (1+nL)*i
22
23 # Bordo 04 (Direita):
24
25 b4 = np.zeros((nL-1,1))
26 for i in range(1,nL):
27   b4[i-1]= (nL+1)*nH+1 + 1*i
28
29 # Vetor que contém todos os nós da malha:
30
31 q55 = np.zeros((nH+1,1))
32 for i in range(nH+1):
33   q55[i]= 1 + 1*i
34 #print(q55)
35 Matrixx5 = []
36 for i in range(nL+1):
37   A55 = q55+51*i
38   Matrixx5.append(A55)
39   #print(A5)
40 Q55 = np.concatenate(Matrixx5)
41 #print(Q5)
42
43 # LOCALIZAÇÃO DOS GRAUS DE LIBERDADE:
44
45 #Bordo 01
46
47 rrx1 = 3 * b1  - 0     #rotação em x
48 rry1 =   rrx1  - 1     #rotação em y
49 rw1  =   rrx1  - 2     #Deslocamento
50
51 #Bordo 02
52
53 rrx2 = 3 * b2  - 0     #rotação em x
54 rry2 =   rrx2  - 1     #rotação em y
55 rw2  =   rrx2  - 2     #Deslocamento
56
57 #Bordo 03
58
59 rrx3 = 3 * b3  - 0     #rotação em x
60 rry3 =   rrx3  - 1     #rotação em y
61 rw3  =   rrx3  - 2     #Deslocamento
62
63 #Bordo 04
64
65 rrx4 = 3 * b4  - 0     #rotação em x
66 rry4 =   rrx4  - 1     #rotação em y
67 rw4  =   rrx4  - 2     #Deslocamento
```

```
68
69 #Bordo dos extremos:
70
71 rrx5 = 3 * b5  - 0     #rotação em x
72 rry5 =   rrx5  - 1     #rotação em y
73 rw5  =   rrx5  - 2     #Deslocamento
74
75 # VETOR DE GRAUS DE LIBERDADE RESTRIGIDOS:
76
77 GLR1 = np.concatenate((rw1, rry1,
78                        rw2, rrx2,
79                        rw3, rrx3,
80                        rw4, rry4,
81                        rw5,))
82
83 GLR = GLR1.astype(int)    # Transformação dos elementos do vetor em elementos inteiros!!!
```

1.8.3 - Vetor de cargas externas

```
1 # VETOR DE CARGAS EXTERNAS (Cargas em todos os nós da malha):
2
3 Nq = np.zeros((nnos,1))
4 for i in range(nnos):
5   Nq[i] = i+1
6
7 nq1 = 3 * Nq - 2         # Grau de liberdade de deslocamento
8
9 nq = nq1.astype(int)     # Transformando a matriz em numeros inteiros
10
11 f = np.zeros((GL,1))
12 f[nq-1] = - P        # Carga aplicada nos nós  dirigida para baixo!
13
14 # DADOS PARAVIEW:
15 nome = 'OitavoProblema (Araújo)'
16 FatorEscala = 300
17
18 print("## VETOR DE CARGAS EXTERNAS: =========================================")
19 print("Nº de linhas do vetor:",f.shape[0])
20 print("Nº de colunas do vetor:",f.shape[1])
21 print("Nº de elementos do vetor:",f.size)
```

```
## VETOR DE CARGAS EXTERNAS: =========================================
Nº de linhas do vetor: 6273
Nº de colunas do vetor: 1
Nº de elementos do vetor: 6273
```

2 - PROCESSAMENTO

2.1 - Vetores dos dados de entrada

```
1 # VETOR DE ESPESSURA DOS ELEMENTOS:
2
3 t = np.zeros((ne, 1))
4 for i in range(0, ne):
5   t[i]= esp
6
7 # VETOR DO MÓDULO DE ELASTICIDADE DE CADA ELEMENTO:
8
9 E = np.zeros((ne, 1))
10 for i in range(0, ne):
11   E[i]= El
12
13 # VETOR DO COEFICIENTE DE POISSON DE CADA ELEMENTO:
14
15 v = np.zeros((ne, 1))
16 for i in range(0, ne):
17   v[i]= vp
18
```

2.2 - Coordenadas dos nós

```
1 # VETOR DAS COORDENADAS X GLOBAIS:
2
3 def xg(e):
4   XG = np.array([[coord[int(conec[(e-1),0]-1), 0]],
```

```
 5                [coord[int(conec[(e-1),1]-1), 0]],
 6                [coord[int(conec[(e-1),2]-1), 0]],
 7                [coord[int(conec[(e-1),3]-1), 0]]])
 8   return XG
 9
10 # VETOR DAS COORDENADAS Y GLOBAIS:
11
12 def yg(e):
13   YG = np.array([[coord[int(conec[(e-1),0]-1), 1]],
14                [coord[int(conec[(e-1),1]-1), 1]],
15                [coord[int(conec[(e-1),2]-1), 1]],
16                [coord[int(conec[(e-1),3]-1), 1]]])
17   return YG
18
19 # VETOR DAS COORDENADAS X LOCAIS:
20
21 def xl(e):
22   XL = np.array([[0],
23                [xg(e)[1][0] - xg(e)[0][0]],
24                [xg(e)[2][0] - xg(e)[3][0]],
25                [0]])
26   return XL
27
28 # VETOR DAS COORDENADAS Y LOCAIS:
29
30 def yl(e):
31   YL = np.array([[0],
32                [0],
33                [yg(e)[2][0] - yg(e)[1][0]],
34                [yg(e)[3][0] - yg(e)[0][0]]])
35   return YL
36
37 # >>> Matriz de coordenadas dos elementos [coorde]
38
39 def coorde(e):    # Coordenadas globais
40   xG_val = xl(e)
41   yG_val = yl(e)
42   coorde = np.array([[xG_val[1 -1][0], yG_val[1 -1][0]],
43                     [xG_val[2 -1][0], yG_val[2 -1][0]],
44                     [xG_val[3 -1][0], yG_val[3 -1][0]],
45                     [xG_val[4 -1][0], yG_val[4 -1][0]]])
46   return coorde
47
```

2.3 - Matrizes constitutivas do material

```
 1 # MATRIZ [D]:
 2 def D(e):
 3   e = e - 1 # Ajuste da numeração do elemento
 4   D = (t[e][0]**3*E[e][0])/(12*(1-v[e][0]**2)) * np.array([[1,        v[e][0], 0],
 5                                                          [v[e][0],  1,       0],
 6                                                          [ 0,       0,       (1-v[e][0])/2]])
 7   return D
 8
 9 #  MATRIZ [C]:
10 def C(e):
11   e = e - 1 # Ajuste da numeração do elemento
12   C = (E[e][0]/((1-v[e][0]**2))) * np.array([[1,        v[e][0], 0],
13                                            [v[e][0],  1,       0],
14                                            [ 0,       0,       (1-v[e][0])/2]])
15   return C
```

2.4 - Formulação das matrizes "A" e "Q":

```
 1 # MATRIZ [Q]:
 2
 3 def Q(e,  ξ , η):
 4   x = xe1(e , ξ , η)
 5   y = ye1(e , ξ , η)
 6   Q = np.array([[ 0, 0, 0, 2, 0, 0, 6*x, 2*y,   0,   0,  6*x*y,      0],
 7                [ 0, 0, 0, 0, 0, 2,   0,   0, 2*x, 6*y,      0,  6*x*y],
 8                [ 0, 0, 0, 0, 2, 0,   0, 4*x, 4*y,   0, 6*x**2, 6*y**2]])
 9   return Q
10
11 # MATRIZ [A]:
12
13 # Matriz A de cada elemento.
14 def A(e):
```

