UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT
DEPARTAMENTO DE ENGENHARIA MECÂNICA – DEM

LUCAS BUBLITZ

PHILLIPO: APLICAÇÃO DO MÉTODO DE ELEMENTOS FINITOS NA ANÁLISE ESTÁTICA DE ESTRUTURAS SÓLIDAS UTILIZANDO ALGUNS ASPECTOS DE PROCESSAMENTO PARALELO E EMPACOTAMENTO

JOINVILLE
2023

LUCAS BUBLITZ

PHILLIPO: APLICAÇÃO DO MÉTODO DE ELEMENTOS FINITOS NA ANÁLISE ESTÁTICA DE ESTRUTURAS SÓLIDAS UTILIZANDO ALGUNS ASPECTOS DE PROCESSAMENTO PARALELO E EMPACOTAMENTO

Dissertação apresentada ao Programa de graduação em Engenharia Mecânica do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, como requisito parcial para a obtenção do grau de Bacharel em Engenharia Mecânica.

Orientador: Pablo Muñoz

JOINVILLE
2023

Para gerar a ficha catalográfica de teses e
dissertações acessar o link:
https://www.udesc.br/bu/manuais/ficha

Bublitz, Lucas
PHILLIPO: aplicação do método de elementos finitos na análise estática de estruturas sólidas utilizando alguns aspectos de processamento paralelo e empacotamento / Lucas Bublitz. - Joinville, 2023.
116 p. : il. ; 30 cm.

Orientador: Pablo Muñoz.
.
Dissertação - Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Bacharelado em Engenharia Mecânica, Joinville, 2023.

1. Método dos Elementos Finitos (MEF). 2. Julia (Linguagem de Programação de Computador). 3. Análise estrutural. 5. Engenharia Mecânica. I. Muñoz, Pablo . II. , . III. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Bacharelado em Engenharia Mecânica. IV. Título.

ERRATA

Elemento opcional.

Exemplo:

SOBRENOME, Prenome do Autor. Título de obra: subtítulo (se houver). Ano de depósito. Tipo do trabalho (grau e curso) - Vinculação acadêmica, local de apresentação/defesa, data.

Folha	**Linha**	**Onde se lê**	**Leia-se**
1	10	auto-conclavo	autoconclavo

LUCAS BUBLITZ

PHILLIPO: APLICAÇÃO DO MÉTODO DE ELEMENTOS FINITOS NA ANÁLISE ESTÁTICA DE ESTRUTURAS SÓLIDAS UTILIZANDO ALGUNS ASPECTOS DE PROCESSAMENTO PARALELO E EMPACOTAMENTO

Dissertação apresentada ao Programa de graduação em Engenharia Mecânica do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, como requisito parcial para a obtenção do grau de Bacharel em Engenharia Mecânica.

Orientador: Pablo Muñoz

BANCA EXAMINADORA:

Nome do Orientador e Titulação
Nome da Instituição

Membros:

Nome do Orientador e Titulação
Nome da Instituição

Nome do Orientador e Titulação
Nome da Instituição

Nome do Orientador e Titulação
Nome da Instituição

Joinville, 01 de maio de 2023

Dedico este trabalho à Norma Kaizer Streit (*in memoriam*).

AGRADECIMENTOS

Agradeço, primeiramente, aos meus pais, Gerson e Klissia, e à minha vó, Norma, pelo suporte. Agradeço aos meus amigos: Ana, Gabriel, Willian, Lucas, Wesley (e o outro também!), Filipe e Gustavo, por estarem presente nessa longa caminhada da graduação. Agradeço também ao Prof. Dr. Eduardo Lenz pelas explicações sempre elucidantes.

Por fim, agradeço especialmente ao meu orientador, Prof. Dr. Pablo Muñoz, pela dedicação e paciência durante o desenvolvimento deste trabalho.

"Mas o contraste não me esmaga liberta-me; e a ironia que há nele é sangue meu. O que deveria humilhar-me é a minha bandeira, que desfraldo; e o riso com que deveria rir de mim, é um clarim com que saúdo e gero uma alvorada em que me faço." (Fernando Pessoa em Livro do Desassossego – *com uma pequena alteração minha*)

RESUMO

O Método dos Elementos Finitos (MEF) é uma ferramenta matemática poderosa para analisar fenômenos modelados por equações diferenciais, tais como a avaliação de tensão e deformação em sólidos. Este método consiste na divisão do domínio do campo incógnito em elementos, nos quais a descrição do fenômeno físico é simplificada por meio de funções de interpolação, gerando sistemas de equações algébricas, de forma que, quando justapostos, sua solução aproxima a própria solução da equação diferencial. Esse procedimento requer que sejam feitos muitos cálculo para construir e resolver o sistema, o que leva à necessidade de implementar o método num ambiente computacional. O presente trabalho visa a implementação computacional do Método dos Elementos Finitos, para análise de tensão e deformação em corpos sólidos sob carregamentos estáticos em regime elástico linear, para este fim, foi desenvolvido um módulo escrito na linguagem de programação Julia, utilizando aspectos de processamento paralelo, e algumas características de empacotamento, com o objetivo de expor o método e servir de exemplo menor. O módulo desenvolvido foi PHILLIPO.jl, e encontra-se disponível pelo gerenciador de pacotes Pkg.jl num repositório público hospedado no GitHub, com paralelismo aplicado na montagem da matriz global de rigidez, e integrado na interface de pré e pós-processamento GID. Foram implementados dois tipos de elementos: o triângulo de deformações constantes (CST) e o tetraedro linear, bem como três tipos de condições de contorno: deslocamentos prescritos, forças concentradas nodais e carregamentos em linhas e superfícies. O programa passou por uma etapa de Verificação & Validação (V & V), comparando resultados de deslocamento e tensão com o software Abaqus e o modelo de viga de Euler-Bernoulli, demonstrando que o algoritmo matemático do MEF foi implementado corretamente. Por fim, foram elencados pontos de melhoria para projetos futuros, que frisam o aprimoramento do código em quesitos de legibilidade e empacotamento, e a implementação de novas funcionalidades, como também a realização de testes de desempenho.

Palavras-chave: Método dos Elementos Finitos (MEF). Julia (Linguagem de Programação de Computador). Análise estrutural. Engenharia Mecânica.

ABSTRACT

The Finite Element Method (FEM) is a robust mathematical tool for analyzing physical phenomena described by differential equations, such as the evaluation of stress and strain in solids. It involves discretizing the domain of the unknown field into elements, where the physical phenomenon is approximated using interpolation functions. This process leads to systems of algebraic equations, and their solution, when assembled, approximates the solution of the original differential equation. Given the computational intensity involved in formulating and solving these systems, there arises a necessity for their implementation in a computational environment. This work focuses on the implementation of the Finite Element Method for analyzing stress and strain in solid bodies under static loads in a linear elastic regime. A module is developed in the Julia programming language, incorporating aspects of parallel processing and packaging as a demonstration. The developed module, named PHILLIPO.jl, is accessible via the Pkg.jl package manager in a public repository on GitHub. Parallelism is applied during the assembly of the global stiffness matrix, and integration with the pre and post-processing interface GID is achieved. Two types of elements are implemented: the Constant Strain Triangle (CST) and the linear tetrahedron. Additionally, three types of boundary conditions are considered: prescribed displacements, nodal concentrated forces, and line and surface loads. The program undergoes Verification & Validation (V & V) by comparing displacement and stress results with Abaqus software and the Euler-Bernoulli beam model, demonstrating the correct implementation of the FEM mathematical algorithm. Finally, potential areas for improvement in future projects are identified, emphasizing code enhancement in terms of readability and packaging, as well as the implementation of new features and performance testing.

Keywords: Finite Element Method (FEM), Julia (Computer Programming Language), Structural Analysis, Mechanical Engineering.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

MEF	Método dos Elementos Finitos
FEM	Finite Element Method
SI	Sistema Internacional de Unidades
COO	COOdinate list
CSC	Compressed Sparse Column
CST	Constant Strain Triangle

LISTA DE SÍMBOLOS

$\boldsymbol{\sigma}$	tensor de tensões
$\boldsymbol{\varepsilon}$	tensor de deformações
$\mathscr{B}$	um corpo sólido, elástico, homogêneo e isotrópico, em equilíbrio
σ_{ij}	tensão na direção i e sentido j
ε_{ij}	deformação na direção i e sentido j
Ω	domínio da função deslocamento que compreende os pontos de um sólido ou elemento
$\partial\Omega$	fronteira do domínio Ω
$\hat{\boldsymbol{n}}$	um versor
$\boldsymbol{\varphi}$	função de deslocamento
$\boldsymbol{x}$	vetor posição
u, v, w	componentes da função de deslocamento, nas direções x, y, z, respectivamente.
$\boldsymbol{C}$	matriz constitutiva
E	módulo de elasticidade
ν	coeficiente de Poisson

SUMÁRIO

1 INTRODUÇÃO

> I think of myself as an engineer, not as a visionary or 'big thinker.' I don't have any lofty goals. (Linus Torvalds)

A limitação do ser humano em captar integralmente os fenômenos ao seu redor é evidente, a ponto de não conseguir compreender como eles se dão. Analisar um fenômeno, portanto, separando-o em pequenas partes (ou elementos) cujo comportamento é mais facilmente determinado, e a partir da justaposição delas reconstruir o funcionamento do próprio fenômeno, é um modo intuitivo que engenheiros e cientistas procedem em seus estudos (ZIENKIEWICZ, 2000, p. 2).

O Método dos Elementos Finitos consiste, basicamente, na ideia apresentada de análise, em que o domínio contínuo de uma equação diferencial é subdividido em elementos discretos, descritos por um conjunto de nós formando uma malha. Os elementos têm suas propriedades herdadas do domínio (características, condições de contorno etc.), porém, a descrição do fenômeno físico é simplificada por meio de funções de interpolação, que interpolam os valores do campo. A equação diferencial então é aplicada sobre essas funções, gerando um conjunto de equações algébricas que são, por fim, justapostas sobre os nós da malha, de modo a garantir continuidade, formando um sistema, cuja solução é uma aproximação da solução da própria equação diferencial (QUEK; LIU, 2003, pág. 1 e 2). Esse procedimento é custoso em termos de cálculo, visto que para cada elemento é necessário calcular suas funções de interpolação, e depois justapor todos em um grande sistema de equações algébricas, cuja solução também é custosa. É evidente, então, que o Método dos Elementos Finitos, ou os métodos numéricos em geral, acompanham o desenvolvimento da programação, que tem sido impulsionada pelo avanço do processamento computacional (OñATE, 2009, pág. 2). O poder computacional permite que se trabalhe com um volume inconcebível para a capacidade humana.

Em um programa de elementos finitos, os algoritmos e a estrutura de dados implementados são tão relevantes quanto a própria equação diferencial. É de se esperar que uma aplicação desse método seja acompanhada de aspectos de estruturação de código, cujo objetivo não seja só a otimização computacional, mas a legibilidade e empacotamento, que são características úteis quando se espera a reutilização, aprimoramento continuado e, acima de tudo, a comunicação e distribuição do código-fonte.

A escolha da linguagem de programação para uma aplicação do MEF, é o passo fundamental para se planejar a estrutura do código, pois, são as ferramentas de sintaxe e processamento que a linguagem e seu compilador/interpretador oferecem que vão ditar, em parte, a forma como os algoritmos são implementados, além de outros aspectos de execução, como otimização e estrutura de dados. Comumente, programas comerciais de análise por elementos finitos, como Abaqus e Ansys, são escritos em linguagens compiladas, basicamente, C e FORTRAN, que são sinônimos de robustez e desempenho. Entretanto, essas linguagens também são conhecidas pela sua prolixidade, complexidade de sintaxe de distribuição de bibliotecas.

Na atualidade, linguagens compiladas não se mostram mais atrativas para se desenvolver aplicações práticas na vida dos engenheiros e cientistas (automatização de tarefas, análise de dados e até computação algébrica simbólica). Linguagens como Python e suas bibliotecas, como Pandas e NumPy, utilizando-se de sua sintaxe simplificada, tipagem dinâmica e popularidade (ocupando o TIOBE Index três vezes nos últimos cinco anos), oferecem uma praticidade maior, e possibilitam uma produtividade maior na construção de programas. A conveniência dessas linguagens, porém, é acompanhada de um preço: o desempenho.

A eficiência de linguagens interpretadas é notavelmente inferior a linguagens compiladas, quando são implementadas em problemas grandes e complexos, envolvendo muito cálculos e um grande volume de variáveis, que é o caso de aplicações do MEF. Esse dilema, entre eficiência (de linguagens como C e FORTRAN) e produtividade (de linguagens como Python) é conhecido como *The Two language Problem*, em tradução livre, O Problema de Duas Linguagens.

Visando unificar esses dois mundos, e diminuir a distância entre as linguagens, engenheiros do MIT desenvolveram Julia, "Uma linguagem de programação para a comunidade científica que combina características de linguagens produtivas, como Python ou MATLAB, com características de linguagens focadas em desempenho, como C++ ou Fortran."(BEZANSON et al., 2018, tradução livre). Por conta do sucesso de Julia, e de sua comunidade engajada, a linguagem vem sendo adotada mais e mais no âmbito acadêmico, incluindo a área de elementos finitos, o que motivou a escolha dela para o desenvolvimento deste trabalho.

O Método dos Elementos Finitos é uma ferramenta numérica poderosa para a análise em sólidos, e o seu desenvolvimento em linguagens como Julia oferece uma porta de entrada muito convidativa para novos engenheiros, assim como impulsiona novas pesquisas no campo. Esta monografia aborda o desenvolvimento de um desses programas: PHILLIPO.jl, cujo objetivo é expor e aplicar o MEF, com alguns aspectos de programação paralela e empacotamento.

1.1 MOTIVAÇÃO

O tema surgiu quando o autor se encontrou na tarefa de adicionar uma funcionalidade em um software já existente de elementos finitos e, já tendo visto uma introdução ao assunto na graduação, teve o interesse de se aprofundar. Resolveu, para tanto, por criar seu próprio programa, em Julia, aplicando seus conhecimento prévios de programação, desenvolvendo mais o seu entendimento sobre o Método dos Elementos Finitos, assim como de aspectos numéricos computacionais.

1.2 OBJETIVO

O objetivo geral deste trabalho foi desenvolver uma aplicação de MEF para a análise de tensão e deformação em estruturas sólidas sob carregamentos estáticos em regime elástico linear, utilizando para tanto, aspectos de processamento paralelo, focando em algumas caracte-

rísticas de empacotamento e de legibilidade, com o intuito secundário de expor as facilidades e vantagens da linguagem Julia, como também servir de exemplo menor.

1.2.1 Objetivos específicos

Foram propostos os seguintes objetivos específicos:

1. estudar o MEF aplicado na determinação de deformações de estruturas sólidas em regime elástico e linear, sob carregamentos estáticos (implementando os elementos triangulares e tetraédricos, de deformações constantes);
2. programar os algoritmos de MEF em Julia visando a legibilidade;
3. desenvolver um módulo que seja distribuível pelo gerenciador de pacotes Pkg.jl, em um repositório público hospedado no GitHub;
4. aplicar processamento paralelo em determinadas partes do programa em que as funções nativas não o fazem, a fim de utilizar mais da capacidade de processamento do computador que um código feito sob o paradigma estruturado.

1.3 ORGANIZAÇÃO DO DOCUMENTO

Este documento aborda o desenvolvimento de um módulo em Julia, denominado PHILLIPO.jl, que aplica o Método de Elementos Finitos, integrado à ferramenta de pré e pós-processamento GiD, para realizar a análise das tensões em estruturas sólidas e elásticas sob carregamentos estáticos, e está organizado em capítulos que abordam:

1. A mecânica dos sólidos: tensão e deformação no regime elástico;
2. O método de elementos finitos aplicado no equilíbrio estático de estruturas sólidas;
3. A linguagem Julia;
4. PHILLIPO.jl;
5. Validação e verificação de resultados;
6. Objetivos alcançados e melhorias em projetos futuros;
7. Conclusão.

O código fonte de PHILLIPO.jl, sob a licença LGPL, assim como o das interfaces de integração com o GiD, estão impressas em anexos, cujos arquivos, incluindo o LaTeX deste documento, podem ser acessados no repositório: <https://github.com/lucas-bublitz/PHILLIPO.jl>.

Todas as figuras foram criadas pelo próprio autor.

2 A MECÂNICA DOS SÓLIDOS: TENSÃO, DEFORMAÇÃO E DESLOCAMENTO

A Mecânica dos Sólidos estuda o comportamento de objetos sólidos sob carregamentos externos, aplicando métodos analíticos para determinar características de resistência, rigidez e estabilidade. Sua aplicação é voltada ao projeto de estruturas a fim de que cumpram determinadas exigências, tais como deformação máxima, capacidade de carga e peso, ou economia de materiais. E, por meio de ferramentas matemáticas, estuda os efeitos de tensão e deformação no interior de corpos sólidos (POPOV, 1990, pág. 2).

Um corpo sólido por ser descrito como um conjunto de pontos materiais que resistem a forças cisalhantes, ou seja, que resistem a trações tangenciais às suas superfícies. Neste trabalho são considerados corpos sólidos elásticos, homogêneos e isotrópicos. Aqui, um corpo com essas características é denominado $\mathscr{B}$.

Corpos sólidos elásticos são aqueles que, quando submetidos a carregamentos, deformam-se, mas quando o carregamento é retirado, retornam à sua forma original. Corpos sólidos homogêneos são aqueles que possuem as mesmas propriedades físicas em todos os pontos materiais (tais como massa específica, rigidez etc.), de modo que uma porção do corpo seja indistinta do restante. Corpos sólidos isotrópicos são aqueles que possuem as mesmas propriedades físicas em todas as direções do espaço.