```
15   x1, x2, x3, x4 = xl(e)[0][0], xl(e)[1][0], xl(e)[2][0], xl(e)[3][0]
16   y1, y2, y3, y4 = yl(e)[0][0], yl(e)[1][0], yl(e)[2][0], yl(e)[3][0]
17   A = np.array([[ 1, x1,  y1, x1**2, x1*y1, y1**2,   x1**3,  y1*x1**2,  x1*y1**2,    y1**3,   y1*x1**3,    x1*y1**3],
18                 [ 0,  0,  -1,     0,   -x1, -2*y1,       0,    -x1**2,  -2*x1*y1, -3*y1**2,     -x1**3, -3*x1*y1**2],
19                 [ 0,  1,   0,  2*x1,    y1,     0, 3*x1**2,   2*y1*x1,     y1**2,        0, 3*y1*x1**2,       y1**3],
20                 [ 1, x2,  y2, x2**2, x2*y2, y2**2,   x2**3,  y2*x2**2,  x2*y2**2,    y2**3,   y2*x2**3,    x2*y2**3],
21                 [ 0,  0,  -1,     0,   -x2, -2*y2,       0,    -x2**2,  -2*x2*y2, -3*y2**2,     -x2**3, -3*x2*y2**2],
22                 [ 0,  1,   0,  2*x2,    y2,     0, 3*x2**2,   2*y2*x2,     y2**2,        0, 3*y2*x2**2,       y2**3],
23                 [ 1, x3,  y3, x3**2, x3*y3, y3**2,   x3**3,  y3*x3**2,  x3*y3**2,    y3**3,   y3*x3**3,    x3*y3**3],
24                 [ 0,  0,  -1,     0,   -x3, -2*y3,       0,    -x3**2,  -2*x3*y3, -3*y3**2,     -x3**3, -3*x3*y3**2],
25                 [ 0,  1,   0,  2*x3,    y3,     0, 3*x3**2,   2*y3*x3,     y3**2,        0, 3*y3*x3**2,       y3**3],
26                 [ 1, x4,  y4, x4**2, x4*y4, y4**2,   x4**3,  y4*x4**2,  x4*y4**2,    y4**3,   y4*x4**3,    x4*y4**3],
27                 [ 0,  0,  -1,     0,   -x4, -2*y4,       0,    -x4**2,  -2*x4*y4, -3*y4**2,     -x4**3, -3*x4*y4**2],
28                 [ 0,  1,   0,  2*x4,    y4,     0, 3*x4**2,   2*y4*x4,     y4**2,        0, 3*y4*x4**2,       y4**3]])
29   return A
30
31 # MATRIZ [Kop]:
32
33 def Kop(e, ξ, η):
34   Kop = np.dot(np.transpose(Q(e , ξ , η)), np.dot(D(e),Q(e , ξ , η)))
35   return Kop
36
37 # Determinante da matriz jacobiana:
38
39 #def Det(e):
40   #e = e - 1
41   #Det = a[e][0] * b[e][0]
42   #return Det
```

2.5 - Integração via quadratura Gaussiana

2.5.1 - Pontos e pesos de Gauss:

```
1 # PONTOS DE GAUSS PARAINTEGRAÇÃO 2X2:
2
3 eg = np.array([[-1/np.sqrt(3)],[1/np.sqrt(3)]])
4 ng = np.array([[-1/np.sqrt(3)],[1/np.sqrt(3)]])
5
6 # PESOS DE GAUSS:
7
8 wg = np.array([[1],[1]])
```

2.5.2 - Funções de forma

```
1 # FUNÇÕES DE FORMA:
2
3 def N1(ξ , η):
4   N1 = (1/4)*((1 - ξ)*(1 - η))
5   return N1
6
7 def N2(ξ , η):
8   N2 = (1/4)*((1 + ξ)*(1 - η))
9   return N2
10
11 def N3(ξ , η):
12   N3 = (1/4)*((1 + ξ)*(1 + η))
13   return N3
14
15 def N4(ξ , η):
16   N4 = (1/4)*((1 - ξ)*(1 + η))
17   return N4
18
19
20 def xe1(e , ξ , η):  # Coordenada x:
21   xe1 = N1(ξ , η)*xl(e)[0][0] + N2(ξ , η)*xl(e)[1][0] + N3(ξ , η)*xl(e)[2][0] + N4(ξ , η)*xl(e)[3][0]
22   return xe1
23
24 def ye1(e , ξ , η):  #Coordenada y:
25   ye1 = N1(ξ , η)*yl(e)[0][0] + N2(ξ , η)*yl(e)[1][0] + N3(ξ , η)*yl(e)[2][0] + N4(ξ , η)*yl(e)[3][0]
26   return ye1
27
28 # Funções de substituição utilizadas por Melosh (1990):
29
30 #def xe(e, eg):  # Coordenada x:
31   #e = e - 1
32   #xe = a[e][0] + a[e][0]*eg
33   #return xe
```

```
34
35
36 #def ye(e, ng):  #Coordenada y:
37   #e = e - 1
38   #ye = b[e][0] + b[e][0]*ng
39   #return ye
```

2.5.3 - Matriz jacobiana

```
1 # Vetores das derivadas parciais das funções de forma
2
3 def Nξ(ξ, η):       # Derivadas parciais em relação a ξ.
4   Nξ = np.array([[-(1 - η) / 4],
5                  [+(1 - η) / 4],
6                  [+(1 + η) / 4],
7                  [-(1 + η) / 4]])
8   return Nξ
9
10 def Nη(ξ, η):       # Derivadas parciais em relação a η
11   Nη = np.array([[-(1 - ξ) / 4],
12                  [-(1 + ξ) / 4],
13                  [+(1 + ξ) / 4],
14                  [+(1 - ξ) / 4]])
15   return Nη
16
17
18 #  Matriz com as derivadas das funções de interpolação
19
20 def DNx(ξ, η):
21     Nξ_val = Nξ(ξ, η)
22     Nη_val = Nη(ξ, η)
23     DNx = np.array([Nξ_val[:, 0], Nη_val[:, 0]])
24     return DNx
25
26 # >>> Matriz Jacobiana [J]
27
28 def J(e, ξ, η):
29   J = DNx(ξ, η) @ coorde(e)
30   return J
31
32 # >>> Determinante da matriz Jacobiana
33
34 def DetJ(e, ξ, η):
35   DetJ = np.linalg.det(J(e, ξ, η))
36   return DetJ
37
38
39 # >>> Matriz Jocobiana Inversa [Γ]
40
41 def Γ(e, ξ, η):
42   Γ = np.linalg.inv(J(e, ξ, η))
43   return Γ
```

2.5.4 - Integração

```
1 def Ki(e):
2   Kii = np.zeros((12,12))
3   for i in range(2):
4     for j in range(2):
5       ξa =  eg[i][0]
6       ηa =  eg[j][0]
7       Kii = Kii + wg[i][0] * wg[j][0] * Kop(e, ξa, ηa) * DetJ(e, ξa, ηa)
8   return(Kii)
```

Clique duas vezes (ou pressione "Enter") para editar

2.6 - Matriz de rigidez dos elementos

```
1 def Ke(e):
2   InvA = np.linalg.inv(A(e))
3   Kee = np.dot(np.transpose(InvA),   np.dot(Ki(e),  InvA))
4   return Kee
```

2.7 - Matriz de rigidez global

```
1 # VETOR DE GRAUS DE LIBERDADE:
2
3 def GDL(e):
4   e = e - 1
5   GDL = np.array(([3*int(conec[e][0]) - 2],
6                   [3*int(conec[e][0]) - 1],
7                   [3*int(conec[e][0]) - 0],
8                   [3*int(conec[e][1]) - 2],
9                   [3*int(conec[e][1]) - 1],
10                  [3*int(conec[e][1]) - 0],
11                  [3*int(conec[e][2]) - 2],
12                  [3*int(conec[e][2]) - 1],
13                  [3*int(conec[e][2]) - 0],
14                  [3*int(conec[e][3]) - 2],
15                  [3*int(conec[e][3]) - 1],
16                  [3*int(conec[e][3]) - 0]))
17   return GDL
18
19 # MATRIZ DE RIGIDEZ GLOBAL:
20
21 K = np.zeros((GL,GL))
22 for e in range(1,(ne+1)):
23   GDL_e = GDL(e)
24   Ke_e = Ke(e)
25   for i in range(12):
26     for j in range(12):
27       K[GDL_e[i][0]-1][GDL_e[j][0]-1] += Ke_e[i][j]
28
29 print("## MATRIZ DE RIGIDEZ GLOBAL: ============================================")
30 print("# Quantidade de colunas:", K.shape[0])
31 print("# Quantidade de linhas:", K.shape[1])
32 print("# Quantidade de elementos da matriz:", K.size)
```

```
## MATRIZ DE RIGIDEZ GLOBAL: ============================================
# Quantidade de colunas: 12
# Quantidade de linhas: 12
# Quantidade de elementos da matriz: 144
```

2.8 - Formulação das condições de contorno

```
1 # Aplicação das condições de contorno:
2
3 for i in range(GLR.shape[0]):
4   K[GLR[i,0]-1,GLR[i,0]-1] = 10**7*K[GLR[i,0]-1,GLR[i,0]-1]
```

2.9 - Solução do sistema de equações

```
1 # Deslocamentos nodais globais:
2
3 #d = np.linalg.solve(K , f)
4
5 # >>> Vetor dos deslocamentos nodais globais
6 def solve_large_system(K, f):  # Função para resolução so sistema linear
7     K_sparse = csc_matrix(K)   # Converter K para formato esparso CSC
8
9     try:
10        d_sparse = spsolve(K_sparse, f)
11        d = np.array([d_sparse]).T
12        print(f"Solução encontrada com sucesso!")
13        return d
14    except ValueError as e:
15        print(f"Erro ao resolver o sistema: {e}")
16        return None
17    except Exception as e:
18        print(f"Ocorreu um erro inesperado: {e}")
19        return None
20
21 d = solve_large_system(K, f)  # Resolvendo o sistema
22
23 if d is not None:              # Verificando se a solução foi encontrada
24     pass                       # Faça algo com a solução, se necessário
25 else:                          # Trate o caso em que não foi possível resolver o sistema
26     print("Não foi possível encontrar uma solução para o sistema.")
27
28 # Deslocamentos nodais de cada elemento:
```

```
29
30 def de(e):
31   dee = np.zeros((12,1))
32   for i in range (12):
33     dee[i, 0] = d[GDL(e)[i][0]-1][0]
34   return dee
35
36 ## TESTE:
37
38 print("==============================================================================")
39 print("VETOR DE DESLOCAMENTO DO ELEMENTO 33:")
40 print(de(1))
41 print("==============================================================================")
```

```
Solução encontrada com sucesso!
==============================================================================
VETOR DE DESLOCAMENTO DO ELEMENTO 33:
[[ 1.50000015e-09]
 [ 8.88980455e-26]
 [ 1.00000010e-09]
 [ 1.00000012e-01]
 [-2.22044628e-18]
 [ 1.50000010e-02]
 [ 1.00000012e-01]
 [-2.22044627e-18]
 [ 1.50000010e-02]
 [ 1.50000015e-09]
 [ 1.06579924e-27]
 [ 1.00000010e-09]]
==============================================================================
```

```
1 # Vetor de deslocamentos
2
3 #desl= np.zeros((nnos,1))
4 #for i in range(nnos):
5 #  desl[i] = d[3*i]*1000
6
7 #ML1 = np.zeros((Q6.shape[0],1))
8 #for i in range(Q6.shape[0]):
9 #  cc = Q6.astype(int)-1
10 #  ML1[i] = desl[cc[i]][0]
11
12 #ff = np.min(ML1)
13 #gg = np.max(ML1)
14 #print("MÁXIMO:======")
15 #print(ff)
16 #print("Mínimo:======")
17 #print(gg)
```

2.10 - Matriz de compatibilidade cinemática

```
1 def B(e, ξ, η):
2   InvA = np.linalg.inv(A(e))
3   B = np.matmul(Q(e, ξ , η), InvA)
4   return B
```

2.11 - VETOR DOS MOMENTOS

2.11.1 - Função dos momentos

```
1 # FUNÇÃO QUE CALCULA O MOMENTO EM FUNÇÃO DOS ELEMENTOS:
2
3 def Mk(e, ξ, η):
4   Mk = np.dot(-D(e), np.dot(B(e, ξ, η), de(e)))
5   return Mk
6
7 ## TESTE:
8
9 print("==============================================================================")
10 print("# VETOR DOS MOMENTOS DO ELEMENTO 37:")
11 print(Mk(1,0,0))
12 print("==============================================================================")
```

```
==============================================================================
# VETOR DOS MOMENTOS DO ELEMENTO 37:
[[-1.00000000e+00]
 [-2.29246683e-24]
 [ 1.44560290e-16]]
```

```
===============================================================================
```

2.11.2 - Vetor dos momentos

```
1 contador = np.zeros((nnos,1))          # Vetor nulo que armazena quandos elementos compartilha cada nó
2 momentos1 = np.zeros((nnos,1))         # Vetor nulo que armazena os momentos em x
3 momentos2 = np.zeros((nnos,1))         # Vetor nulo que armazena os momentos em y
4 momentos3 = np.zeros((nnos,1))         # Vetor nulo que armazena os momentos torsores
5
6 for i in range (1, nnos+1):            # PERCORRE OS NÓS GLOBAIS DA MALHA
7   #print("===================")
8   #print(f"Nó Global:{i}")
9   indicess = np.argwhere(conec == i)
10  contador2 = len(indicess)
11  #print(indicess)
12  M1 = np.zeros((len(indicess),1))
13  M2 = np.zeros((len(indicess),1))
14  M3 = np.zeros((len(indicess),1))
15  #print("Elementos que compartilham o mesmo nó:",len(indicess))
16  for j in range(len(indicess)):  # CONTA OS ELEMENTOS QUE COMPARTILHAM O MESMO NO.
17    Elem = indicess[j][0] + 1
18    #print("ELEME",Elem)
19    no = indicess[j][1] + 1
20    #print("no",no)
21    if no == 1:
22     (x,y) = (-1,-1)
23    elif no == 2:
24     (x,y) = (1 ,-1)
25    elif no == 3:
26     (x,y) = (1 , 1)
27    else:
28     (x,y) = (-1, 1)
29    M1[j] = Mk(Elem,x,y)[0]
30    M2[j] = Mk(Elem,x,y)[1]
31    M3[j] = Mk(Elem,x,y)[2]
32    #print(M1)
33    Mm1 = np.sum(M1)
34    Mm2 = np.sum(M2)
35    Mm3 = np.sum(M3)
36
37  contador[i-1] = contador2
38  momentos1[i-1] = Mm1
39  momentos2[i-1] = Mm2
40  momentos3[i-1] = Mm3
41 print("Vetor dos momentos:========")
42 print(momentos3[1])
43 print("Vetor contador:============")
44 print(contador[1])
45
46 mx = np.zeros((nnos,1))
47 my = np.zeros((nnos,1))
48 mxy = np.zeros((nnos,1))
49 for i in range(nnos):
50  mx[i]  = -momentos1[i]/contador[i]
51  my[i]  = -momentos2[i]/contador[i]
52  mxy[i] = -momentos3[i]/contador[i]
```

```
Vetor dos momentos:========
[-1.4456029e-16]
Vetor contador:============
[1.]
```

```
1 #nn = ((ne_y/2)+1)+(ne_y/2)*(ne_x+1)
2 #nn=1
3 #print(nn)
4
5 #print("Momento em x:============")
6 #print(mx[int(nn-1)])
7 #print("Momento em y:============")
8 #print(my[int(nn-1)])
9 #print("Momento torsor:==========")
10 #print(mxy[int(nn-1)])
11 #print("=======================")
12
```

```
1
2 #ML1 = np.zeros((b5.shape[0],1))
3 #for i in range(b5.shape[0]):
4 #  cc = b5.astype(int)-1
```

```
5 #  ML1[i] = mx[cc[i]][0]
6
7 #ff = np.max(ML1)
8 #gg = np.min(ML1)
9 #print("MÁXIMO:======")
10 #print(ff)
11 #print("Mínimo:======")
12 #print(gg)
```

3 - ARQUIVO VTU

3.1 - Códigos do módulo EVTK - Herrera (2016)

```
1 ################
2 ###### VTK ######
3 ################
4
5 # **************************************
6 # *  Low level Python library to       *
7 # *  export data to binary VTK file.   *
8 # **************************************
9
10 import sys
11 import os
12
13 # ================================
14 #            VTK Types
15 # ================================
16
17 #     FILE TYPES
18 class VtkFileType:
19
20     def __init__(self, name, ext):
21         self.name = name
22         self.ext  = ext
23
24     def __str__(self):
25         return "Name: %s  Ext: %s \n" % (self.name, self.ext)
26
27 VtkImageData        = VtkFileType("ImageData", ".vti")
28 VtkPolyData         = VtkFileType("PolyData", ".vtp")
29 VtkRectilinearGrid  = VtkFileType("RectilinearGrid", ".vtr")
30 VtkStructuredGrid   = VtkFileType("StructuredGrid", ".vts")
31 VtkUnstructuredGrid = VtkFileType("UnstructuredGrid", ".vtu")
32
33 #    DATA TYPES
34 class VtkDataType:
35
36     def __init__(self, size, name):
37         self.size = size
38         self.name = name
39
40     def __str__(self):
41         return "Type: %s  Size: %d \n" % (self.name, self.size)
42
43 VtkInt8    = VtkDataType(1, "Int8")
44 VtkUInt8   = VtkDataType(1, "UInt8")
45 VtkInt16   = VtkDataType(2, "Int16")
46 VtkUInt16  = VtkDataType(2, "UInt16")
47 VtkInt32   = VtkDataType(4, "Int32")
48 VtkUInt32  = VtkDataType(4, "UInt32")
49 VtkInt64   = VtkDataType(8, "Int64")
50 VtkUInt64  = VtkDataType(8, "UInt64")
51 VtkFloat32 = VtkDataType(4, "Float32")
52 VtkFloat64 = VtkDataType(8, "Float64")
53
54 # Map numpy to VTK data types
55 np_to_vtk = { 'int8'    : VtkInt8,
56               'uint8'   : VtkUInt8,
57               'int16'   : VtkInt16,
58               'uint16'  : VtkUInt16,
59               'int32'   : VtkInt32,
60               'uint32'  : VtkUInt32,
61               'int64'   : VtkInt64,
62               'uint64'  : VtkUInt64,
63               'float32' : VtkFloat32,
64               'float64' : VtkFloat64 }
65
66 #    CELL TYPES
```