Este capítulo aborda os seguintes temas de Mecânica dos Sólidos, relevantes para o desenvolvimento inicial do módulo PHILLIPO.jl voltado à análise de estruturas elásticas sob carregamentos constantes:

1. conceito de tensão;
2. conceito de deslocamento e deformação;
3. comportamento dos materiais: a lei de Hooke generalizada;

2.1 TENSÃO

Um corpo sólido se deforma quando submetido a carregamentos externos[1] sobre sua geometria, de modo que qualquer seção plana arbitrária do corpo revele forças internas que estejam em equilíbrio entre si, e que sejam balanceadas pelos carregamentos externos. As forças internas geralmente variam ao longo do corpo, como também dependem da orientação do plano dessa seção. Seja um corpo $\mathscr{B}$, sólido, em equilíbrio e de geometria qualquer, descrito sobre um sistema de referência (xyz com origem em o), submetido a forças externas na forma do carregamento $\boldsymbol{F}$ (veja figura 1). Sejam também as seções $S_{1,2,3}$, planos de corte através desse corpo (normais aos versores do sistema de referência), em que atuam as forças internas $\boldsymbol{P}$, conforme a figura 1b. $\Delta\boldsymbol{P}$ é a resultante de forças que atuam sobre uma área ΔA (centrada em

[1] Expressão que se refere tanto a carregamentos térmicos, quanto mecânicos, embora o primeiro não seja assunto deste texto.

um certo ponto p), pertencente a S. O limite da razão entre cada componente de $\Delta \boldsymbol{P}$ (tangenciais e normais) e a área ΔA, quando $\Delta A \to 0$, define o vetor tração, cuja decomposição nos eixos x, y e z, define as componentes do tensor tensão sobre o ponto p, de forma que

$$\tau_{xx} = \lim_{\Delta A_x \to 0} \frac{\Delta P_x}{\Delta A_x}, \qquad \tau_{xy} = \lim_{\Delta A_x \to 0} \frac{\Delta P_y}{\Delta A_x}, \qquad \tau_{xz} = \lim_{\Delta A_x \to 0} \frac{\Delta P_z}{\Delta A_x}, \tag{1}$$

em que os índices de τ indicam, o primeiro, a normal do plano infinitesimal em que a tensão atua, e, o segundo, sua direção. Por conveniência, as tensões normais (aquelas que atuam perpendicularmente ao plano) são representadas por σ, ao invés de se utilizar τ com índices repetidos ($\tau_{xx} \equiv \sigma_x$). O símbolo τ, então, é reservado às tensões de cisalhamento, que atuam tangencialmente ao plano infinitesimal. No SI, a tensão é mensurada em Pascal ([Pa] = [N/mš]) (POPOV, 1990, pág. 5).

Se o mesmo procedimento for realizado para cada face do elemento cúbico, formado por mais três seções paralelas a $S_{1,2,3}$ da figura 1c, teremos a configuração da tração em três planos perpendiculares entre si para um certo ponto p em $\mathscr{B}$, conforme a figura 2, o que descreve o estado de tensão para aquele ponto. As componentes do estado de tensão podem ser dispostas na forma de uma matriz representativa do tensor de segunda ordem, denominado tensor de tensões, e, de acordo com Popov (1990), é

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_x & \tau_{yx} & \tau_{zx} \\ \tau_{xy} & \sigma_y & \tau_{zy} \\ \tau_{xz} & \tau_{yz} & \sigma_z \end{bmatrix}, \tag{2}$$

em que a linha indica o plano em que a componente age, e a coluna, sua direção.

O tensor de tensões é simétrico, o que pode ser demonstrado realizando o somatório de momentos sobre o elemento infinitesimal de tensão, de modo que esteja em equilíbrio. Oportunamente, escolhendo o ponto central para a análise do equilíbrio angular, podemos descrever as seguintes relações (POPOV, 1990, pág. 8):

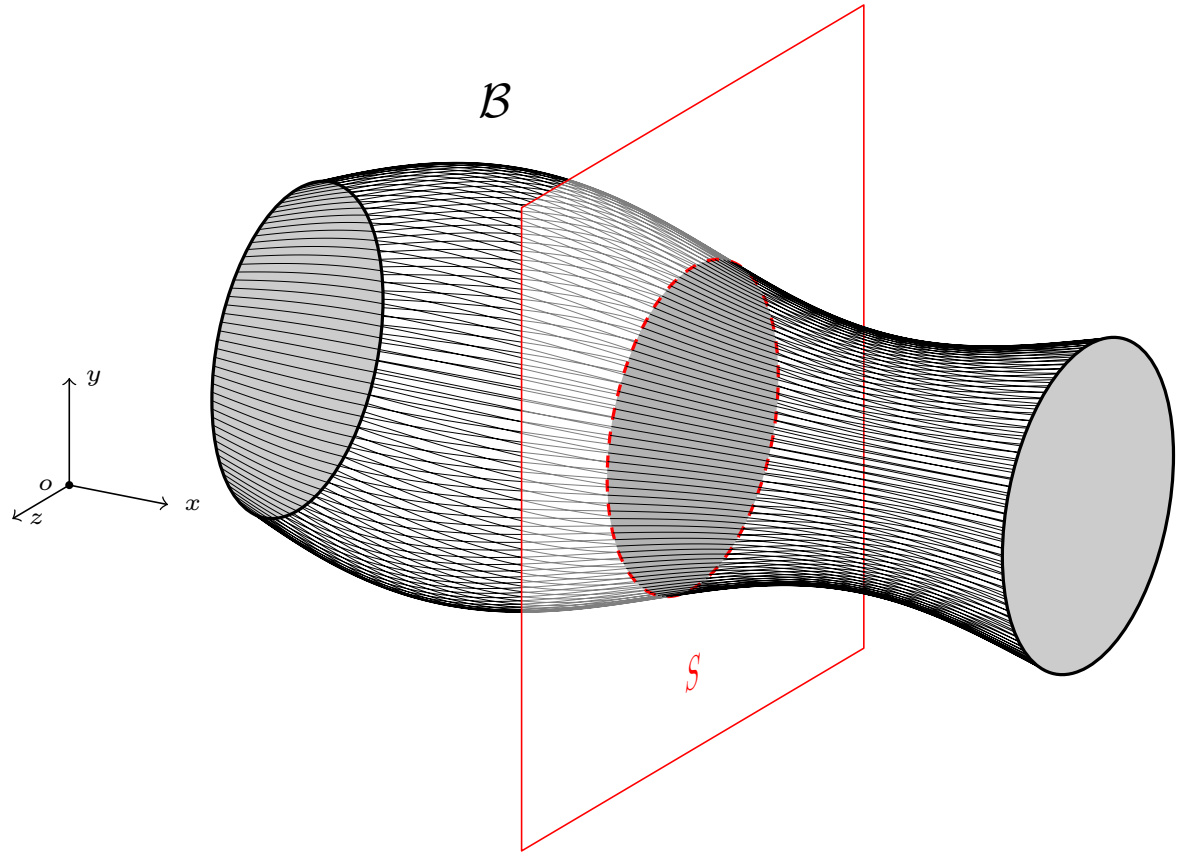
$$\begin{cases} \vec{i} : \tau_{zy}(dxdy)\dfrac{dz}{2} - \tau_{yz}(dxdz)\dfrac{dy}{2} - \tau_{zy}(dydx)\dfrac{dz}{2} + \tau_{yz}(dxdy)\dfrac{dy}{2} = 0 \\ \vec{j} : \tau_{xz}(dydz)\dfrac{dx}{2} - \tau_{zx}(dxdy)\dfrac{dz}{2} - \tau_{zx}(dxdy)\dfrac{dz}{2} + \tau_{xz}(dydz)\dfrac{dx}{2} = 0 \\ \vec{k} : \tau_{yx}(dxdz)\dfrac{dy}{2} - \tau_{xy}(dydz)\dfrac{dx}{2} - \tau_{xy}(dydz)\dfrac{dx}{2} + \tau_{yx}(dxdz)\dfrac{dy}{2} = 0 \end{cases} \implies \begin{cases} \tau_{zy} = \tau_{yz} \\ \tau_{xz} = \tau_{zx} \\ \tau_{yx} = \tau_{xy} \end{cases}. \tag{3}$$

Portanto,

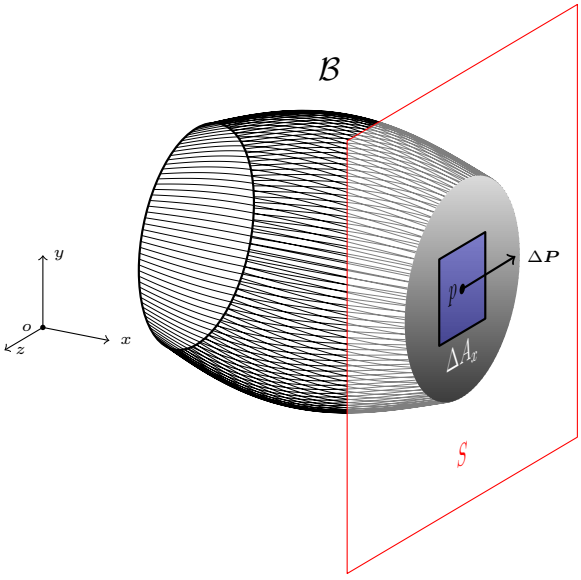
$$\tau_{ij} = \tau_{ji} \iff \boldsymbol{\sigma} = \boldsymbol{\sigma}^t. \tag{4}$$

Essa propriedade implica que o tensor de tensões possua apenas seis componentes independentes, ao invés de nove. Aproveitando-se disso, a notação de Voigt distribui as componentes independentes em uma matriz coluna de seis elementos, tal que, de acordo com Roylance (1995),

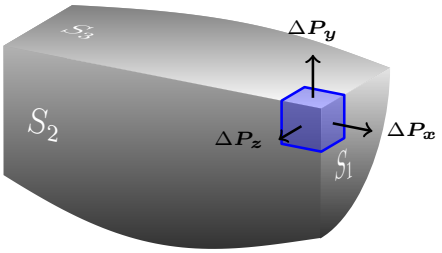
Figura 1 – Forças internas: seção em um sólido qualquer



(a)

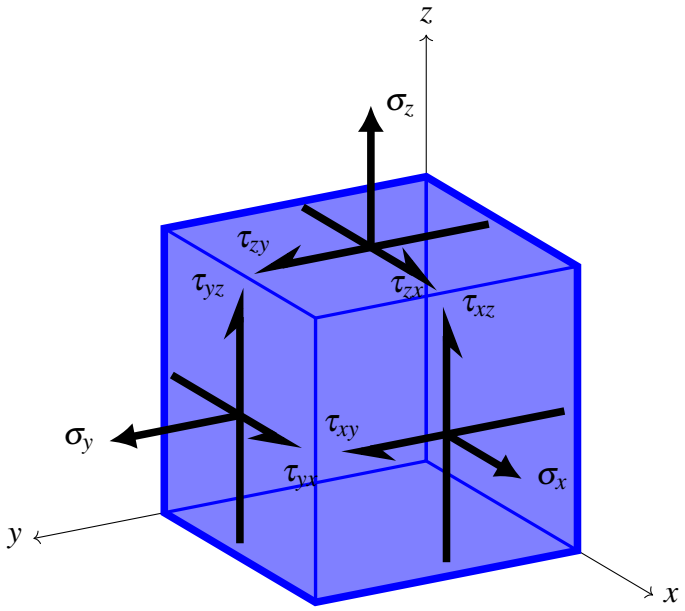


(b)



(c)

Figura 2 – Estado de tensão



$$\{\boldsymbol{\sigma}\} = \begin{bmatrix} \sigma_x & \sigma_y & \sigma_z & \tau_{xy} & \tau_{xz} & \tau_{yz} \end{bmatrix}^t. \tag{5}$$

2.1.1 Equações diferenciais governantes do equilíbrio estático

Outro fato importante sobre o estado de tensão vem do equilíbrio de forças. Assumindo que a distribuição de tensão $\boldsymbol{\sigma}(x,y,z)$ é contínua e diferenciável ao longo do domínio Ω do sólido $\mathscr{B}$, podemos analisar sua variação sobre um elemento cúbico infinitesimal, de modo que a força resultante sobre ele seja nula. Como o tensor de tensões representa a decomposição das forças internas agindo sobre as faces de um cubo infinitesimal, o somatório de forças é a própria integral da tensão ao longo dessas superfícies, ou seja,

$$\oiint_A \boldsymbol{\sigma} \cdot d\boldsymbol{A} = \mathbf{0}. \tag{6}$$

São desconsideradas, aqui, as forças de campo, como gravidade ou eletromagnética (ROYLANCE, 1995, pág. 4, The Equilibrium Equations).

A integração é realizada sobre uma região fechada, fronteira de um subdomínio de Ω. Como a tensão foi assumida contínua e diferenciável em todo Ω, podemos aplicar o teorema da divergência[2] à equação 6, obtendo que

$$\int_V \nabla \cdot \boldsymbol{\sigma} dV = \mathbf{0}. \tag{7}$$

[2] O teorema da divergência, também conhecido como teorema de Gauss, afirma que, dada uma função vetorial contínua e diferenciável sobre uma região fechada: $\oiint_{\partial\omega} \boldsymbol{f} d\boldsymbol{A} = \iiint_\Omega \nabla \cdot \boldsymbol{f} dV$, em que $\partial\Omega$ representa a fronteira da região Ω.

Essa relação é válida para qualquer volume infinitesimal no sólido, independente de sua orientação (ou seja, independente da escolha do sistema de referência), portanto o domínio de integração V é um volume arbitrário. Deste modo, como a integração deve ser nula independentemente do subdomínio de Ω escolhido para compor V, a função integrada deve ser nula em todo domínio, ou seja,

$$\nabla \cdot \boldsymbol{\sigma} = \mathbf{0}. \tag{8}$$

Esse resultado é o sistema de equações diferenciais parciais de equilíbrio, que governa o estado de tensão. A partir dele, é possível obter a distribuição de tensão sobre o sólido, desde que sejam conhecidas as condições de contorno. Comumente esta distribuição é solucionada por métodos numéricos, como o MEF, devido à dificuldade em encontrar soluções analíticas para geometrias muito complicadas.

Explicitamente, para três dimensões, o sistema de equações diferenciais de equilíbrio é

$$\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} = 0, \tag{9}$$

$$\frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} = 0, \tag{10}$$

$$\frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \sigma_z}{\partial z} = 0. \tag{11}$$

A direção em que o elemento infinitesimal é orientado altera as componentes do seu estado de tensão, de modo que sua rotação evidencia direções nas quais as tensões não têm componentes tangenciais, ou seja, têm cisalhamento nulo. Essas tensões são chamadas, então, tensões principais.

O tensor de tensões é uma transformação linear que recebe um vetor unitário $\hat{\boldsymbol{n}}$ e retorna o vetor da tração resultante, $\boldsymbol{t}$, agindo sobre um plano normal a $\hat{\boldsymbol{n}}$.[3] Caso exista um vetor tração resultante que tenha a mesma direção de $\hat{\boldsymbol{n}}$, a tensão não terá componentes tangenciais, uma vez que, sendo colinear ao vetor unitário, é normal ao plano definido por ele. Em termos matemáticos, é o mesmo que $\boldsymbol{t} = \sigma\hat{\boldsymbol{n}}$[4], ou, aplicando a transformação linear $\boldsymbol{\sigma}$,

$$\boldsymbol{\sigma}\hat{\boldsymbol{n}} = \sigma\hat{\boldsymbol{n}}. \tag{12}$$

Observando a forma dessa equação, é evidente que $\hat{\boldsymbol{n}}$ é um autovetor de $\boldsymbol{\sigma}$, e σ é o autovalor correspondente, portanto, determiná-los é equivalente a encontrar as tensões principais, ou seja, as raízes do polinômio característica do tensor de tensões:

[3] Aplicando um vetor $\hat{\boldsymbol{n}}$ trivial $(\hat{\boldsymbol{i}}, \hat{\boldsymbol{j}}, \hat{\boldsymbol{k}})$ à trasnformação $\boldsymbol{\sigma}$, obtém-se as próprias tensões mostradas na figura 2, como já era de se esperar.