```
67 class VtkCellType:
68 
69     def __init__(self, tid, name):
70         self.tid = tid
71         self.name = name
72 
73     def __str__(self):
74         return "VtkCellType( %s ) \n" % ( self.name )
75 
76 VtkVertex = VtkCellType(1, "Vertex")
77 VtkPolyVertex = VtkCellType(2, "PolyVertex")
78 VtkLine = VtkCellType(3, "Line")
79 VtkPolyLine = VtkCellType(4, "PolyLine")
80 VtkTriangle = VtkCellType(5, "Triangle")
81 VtkTriangleStrip = VtkCellType(6, "TriangleStrip")
82 VtkPolygon = VtkCellType(7, "Polygon")
83 VtkPixel = VtkCellType(8, "Pixel")
84 VtkQuad = VtkCellType(9, "Quad")
85 VtkTetra = VtkCellType(10, "Tetra")
86 VtkVoxel = VtkCellType(11, "Voxel")
87 VtkHexahedron = VtkCellType(12, "Hexahedron")
88 VtkWedge = VtkCellType(13, "Wedge")
89 VtkPyramid = VtkCellType(14, "Pyramid")
90 VtkQuadraticEdge = VtkCellType(21, "Quadratic_Edge")
91 VtkQuadraticTriangle = VtkCellType(22, "Quadratic_Triangle")
92 VtkQuadraticQuad = VtkCellType(23, "Quadratic_Quad")
93 VtkQuadraticTetra = VtkCellType(24, "Quadratic_Tetra")
94 VtkQuadraticHexahedron = VtkCellType(25, "Quadratic_Hexahedron")
95 
96 # ==============================
97 #       Helper functions
98 # ==============================
99 def _mix_extents(start, end):
100     assert (len(start) == len(end) == 3)
101     string = "%d %d %d %d %d %d" % (start[0], end[0], start[1], end[1], start[2], end[2])
102     return string
103 
104 def _array_to_string(a):
105     s = "".join([repr(num) + " " for num in a])
106     return s
107 
108 def _get_byte_order():
109     if sys.byteorder == "little":
110         return "LittleEndian"
111     else:
112         return "BigEndian"
113 
114 # ================================
115 #        VtkGroup class
116 # ================================
117 class VtkGroup:
118 
119     def __init__(self, filepath):
120         """ Creates a VtkGroup file that is stored in filepath.
121 
122             PARAMETERS:
123                 filepath: filename without extension.
124         """
125         self.xml = XmlWriter(filepath + ".pvd")
126         self.xml.openElement("VTKFile")
127         self.xml.addAttributes(type = "Collection", version = "0.1",  byte_order = _get_byte_order())
128         self.xml.openElement("Collection")
129         self.root = os.path.dirname(filepath)
130 
131     def save(self):
132         """ Closes this VtkGroup. """
133         self.xml.closeElement("Collection")
134         self.xml.closeElement("VTKFile")
135         self.xml.close()
136 
137     def addFile(self, filepath, sim_time, group = "", part = "0"):
138         """ Adds file to this VTK group.
139 
140             PARAMETERS:
141                 filepath: full path to VTK file.
142                 sim_time: simulated time.
143                 group: This attribute is not required; it is only for informational purposes.
144                 part: It is an integer value greater than or equal to 0.
145 
146             See: http://www.paraview.org/Wiki/ParaView/Data_formats#PVD_File_Format for details.
147         """
148         # TODO: Check what the other attributes are for.
```

```
149         filename = os.path.relpath(filepath, start = self.root)
150         self.xml.openElement("DataSet")
151         self.xml.addAttributes(timestep = sim_time, group = group, part = part, file = filename)
152         self.xml.closeElement()
153 
154 #offset
155 
156 # ================================
157 #        VtkFile class
158 # ================================
159 class VtkFile:
160 
161     def __init__(self, filepath, ftype, largeFile = False):
162         """
163             PARAMETERS:
164                 filepath: filename without extension.
165                 ftype: file type, e.g. VtkImageData, etc.
166                 largeFile: If size of the stored data cannot be represented by a UInt32.
167         """
168         self.ftype = ftype
169         self.filename = filepath + ftype.ext
170         self.xml = XmlWriter(self.filename)
171         self.offset = 0  # offset in bytes after beginning of binary section
172         self.appendedDataIsOpen = False
173 #        self.largeFile = largeFile
174 
175 #       if largeFile == False:
176 #            self.xml.openElement("VTKFile").addAttributes(type = ftype.name,
177 #                                                          version = "0.1",
178 #                                                          byte_order = _get_byte_order())
179 #        else:
180 #           print "WARNING: output file only compatible with VTK 6.0 and later."
181         self.xml.openElement("VTKFile").addAttributes(type = ftype.name,
182                                                          version = "1.0",
183                                                          byte_order = _get_byte_order(),
184                                                          header_type = "UInt64")
185 
186     def getFileName(self):
187         """ Returns absolute path to this file. """
188         return  os.path.abspath(self.filename)
189 
190     def openPiece(self, start = None, end = None,
191                         npoints = None, ncells = None,
192                         nverts = None, nlines = None, nstrips = None, npolys = None):
193         """ Open piece section.
194 
195             PARAMETERS:
196                 Next two parameters must be given together.
197                 start: array or list with start indexes in each direction.
198                 end:   array or list with end indexes in each direction.
199 
200                 npoints: number of points in piece (int).
201                 ncells: number of cells in piece (int). If present,
202                         npoints must also be given.
203 
204                 All the following parameters must be given together with npoints.
205                 They should all be integer values.
206                 nverts: number of vertices.
207                 nlines: number of lines.
208                 nstrips: number of strips.
209                 npolys: number of .
210 
211             RETURNS:
212                 this VtkFile to allow chained calls.
213         """
214         # TODO: Check what are the requirements for each type of grid.
215 
216         self.xml.openElement("Piece")
217         if (start and end):
218             ext = _mix_extents(start, end)
219             self.xml.addAttributes( Extent = ext)
220 
221         elif (ncells and npoints):
222             self.xml.addAttributes(NumberOfPoints = npoints, NumberOfCells = ncells)
223 
224         elif npoints or nverts or nlines or nstrips or npolys:
225             if npoints is None: npoints = str(0)
226             if nverts is None: nverts = str(0)
227             if nlines is None: nlines = str(0)
228             if nstrips is None: nstrips = str(0)
229             if npolys is None: npolys = str(0)
230             self.xml.addAttributes(NumberOfPoints = npoints, NumberOfVerts = nverts,
```

```
231                                NumberOfLines = nlines, NumberOfStrips = nstrips, NumberOfPolys = npolys)
232         else:
233             assert(False)
234 
235         return self
236 
237     def closePiece(self):
238         self.xml.closeElement("Piece")
239 
240     def openData(self, nodeType, scalars=None, vectors=None, normals=None, tensors=None, tcoords=None):
241         """ Open data section.
242 
243             PARAMETERS:
244                 nodeType: Point or Cell.
245                 scalars: default data array name for scalar data.
246                 vectors: default data array name for vector data.
247                 normals: default data array name for normals data.
248                 tensors: default data array name for tensors data.
249                 tcoords: dafault data array name for tcoords data.
250 
251             RETURNS:
252                 this VtkFile to allow chained calls.
253         """
254         self.xml.openElement(nodeType + "Data")
255         if scalars:
256             self.xml.addAttributes(scalars = scalars)
257         if vectors:
258             self.xml.addAttributes(vectors = vectors)
259         if normals:
260             self.xml.addAttributes(normals = normals)
261         if tensors:
262             self.xml.addAttributes(tensors = tensors)
263         if tcoords:
264             self.xml.addAttributes(tcoords = tcoords)
265 
266         return self
267 
268     def closeData(self, nodeType):
269         """ Close data section.
270 
271             PARAMETERS:
272                 nodeType: Point or Cell.
273 
274             RETURNS:
275                 this VtkFile to allow chained calls.
276         """
277         self.xml.closeElement(nodeType + "Data")
278 
279 
280     def openGrid(self, start = None, end = None, origin = None, spacing = None):
281         """ Open grid section.
282 
283             PARAMETERS:
284                 start: array or list of start indexes. Required for Structured, Rectilinear and ImageData grids.
285                 end: array or list of end indexes. Required for Structured, Rectilinear and ImageData grids.
286                 origin: 3D array or list with grid origin. Only required for ImageData grids.
287                 spacing: 3D array or list with grid spacing. Only required for ImageData grids.
288 
289             RETURNS:
290                 this VtkFile to allow chained calls.
291         """
292         gType = self.ftype.name
293         self.xml.openElement(gType)
294         if (gType == VtkImageData.name):
295             if (not start or not end or not origin or not spacing): assert(False)
296             ext = _mix_extents(start, end)
297             self.xml.addAttributes(WholeExtent = ext,
298                                    Origin = _array_to_string(origin),
299                                    Spacing = _array_to_string(spacing))
300 
301         elif (gType == VtkStructuredGrid.name or gType == VtkRectilinearGrid.name):
302             if (not start or not end): assert (False)
303             ext = _mix_extents(start, end)
304             self.xml.addAttributes(WholeExtent = ext)
305 
306         return self
307 
308     def closeGrid(self):
309         """ Closes grid element.
310 
311             RETURNS:
312                 this VtkFile to allow chained calls.
```

```
313         """
314         self.xml.closeElement(self.ftype.name)
315 
316 
317     def addHeader(self, name, dtype, nelem, ncomp):
318         """ Adds data array description to xml header section.
319 
320             PARAMETERS:
321                 name: data array name.
322                 dtype: string describing type of the data.
323                        Format is the same as used by numpy, e.g. 'float64'.
324                 nelem: number of elements in the array.
325                 ncomp: number of components, 1 (=scalar) and 3 (=vector).
326 
327             RETURNS:
328                 This VtkFile to allow chained calls.
329 
330             NOTE: This is a low level function. Use addData if you want
331                   to add a numpy array.
332         """
333         dtype = np_to_vtk[dtype]
334 
335         self.xml.openElement( "DataArray")
336         self.xml.addAttributes( Name = name,
337                                 NumberOfComponents = ncomp,
338                                 type = dtype.name,
339                                 format = "appended",
340                                 offset = self.offset)
341         self.xml.closeElement()
342 
343         #TODO: Check if 4/8 is platform independent
344         #if self.largeFile == False:
345         #    self.offset += nelem * ncomp * dtype.size + 4 # add 4 to indicate array size
346         #else:
347         self.offset += nelem * ncomp * dtype.size + 8 # add 8 to indicate array size
348         return self
349 
350     def addData(self, name, data):
351         """ Adds array description to xml header section.
352 
353              PARAMETERS:
354                 name: data array name.
355                 data: one numpy array or a tuple with 3 numpy arrays. If a tuple, the individual
356                       arrays must represent the components of a vector field.
357                       All arrays must be one dimensional or three-dimensional.
358         """
359         if type(data).__name__ == "tuple": # vector data
360             assert (len(data) == 3)
361             x = data[0]
362             self.addHeader(name, x.dtype.name, x.size, 3)
363         elif type(data).__name__ == "ndarray":
364             if data.ndim == 1 or data.ndim == 3:
365                 self.addHeader(name, data.dtype.name, data.size, 1)
366             else:
367                 assert False, "Bad array shape: " + str(data.shape)
368         else:
369             assert False, "Argument must be a Numpy array"
370 
371     def appendHeader(self, dtype, nelem, ncomp):
372         """ This function only writes the size of the data block that will be appended.
373             The data itself must be written immediately after calling this function.
374 
375             PARAMETERS:
376                 dtype: string with data type representation (same as numpy). For example, 'float64'
377                 nelem: number of elements.
378                 ncomp: number of components, 1 (=scalar) or 3 (=vector).
379         """
380         self.openAppendedData()
381         dsize = np_to_vtk[dtype].size
382         block_size = dsize * ncomp * nelem
383         if self.largeFile == False:
384             writeBlockSize(self.xml.stream, block_size)
385         else:
386             writeBlockSize64Bit(self.xml.stream, block_size)
387 
388 
389     def appendData(self, data):
390         """ Append data to binary section.
391             This function writes the header section and the data to the binary file.
392 
393             PARAMETERS:
394                 data: one numpy array or a tuple with 3 numpy arrays. If a tuple, the individual
```

```
395                       arrays must represent the components of a vector field.
396                       All arrays must be one dimensional or three-dimensional.
397                       The order of the arrays must coincide with the numbering scheme of the grid.
398
399             RETURNS:
400                 This VtkFile to allow chained calls
401
402             TODO: Extend this function to accept contiguous C order arrays.
403         """
404         self.openAppendedData()
405
406         if type(data).__name__ == 'tuple': # 3 numpy arrays
407             ncomp = len(data)
408             assert (ncomp == 3)
409             dsize = data[0].dtype.itemsize
410             nelem = data[0].size
411             block_size = ncomp * nelem * dsize
412             #if self.largeFile == False:
413             writeBlockSize(self.xml.stream, block_size)
414             #else:
415             #    writeBlockSize64Bit(self.xml.stream, block_size)
416             x, y, z = data[0], data[1], data[2]
417             writeArraysToFile(self.xml.stream, x, y, z)
418
419         elif type(data).__name__ == 'ndarray' and (data.ndim == 1 or data.ndim == 3): # single numpy array
420             ncomp = 1
421             dsize = data.dtype.itemsize
422             nelem = data.size
423             block_size = ncomp * nelem * dsize
424             #if self.largeFile == False:
425             writeBlockSize(self.xml.stream, block_size)
426             #else:
427             #    writeBlockSize64Bit(self.xml.stream, block_size)
428             writeArrayToFile(self.xml.stream, data)
429
430         else:
431             assert False
432
433         return self
434
435     def openAppendedData(self):
436         """ Opens binary section.
437
438             It is not necessary to explicitly call this function from an external library.
439         """
440         if not self.appendedDataIsOpen:
441             self.xml.openElement("AppendedData").addAttributes(encoding = "raw").addText("_")
442             self.appendedDataIsOpen = True
443
444     def closeAppendedData(self):
445         """ Closes binary section.
446
447             It is not necessary to explicitly call this function from an external library.
448         """
449         self.xml.closeElement("AppendedData")
450
451     def openElement(self, tagName):
452         """ Useful to add elements such as: Coordinates, Points, Verts, etc. """
453         self.xml.openElement(tagName)
454
455     def closeElement(self, tagName):
456         self.xml.closeElement(tagName)
457
458     def save(self):
459         """ Closes file """
460         if self.appendedDataIsOpen:
461             self.xml.closeElement("AppendedData")
462         self.xml.closeElement("VTKFile")
463         self.xml.close()
464
465 ################
466 ###### HL ######
467 ################
468
469 # **************************************
470 # *  High level Python library to      *
471 # *  export data to binary VTK file.   *
472 # **************************************
473
474 # ================================
475 #       Helper functions
476 # ================================
```

```
477 def _addDataToFile(vtkFile, cellData, pointData):
478     # Point data
479     if pointData:
480         keys = list(pointData.keys())
481         vtkFile.openData("Point", scalars = keys[0])
482         for key in keys:
483             data = pointData[key]
484             vtkFile.addData(key, data)
485         vtkFile.closeData("Point")
486 
487     # Cell data
488     if cellData:
489         keys = list(cellData.keys())
490         vtkFile.openData("Cell", scalars = keys[0])
491         for key in keys:
492             data = cellData[key]
493             vtkFile.addData(key, data)
494         vtkFile.closeData("Cell")
495 
496 def _appendDataToFile(vtkFile, cellData, pointData):
497     # Append data to binary section
498     if pointData != None:
499         keys = list(pointData.keys())
500         for key in keys:
501             data = pointData[key]
502             vtkFile.appendData(data)
503 
504     if cellData != None:
505         keys = list(cellData.keys())
506         for key in keys:
507             data = cellData[key]
508             vtkFile.appendData(data)
509 
510 # ================================
511 #       High level functions
512 # ================================
513 def imageToVTK(path, origin = (0.0,0.0,0.0), spacing = (1.0,1.0,1.0), cellData = None, pointData = None ):
514     """ Exports data values as a rectangular image.
515 
516         PARAMETERS:
517             path: name of the file without extension where data should be saved.
518             origin: grid origin (default = (0,0,0))
519             spacing: grid spacing (default = (1,1,1))
520             cellData: dictionary containing arrays with cell centered data.
521                       Keys should be the names of the data arrays.
522                       Arrays must have the same dimensions in all directions and must contain
523                       only scalar data.
524             nodeData: dictionary containing arrays with node centered data.
525                       Keys should be the names of the data arrays.
526                       Arrays must have same dimension in each direction and
527                       they should be equal to the dimensions of the cell data plus one and
528                       must contain only scalar data.
529 
530          RETURNS:
531             Full path to saved file.
532 
533         NOTE: At least, cellData or pointData must be present to infer the dimensions of the image.
534     """
535     assert (cellData != None or pointData != None)
536 
537     # Extract dimensions
538     start = (0,0,0)
539     end = None
540     if cellData != None:
541         keys = list(cellData.keys())
542         data = cellData[keys[0]]
543         end = data.shape
544     elif pointData != None:
545         keys = list(pointData.keys())
546         data = pointData[keys[0]]
547         end = data.shape
548         end = (end[0] - 1, end[1] - 1, end[2] - 1)
549 
550     # Write data to file
551     w = VtkFile(path, VtkImageData)
552     w.openGrid(start = start, end = end, origin = origin, spacing = spacing)
553     w.openPiece(start = start, end = end)
554     _addDataToFile(w, cellData, pointData)
555     w.closePiece()
556     w.closeGrid()
557     _appendDataToFile(w, cellData, pointData)
558     w.save()
```