[4] $|\sigma|$, nesse sentido, seria a norma da tração resultante, uma vez que $\hat{\boldsymbol{n}}$ é unitário e adimensional. $|\boldsymbol{t}| = |\sigma\hat{\boldsymbol{n}}| = |\sigma||\hat{\boldsymbol{n}}| = |\sigma|$

$$\det(\boldsymbol{\sigma} - \sigma \boldsymbol{I}) = 0, \qquad \text{ou} \qquad \begin{vmatrix} \sigma_x - \sigma & \tau_{yx} & \tau_{zx} \\ \tau_{xy} & \sigma_y - \sigma & \tau_{zy} \\ \tau_{xz} & \tau_{yz} & \sigma_z - \sigma \end{vmatrix} = 0. \tag{13}$$

Para o caso bidimensional, a solução desses sistema é bem conhecida, sendo dada por:

$$\sigma_{1,2} = \frac{\sigma_x + \sigma_y}{2} \pm \sqrt{\left(\frac{\sigma_x - \sigma_y}{2}\right)^2 + \tau_{xy}^2}. \tag{14}$$

2.2 DESLOCAMENTO E DEFORMAÇÃO

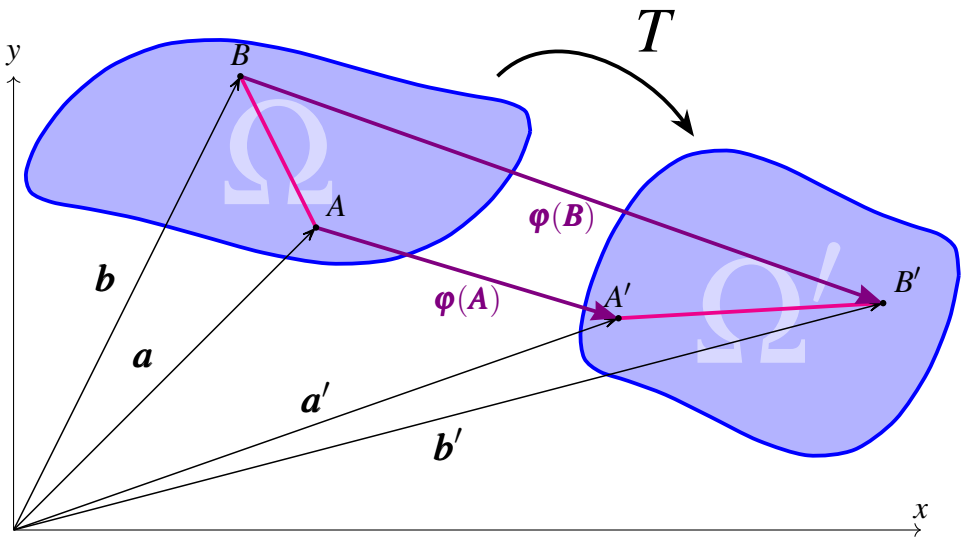
O deslocamento de um sólido é uma função vetorial que mapeia a cada ponto do seu domínio à variação entre sua posição original e a deslocada, de modo que se possa descrever sua posição final sofre em termos do deslocamento e de sua posição original.

Seja um corpo $\mathscr{B}$ definido sobre uma região Ω, e a função $\boldsymbol{\varphi}(\boldsymbol{x})$, a representação de seu deslocamento, a transformação que mapeia a posição original na posição deformada de cada ponto, mapeando Ω para Ω', é dada por

$$T(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{\varphi}(\boldsymbol{x}), \qquad \boldsymbol{\varphi}(x,y,z) = \begin{Bmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{Bmatrix}. \tag{15}$$

em que u, v, w são as componentes do deslocamento nas direções de x, y, z, respectivamente, $\boldsymbol{x} = (x,y,z)$ é o vetor posição do ponto (ver figura 3).

Figura 3 – Função de deslocamento sobre a região de um sólido



Quando um sólido passa por uma transformação de deslocamentos, pode sofrer translações e deformações, ambas caracterizadas pelas distâncias entre pontos do corpo antes e após a transformação. A figura 3 exibe a transformação sobre um corpo $\mathscr{B}$, e como o segmento de reta AB é mapeado para sua nova configuração sobre Ω'.

Nesse sentido, são duas as possibilidades:

1. As distâncias entre os pontos permanecem a mesmas. Nesse caso, podemos dizer que a transformação é uma translação[5], e que o corpo não sofreu deformação, pois sua geometria foi conservada. Em termos matemáticos,

$$||T(\boldsymbol{a}) - T(\boldsymbol{b})|| = ||\boldsymbol{a} - \boldsymbol{b}||, \ \forall \boldsymbol{a}, \boldsymbol{b} \in \Omega. \tag{16}$$

2. As distâncias entre os pontos não se conservam. Quando isso ocorre, a geometria do corpo é alterada, e podemos dizer que houve deformação [6]. Em termos matemáticos, podemos dizer que

$$\exists \boldsymbol{a}, \boldsymbol{b} \in \Omega : \ ||T(\boldsymbol{a}) - T(\boldsymbol{b})|| \neq ||\boldsymbol{a} - \boldsymbol{b}||. \tag{17}$$

A deformação normal é o alongamento ou encurtamento sofrido pelas linhas entre os pontos do corpo, de forma que é descrita entre a variação do comprimento do segmentos e o comprimento original, ou seja, a varição relativa do comprimento. Como deformação geralmente varia ao longo do sólido, podemos descrevê-la observado seus efeitos sobre um elemento infinitesimal, e, similarmente à tensão, descrevendo um tensor de segunda ordem, com componentes normais e cisalhantes. (figura 4) (POPOV, 1990, pág. 143).

A figura 4a mostra um elemento infinitesimal ($dx \times dy$), em que o segmento AD sofreu tanto uma translação quanto uma deformação, dados pelo campo de deslocamento $\boldsymbol{\varphi}$, de modo a se tornar $A'D'$. A deformação normal desse segmento é dado por

$$\varepsilon_x = \frac{||A'D'|| - ||AD||}{||AD||}. \tag{18}$$

O comprimento do segmento antes da deformação AD é o próprio infinitesimal dx. Já o comprimento do deformado, é dado pela diferença das posições dos pontos deslocados $||A'D'||$. Em termos do campo de deslocamento, é o mesmo que

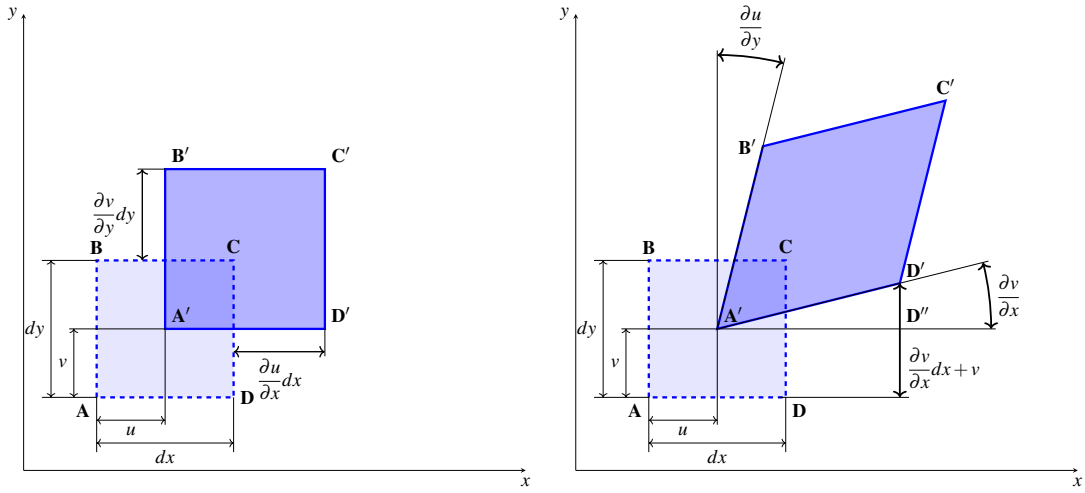
$$||A'D'|| = (D_x + u(D_x)) - (A_x + u(A_x)), \tag{19}$$

em que o subscrito x denota a respectiva componente da posição do ponto.

[5] Esse transformação também é uma isomeria, pois preserva a métrica do espaço, ou seja, o produto interno, de forma que $T(\boldsymbol{a} \cdot \boldsymbol{b}) = T(\boldsymbol{a}) \cdot T(\boldsymbol{b})$.

[6] Deformação e traslação não são excludentes. Um corpo pode deformar e também trasladar.

Figura 4 – Deformação de um elemento quadrado infinitesimal



(a) deformação linear (b) deformação angular

Como $D_x = A_x + dx$, podemos expandir a função u ao redor de A_x, obtendo que[7]

$$u(D_x) = u(A_x + dx) = u + \frac{\partial u}{\partial x} dx, \tag{20}$$

Portanto,

$$||A'D'|| - ||AD|| = (D_x + u + \frac{\partial u}{\partial x} dx - u - A_x) - (D_x - A_x) = \frac{\partial u}{\partial x} dx \tag{21}$$

em que A_x representa a projeção do ponto A em x. Agora, substituindo essa expressão na equação 18, temos a definição da deformação normal na direção de x, em que

$$\varepsilon_x = \frac{\partial u}{\partial x}. \tag{22}$$

O mesmo procedimento pode ser feito tanto na direção de y, com o segmento AB,quanto na direção de z, assumindo um elemento infinitesimal cúbico, tal qual a figura 2 do estado de tensão, obtendo-se, assim, todas as definições básicas de deformação normal (POPOV, 1990):

$$\varepsilon_x = \frac{\partial u}{\partial x}, \qquad \varepsilon_y = \frac{\partial u}{\partial y}, \qquad \varepsilon_z = \frac{\partial u}{\partial z}. \tag{23}$$

Na figura 4b, ocorre a deformação por cisalhamento, em que os segmentos AD e AB são, além de transladados, rotacionados em torno de A', de modo a distorcer a geometria do elemento. Agora, a deformação ocorre tanto na direção em x, quanto em y, e é definida pela alteração do ângulo reto $\angle BAD$, determinada, em termos do campo de deslocamentos (tal como na deformação normal) analisando o triângulo $A'D'D''$.

[7] Os termos $O(3)$ da expansão são desconsiderados pois a diferença entre os pontos é infinitesimal, logo $dx >> dx^2$.

A função v, quando variada em x da posição de A até D, descreve o deslocamento dos pontos da face inferior do elemento infinitesimal na direção de y, ou seja, a hipotenusa do triângulo $A'D'D''$. A inclinação, portanto, dessa reta é própria derivada de v na direção x, ou seja,

$$\angle D'A'D'' = \tan\frac{\partial v}{\partial x} \approx \frac{\partial v}{\partial x}.^{8} \tag{24}$$

Outro modo de se obter a mesma expressão é aplicar a definição trigonométrica da tangente sobre o triângulo $A'D'D''$, de forma que, em suma,

$$|A'D''| = \sqrt{dx^2 - \left(\frac{\partial v}{\partial x}dx\right)^2}, \text{Teorema de Pitágoras} \tag{25}$$

$$= \sqrt{dx^2}, \left(\frac{\partial v}{\partial x}dx\right)^2 << dx, \tag{26}$$

$$|A'D''| = dx, \tag{27}$$

$$\tan\angle D'A'D'' = \frac{|D'D''|}{|A'D''|}, \tag{28}$$

$$= \frac{\partial v}{\partial x}dx\frac{1}{dx}, \tag{29}$$

$$= \frac{\partial v}{\partial x}. \tag{30}$$

$$\tag{31}$$

O mesmo pode ser feito na direção de y, para encontrar a inclinação do segmento $A'B'$, como também para z, considerando um elemento infinitesimal cúbico.

Portanto, a deformação cisalhante do elemento infinitesimal é dada, em termos do deslocamento, por

$$\gamma_{xy} = \gamma_{yx} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \tag{32}$$

$$\gamma_{xz} = \gamma_{zx} = \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \tag{33}$$

$$\gamma_{yz} = \gamma_{zy} = \frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \tag{34}$$

$$\tag{35}$$

Por convenção[9]

[8] É essa aproximação é devida pois para ângulos suficientemente pequenos: $\tan\theta = \theta$, o que pode ser verificado expandindo a série de Taylor ao redor de $x = 0$.

[9] Essa convenção não é mero simbolismo, mas faz com que o tensor de deformações tenha propriedades interessantes, como a possibilidade de se descrever a energia interna em termos do produto escalar entre os tensores de deformação e de tensão na notação de Voigt.

O tensor de deformações é

$$[\varepsilon] = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{zz} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) & \frac{1}{2}\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\ \frac{1}{2}\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) & \frac{\partial v}{\partial y} & \frac{1}{2}\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \frac{1}{2}\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) & \frac{1}{2}\left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right) & \frac{\partial w}{\partial z} \end{bmatrix}. \tag{36}$$

Tal como o tensor de tensões, o tensor de deformações é simétrico, e, portanto, só possui seis componentes independentes. Na notação de Voigt, de acordo com Roylance (1995),

$$\{\varepsilon\} = \begin{bmatrix} \varepsilon_x & \varepsilon_y & \varepsilon_z & \gamma_{xy} & \gamma_{xz} & \gamma_{yz} \end{bmatrix}^t \tag{37}$$

2.3 A LEI DE HOOKE

Em corpos sólidos, a deformação está relacionada diretamente com a tensão em seu interior, de modo que se possa, dentro de certas condições, descrever uma transformação linear entre elas. Essa transformação é nomeada *Lei de Hooke*, e, utilizando a notação de Voigt, é descrita por

$$\{\sigma\} = [\boldsymbol{C}]\{\varepsilon\}. \tag{38}$$

em que $[\boldsymbol{C}]$ é denominada *matriz constitutiva*[10], e é definida em termos das características do material do corpo, tais como módulo de elasticidade e coeficiente de Poisson.

2.3.1 A Lei de Hooke Uniaxial

Sejam o quadrilátero $\mathscr{B}$, um corpo sólido, homogêneo, em equilíbrio, de comprimento L, engastado em sua face esquerda, e de seção transversal A. Seja também F, uma força constante que atua sobre a face direita de $\mathscr{B}$, na direção de x, que a deforma em $\mathscr{B}'$ até um comprimento $L + \Delta L$, tal como na figura 5a.

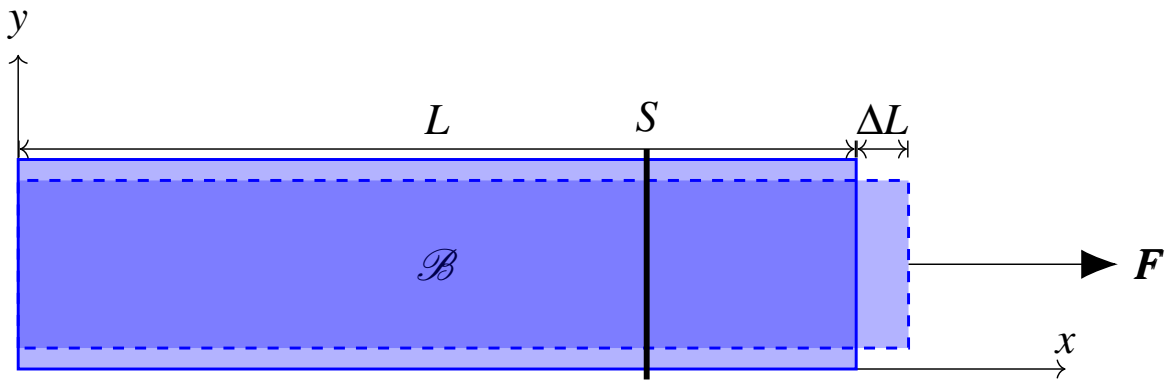
A tensão desenvolvida em um ponto, suficientemente longe da região de aplicação da carga, de uma seção S de $\mathscr{B}$, perpendicular à força $\vec{F}$, pode ser descrita em termos do módulo de elasticidade, E (também denominado módulo de Young), que é uma característica intrínseca do material de $\mathscr{B}$, e da deformação, ε_x, atuando na mesma direção da força. Essa relação, denominada *Lei de Hooke Uniaxial*, é linear da forma

$$\sigma_x = E\varepsilon_x. \tag{39}$$

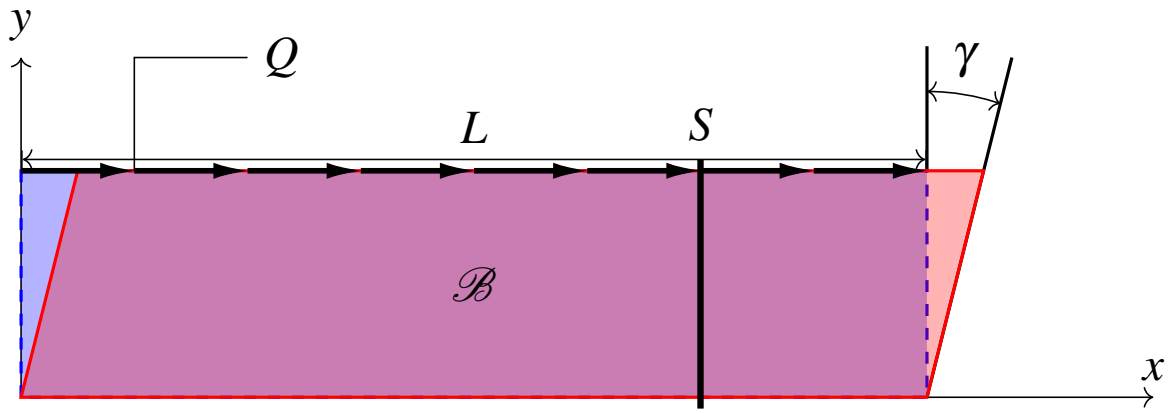
A unidade de E é a mesma de σ_x, o que é coerente, pois ε_x é adimensional, ou seja, o módulo de elasticidade é medido, no SI, em Pascal.

[10] A matriz constitutiva é uma forma de notação abreviada para descrever essa relação. Da mesma forma que o tensor de tensões é abreviado por uma matrix coluna na notação de Voigt, devido a sua simetria, a matriz constitutiva é a abreviação de do tensor de elasticidade, um tensor de quarta ordem que mapeia o espaço das deformações no das tensões.

Figura 5 – quadrilátero deformado



(a) por tração



(b) por cisalhamento

Essa relação desconsidera outros efeitos de deformação no interior do sólido, como a deformação transversal, ε_y (que pode ser observada como o encurtamento da altura da barra na figura 5a), e a por cisalhamento, γ_{xy}. [11].

2.3.2 A Lei de Hooke em Cisalhamento

Seja um quadrilátero $\mathscr{B}$, um corpo sólido, homogêneo, em equilíbrio, de comprimento L, engastado em sua face inferior, e de seção transversal S, e seja Q, um carregamento constante que atua sobre a face superior de $\mathscr{B}$, tangencialmente, na direção de x, que a deforma em $\mathscr{B}'$, inclinando-a, até um ângulo γ, tal como na figura 5b.

A tensão desenvolvida em uma seção S de $\mathscr{B}$, paralela ao carregamento Q, pode ser descrita em termos do módulo de cisalhamento, G, que é uma característica intrínseca do material de $\mathscr{B}$, e da deformação angular, γ, atuando na inclinação das seções verticais. Essa relação, denominada *Lei de Hooke em Cisalhamento*, é linear da forma

[11] O limite de elasticidade do material é determinado experimentalmente, observando como se deforma sobre carregamentos controlados, determinando a região de deformações em que o material preserva-se na Lei de Hooke, ou seja, mantém uma relação linear entre deformação e tensão, e ao ser aliviado dos carregamentos externos, volta à geometria original.

$$\tau_{xy} = G\gamma_{xy} \implies \tau_{xy} = 2G\varepsilon_{xy}. \tag{40}$$

O módulo de cisalhamento tem a mesma unidade de tensão, e, assim como o módulo de Young (γ é adimensional), não depende da geometria do corpo, mas do material de que é feito.

2.3.3 O Coeficiente de Poisson

Na deformação uniaxial de um corpo sólido, tal como na figura 5a, é razoável que o corpo também se deforme nas direções transversais, de forma que existe, dentro de determinados limites do regime elástico, uma relação entre essa deformações. De acordo com Lubliner (2017), para deformação axial em x observa-se uma deformação transversal em y, de modo que

$$\nu = -\frac{\varepsilon_y}{\varepsilon_x}, \tag{41}$$

em que ν representa o coeficiente de Poisson, a razão entre a deformação transversal e a deformação axial.

O coeficiente de Poisson é uma característica intrínseca do material, e, assim como os módulos de Young e de cisalhamento, não depende da geometria do corpo, mas do material de que é feito (dentre outras condições mais específicas), entretanto, pode variar conforme a direção da deformação, para materiais que não são isotrópicos, diferentemente, de $\mathscr{B}$. Aqui ν é tratado como constante.

2.3.4 O Princípio da Sobreposição & A Lei de Hooke Generalizada

> O *princípio da sobreposição* permite a aditividade de efeitos (leia-se, deformações) na presença de múltiplas causa (leia-se, tensões). Invocando esse princípio, nós podemos expressão o total de deformação percebida pelo corpo como a soma de todas as deformações devidas aos componentes individuais de tensão presentes no corpo. (LUBLINER, 2017, pág. 252, tradução livre)

Esse princípio é válido quando, de acordo com (LUBLINER, 2017):

1. as equações de equilíbrio são lineares nas tensões;

   Observando a equação 11, é possível notar que, dado dois conjuntos de tensões que satisfazem as equações de equilíbrio, a soma desses dois conjuntos também o faz, ou seja, as equações de equilíbrio são lineares nas tensões. Em termos matemáticos, é o mesmo que demonstrar a linearidade do operador divergência ($\nabla \cdot (*)$).

2. as relações entre deformação e deslocamento são lineares;

   Tal como no item anterior, é possível demonstrar essa propriedade tomando dos conjuntos de deslocamentos arbitrários, que, quando somados, levam um conjunto de deformações

que equivale ao somatório das deformações de cada conjunto de deslocamentos individualmente.

3. as relações entre tensão e deformação são lineares.

   Observando as relações definidas para o módulo de Young, o módulo de cisalhamento e o coeficiente de Poisson, fica evidente que todas são lineares.

Assumindo agora que sobre um elemento infinitesimal cúbico, tal qual a figura 2, atuam todas as componentes do tensor de tensões, de modo que, utilizando as relações entre deformação e tensão, das equações 39 e 40, como também a relação entre deformações, equação 41, é possível determinar o efeito de deformação causado por cada componente de tensão, tal que, devido à tensão σ_x, o elemento infinitesimal percebe as deformações

$$\varepsilon_x = \frac{\sigma_x}{E}, \qquad \varepsilon_y = -\nu\frac{\sigma_x}{E}, \qquad \varepsilon_z = -\nu\frac{\sigma_x}{E}. \tag{42}$$

As outras tensões, σ_y e σ_z, se comportam de forma análoga, de modo que

$$\varepsilon_x = -\nu\frac{\sigma_y}{E}, \qquad \varepsilon_y = \frac{\sigma_y}{E}, \qquad \varepsilon_z = -\nu\frac{\sigma_y}{E}, \tag{43}$$

$$\varepsilon_x = -\nu\frac{\sigma_z}{E}, \qquad \varepsilon_y = -\nu\frac{\sigma_z}{E}, \qquad \varepsilon_z = \frac{\sigma_z}{E}. \tag{44}$$

As tensões de cisalhamento, τ_{xy}, τ_{xz} e τ_{yz}, por sua vez, causam as deformações angulares,

$$\gamma_{xy} = \frac{\tau_{xy}}{,} \qquad \gamma_{xz} = \frac{\tau_{xz}}{G}, \qquad \gamma_{yz} = \frac{\tau_{yz}}{G}. \tag{45}$$

Sobrepondo as deformações axiais para cada eixo, e as deformações angulares, é possível determinar as relações entre todas as tensões e todas as deformações, denominada *lei de Hooke generalizada*:

$$\varepsilon_x = \frac{\sigma_x}{E} - \nu\frac{\sigma_y}{E} - \nu\frac{\sigma_z}{E}, \tag{46}$$

$$\varepsilon_y = -\nu\frac{\sigma_x}{E} + \frac{\sigma_y}{E} - \nu\frac{\sigma_z}{E}, \tag{47}$$

$$\varepsilon_z = -\nu\frac{\sigma_x}{E} - \nu\frac{\sigma_y}{E} + \frac{\sigma_z}{E}, \tag{48}$$

$$\gamma_{xy} = \frac{\tau_{xy}}{G}, \tag{49}$$

$$\gamma_{xz} = \frac{\tau_{xz}}{G}, \tag{50}$$

$$\gamma_{yz} = \frac{\tau_{yz}}{G}. \tag{51}$$

O módulo de cisalhamento pode ser determinado em termos da razão de Poisson e do módulo de Young, de modo que[12]

$$G = \frac{E}{2(1+\nu)}. \tag{52}$$

[12] A demonstração dessa relação se utiliza daas fórmulas de rotação dos tensores de deformações e de tensores, que não são tratadas aqui.

Utilizando a notação de Voigt, a Lei de Hooke generalizada pode ser em forma compacta como

$$\begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ \gamma_{xy} \\ \gamma_{xz} \\ \gamma_{yz} \end{Bmatrix} = \frac{1}{E} \begin{bmatrix} 1 & -\nu & -\nu & 0 & 0 & 0 \\ -\nu & 1 & -\nu & 0 & 0 & 0 \\ -\nu & -\nu & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2(1+\nu) & 0 & 0 \\ 0 & 0 & 0 & 0 & 2(1+\nu) & 0 \\ 0 & 0 & 0 & 0 & 0 & 2(1+\nu) \end{bmatrix} \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \tau_{xy} \\ \tau_{xz} \\ \tau_{yz} \end{Bmatrix}, \tag{53}$$

Invertendo esse sistema, obtemos a matriz constitutiva da equação 38,

$$\{\sigma\} = \underbrace{\frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}}_{[\boldsymbol{C}]} \{\varepsilon\} \tag{54}$$

Para que essa matriz exista, é necessário que $-1 < \nu < \frac{1}{2}$.[13]

2.3.5 Estado Plano de Deformação e de Tensão

Quando se analisa um problema plano, é possível simplificar as relações descritas pela matriz $[\boldsymbol{C}]$ em dois casos, observando a rigidez do corpo nas direção transversal ao plano de análise.

O Estado Plano de Tensão (EPT) ocorre quando a rigidez do corpo na direção transversal é muito baixa (como uma chapa ou uma placa), fazendo com que a tensão nessa direção possa ser neglicenciada, ou seja, $\sigma_z = \tau_{xz} = \tau_{yz} = 0$. Nesse caso, a lei de Hooke generalizada se reduz a, [14]

$$\begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix} = \frac{1}{E} \begin{bmatrix} 1 & -\nu & 0 \\ -\nu & 1 & 0 \\ 0 & 0 & (1+\nu) \end{bmatrix} \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix}, \tag{55}$$

$$\varepsilon_z = -\nu\frac{\sigma_x}{E} - \nu\frac{\sigma_y}{E}. \tag{56}$$

[13] Materiais com $\nu = 0.5$ são chamados de incompressíveis, pois não sofrem deformações volumétricas.

[14] Tanto a matriz constitutiva do EPT e quando do EPD são facilmente deduzidas da lei de Hooke generalizada, apenas atribuindo os valores nulos e trabalhando com as inversas da matriz $[\boldsymbol{C}]$.

ε_z passou a ser uma variável dependente, pois é determinada, totalmente, pelas outras deformações. (ZIENKIEWICZ, 2000, pág. 90)

As relações que definem a matriz constitutiva para o estado plano de tensão são

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = \frac{E}{1-\nu^2} \underbrace{\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}}_{[\boldsymbol{C}]} \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_{xy} \end{Bmatrix}. \tag{57}$$

O Estado de Plano de Deformação (EPD), por sua vez, ocorre quando o corpo é muito rígido na direção transversal (como uma barragem ou um muro), fazendo com que a deformação nessa direção possa ser neglicenciada, ou seja, $\varepsilon_z = \gamma_{xz} = \gamma_{yz} = 0$. Nesse caso, a lei de Hooke generalizada se reduz a

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \underbrace{\begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}}_{[\boldsymbol{C}]} \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix}, \tag{58}$$

$$\sigma z = \nu(\sigma_x + \sigma_y) \tag{59}$$

σ_z passou a ser uma variável dependente, pois é determinada, totalmente, pelas outras deformações. (ZIENKIEWICZ, 2000, pág. 91)

2.4 TEORIA DA ENERGIA DE DISTORÇÃO MÁXIMA: A TENSÃO EQUIVALENTE DE VON MISES

A Teoria da Energia de Distorção máxima propõe que:

> o escoamento em um material dúctil ocorre quando a energia de distorção por unidade de volume do material ultrapassa a energia de distorção por unidade de volume do mesmo material quando submetido a escoamento em um ensaio de tração simples. (HIBBELER, 2010)

Essa proposição surge da observação que materiais dúcteis sob carregamentos hidrostáticos exibem uma resistência muito maior que em simples ensaios de tração uniaxial, e que, portanto, o escoamento não é um fenômeno compreendido, simplesmente, pela tração ou compressão, mas, majoritariamente, pela distorção do material. (HIBBELER, 2010)

A energia por unidade de volume, devida à deformação do material, é definida por um produto da tensão pela deformação. Em um elemento orientado sob tensões principais, de acordo com (HIBBELER, 2010)

$$\mathscr{U} = \frac{1}{2}(\sigma_1\varepsilon_1 + \sigma_2\varepsilon_2 + \sigma_3\varepsilon_3), \tag{60}$$

em que $\mathscr{U}$ denota a energia por unidade de volume, σ_i as tensões principais.

Substituindo as deformações pelas tensões, utilizando a lei de Hooke generalizada, equação 51, temos que

$$\mathscr{U} = \frac{1}{2E}\left[\sigma_1^2 + \sigma_2^2 + \sigma_3^2 - 2\nu(\sigma_1\sigma_2 + \sigma_1\sigma_3 + \sigma_2\sigma_3)\right]. \tag{61}$$

Essa energia pode ser compreendida como a soma de duas parcelas: a energia de deformação volumétrica, u_v, e a energia de deformação por cisalhamento, u_d. A energia de deformação volumétrica é a energia de deformação que tende a alterar o volume do elemento, sem distorcer sua geometria, ou seja, a energia de deformação devido à tensão média das tensões principais. Substituindo, então, as tensões principais pela tensão média na equação anterior, temos que

$$\mathscr{U}_v = \frac{3\bar{\sigma}^2}{2E}(1-2\nu), \tag{62}$$

em que $\bar{\sigma} = \frac{1}{3}(\sigma_1 + \sigma_2 + \sigma_3)$ é a tensão média.

A energia de distorção, portanto, é a diferença entre a energia total e a energia volumétrica, de modo que

$$\mathscr{U}_d = \mathscr{U} - \mathscr{U}_v = \frac{1+\nu}{3E}\left[\frac{(\sigma_1-\sigma_2)^2 + (\sigma_2-\sigma_3)^2 + (\sigma_3-\sigma_1)^2}{2}\right]. \tag{63}$$

Fazendo $\sigma_2 = \sigma_3 = 0$, temos a energia de deformação para o caso de tração uniaxial, de modo que

$$\mathscr{U} = \frac{1+\nu}{3}\sigma_e^2, \tag{64}$$

em que σ_e é a tensão admissível do material num ensaio de tração. (HIBBELER, 2010)

Portanto, de acordo com o princípio da energia de distorção máxima, o escoamento ocorre quando a energia de distorção ultrapassa a energia de distorção de um ensaio de tração simples, ou seja, quando (HIBBELER, 2010)

$$\sigma_e \leq \left[\frac{(\sigma_1-\sigma_2)^2 + (\sigma_2-\sigma_3)^2 + (\sigma_3-\sigma_1)^2}{2}\right]^{1/2}. \tag{65}$$

O lado direito dessa inequação pode ser interpretado como uma tensão equivalente ou efetiva do estado de tensão dado pelas tensões principais. De acordo com (HIBBELER, 2010), essa tensão é denominada *tensão equivalente de von Mises*, definida por

$$\sigma_{eq} = \left[\frac{(\sigma_1-\sigma_2)^2 + (\sigma_2-\sigma_3)^2 + (\sigma_3-\sigma_1)^2}{2}\right]^{1/2}. \tag{66}$$

Em termos de um sistema *xyz*, e utilizando todas as componentes do tensor de tensões, a tensão equivalente de von Mises pode ser expressa por

$$\sigma_{eq} = \frac{1}{\sqrt{2}} \left[(\sigma_x - \sigma_y)^2 + (\sigma_y - \sigma_z)^2 + (\sigma_z - \sigma_x)^2 + 6(\tau_{xy}^2 + \tau_{yz}^2 + \tau_{zx}^2) \right]. \tag{67}$$

3 O MÉTODO DOS ELEMENTOS FINITOS

> "As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality."(Albert Einstein)

O método dos elementos finitos (MEF) é um procedimento numérico utilizada para encontrar soluções de equações que modelam a natureza. É amplamente utilizado na simulação de fenômenos físicos, como mecânica dos sólidos, transferência de calor, eletromagnetismo e muitos outros. (OñATE, 2009)

A abordagem do MEF envolve a subdivisão de uma estrutura ou domínio contínuo em pequenos elementos geométricos finitos, definidos por nós, caracterizados por suas propriedades físicas e geométricas, em que são aplicadas as equações governantes do problema, cujo campo é aproximado, dentro de cada um, por meio de funções de interpolação, que vão discretizar o campo sobre os valores nodais. Em seguida, essas equações aproximadas são montadas, justapondo os elementos, em um sistema global, levando em consideração as condições de contorno e as restrições do problema. (QUEK; LIU, 2003)

Em suma, o método de elementos finitos segue o seguinte procedimento, de acordo com Oñate (2009):

1. definição do domínio, e das condições de contorno;
2. discretização do domínio em uma malha formada por nós que constituem os elemento;
3. aplicação das equações de governo sobre cada elemento, formando um sistema local de rigidez;
4. montagem dessas equações em um único grande sistema global;
5. resolução do sistema, encontrando os valores nodais do campo.

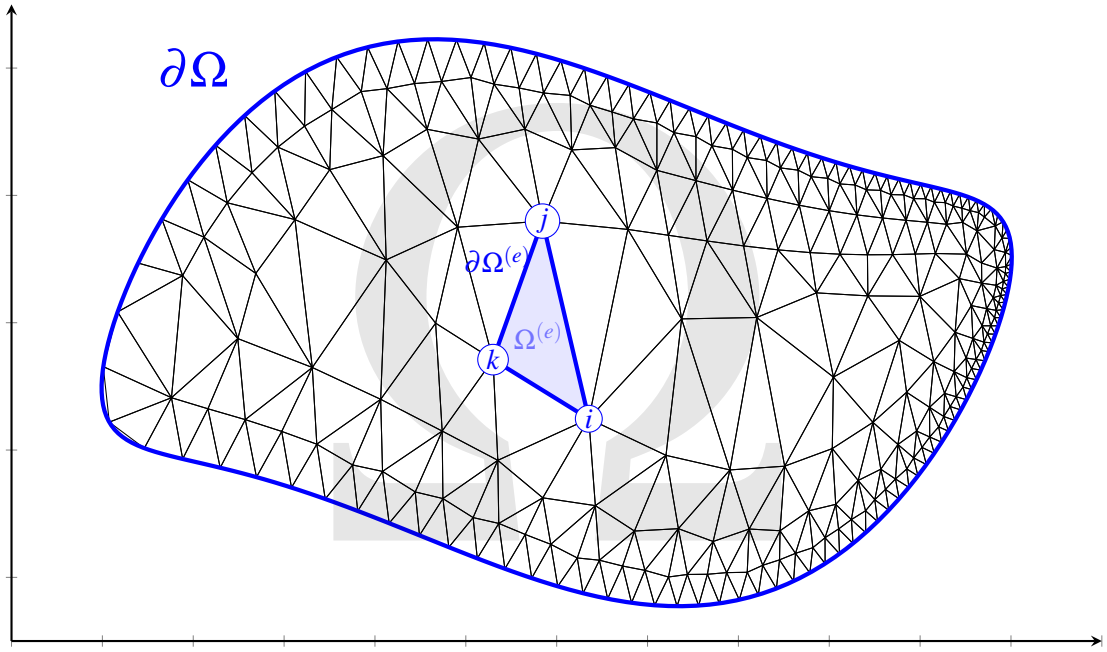
Na análise estrutural aqui abordada, o objetivo da aplicação do MEF é determinar o deslocamento de um sólido em equilíbrio sujeito a carregamentos e restrições, dentro do regime elástico modelado pela Lei de Hooke, utilizando funções de interpolações lineares em elementos triangulares (na análise 2D: EPT ou EPD) e tetraédricos (na análise 3D). Para tanto, resolver um sistema da forma

$$\boldsymbol{KU} = \boldsymbol{F}, \tag{68}$$

em que $\boldsymbol{K}$ é a matriz de rigidez global, $\boldsymbol{U}$ é o vetor de deslocamentos nodais, e $\boldsymbol{F}$ é o vetor de forças nodais.