```
559     return w.getFileName()
560 
561 # ==============================================================================
562 def gridToVTK(path, x, y, z, cellData = None, pointData = None):
563     """
564         Writes data values as a rectilinear or rectangular grid.
565 
566         PARAMETERS:
567             path: name of the file without extension where data should be saved.
568             x, y, z: coordinates of the nodes of the grid. They can be 1D or 3D depending if
569                      the grid should be saved as a rectilinear or logically structured grid, respectively.
570                      Arrays should contain coordinates of the nodes of the grid.
571                      If arrays are 1D, then the grid should be Cartesian, i.e. faces in all cells are orthogonal.
572                      If arrays are 3D, then the grid should be logically structured with hexahedral cells.
573                      In both cases the arrays dimensions should be equal to the number of nodes of the grid.
574             cellData: dictionary containing arrays with cell centered data.
575                       Keys should be the names of the data arrays.
576                       Arrays must have the same dimensions in all directions and must contain
577                       only scalar data.
578             pointData: dictionary containing arrays with node centered data.
579                        Keys should be the names of the data arrays.
580                        Arrays must have same dimension in each direction and
581                        they should be equal to the dimensions of the cell data plus one and
582                        must contain only scalar data.
583 
584         RETURNS:
585             Full path to saved file.
586 
587     """
588     # Extract dimensions
589     start = (0,0,0)
590     nx = ny = nz = 0
591 
592     if (x.ndim == 1 and y.ndim == 1 and z.ndim == 1):
593         nx, ny, nz = x.size - 1, y.size - 1, z.size - 1
594         isRect = True
595         ftype = VtkRectilinearGrid
596     elif (x.ndim == 3 and y.ndim == 3 and z.ndim == 3):
597         s = x.shape
598         nx, ny, nz = s[0] - 1, s[1] - 1, s[2] - 1
599         isRect = False
600         ftype = VtkStructuredGrid
601     else:
602         assert(False)
603     end = (nx, ny, nz)
604 
605 
606     w =  VtkFile(path, ftype)
607     w.openGrid(start = start, end = end)
608     w.openPiece(start = start, end = end)
609 
610     if isRect:
611         w.openElement("Coordinates")
612         w.addData("x_coordinates", x)
613         w.addData("y_coordinates", y)
614         w.addData("z_coordinates", z)
615         w.closeElement("Coordinates")
616     else:
617         w.openElement("Points")
618         w.addData("points", (x,y,z))
619         w.closeElement("Points")
620 
621     _addDataToFile(w, cellData, pointData)
622     w.closePiece()
623     w.closeGrid()
624     # Write coordinates
625     if isRect:
626         w.appendData(x).appendData(y).appendData(z)
627     else:
628         w.appendData( (x,y,z) )
629     # Write data
630     _appendDataToFile(w, cellData, pointData)
631     w.save()
632     return w.getFileName()
633 
634 
635 # ==============================================================================
636 def pointsToVTK(path, x, y, z, data = None):
637     """
638         Export points and associated data as an unstructured grid.
639 
640         PARAMETERS:
```

```
641             path: name of the file without extension where data should be saved.
642             x, y, z: 1D arrays with coordinates of the points.
643             data: dictionary with variables associated to each point.
644                   Keys should be the names of the variable stored in each array.
645                   All arrays must have the same number of elements.
646 
647         RETURNS:
648             Full path to saved file.
649 
650     """
651     assert (x.size == y.size == z.size)
652     npoints = x.size
653 
654     # create some temporary arrays to write grid topology
655     offsets = np.arange(start = 1, stop = npoints + 1, dtype = 'int32')   # index of last node in each cell
656     connectivity = np.arange(npoints, dtype = 'int32')                    # each point is only connected to itself
657     cell_types = np.empty(npoints, dtype = 'uint8')
658 
659     cell_types[:] = VtkVertex.tid
660 
661     w = VtkFile(path, VtkUnstructuredGrid)
662     w.openGrid()
663     w.openPiece(ncells = npoints, npoints = npoints)
664 
665     w.openElement("Points")
666     w.addData("points", (x,y,z))
667     w.closeElement("Points")
668     w.openElement("Cells")
669     w.addData("connectivity", connectivity)
670     w.addData("offsets", offsets)
671     w.addData("types", cell_types)
672     w.closeElement("Cells")
673 
674     _addDataToFile(w, cellData = None, pointData = data)
675 
676     w.closePiece()
677     w.closeGrid()
678     w.appendData( (x,y,z) )
679     w.appendData(connectivity).appendData(offsets).appendData(cell_types)
680 
681     _appendDataToFile(w, cellData = None, pointData = data)
682 
683     w.save()
684     return w.getFileName()
685 
686 # ==============================================================================
687 def linesToVTK(path, x, y, z, cellData = None, pointData = None):
688     """
689         Export line segments that joint 2 points and associated data.
690 
691         PARAMETERS:
692             path: name of the file without extension where data should be saved.
693             x, y, z: 1D arrays with coordinates of the vertex of the lines. It is assumed that each line.
694                      is defined by two points, then the lenght of the arrays should be equal to 2 * number of lines.
695             cellData: dictionary with variables associated to each line.
696                   Keys should be the names of the variable stored in each array.
697                   All arrays must have the same number of elements.
698             pointData: dictionary with variables associated to each vertex.
699                   Keys should be the names of the variable stored in each array.
700                   All arrays must have the same number of elements.
701 
702         RETURNS:
703             Full path to saved file.
704 
705     """
706     assert (x.size == y.size == z.size)
707     assert (x.size % 2 == 0)
708     npoints = x.size
709     ncells = x.size / 2
710 
711     # Check cellData has the same size that the number of cells
712 
713     # create some temporary arrays to write grid topology
714     offsets = np.arange(start = 2, step = 2, stop = npoints + 1, dtype = 'int32')   # index of last node in each cell
715     connectivity = np.arange(npoints, dtype = 'int32')                              # each point is only connected to itself
716     cell_types = np.empty(npoints, dtype = 'uint8')
717 
718     cell_types[:] = VtkLine.tid
719 
720     w = VtkFile(path, VtkUnstructuredGrid)
721     w.openGrid()
722     w.openPiece(ncells = ncells, npoints = npoints)
```