Deste modo, convém definir os termos e símbolos dessas entidades matemáticas. Seja um corpo $\mathscr{B}$, definido por uma geometria sobre o domínio Ω, que é repartido em pequenos

Figura 6 – Domínio discretizado em elementos triangulares.



Fonte: Elaborado pelo autor (2022).

elementos, de domínio $\Omega^{(e)}$, cuja fronteira se denomina $\partial\Omega^{(e)}$. A figura 6 mostra um domínio bidimensional Ω discretizado em elementos triangulares, cujos nós são representados por i, j e k. Quando uma entidade é definida sobre um elemento, ela é representada por um superescrito (e), como por exemplo o deslocamento $\boldsymbol{\varphi}^{(e)}$ sobre o elemento e.

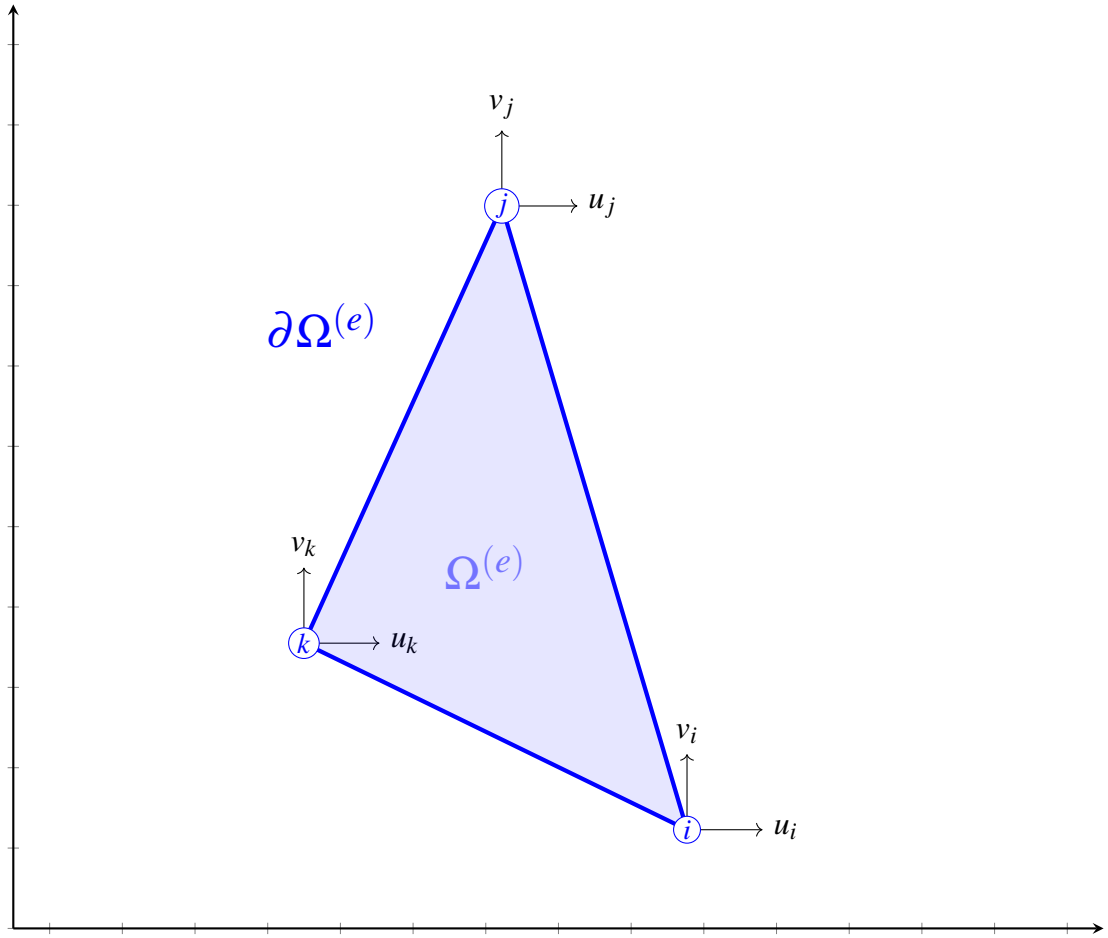
Seguindo Logan (2022), neste capítulo aborda-se como

1. selecionar as funções de interpolação;
2. definir as relações de tensão-deformação-deslocamento;
3. derivar a forma da matriz de rigidez;
4. montar o sistema global e introduzir as condições de contorno.

3.1 AS FUNÇÕES DE INTERPOLAÇÃO

As funções de interpolação são funções matemáticas que aproximam o campo de interesse, neste caso o deslocamento, dentro de um elemento, por meio de uma combinação linear de funções conhecidas, definidas sobre os nós do elemento. Essas funções são definidas, e depois justapostas, de modo que o campo seja contínuo. O objetivo dessa ferramente é alterar os valores incógnitos do campo de contínuos para discretos, para que o deslocamento seja bem definido pelo seu valor sobre a posição cada nó, e que, por sobre o domínio de cada elemento, o campo seja interpolado. Esses valores discretos desconhecidos do campo nos nós, em que o campo tem "liberdade"pra variar, denominam-se *graus de liberdade*. (LOGAN, 2022)

Figura 7 – Um triângulo de deformações constantes (CST).



Fonte: Elaborado pelo autor (2022).

O CST (*Constant Strain Triangle*), por exemplo, tem seis graus de liberdade, uma vez que para cada nó, o vetor de deslocamento tem duas componentes. O tetraedro linear, por sua vez, tem doze graus de liberdade, uma vez que para cada nó, o vetor de deslocamento tem três componentes.

Define-se, então, a função de deslocamento sobre um elemento, representado pela figura 7, assim como a proposta linear de interpolação,

$$\boldsymbol{\varphi}^{(e)} = \begin{Bmatrix} u(x,y) \\ v(x,y) \end{Bmatrix}^{(e)} = \begin{Bmatrix} a_1 + a_2 x + a_3 y \\ b_1 + b_2 x + b_3 y \end{Bmatrix}^{(e)} . \tag{69}$$

No caso bidimensional, $\boldsymbol{\varphi}$ é uma função vetorial de campo que mapeia cada ponto do sólido para seu respectivo deslocamento nos eixos do sistema xy (u e v respectivamente).

Para determinar as constantes a e b, e termos dos deslocamentos nodais basta aplicar a condição de que em cada nó a função deve assumir o valor do deslocamento respectivo. Isto é, analisando somente a componente x, ou seja, $u(x,y)$, temos que

$$u(x_i, y_i) = u_i = a_1 + a_2 x_i + a_3 y_i, \tag{70}$$

$$u(x_j, y_j) = u_j = a_1 + a_2 x_j + a_3 y_j, \tag{71}$$

$$u(x_k, y_k) = u_k = a_1 + a_2 x_k + a_3 y_k, \tag{72}$$

em que u_i é o valor do deslocamento nodais i, assim como x_i e y_i é sua posição sobre o domínio Ω.

Reescrevendo esse sistema na forma matricial, temos que

$$\begin{bmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{Bmatrix} u_i \\ u_j \\ u_k \end{Bmatrix} \implies \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \frac{1}{2A} \begin{bmatrix} \alpha_i & \alpha_j & \alpha_k \\ \beta_i & \beta_j & \beta_k \\ \gamma_i & \gamma_j & \gamma_k \end{bmatrix} \begin{Bmatrix} u_i \\ u_j \\ u_k \end{Bmatrix}, \tag{73}$$

sendo que[1]

$$\frac{1}{2A} \begin{bmatrix} \alpha_i & \alpha_j & \alpha_k \\ \beta_i & \beta_j & \beta_k \\ \gamma_i & \gamma_j & \gamma_k \end{bmatrix} = \begin{bmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{bmatrix}^{-1} = \boldsymbol{X}^{-1}, \tag{74}$$

em que A é área do elemento triangular. [2]

Deste modo, a função de interpolação para $u(x,y)$ pode ser descrita em termos dos deslocamentos nodais na forma, utilizando o sistema 73,

$$u(x,y) = \begin{bmatrix} 1 & x & y \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \begin{bmatrix} 1 & x & y \end{bmatrix} \frac{1}{2A} \begin{bmatrix} \alpha_i & \alpha_j & \alpha_k \\ \beta_i & \beta_j & \beta_k \\ \gamma_i & \gamma_j & \gamma_k \end{bmatrix} \begin{Bmatrix} u_i \\ u_j \\ u_k \end{Bmatrix}. \tag{75}$$

Expandido essas expressões, multiplicando as matrizes e rearranjado os termos, temos que

$$u(x,y) = \frac{1}{2A} \left[\alpha_i + \beta_i x + \gamma_i y\right] u_i + \frac{1}{2A} \left[\alpha_j + \beta_j x + \gamma_j y\right] u_j + \frac{1}{2A} \left[\alpha_k + \beta_k x + \gamma_k y\right] u_k. \tag{76}$$

É possível, também, definir a função de interpolação para a componente $v(x,y)$, de modo análogo, obtendo, apenas substituindo a função u por v nas equações anteriores, que as mesmas relações de interpolações são válidas também para a outra componente de $\boldsymbol{\varphi}$. Então,

$$v(x,y) = \frac{1}{2A} \left[\alpha_i + \beta_i x + \gamma_i y\right] v_i + \frac{1}{2A} \left[\alpha_j + \beta_j x + \gamma_j y\right] v_j + \frac{1}{2A} \left[\alpha_k + \beta_k x + \gamma_k y\right] v_k. \tag{77}$$

[1] É fácil demonstrar que esse sistema sempre é possível e determinado apenas observando o fato de que as posições dos nós são distintas e não colineares, afinal os elementos são triangulares.

[2] Essa forma de escrever a inversa de $\boldsymbol{X}$ é interessante pois simplifica os termos α, β e γ pelo determinante $2A$.

Para simplificar a notação das equações 76 e 77, definem-se as funções de interpolação N da forma

$$N_i = \frac{1}{2A}\left[\alpha_i + \beta_i x + \gamma_i y\right], \tag{78}$$

$$N_j = \frac{1}{2A}\left[\alpha_j + \beta_j x + \gamma_j y\right], \tag{79}$$

$$N_k = \frac{1}{2A}\left[\alpha_k + \beta_k x + \gamma_k y\right]. \tag{80}$$

Por fim, $\boldsymbol{\varphi}$ pode ser reescrito na forma matricial, em termos dessas funções N e dos deslocamentos nodais e u e v, como

$$\boldsymbol{\varphi}^{(e)} = \begin{Bmatrix} u(x,y) \\ v(x,y) \end{Bmatrix}^{(e)} = \begin{bmatrix} N_i & 0 & N_j & 0 & N_k & 0 \\ 0 & N_i & 0 & N_j & 0 & N_k \end{bmatrix} \begin{Bmatrix} u_i \\ v_i \\ u_j \\ v_j \\ u_k \\ v_k \end{Bmatrix} = \boldsymbol{N}\boldsymbol{U}^{(e)}, \tag{81}$$

em que $\boldsymbol{N}$ é a matrix de funções de interpolação, e $\boldsymbol{U}^{(e)}$ é o vetor de deslocamentos nodais do elemento.

3.2 AS RELAÇÕES DE TENSÃO-DEFORMAÇÃO-DESLOCAMENTO

No caso bidimensional, o tensor de deformação, na notação de Voigt, é dado, conforme definido no capítulo anterior,

$$\{\varepsilon\} = \begin{Bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \gamma_{xy} \end{Bmatrix} = \begin{Bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \end{Bmatrix} \tag{82}$$

Aplicando nessas derivadas parciais as funções de interpolação de $\boldsymbol{\varphi}$, das equações 76 e 77, temos que, conforme Logan (2022),

$$\frac{\partial u}{\partial x} = \frac{1}{2A}\left[\beta_i u_i + \beta_j u_j + \beta_k u_k\right], \tag{83}$$

$$\frac{\partial v}{\partial y} = \frac{1}{2A}\left[\gamma_i v_i + \gamma_j v_j + \gamma_k v_k\right], \tag{84}$$

$$\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} = \frac{1}{2A}\left[\gamma_i u_i + \gamma_j u_j + \gamma_k u_k + \beta_i v_i + \beta_j v_j + \beta_k v_k\right]. \tag{85}$$

Utilizando essas equações, é possível reescrever o tensor de deformação $\{\varepsilon\}$ em termos dos coeficientes β e γ, como também do vetor de deslocamentos nodais do elemento $\{U\}^{(e)}$, na forma

$$\{\varepsilon\} = \frac{1}{2A}\begin{bmatrix} \beta_i & 0 & \beta_j & 0 & \beta_k & 0 \\ 0 & \gamma_i & 0 & \gamma_j & 0 & \gamma_k \\ \gamma_i & \beta_i & \gamma_j & \beta_j & \gamma_k & \beta_k \end{bmatrix} \begin{Bmatrix} u_i \\ v_i \\ u_j \\ v_j \\ u_k \\ v_k \end{Bmatrix} = \boldsymbol{B}\boldsymbol{U}^{(e)}. \tag{86}$$

A matrix $\boldsymbol{B}$ é uma função das coordenadas dos nós do elemento, e relaciona o vetor de deslocamentos nodais do elemento $\boldsymbol{U}^{(e)}$ com o tensor de deformações $\{\varepsilon\}$. Como as funções de interpolação são lineares, os coeficientes da matriz $\boldsymbol{B}$ são constantes ao longo do elemento e, consequentemente, as deformação também são constantes. Por causa dessa propriedade, esse elemento é denominado triângulo de deformações constantes, ou CST (*Constant Strain Triangle*).

Por fim, ao passo que a relação entre deformação e tensão é dada pela Lei de Hooke generalizada, na forma da matriz constitutiva $\boldsymbol{C}$ (por conveniência, a matriz constitutiva será expressa sem a notação usual $[C]$), expressar a relação entre tensão e deslocamento, ou tensão-Deformação-Deslocamento, é simplesmente uma questão de multiplicar a matriz constitutiva pela matriz $\boldsymbol{B}$, obtendo assim a relação

$$\{\sigma\} = \boldsymbol{C}\boldsymbol{B}\boldsymbol{U}^{(e)}. \tag{87}$$

3.3 A MATRIZ DE RIGIDEZ LOCAL

A matriz de rigidez local é uma matriz quadrada, simétrica, que relaciona o vetor de forças nodais $\boldsymbol{F}^{(e)}$ com o vetor de deslocamentos nodais $\boldsymbol{U}^{(e)}$, na forma

$$\boldsymbol{F}^{(e)} = \boldsymbol{K}^{(e)}\boldsymbol{U}^{(e)}, \tag{88}$$

cuja derivação é feita aplicando a equação de governo, o equilíbrio estático nesse caso, sobre o elemento, utilizando as funções de interpolação para o campo de deslocamentos. (LOGAN, 2022)

$\boldsymbol{F}^{(e)}$ é um vetor que armazena as forças aplicadas sobre os nós do elemento, analogamente a $\boldsymbol{U}^{(e)}$ que armazena os deslocamentos nodais, e tem a forma

$$\boldsymbol{F}^{(e)} = \begin{Bmatrix} fx_i \\ fy_j \\ fx_j \\ fy_j \\ fx_k \\ fy_k \end{Bmatrix}. \tag{89}$$

Um dos métodos para derivar essa matriz é utilizar o *Princípio dos Trabalhos Virtuais*, que pode ser enunciado como:

> Se um corpo deformável em equilíbrio é submetido a deslocamentos virtuais arbitrários (imaginários) associados a uma deformação compatível do corpo, o trabalho virtual das forças externas no corpo é igual à energia virtual de deformação das tensões internas. (LOGAN, 2022, pág. 876, tradução livre)

Nesse contexto, o trabalho virtual interno é causado por uma deformação do próprio corpo, denotada $\delta\boldsymbol{\varepsilon}$, devida a uma deformação virtual $\delta\boldsymbol{u}$, enquanto o trabalho virtual externo é causado diretamente pelas forças que atuam sobre as fronteiras do corpo. Em um corpo elástico, modelado pela Lei de Hooke, o trabalho virtual interno pode ser descrito em termos do deslocamento virtual, como a energia armazenada na forma elástica, a fim de desenvolver as equações de equilíbrio estático apresentadas a seguir. [3]

Seja um elemento triangular, como da figura 8, que sofre um deslocamento virtual $\delta\boldsymbol{U}^{(e)}$. De acordo com Zienkiewicz (2000), o trabalho interno do elemento é dado por[4]

$$\delta\mathscr{U} = \int_{\Omega^{(e)}} \delta\{\varepsilon\}^t\{\sigma\}dV, \tag{90}$$