```
723
724     w.openElement("Points")
725     w.addData("points", (x,y,z))
726     w.closeElement("Points")
727     w.openElement("Cells")
728     w.addData("connectivity", connectivity)
729     w.addData("offsets", offsets)
730     w.addData("types", cell_types)
731     w.closeElement("Cells")
732
733     _addDataToFile(w, cellData = cellData, pointData = pointData)
734
735     w.closePiece()
736     w.closeGrid()
737     w.appendData( (x,y,z) )
738     w.appendData(connectivity).appendData(offsets).appendData(cell_types)
739
740     _appendDataToFile(w, cellData = cellData, pointData = pointData)
741
742     w.save()
743     return w.getFileName()
744
745 # ==============================================================================
746 def polyLinesToVTK(path, x, y, z, pointsPerLine, cellData = None, pointData = None):
747     """
748         Export line segments that joint 2 points and associated data.
749
750         PARAMETERS:
751             path: name of the file without extension where data should be saved.
752             x, y, z: 1D arrays with coordinates of the vertices of the lines. It is assumed that each line.
753                      has diffent number of points.
754             pointsPerLine: 1D array that defines the number of points associated to each line. Thus,
755                            the length of this array define the number of lines. It also implicitly
756                            defines the connectivity or topology of the set of lines. It is assumed
757                            that points that define a line are consecutive in the x, y and z arrays.
758             cellData: Dictionary with variables associated to each line.
759                       Keys should be the names of the variable stored in each array.
760                       All arrays must have the same number of elements.
761             pointData: Dictionary with variables associated to each vertex.
762                        Keys should be the names of the variable stored in each array.
763                        All arrays must have the same number of elements.
764
765         RETURNS:
766             Full path to saved file.
767
768     """
769     assert (x.size == y.size == z.size)
770     npoints = x.size
771     ncells = pointsPerLine.size
772
773     # create some temporary arrays to write grid topology
774     offsets = np.zeros(ncells, dtype = 'int32')         # index of last node in each cell
775     ii = 0
776     for i in range(ncells):
777         ii += pointsPerLine[i]
778         offsets[i] = ii
779
780     connectivity = np.arange(npoints, dtype = 'int32')      # each line connects points that are consecutive
781
782     cell_types = np.empty(npoints, dtype = 'uint8')
783     cell_types[:] = VtkPolyLine.tid
784
785     w = VtkFile(path, VtkUnstructuredGrid)
786     w.openGrid()
787     w.openPiece(ncells = ncells, npoints = npoints)
788
789     w.openElement("Points")
790     w.addData("points", (x,y,z))
791     w.closeElement("Points")
792     w.openElement("Cells")
793     w.addData("connectivity", connectivity)
794     w.addData("offsets", offsets)
795     w.addData("types", cell_types)
796     w.closeElement("Cells")
797
798     _addDataToFile(w, cellData = cellData, pointData = pointData)
799
800     w.closePiece()
801     w.closeGrid()
802     w.appendData( (x,y,z) )
803     w.appendData(connectivity).appendData(offsets).appendData(cell_types)
804
```

```
805     _appendDataToFile(w, cellData = cellData, pointData = pointData)
806
807     w.save()
808     return w.getFileName()
809
810 # ==============================================================================
811 def unstructuredGridToVTK(path, x, y, z, connectivity, offsets, cell_types, cellData = None, pointData = None):
812     """
813         Export unstructured grid and associated data.
814
815         PARAMETERS:
816             path: name of the file without extension where data should be saved.
817             x, y, z: 1D arrays with coordinates of the vertices of cells. It is assumed that each element
818                      has diffent number of vertices.
819             connectivity: 1D array that defines the vertices associated to each element.
820                           Together with offset define the connectivity or topology of the grid.
821                           It is assumed that vertices in an element are listed consecutively.
822             offsets: 1D array with the index of the last vertex of each element in the connectivity array.
823                      It should have length nelem, where nelem is the number of cells or elements in the grid.
824             cell_types: 1D array with an integer that defines the cell type of each element in the grid.
825                         It should have size nelem. This should be assigned from evtk.vtk.VtkXXXX.tid, where XXXX represent
826                         the type of cell. Please check the VTK file format specification for allowed cell types.
827             cellData: Dictionary with variables associated to each line.
828                       Keys should be the names of the variable stored in each array.
829                       All arrays must have the same number of elements.
830             pointData: Dictionary with variables associated to each vertex.
831                        Keys should be the names of the variable stored in each array.
832                        All arrays must have the same number of elements.
833
834         RETURNS:
835             Full path to saved file.
836
837     """
838     assert (x.size == y.size == z.size)
839     npoints = x.size
840     ncells = cell_types.size
841     assert (offsets.size == ncells)
842
843     w = VtkFile(path, VtkUnstructuredGrid)
844     w.openGrid()
845     w.openPiece(ncells = ncells, npoints = npoints)
846
847     w.openElement("Points")
848     w.addData("points", (x,y,z))
849     w.closeElement("Points")
850     w.openElement("Cells")
851     w.addData("connectivity", connectivity)
852     w.addData("offsets", offsets)
853     w.addData("types", cell_types)
854     w.closeElement("Cells")
855
856     _addDataToFile(w, cellData = cellData, pointData = pointData)
857
858     w.closePiece()
859     w.closeGrid()
860     w.appendData( (x,y,z) )
861     w.appendData(connectivity).appendData(offsets).appendData(cell_types)
862
863     _appendDataToFile(w, cellData = cellData, pointData = pointData)
864
865     w.save()
866     return w.getFileName()
867
868 # ==============================================================================
869 def cylinderToVTK(path, x0, y0, z0, z1, radius, nlayers, npilars = 16, cellData=None, pointData=None):
870     """
871         Export cylinder as VTK unstructured grid.
872
873       PARAMETERS:
874         path: path to file without extension.
875         x0, yo: center of cylinder.
876         z0, z1: lower and top elevation of the cylinder.
877         radius: radius of cylinder.
878         nlayers: Number of layers in z direction to divide the cylinder.
879         npilars: Number of points around the diameter of the cylinder.
880                  Higher value gives higher resolution to represent the curved shape.
881         cellData: dictionary with 1D arrays that store cell data.
882                   Arrays should have number of elements equal to ncells = npilars * nlayers.
883         pointData: dictionary with 1D arrays that store point data.
884                   Arrays should have number of elements equal to npoints = npilars * (nlayers + 1).
885
886       RETURNS:
```

```
887             Full path to saved file.
888 
889         NOTE: This function only export vertical shapes for now. However, it should be easy to
890               rotate the cylinder to represent other orientations.
891     """
892     import math as m
893 
894     # Define x, y coordinates from polar coordinates.
895     dpi = 2.0 * m.pi / npilars
896     angles = np.arange(0.0, 2.0 * m.pi, dpi)
897 
898     x = radius * np.cos(angles) + x0
899     y = radius * np.sin(angles) + y0
900 
901     dz = (z1 - z0) / nlayers
902     z = np.arange(z0, z1+dz, step = dz)
903 
904     npoints = npilars * (nlayers + 1)
905     ncells  = npilars * nlayers
906 
907     xx = np.zeros(npoints)
908     yy = np.zeros(npoints)
909     zz = np.zeros(npoints)
910 
911     ii = 0
912     for k in range(nlayers + 1):
913         for p in range(npilars):
914             xx[ii] = x[p]
915             yy[ii] = y[p]
916             zz[ii] = z[k]
917             ii = ii + 1
918 
919     # Define connectivity
920     conn = np.zeros(4 * ncells, dtype = np.int64)
921     ii = 0
922     for l in range(nlayers):
923         for p in range(npilars):
924             p0 = p
925             if(p + 1 == npilars):
926                 p1 = 0
927             else:
928                 p1 = p + 1 # circular loop
929 
930             n0 = p0 + l * npilars
931             n1 = p1 + l * npilars
932             n2 = n0 + npilars
933             n3 = n1 + npilars
934 
935             conn[ii + 0] = n0
936             conn[ii + 1] = n1
937             conn[ii + 2] = n3
938             conn[ii + 3] = n2
939             ii = ii + 4
940 
941     # Define offsets
942     offsets = np.zeros(ncells, dtype = np.int64)
943     for i in range(ncells):
944         offsets[i] = (i + 1) * 4
945 
946     # Define cell types
947     ctype = np.ones(ncells) + VtkPixel.tid
948 
949     return unstructuredGridToVTK(path, xx, yy, zz, connectivity = conn, offsets = offsets, cell_types = ctype, cellData = cellData,
950 
951 #################
952 ###### XML ######
953 #################
954 
955 # **************************************
956 # *  Simple class to generate a        *
957 # *  well-formed XML file.             *
958 # **************************************
959 
960 class XmlWriter:
961     def __init__(self, filepath, addDeclaration = True):
962         self.stream = open(filepath, "wb")
963         self.openTag = False
964         self.current = []
965         if (addDeclaration): self.addDeclaration()
966 
967     def close(self):
968         assert(not self.openTag)
```