Utilizando as relações de tensão-deformação-deslocamento, descritas nas equações 86 e 87, temos que[5]

$$\delta\mathscr{U} = \int_{\Omega^{(e)}} (\boldsymbol{B}\delta\boldsymbol{U}^{(e)})^t\boldsymbol{C}\boldsymbol{B}\boldsymbol{U}^{(e)}dV \implies \mathscr{U} = \delta(\boldsymbol{U}^{(e)})^t \int_{\Omega^{(e)}} \boldsymbol{B}^t\boldsymbol{C}\boldsymbol{B}dV\boldsymbol{U}^{(e)}. \tag{91}$$

O trabalho desenvolvido pelas forças externas é dado, também de acordo com Zienkiewicz (2000), pelo produto dos deslocamentos nodais e das forças nodais, na forma

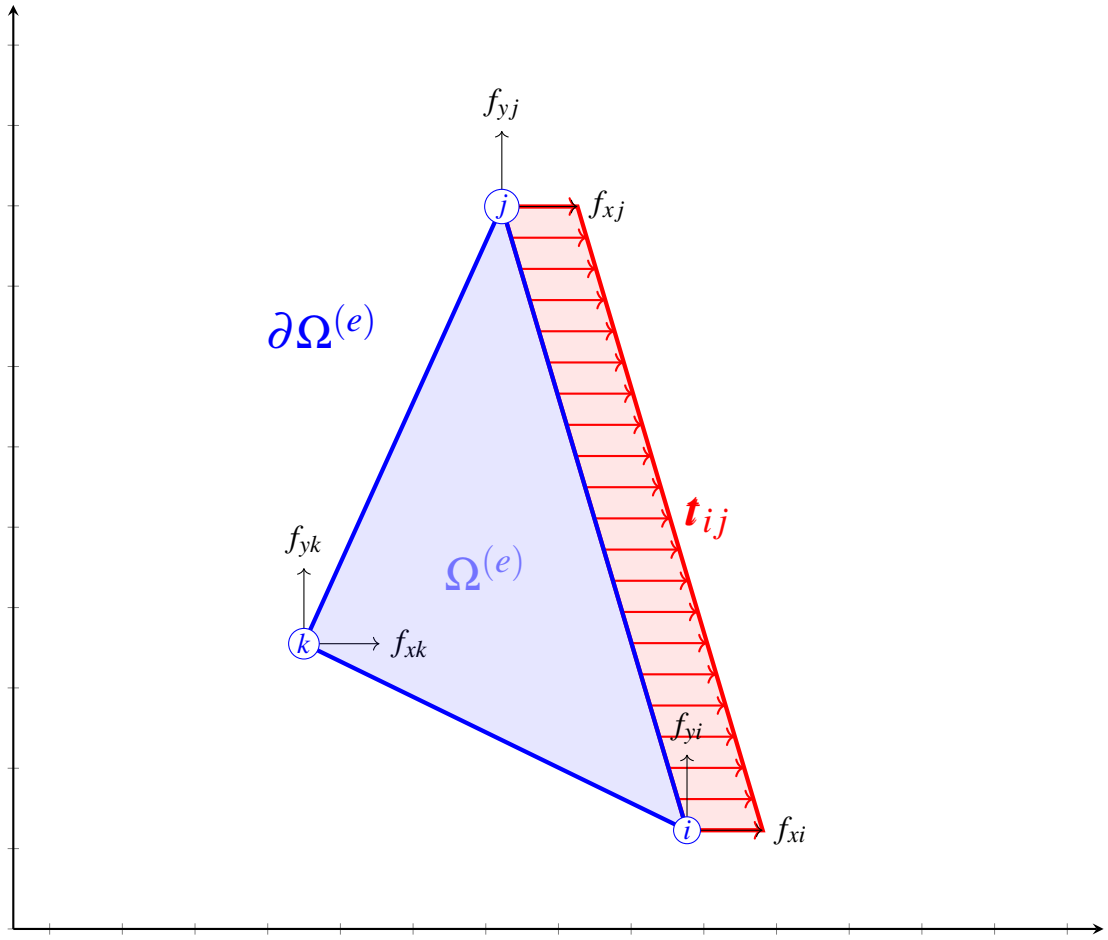
$$\delta\mathscr{W} = \delta(\boldsymbol{U}^{(e)})^t\boldsymbol{F}^{(e)}. \tag{92}$$

[3] A notação δ se refere a um variacional, que representa uma variação infinitesimal sobre o todo o contínuo da função, de modo que seja nulo na região em que são aplicadas as condições de contorno. O cálculo variacional, entretanto, não é abordado diretamente nesta monografia.

[4] Poder escrever a energia interna desta forma simples é devido à escolha de que, na notação de Voigt do tensor de deformação, foi utilizado a convenção de $\gamma = 2\varepsilon$.

[5] Utilizando também a propriedade de transposição do produto de matrizes, $(\boldsymbol{AB})^t = \boldsymbol{B}^t\boldsymbol{A}^t$.

Figura 8 – Um triângulo de deformações constantes sob carregamentos (CST) sob carregamentos uniformes.



Fonte: Elaborado pelo autor (2022).

Igualando as duas parcelas de trabalho, temos que

$$\delta\mathscr{U} = \delta\mathscr{W} \implies \delta(\boldsymbol{U}^{(e)})^t \int_{\Omega^{(e)}} \boldsymbol{B}^t\boldsymbol{C}\boldsymbol{B}dV\boldsymbol{U}^{(e)} = \delta(\boldsymbol{U}^{(e)})^t\boldsymbol{F}^{(e)}. \tag{93}$$

Portanto,

$$\boldsymbol{F}^{(e)} = \int_{\Omega^{(e)}} \boldsymbol{B}^t\boldsymbol{C}\boldsymbol{B}dV\boldsymbol{U}^{(e)}. \tag{94}$$

Comparando com a forma da equação 68, fica evidente que a matriz de rigidez local é dada por

$$\boldsymbol{K}^{(e)} = \int_{\Omega^{(e)}} \boldsymbol{B}^t\boldsymbol{C}\boldsymbol{B}dV. \tag{95}$$

O elemento tratado aqui é o CST, portanto, a matriz $\boldsymbol{B}$ é constante em todo o domínio $\Omega^{(e)}$, como também é constante a matriz constitutiva $\boldsymbol{C}$. A integral, portanto, não precisa ser

computada em um sistema local, pois todos os termos que a compõem são constantes. Logo, a matriz de rigidez local pode ser simplificada ainda mais, tornando-se, de acordo com Logan (2022)

$$\boldsymbol{K}^{(e)} = \boldsymbol{B}^t \boldsymbol{C} \boldsymbol{B} \int_{\Omega^{(e)}} dV = \boldsymbol{B}^t \boldsymbol{C} \boldsymbol{B} V, \tag{96}$$

em que V é o volume do elemento, definido, no caso bidimensional, por $V = Ad$, em que d é a espessura.

Outra forma de derivar essa relação diretamente da equação de equilíbrio estático é por meio do método de Galerkin, em que as bases da função de interpolação são as mesmas da função de ponderação, utilizando a chamada formulação fraca da equação de governo. Esse método leva à mesma formulação mostrada acima.

Vale ressaltar que não foi preciso definir um sistema local de coordenadas para derivar essas relações, uma vez que o elemento tradado aqui, como também o tetraedro (tratado mais adiante), tem funções de interpolação simples o suficiente para que um sistema orientado a cada elemento não seja necessário. Elementos mais sofisticados necessitam, por uma questão de manipulação algébrica, de um sistema de referência local. Esses elementos não são tratados aqui.

3.4 MATRIZ DE RIGIDEZ GLOBAL E CONDIÇÕES DE CONTORNO

Para cada elemento do domínio é realizado o procedimento descrito na seção anterior, de encontrar a matriz de rigidez local que descreve o equilíbrio do elemento em termos das forças externas e do campo de deslocamento sobre os nós. Para resolver o problema, encontrar os deslocamentos de todos os nós, é necessário justapor essas matrizes locais em uma matriz global, ou seja,

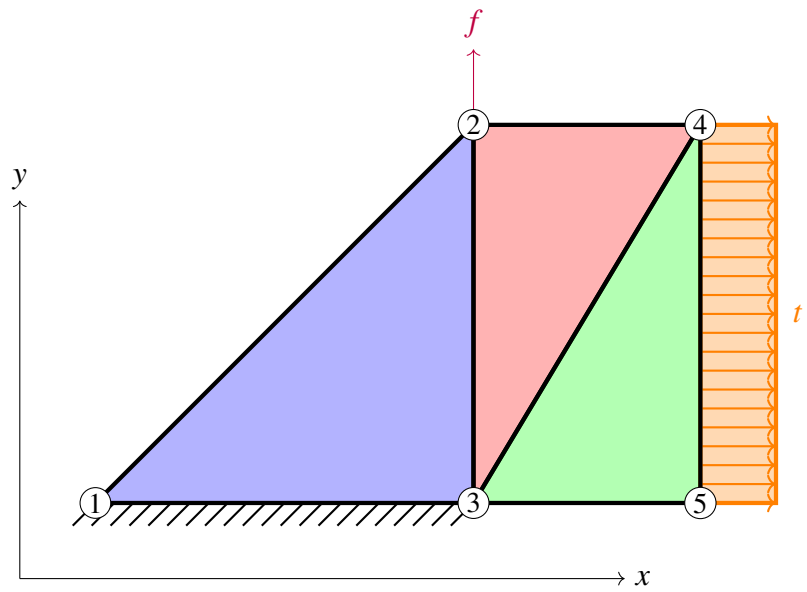
$$\boldsymbol{K} = \sum_e \boldsymbol{K}^{(e)}. \tag{97}$$

Nesse contexto, Σ representa, não uma soma ordinária, mas sim a sobreposição dos efeitos de rigidez sobre os nós. (LOGAN, 2022)

Em cada elemento, a matriz local de rigidez $\boldsymbol{K}^{(e)}$ define a interação de forças e deslocamentos entre os graus de liberdade dos nós. Analisando a estrutura como um todo, cada nó pode pertencer a vários elementos, pois é assim que a malha é constituída. O efeito de rigidez, então, sobre cada nó é a soma dos efeitos de todos os elementos que o contém. Isso vale para os deslocamentos nodais, como para as forças nodais. O procedimento, portanto, de montagem da matriz global de rigidez é a soma das matrizes locais de rigidez sobre as posições dos graus de liberdade respe ctivos das matrizes locais (mapeamento local-global).

Seja um sólido $\mathscr{B}$ discretizado por uma malha composta de cinco nós, formando três elementos, conforme a figura 9, engastado na superfície inferior do elemento azulado e sujeito

Figura 9 – Uma estrutura discretizada em elementos CST.



Fonte: Elaborado pelo autor (2022).

a uma força f concentrada no nó 2, como também um carregamento distribuído t na superfície entre os nós 4 e 5.

As matrizes locais de rigidez desses elementos são quadradas 6×6, pois no CST existem três graus de liberdade, dois para cada nó, referente ao deslocamento nas direções de x e y. Os sistemas locas, então, para elemento da figura 9, têm a forma

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} \\ k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} \\ k_{31} & k_{32} & k_{33} & k_{34} & k_{35} & k_{36} \\ k_{41} & k_{42} & k_{43} & k_{44} & k_{45} & k_{46} \\ k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} \\ k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66} \end{bmatrix}^{(1)} \begin{Bmatrix} u_1 \\ v_1 \\ u_3 \\ v_3 \\ u_2 \\ v_2 \end{Bmatrix} = \begin{Bmatrix} f_{x1} \\ f_{y1} \\ f_{x2} \\ f_{y2} \\ f_{x3} \\ f_{y3} \end{Bmatrix} \iff \boldsymbol{K}^{(1)}\boldsymbol{U}^{(1)} = \boldsymbol{F}^{(1)}, \tag{98}$$

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} \\ k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} \\ k_{31} & k_{32} & k_{33} & k_{34} & k_{35} & k_{36} \\ k_{41} & k_{42} & k_{43} & k_{44} & k_{45} & k_{46} \\ k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} \\ k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66} \end{bmatrix}^{(2)} \begin{Bmatrix} u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \end{Bmatrix} = \begin{Bmatrix} f_{x2} \\ f_{y2} \\ f_{x3} \\ f_{y3} \\ f_{x4} \\ f_{y4} \end{Bmatrix} \iff \boldsymbol{K}^{(2)}\boldsymbol{U}^{(2)} = \boldsymbol{F}^{(2)}, \tag{99}$$

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} \\ k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} \\ k_{31} & k_{32} & k_{33} & k_{34} & k_{35} & k_{36} \\ k_{41} & k_{42} & k_{43} & k_{44} & k_{45} & k_{46} \\ k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} \\ k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66} \end{bmatrix}^{(3)} \begin{Bmatrix} u_3 \\ v_3 \\ u_5 \\ v_5 \\ u_4 \\ v_4 \end{Bmatrix} = \begin{Bmatrix} f_{x3} \\ f_{y3} \\ f_{x5} \\ f_{y5} \\ f_{x4} \\ f_{y4} \end{Bmatrix} \iff \boldsymbol{K}^{(3)}\boldsymbol{U}^{(3)} = \boldsymbol{F}^{(3)}. \tag{100}$$

A montagem do sistema global é somar essas matrizes de rigidez sobre os graus de liberdade. Observemos que o primeiro nó só faz parte de um elemento, o azul, portanto sua rigidez só tem contribuição desse elemento; o nó 3, por sua vez, pertence aos três elementos (azul, vermelho e verde), e, portanto, seu termo de rigidez tem contribuição de todos eles. Realizando esse procedimento, é possível montar a matriz global de rigidez, utilizando, para tanto, a notação dos vetores de deslocamentos nodais $\boldsymbol{U}$ e de forças nodais $\boldsymbol{F}$.

A matriz de rigidez global é quadrada 10×10, pois existem cinco nós, com dois graus de liberdade cada, e é dada por

$$
\begin{bmatrix}
k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} & 0 & 0 & 0 & 0 \\
k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} & 0 & 0 & 0 & 0 \\
k_{31} & k_{32} & k_{33}+k_{11} & k_{34}+k_{12} & k_{35}+k_{13} & k_{36}+k_{14} & k_{15} & k_{16} & 0 & 0 \\
k_{41} & k_{42} & k_{43}+k_{21} & k_{44}+k_{22} & k_{45}+k_{23} & k_{46}+k_{24} & k_{25} & k_{26} & 0 & 0 \\
k_{51} & k_{52} & k_{53}+k_{31} & k_{54}+k_{32} & k_{55}+k_{33}+k_{11} & k_{56}+k_{34}+k_{12} & k_{35}+k_{13} & k_{36}+k_{14} & k_{15} & k_{16} \\
k_{61} & k_{62} & k_{63}+k_{41} & k_{64}+k_{42} & k_{65}+k_{43}+k_{21} & k_{66}+k_{44}+k_{22} & k_{45}+k_{23} & k_{46}+k_{24} & k_{25} & k_{26} \\
0 & 0 & k_{51} & k_{52} & k_{53}+k_{31} & k_{54}+k_{32} & k_{55}+k_{33} & k_{56}+k_{34} & k_{35} & k_{36} \\
0 & 0 & k_{61} & k_{62} & k_{63}+k_{41} & k_{64}+k_{42} & k_{65}+k_{43} & k_{66}+k_{44} & k_{45} & k_{46} \\
0 & 0 & 0 & 0 & k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} \\
0 & 0 & 0 & 0 & k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66}
\end{bmatrix}
\begin{Bmatrix}
u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \\ u_5 \\ v_5
\end{Bmatrix}
=
\begin{Bmatrix}
R_{x1} \\ R_{y1} \\ 0 \\ f \\ R_{x3} \\ R_{y3} \\ \frac{1}{2}\ell_{4-5}t \\ 0 \\ \frac{1}{2}\ell_{4-5}t \\ 0
\end{Bmatrix}. \tag{101}
$$

Nas duas primeiras linhas do sistema, que representam as forças nodais sobre o primeiro nó, é possível notar que somente os deslocamentos do elemento azulado que impactam diretamente. Nos graus de u_3 e v_3, fica evidente que todos os elementos contribuem para as forças nodais do nó 3, conforme descrito anteriormente.

Um outro passo nessa montagem do sistema global é a aplicação das condições de contorno. As condições de contorno, tradas aqui, podem ser divididas em dois tipos:

1. de Dirichlet, ou de primeiro tipo, e
2. de Neumann, ou de segundo tipo.

As condições de Dirichlet são aquelas que definem valores da variável incógnita na fronteira do domínio. Já as de Neumann, são aquelas que definem valores de contorno sobre a derivada da variável incógnita nas fronteiras do domínio (MUFTU, 2022). Aqui, as condições de contorno de Dirichlet são restrições de deslocamento, enquanto as condições de Neumann são trações aplicados sobre a estrutura.

As condições de contorno sobre os deslocamentos, então, dividem os graus de liberdade entre

1. livre; Quando não há informação prévia de seus valores, isto é, o campo pode variar livremente.
2. prescritos; prescritos quando há informação prévia de seus valores, e portanto, não podem variar. (LOGAN, 2022)

Um engaste, por exemplo, gera uma condições de contorno de Dirichlet, definindo graus de liberdade prescritos, pois impõe que o deslocamento naquela região do sólido é nulo. Um deslocamento conhecido, também o faz, determinando que os graus correspondentes recebam o valor deslocado.

O vetor de forças $\boldsymbol{F}$ é nulo em regra sobre os graus de liberdade livres, pois essa entidade representa as forças externas sobre o elemento. Entretanto, como no MEF os deslocamentos são discretizados nos nós, o mesmo é feito com os carregamentos. Quando se aplicam condições de contorno de Neumann, os graus correspondentes devem receber forças nodais equivalentes, de forma que representem os carregamentos por forças externas concentradas nos nó. Pode-se encontrar as forças nodais equivalentes utilizando o mesmo método da seção anterior: o princípio dos trabalhos virtuais, fazendo com que o trabalho de carregamentos sobre a fronteira de um elemento $\partial\Omega^{(e)}$, causado por um deslocamento virtual $\delta\boldsymbol{U}^{(e)}$, seja igual ao trabalho da forças nodais equivalentes. Em termos matemáticos, de acordo com Zienkiewicz (2000),

$$\delta(\boldsymbol{U}^{(e)})^t\boldsymbol{F}^{(e)} = \int_{\partial\Omega^{(e)}} (\boldsymbol{N}\delta\boldsymbol{U}^{(e)})^t\boldsymbol{t}d\ell, \tag{102}$$

em que $\boldsymbol{t}$ é o vetor do carregamento sobre a fronteira do elemento, na forma

$$\boldsymbol{t} = \begin{Bmatrix} t_x(x,y) \\ t_y(x,y), \end{Bmatrix}. \tag{103}$$

e $d\ell$ é o infinitesimal de área da fronteira do elemento.

Aqui os carregamentos são constantes sobre as fronteiras dos elementos, isto é, são carregamentos distribuídos e uniformes. Deste modo, a equação anterior pode ser simplificada para

$$\delta(\boldsymbol{U}^{(e)})^t \boldsymbol{F}^{(e)} = \delta \boldsymbol{U}^{(e)} \int_{\partial\Omega^{(e)}} \boldsymbol{N}^t d\ell \boldsymbol{t}. \tag{104}$$

Portanto, as forças nodais equivalentes, devidas a carregamentos uniformes sobre a fronteira do elemento, são dadas por

$$\boldsymbol{F}^{(e)} = \int_{\partial\Omega^{(e)}} \boldsymbol{N}^t dl \boldsymbol{t}. \tag{105}$$

No elemento CST, pela simplicidade das funções de interpolação, é possível realizar essa integração analiticamente. A solução mostra que o carregamento uniforme sobre a fronteira do elemento é equivalente à distribuição dessa carga duas forças nodais iguais, agindo nas extremidades da face em que atua o carregamento, ou seja, de acordo com (OñATE, 2009),

$$\boldsymbol{F}_{\boldsymbol{t}}^{(e)} = \frac{1}{2} l^{(e)} \boldsymbol{t} \tag{106}$$

em que $\boldsymbol{F}_{\boldsymbol{t}}^{(e)}$ são as forças equivalentes que compreendem a fronteira em que o carregamento $\boldsymbol{t}$ é aplicado, $l^{(e)}$ é o comprimento da fronteira do elemento, e d é a espessura do elemento.

Na equação 101, em que foi montado o sistema global, já foram aplicada essas condições de contorno de carregamentos. A força roxeada f, agindo sobre o nó 2, faz-se presente diretamente no vetor de forças nodais, pois já é uma força concentrada em um nó. O carregamento alaranjado t, por sua vez, foi decomposto em duas parcelas, para os nós 5 e 4, seguindo a expressão da equação anterior.

Vale ressaltar que nos graus de liberdade prescritos as forças nodais desconhecidas são as reações das estruturas, isto é, as forças de reação que as restrições fazem sobre o corpo que se mantenha em equilíbrio. Por conta disso, na equação 101, as forças sobre os graus prescritos foram denominadas R, de reação.

3.4.1 Solução direta do sistema global

Com a determinação da matriz de rigidez global, as condições de contorno devidamente expressas em graus prescritos e livres (sejam deslocamento ou carregamentos), é possível reescrever o sistema geral da equação 68 na forma, de acordo com Rao (2018),

$$\begin{bmatrix} \boldsymbol{K}_{LL} & \boldsymbol{K}_{LP} \\ \boldsymbol{K}_{PL} & \boldsymbol{K}_{PP} \end{bmatrix} \begin{Bmatrix} \boldsymbol{U}_L \\ \boldsymbol{U}_P \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_L \\ \boldsymbol{F}_P \end{Bmatrix}, \tag{107}$$

Nessa expressão, os graus de liberdade subdividem os vetores $\boldsymbol{U}$ e $\boldsymbol{F}$ em dois cada um, em prescritos ($\boldsymbol{U}_P$ e $\boldsymbol{F}_P$), e livres ($\boldsymbol{U}_L$ e $\boldsymbol{F}_L$). A matriz de rigidez, então, é subdividida em quatro, que relacionam os graus de liberdade prescritos e livros dos deslocamentos e das forças nodais, respectivamente.

Portanto, o sistema global pode ser reescrito mais uma vez em termos dessas submatrizes, como

$$\boldsymbol{K}_{LL}\boldsymbol{U}_L + \boldsymbol{K}_{LP}\boldsymbol{U}_P = \boldsymbol{F}_L \tag{108}$$

$$\boldsymbol{K}_{PL}\boldsymbol{U}_L + \boldsymbol{K}_{PP}\boldsymbol{U}_P = \boldsymbol{F}_P. \tag{109}$$

Os valores conhecidos desse sistema são os deslocamentos sobre os graus prescritos $\boldsymbol{U}_P$, e as forças externas sobre o graus livres $\boldsymbol{F}_L$, afinal, são as próprias condições de contorno definidas matematicamente pelo MEF. Logo, como o objetivo de resolver o sistema é encontrar os deslocamentos nodais, é possível isolar $\boldsymbol{U}_L$ na primeira equação, ou seja,

$$\boldsymbol{U}_L = \boldsymbol{K}_{LL}^{-1}(\boldsymbol{F}_L - \boldsymbol{K}_{LP}\boldsymbol{U}_P). \tag{110}$$

A matriz $\boldsymbol{K}_{LL}$ é quadrada, simétrica e sempre possui inversa quando o problema está estaticamente determinado (RAO, 2018).

Desse sistema, também, é possível determinar as forças nodais sobre os graus de liberdade prescritos, isto é, as reações da estrutura, apenas aplicando a equação 109, após o cálculo dos deslocamentos nodais livres.

Aplicando essas expressões à equação 101, referente ao exemplo da figura 9, temos que

$$\begin{bmatrix} k_{33}+k_{11} & k_{34}+k_s12 & k_{15} & k_{16} & 0 & 0 \\ k_{43}+k_{21} & k_{44}+k_{22} & k_{25} & k_{26} & 0 & 0 \\ k_{51} & k_{52} & k_{55}+k_{33} & k_{56}+k_{34} & k_{35} & k_{36} \\ k_{61} & k_{62} & k_{65}+k_{43} & k_{66}+k_{44} & k_{45} & k_{46} \\ 0 & 0 & k_{53} & k_{54} & k_{55} & k_{56} \\ 0 & 0 & k_{63} & k_{64} & k_{65} & k_{66} \end{bmatrix} \begin{Bmatrix} u_2 \\ v_2 \\ u_4 \\ v_4 \\ u_5 \\ v_5 \end{Bmatrix} = \begin{Bmatrix} 0 \\ f \\ \frac{1}{2}\ell_{4-5}t \\ 0 \\ \frac{1}{2}\ell_{4-5}t \\ 0 \end{Bmatrix} + \begin{bmatrix} k_{31} & k_{32} & k_{35}+k_{13} & k_{36}+k_{14} \\ k_{41} & k_{42} & k_{45}+k_{23} & k_{46}+k_{24} \\ 0 & 0 & k_{53}+k_{31} & k_{54}+k_{32} \\ 0 & 0 & k_{63}+k_{41} & k_{64}+k_{42} \\ 0 & 0 & k_{51} & k_{52} \\ 0 & 0 & k_{61} & k_{62} \end{bmatrix} \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}$$

$$\begin{bmatrix} k_{13} & k_{14} & 0 & 0 & 0 & 0 \\ k_{23} & k_{24} & 0 & 0 & 0 & 0 \\ k_{53}+k_{31} & k_{54}+k_{32} & k_{35}+k_{13} & k_{36}+k_{14} & k_{15} & k_{16} \\ k_{63}+k_{41} & k_{64}+k_{42} & k_{45}+k_{23} & k_{46}+k_{24} & k_{25} & k_{26} \end{bmatrix} \begin{Bmatrix} u_2 \\ v_2 \\ u_4 \\ v_4 \\ u_5 \\ v_5 \end{Bmatrix} = \begin{Bmatrix} R_{x1} \\ R_{y1} \\ R_{x3} \\ R_{y3} \end{Bmatrix} + \begin{bmatrix} k_{11} & k_{12} & k_{15} & k_{16} \\ k_{21} & k_{22} & k_{25} & k_{26} \\ k_{51} & k_{52} & k_{55}+k_{33}+k_{11} & k_{56}+k_{34}+k_{12} \\ k_{61} & k_{62} & k_{65}+k_{43}+k_{21} & k_{66}+k_{44}+k_{22} \end{bmatrix} \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}$$

3.5 EXPRESSÕES PARA O TETRAEDRO LINEAR

As mesmas relações gerais da seção anterior podem ser utilizadas para derivar as expressões para o elemento tetraédrico linear (figura 10). A grande diferença é a passagem de uma modelagem bidimensional para uma tridimensional, o que implica em um aumento do número de graus de liberdade, e o abandono de EPT e EPD, como também a introdução de um nova condição de contorno sobre superfícies propriamente. De modo similar ao CST, o tetraedro é definido por três nós, nomeados i, j, k e m, sobre os quais tanto o deslocamento é discretizado.

3.5.1 As funções de interpolação

A primeira grande diferença é que agora a função de deslocamento $\boldsymbol{\varphi}^{(e)}$, tem três componentes, na forma

$$\boldsymbol{\varphi}^{(e)} = \begin{Bmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{Bmatrix}^{(e)} = \begin{Bmatrix} a_1 + a_2 x + a_3 y + a_4 z \\ b_1 + b_2 x + b_3 y + b_4 z \\ c_1 + c_2 x + c_3 y + c_4 z \end{Bmatrix}^{(} e). \tag{111}$$

em que u, v e w são as componentes do deslocamento nos eixos x, y e z, respectivamente.

A proposta de interpolação desses valores continua sendo linear, isto é, a função de deslocamento é interpolada por um polinômio de primeiro grau, que é definido pelos valores do campo nos nós do elemento. No caso do tetraedro linear, há quatro nós, e portanto, quatro valores de deslocamento. Para descrever esses coeficientes em termos do deslocamento discretizado nos nó, basta seguir o mesmo procedimento da seção anterior, e montar um sistema de equações, na forma

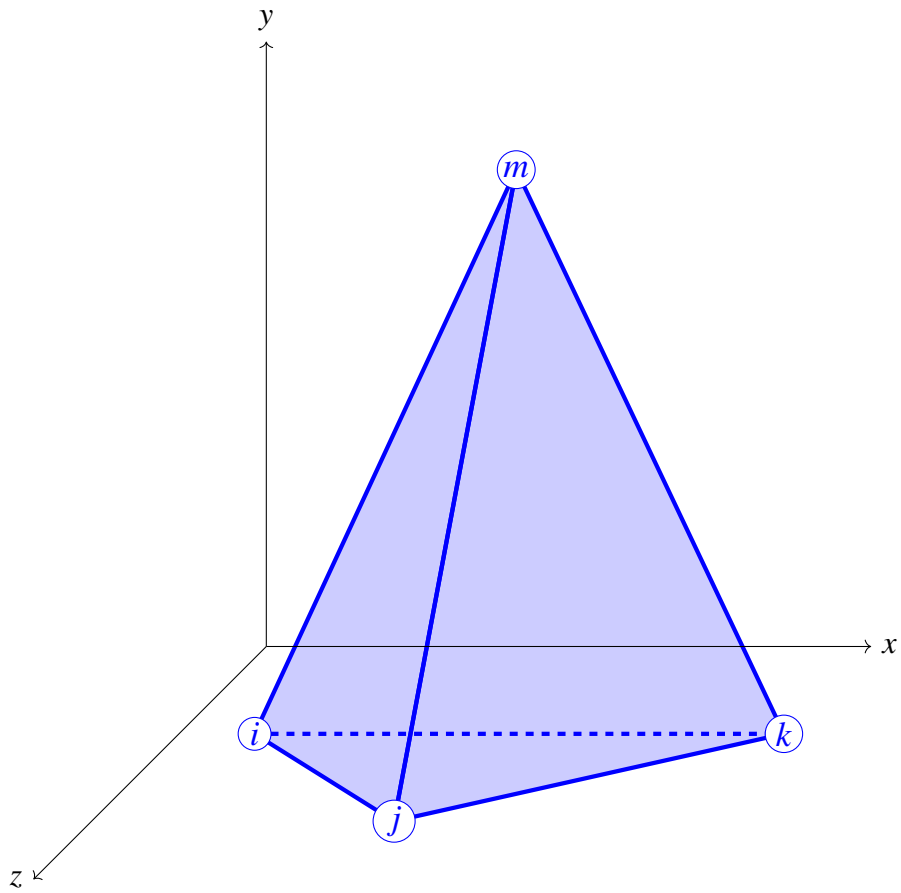
$$\begin{bmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{Bmatrix} = \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{Bmatrix}. \tag{112}$$

Logo, os coeficientes são expressos por

$$\begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{Bmatrix} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \\ \beta_1 & \beta_2 & \beta_3 & \beta_4 \\ \gamma_1 & \gamma_2 & \gamma_3 & \gamma_4 \\ \delta_1 & \delta_2 & \delta_3 & \delta_4 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{Bmatrix}. \tag{113}$$

Portanto,

Figura 10 – Elemento tetraédrico



Fonte: Elaborado pelo autor (().2022)

$$u(x,y,z) = \begin{bmatrix} 1 & x & y & z \end{bmatrix} \frac{1}{6V} \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \\ \beta_1 & \beta_2 & \beta_3 & \beta_4 \\ \gamma_1 & \gamma_2 & \gamma_3 & \gamma_4 \\ \delta_1 & \delta_2 & \delta_3 & \delta_4 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{Bmatrix}. \tag{114}$$

em que V é o volume do tetraedro.

Expandindo os termos, temos que

$$\begin{aligned} u(x,y,z) = &\frac{1}{6V}(\alpha_1 + \beta_1 x + \gamma_1 y + \delta_1 z)u_1 + \frac{1}{6V}(\alpha_2 + \beta_2 x + \gamma_2 y + \delta_2 z)u_2 + \\ &\frac{1}{6V}(\alpha_3 + \beta_3 x + \gamma_3 y + \delta_3 z)u_3 + \frac{1}{6V}(\alpha_4 + \beta_4 x + \gamma_4 y + \delta_4 z)u_4. \end{aligned} \tag{115}$$

Desta forma, as funções de interpolação N têm a forma

$$N_i = \frac{1}{6V}(\alpha_1 + \beta_1 x + \gamma_1 y + \delta_1 z). \tag{116}$$

Por fim, a forma discretizada da função de deslocamento $\boldsymbol{\varphi}^{(e)}$, em termos dos deslocamentos nodais, é

$$\boldsymbol{\varphi}^{(e)} = \begin{Bmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{Bmatrix}^{(e)} = \begin{bmatrix} N_i & 0 & 0 & N_j & 0 & 0 & N_k & 0 & 0 & N_m & 0 & 0 \\ 0 & N_i & 0 & 0 & N_j & 0 & 0 & N_k & 0 & 0 & N_m & 0 \\ 0 & 0 & N_i & 0 & 0 & N_j & 0 & 0 & N_k & 0 & 0 & N_m \end{bmatrix} \begin{Bmatrix} u_i \\ v_i \\ w_i \\ u_j \\ v_j \\ w_j \\ u_k \\ v_k \\ w_k \\ u_m \\ v_m \\ w_m \end{Bmatrix}. \tag{117}$$

3.5.2 As Relações de Tensão-Deformação-Deslocamento

Aplicando as definições do tensor de deformações na notação de Voigt, temos que

$$\varepsilon_{xx} = \frac{\partial u}{\partial x} = \frac{1}{6V}[\beta_i u_i + \beta_j u_j + \beta_k u_k + \beta_m u_m] \tag{118}$$

$$\varepsilon_{yy} = \frac{\partial v}{\partial y} = \frac{1}{6V}[\gamma_i v_i + \gamma_j v_j + \gamma_k v_k + \gamma_m v_m] \tag{119}$$

$$\varepsilon_{zz} = \frac{\partial w}{\partial z} = \frac{1}{6V}[\delta_i w_i + \delta_j w_j + \delta_k w_k + \delta_m w_m] \tag{120}$$

$$\gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} = \frac{1}{6V}[\beta_i v_i + \beta_j v_j + \beta_k v_k + \beta_m v_m + \gamma_i u_i + \gamma_j u_j + \gamma_k u_k + \gamma_m u_m] \tag{121}$$

$$\gamma_{xz} = \frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} = \frac{1}{6V}[\delta_i u_i + \delta_j u_j + \delta_k u_k + \delta_m u_m + \beta_i w_i + \beta_j w_j + \beta_k w_k + \beta_m w_m] \tag{122}$$

$$\gamma_{yz} = \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} = \frac{1}{6V}[\gamma_i w_i + \gamma_j w_j + \gamma_k w_k + \gamma_m w_m + \delta_i v_i + \delta_j v_j + \delta_k v_k + \delta_m v_m]. \tag{123}$$