```
969         self.stream.close()
970 
971     def addDeclaration(self):
972         self.stream.write(b'<?xml version="1.0"?>')
973 
974     def openElement(self, tag):
975         if self.openTag: self.stream.write(b">")
976         st = "\n<%s" % tag
977         self.stream.write(str.encode(st))
978         self.openTag = True
979         self.current.append(tag)
980         return self
981 
982     def closeElement(self, tag = None):
983         if tag:
984             assert(self.current.pop() == tag)
985             if (self.openTag):
986                 self.stream.write(b">")
987                 self.openTag = False
988             st = "\n</%s>" % tag
989             self.stream.write(str.encode(st))
990         else:
991             self.stream.write(b"/>")
992             self.openTag = False
993             self.current.pop()
994         return self
995 
996     def addText(self, text):
997         if (self.openTag):
998             self.stream.write(b">\n")
999             self.openTag = False
1000         self.stream.write(str.encode(text))
1001         return self
1002 
1003     def addAttributes(self, **kwargs):
1004         assert (self.openTag)
1005         for key in kwargs:
1006             st = ' %s="%s"'%(key, kwargs[key])
1007             self.stream.write(str.encode(st))
1008         return self
1009 
1010 ##################
1011 ###### EVTK ######
1012 ##################
1013 
1014 import struct
1015 
1016 # Map numpy dtype to struct format
1017 np_to_struct = { 'int8'    : 'b',
1018                  'uint8'   : 'B',
1019                  'int16'   : 'h',
1020                  'uint16'  : 'H',
1021                  'int32'   : 'i',
1022                  'uint32'  : 'I',
1023                  'int64'   : 'q',
1024                  'uint64'  : 'Q',
1025                  'float32' : 'f',
1026                  'float64' : 'd' }
1027 
1028 def _get_byte_order_char():
1029 # Check format in https://docs.python.org/3.5/library/struct.html
1030     if sys.byteorder == "little":
1031         return '<'
1032     else:
1033         return '>'
1034 
1035 # ================================
1036 #        Python interface
1037 # ================================
1038 def writeBlockSize(stream, block_size):
1039     fmt = _get_byte_order_char() + 'Q' # Write size as unsigned long long == 64 bits unsigned integer
1040     stream.write(struct.pack(fmt, block_size))
1041 
1042 def writeArrayToFile(stream, data):
1043     #stream.flush() # this should not be necessary
1044     assert (data.ndim == 1 or data.ndim == 3)
1045     fmt = _get_byte_order_char() + str(data.size) + np_to_struct[data.dtype.name]  # > for big endian
1046 
1047     # Check if array is contiguous
1048     assert (data.flags['C_CONTIGUOUS'] or data.flags['F_CONTIGUOUS'])
1049 
1050     # NOTE: VTK expects data in FORTRAN order
```

```
1051     # This is only needed when a multidimensional array has C-layout
1052     dd = np.ravel(data, order='F')
1053 
1054     bin = struct.pack(fmt, *dd)
1055     stream.write(bin)
1056 
1057 # ================================================================================
1058 def writeArraysToFile(stream, x, y, z):
1059     # Check if arrays have same shape and data type
1060     assert ( x.size == y.size == z.size ), "Different array sizes."
1061     assert ( x.dtype.itemsize == y.dtype.itemsize == z.dtype.itemsize ), "Different item sizes."
1062 
1063     nitems = x.size
1064     itemsize = x.dtype.itemsize
1065 
1066     fmt = _get_byte_order_char() + str(1) + np_to_struct[x.dtype.name]  # > for big endian
1067 
1068     # Check if arrays are contiguous
1069     assert (x.flags['C_CONTIGUOUS'] or x.flags['F_CONTIGUOUS'])
1070     assert (y.flags['C_CONTIGUOUS'] or y.flags['F_CONTIGUOUS'])
1071     assert (z.flags['C_CONTIGUOUS'] or z.flags['F_CONTIGUOUS'])
1072 
1073 
1074     # NOTE: VTK expects data in FORTRAN order
1075     # This is only needed when a multidimensional array has C-layout
1076     xx = np.ravel(x, order='F')
1077     yy = np.ravel(y, order='F')
1078     zz = np.ravel(z, order='F')
1079 
1080     # eliminate this loop by creating a composed array.
1081     for i in range(nitems):
1082         bx = struct.pack(fmt, xx[i])
1083         by = struct.pack(fmt, yy[i])
1084         bz = struct.pack(fmt, zz[i])
1085         stream.write(bx)
1086         stream.write(by)
1087         stream.write(bz)
```

3.2 - Função PreParaview - Silva (2022)

```
 1 'CRIAÇÃO DO ARQUIVO VTK PARA O PARAVIEW'
 2 
 3 def PreParaview(NumElem, NumNos, coord, conec, nome, d, mx, my, mxy, FatorEscala)
 4 
 5     ### Inicializações:
 6     x  = np.zeros(NumNos)       # Coordenadas 'x' de cada nó (deformado)
 7     y  = np.zeros(NumNos)       # Coordenadas 'y' de cada nó (deformado)
 8     z  = np.zeros(NumNos)       # Coordenadas 'z' de cada nó (deformado)
 9     xin  = np.zeros(NumNos)     # Coordenadas 'x' de cada nó (indeformado)
10     yin  = np.zeros(NumNos)     # Coordenadas 'y' de cada nó (indeformado)
11     zin  = np.zeros(NumNos)     # Coordenadas 'z' de cada nó (indeformado)
12     dz = np.zeros(NumNos)       # Vetor com os deslocamentos em 'z'
13     rx = np.zeros(NumNos)       # Vetor com as rotações em 'x'
14     ry = np.zeros(NumNos)       # Vetor com as rotações em 'y'
15     conn = np.zeros(NumElem*4)  # Vetor de conectividade
16     Mx = np.zeros(NumNos)       # Vetor dos momentos em 'x'
17     My = np.zeros(NumNos)       # Vetor dos momentos em 'y'
18     Mxy = np.zeros(NumNos)      # Vetor dos momentos em 'xy'
19     ctype = np.zeros(NumElem)   # Vetor com o tipo de elemento
20     offset = np.zeros(NumElem)  # Vetor 'offset' usados pelo 'EVTK'
21 
22     for i in range(0, NumNos):
23 
24         ### Coordenadas de cada nó (deformado):
25         x[i] = coord[i][0]
26         y[i] = coord[i][1]
27         z[i] = 0           + d[3*i] * FatorEscala
28 
29         ### Coordenadas de cada nó (indeformado):
30         xin[i] = coord[i][0]
31         yin[i] = coord[i][1]
32         zin[i] = 0
33 
34         ### Vetor com os deslocamentos em 'z' (mm):
35         dz[i] = d[3*i] * 1000
36 
37         ### Vetor com as rotações em 'x' (10e-3 rad):
38         rx[i] = d[3*i+1] * 1000
39 
40         ### Vetor com as rotações em 'y' (10e-3 rad):
```

```
41         ry[i] = d[3*i+2] * 1000
42
43         ### Vetor com as rotações em 'y' (10e-3 rad):
44         Mx[i]  = mx[i]
45         My[i]  = my[i]
46         Mxy[i] = mxy[i]
47
48     for e in range(0, NumElem):
49         for j in range(0, 4):
50
51             ### Vetor de conectividade para elementos retangulares:
52             conn[4*e + j] = conec[e][j] - 1
53
54             ### Vetor com o tipo do elemento:
55             ctype[e] = VtkQuad.tid
56
57         ### Vetor 'offset' posição do último nó do elemento (começando de '1'):
58         offset[e] = 4*(e+1)
59
60         ### Vetores com os esforços internos([kN] e [m]):
61     #Nx  = mx[0][:] * 10**-3
62     #Ny  = my[1][:] * 10**-3
63     #Nxy = mxy[2][:] * 10**-3
64
65     ### Criação do arquivo 'VTK':
66     unstructuredGridToVTK(nome, x, y, z, connectivity = conn,
67                           offsets = offset, cell_types = ctype,
68
69                           # Esforços internos nos elementos:
70                           cellData = {"Mx (kip.in/in)" : Mx,
71                                       "My (kip.in/in)" : My,
72                                       "Mxy (kip.in/in)": Mxy,},
73
74                           # Deslocamentos nos nós:
75                           pointData = {"dz (10e-3 in)": dz,
76                                        "rx (10e-3 rad)": rx,
77                                        "ry (10e-3 rad)": ry,
78                                        "Mx (kip.in/in)": Mx,
79                                        "My (kip.in/in)": My,
80                                        "Mxy (kip.in/in)": Mxy})
81
82     unstructuredGridToVTK(nome+'_ind', xin, yin, zin,
83                           connectivity = conn, offsets = offset,
```

3.3 - Criação do arquivo VTU

```
1 PreParaview(ne, nnos, coord, conec, nome, d, mx, my, mxy, FatorEscala)
```

```
<ipython-input-118-d00ed4ebbd91>:27: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will er
  z[i] = 0           + d[3*i] * FatorEscala
<ipython-input-118-d00ed4ebbd91>:35: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will er
  dz[i] = d[3*i] * 1000
<ipython-input-118-d00ed4ebbd91>:38: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will er
  rx[i] = d[3*i+1] * 1000
<ipython-input-118-d00ed4ebbd91>:41: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will er
  ry[i] = d[3*i+2] * 1000
<ipython-input-118-d00ed4ebbd91>:44: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will er
  Mx[i]  = mx[i]
<ipython-input-118-d00ed4ebbd91>:45: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will er
  My[i]  = my[i]
```