Portanto, a relação entre deformação e deslocamento é dada por

$$\begin{Bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ \gamma_{xy} \\ \gamma_{xz} \\ \gamma_{yz} \end{Bmatrix} = \frac{1}{6V} \begin{bmatrix} \beta_i & 0 & 0 & \beta_j & 0 & 0 & \beta_k & 0 & 0 & \beta_m & 0 & 0 \\ 0 & \gamma_i & 0 & 0 & \gamma_j & 0 & 0 & \gamma_k & 0 & 0 & \gamma_m & 0 \\ 0 & 0 & \delta_i & 0 & 0 & \delta_j & 0 & 0 & \delta_k & 0 & 0 & \delta_m \\ \gamma_i & \beta_i & 0 & \gamma_j & \beta_j & 0 & \gamma_k & \beta_k & 0 & \gamma_m & \beta_m & 0 \\ \delta_i & 0 & \beta_i & \delta_j & 0 & \beta_j & \delta_k & 0 & \beta_k & \delta_m & 0 & \beta_m \\ 0 & \delta_i & \gamma_i & 0 & \delta_j & \gamma_j & 0 & \delta_k & \gamma_k & 0 & \delta_m & \gamma_m \end{bmatrix} \begin{Bmatrix} u_i \\ v_i \\ w_i \\ u_j \\ v_j \\ w_j \\ u_k \\ v_k \\ w_k \\ u_m \\ v_m \\ w_m \end{Bmatrix}. \tag{124}$$

A relação entre tensão e deformação é dada pela lei de Hooke generalizada (equação 38). Logo,

$$\{\sigma\} = \boldsymbol{C}\{\varepsilon\} \implies \{\sigma\} = \boldsymbol{CBU}^{(e)}. \tag{125}$$

3.5.3 As condições de contorno

Tal qual o CST, os carregamentos aplicados sobre as fronteiras dos elementos devem ser decompostos em forças nodais equivalentes. O mesmo procedimento descrito na seção anterior, aplicando o PTV, pode ser empregado a integração da equação 94 então deve ser feita sobre o tetraedro o que, similar ao CST, tem uma forma algébrica, devido ao fato da simplicidade das funções de interpolação. Similarmente, as forças nodais equivalentes são dadas pela mera distribuição uniforme do carregamento sobre os nós que compõem a superfície sobre a qual é aplicado. De acordo com Oñate (2009), a força nodal equivalente sobre os nós de um carregamento uniforme na superfície de um elemento tetraédrico linear é dado por

$$\boldsymbol{F}_{\boldsymbol{t}}^{(e)} = \frac{1}{3} A^{(e)} \boldsymbol{t}, \tag{126}$$

em que $A^{(e)}$ é a área da superfície do elemento em que atua o carregamento uniforme $\boldsymbol{t}$.

4 JULIA & SEUS MÓDULOS

A programming language to heal the planet together. (Alan Edelman)

Figura 11 – Logo da linguagem Julia

Julia é uma linguagem de programação dinâmica, opcionalmente tipada, pré-compilada, generalista, de código livre[1] e de alto nível, criada por Jeff Bezanso, Stefan Karpinski, Viral B. Shah e Alan Edelman, em 2012, com o objetivo de minimizar o problema das duas linguagens (*the two language problem*). É voltada para a programação científica, com capacidades de alta performance e sintaxe simples, similar à notação matemática usual. (SHERRINGTON; BALBEART; SENGUPTA, 2015)

Na aplicação do MEF, a linguagem se destaca por sua sintaxe semelhante a do MATLAB, que se mostra ideal na manipulação de matrizes e vetores, e por sua capacidade de processamento paralelo, que permite a otimização na construção e resolução de sistemas algébricos grandes. Além disso, o empacotamento oferecido pela linguagem, por meio do *Pkg.jl*, fornece ferramentas de controle e versionamento, como também, a criação de módulos, uma forma eficiente de organizar e distribuir aplicações.

Este capítulo aborda brevemente alguns conceito dos seguintes tópicos, vinculados a pacotes, que são relevantes para a aplicação do MEF em Julia, e que foram explorados no desenvolvimento de PHILLIPO:

1. matrizes esparsas, o *SparseArrays.jl*;
2. processamento paralelo com *Threads.jl* em estruturas de repetição;

4.1 MATRIZES ESPARSA

As matrizes esparsas desempenham um papel crucial em diversas áreas da computação científica, sendo particularmente relevantes no contexto do MEF quando se refere às matrizes de rigidez globais, que geralmente são grandes. Se n é o número de nós e g o número de graus de liberdade, a matriz de rigidez terá dimensão $ng \times ng$, que pode ser da ordem de milhares

[1] A Linguagem Julia é distribuída, quase integralmente, sob a MIT License, que permite a modificação, utilização e distribuição, seja comercial ou não, de qualquer parte do código, assim como das documentações associadas.

quando a malha é suficientemente refinada. No entanto, a maioria dos elementos da matriz de rigidez é nulo, pois cada nó está conectado apenas a um número limitado de outros nós, devido à conectividade dos elementos. Pode-se, então, definir essa matriz como esparsa, o que tem implicações importantes na sua armazenagem e manipulação (LOGAN, 2022).

Uma matriz esparsa é aquela em que a maioria dos elementos é igual a zero (em contraste com as matrizes densas, onde a maioria dos elementos é diferente de zero). Aqui são apresentadas duas formas de armazenamento de matrizes esparsas, e como elas podem ser utilizadas em Julia pelo módulo *SparseArrays.jl*: COO e CSC.

O armazenamento de matrizes esparsas por coordenadas (COO) consiste em gravar apenas os valores não nulos em um conjunto de tuplas (i, j, v), em que i e j são os índices de linha e coluna, respectivamente, e v é o valor na posição correspondente. Todas outras posições são presumidas nulas (DUFF A. M. ERISMAN, 1989).

Uma matriz esparsa, por exemplo

$$A = \begin{bmatrix} 3 & 0 & 0 & 13 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 11 \\ 0 & 7 & 0 & 9 \end{bmatrix}, \tag{127}$$

pode ser armazenada no conjunto de tuplas

$$A_{COO} = \{(1,1,5),(2,2,5),(3,3,0),(4,4,9),(1,4,13),(3,4,3),(4,2,7)\}. \tag{128}$$

Como tuplas não são comumente utilizadas em programação imperativa para armazenar um grande volume de dados, o conjunto é substituído por uma representação ordenada: um grupo de três vetores (I,J e V). Para a matriz A, nessa forma

$$I = [1,1,2,3,4,4], \qquad J = [1,4,2,4,2,4], \quad V = [3,13,5,11,7,9], \tag{129}$$

em que I são os índices de linha, J os índices de coluna e V os valores não nulos.

Uma vantagem da utilização desse formato é a inserção de novos valores sobre posições não nulas. A inserção é realizada simplesmente concatenado mais uma posição nos vetores I, J e V. Como é visto mais adiante, em outros formatos, como o CSC, essa operação requer mais alocações dependendo da posição nova a ser inserida.

O formato COO infelizmente não está presente no módulo *SparseArrays.jl*, e por conta disso, foi implementado explicitamente em PHILLIPO, no módulo interno *Matrices.jl*[2].

O formato CSC, por sua vez, já consiste em uma forma mais sofisticada, fazendo um mapeamento coluna por coluna. As entradas não nulas da matriz são organizados em ordem crescente de coluna e linha no vetor V, armazenando os índices de linha respectivos no vetor

[2] Esse trecho do código é uma adaptação da implementação do formato COO em *FEMSparse.jl*. Detalhes da implementação não fazem parte do escopo deste trabalho.

I. O vetor J é substituído por um vetor P, que consiste nas posições do vetor V que separam a armazenagem das colunas, e quando uma coluna só tem valores nulos a última posição da coluna não nula é repetida. Por convenção, ao final do vetor P é concatenado a quantidade de entradas não nulas acrescida de um. A matriz A, nessa forma, se torna

$$I = [1,2,4,4,1,3], \qquad V = [3,5,7,9,13,11], \quad P = [1,2,4,4,7]. \tag{130}$$

O módulo *SparseArrays.jl* implementa esse formato como padrão de matrizes esparsas, além de adicionar diversas otimizações matemáticas nas operações definidas pelo módulo *LinearAlgebra.jl*, principalmente quando se trada de resolver grandes sistemas lineares, escolhendo a melhor rotina aplicável [3].

Vale também ressaltar que o módulo *SparseArrays.jl* implementa as matrizes esparsas de modo que ainda sejam compatíveis com as operações convencionais, isto é, a sintaxe e funções definidas para matrizes densas (adição, multiplicação e seleção de entradas...) são aplicáveis também para matrizes esparsas, de forma que o usuário não precise se preocupar com a forma de armazenamento da matriz, e pode se ater apenas na resolução do problema em si.

4.2 PROCESSAMENTO PARALELO

Computação paralela é um método de realizar múltiplos cálculos ou tarefas simultaneamente, utilizando vários processadores ou computadores. Essa abordagem tem o objetivo de aumentar a velocidade e eficiência do processamento, dividindo um problema maior em partes menores e independentes que podem ser resolvidas simultaneamente. Isso difere da computação sequencial tradicional, onde as tarefas são realizadas uma após a outra por um único processador. A computação paralela é frequentemente empregada em tarefas que podem ser divididas em subtarefas paralelas, como simulações científicas e análise de dados, para um aproveitamento maior da capacidade de processamento disponível (TROBEC; SLIVNIK; BULI, 2018).

A linguagem Julia implementa suporte a quatro categorias de programação concorrente e paralela. Conforme a própria documentação da linguagem, são elas:

1. Tarefas Assíncronas, ou Corotinas: As tarefas em Julia permitem a suspensão e retomada de computações para operações de entrada/saída, manipulação de eventos, processos produtor-consumidor e padrões semelhantes. As tarefas podem sincronizar por meio de operações como espera (wait) e obtenção (fetch), e se comunicam por meio de Canais. Embora estritamente não sejam computação paralela por si mesmas, Julia permite agendar tarefas em vários threads.[4]

2. Multi-threading: O suporte a multi-threading em Julia proporciona a capacidade de agendar tarefas simultaneamente em mais de um thread ou núcleo de CPU, compartilhando

[3] Detalhes da implementação de rotinas otimizadas para matrizes esparsas não faz parte do escopo deste trabalho.
[4] Esse conceito é similar as *promises* em JavaScript.

memória. Geralmente, essa é a maneira mais fácil de obter paralelismo em um PC ou em um único servidor com vários núcleos. O suporte a multi-threading em Julia é fornecido pelo módulo *Threads.jl*.

3. Computação Distribuída: A computação distribuída executa vários processos Julia com espaços de memória separados. Esses processos podem estar no mesmo computador ou em vários computadores. A biblioteca padrão Distribuída fornece a capacidade de execução remota de uma função Julia. Com esse bloco de construção básico, é possível criar várias abstrações de computação distribuída. Pacotes como DistributedArrays.jl são exemplos de tais abstrações. Por outro lado, pacotes como MPI.jl e Elemental.jl fornecem acesso ao ecossistema existente de bibliotecas MPI. O suporte a computação distribuída em Julia é fornecido pelo módulo *Distributed.jl* [5].

4. Computação em GPU: O compilador GPU em Julia proporciona a capacidade de executar código Julia nativamente em GPUs. Existe um ecossistema robusto de pacotes Julia voltados para GPUs. O site JuliaGPU.org fornece uma lista de capacidades, GPUs suportadas, pacotes relacionados e documentação.

Neste trabalho foi aplicado um aspecto de Multi-threading diretamente, pelo o módulo *Threads.jl*, para paralelizar a construção da matriz de rigidez global.

O módulo *Threads.jl* fornece a possibilidade de executar um bloco de código em paralelo, por meio da macro *@threads*. As macros são formas de metaprogramação implementadas em Julia, que servem como ferramentas para incluir e manipular o próprio código em tempo de execução[6]. A macro *@threads* é utilizada para transformar um bloco de repetição, leia-se loop, de assíncrono para síncrono, isto é, criando uma pilha de execuções formada pelas iterações do loop que são chamadas para as *threads* conforme vão ficando disponíveis.

Uma *thread*, ou linha de execução, é a menor unidade de processamento que pode ser programada em um sistema. É uma sequência de instruções que pode ser executada independentemente, compartilhando recursos com outras threads que pertencem ao mesmo processo. Em suma, é um fluxo de execução dentro de um processo[7]. (BALBAERT, 2015)

Conforme a sessão Julia é iniciada, uma *thread* é criada para executar o código principal, e também são reservadas outras para executar tarefas síncronas, o que deve ser declarado explicitamente pelas flag *-t*. Por padrão, a sessão nunca reserva mais de uma *thread*.

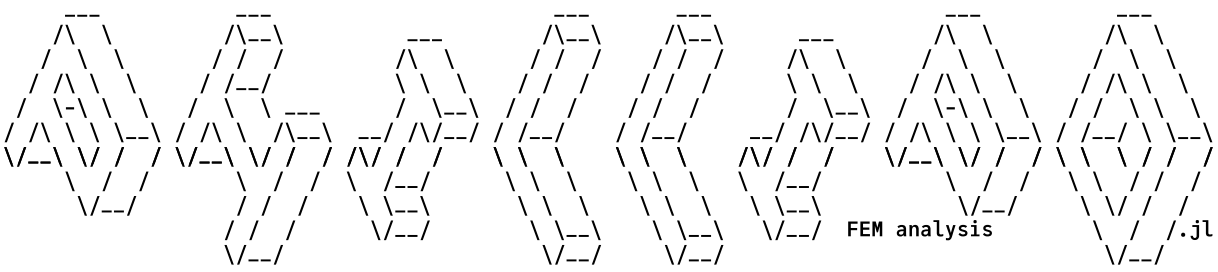
---

[5] *JuliaFEM.jl* utiliza desas abordagem para implementar o MEF de forma escalável.
[6] Isso é devido a homoiconicidade da linguagem.
[7] Processo aqui pode ser compreendido como a sessão Julia.

5 PHILLIPO

PHILLIPO é um *solver* para análise de estruturas discretizadas por elementos finitos, com algumas otimizações computacionais relacionadas a paralelismo e matrizes esparsas, e empacotamento, que visa constituir-se como exemplo menor na implementação legível e concisa dos algoritmos de elementos finitos em Julia no âmbito acadêmico do campus CCT, da UDESC. PHILLIPO é um programa de código aberto, que é distribuído em um repositório público[1] sob a licença LGPL[2]. Portanto, sua utilização é gratuita e livre para fins acadêmicos e comerciais, que incluem a modificação, implementação e venda de qualquer parte do programa, como também da documentação que o acompanha; só se resguarda, entretanto, a devida citação desta monografia. A logo de PHILLIPO é mostrada na figura 12.

Figura 12 – Logo estilizada de PHILLIPO.jl



Neste capítulo é apresentado como é feita a distribuição e a instalação, e como se dá o funcionamento de PHILLIPO, dividido em duas partes. A primeira, descrevendo o fluxo de execução normal do programa, isto é, utilizando o GID como interface de pré e pós-processamento, e, a segunda, esmiuçando o código, tanto do módulo PHILLIPO, quanto dos arquivos de integração com o GID.

5.1 DISTRIBUIÇÃO PELO PKG.JL E IMPORTAÇÃO DOS *PROBLEMS TYPES* NO GID

O Pkg.jl é o gerenciador de pacotes anexado à Julia, tal como PIP é anexado ao Pyhton. Ele é responsável por distribuir, gerir e empacotar os módulos da linguagem, permitindo relacionar dependências e controlar versionamento. PHILLIPO é distribuído por meio do Pkg.jl, porém, não pelo repositório oficial[1], mas pelo próprio repositório deste trabalho, que pode ser utilizado para o mesmo fim, pela função *add*. A utilização do *Pkg.jl* determinou a estrutura de a estrutura dos arquivos de código-fonte de PHILLIPO, conforme o próprio manual do pacote[2].

[1] O repositório é mantido no GitHub, assim como o presente documento em formato Latex: <https://github.com/lucas-bublitz/PHILLIPO>

[2] O GiD, interface de pré e pós-processamento, é um software distribuído comercialmente, e não está sujeito à mesma licença que PHILLIPO.

[1] Há uma série de critérios para que um módulo seja adicionado ao repositório oficial do Pkg.jl, além disso, não é objetivo de PHILLIPO ser distribuido massivamente.

[2] Acessível em <https://pkgdocs.julialang.org/v1/>

Como PHILLIPO foi encapsulado em um módulo, pode ser facilmente distribuído iniciando uma sessão Julia e executando:

```
1    add https://github.com/lucas-bublitz/PHILLIPO.jl
```

O Pkg.jl então trata de buscar as dependências do módulo, isto é, os módulos que são importados para uso interno de PHILLIPO: o *SparseArrays*, que fornece as estruturas e funções para alocar e manipular eficientemente matrizes esparsas, o *LinearAlgebra*, implementação do LAPACK em Julia, e o *JSON*, um parser de objetos em JSON para dicionários. No arquivo *Projecy.toml* é possível encontrar tanto essa lista de dependência, e no *Manifest.toml*, são listadas as dependências das dependências, isto é, quais módulos cada módulo importado por PHILLIPO importa para si. [3]

PHILLIPO utiliza a interface GID para gerar as malhas e definir as condições de contorno. A integração desses programas é feita pelo conjunto de arquivos presente presentes na pasta *\GID connections*: *PHILLIPO.gid* e *PHILLIPO3D.gid*, que são os *Problem types* do GID para PHILLIPO, e *link.jl*, que é o arquivo que é chamado pelo *script* de execução do GID, e que importa o módulo PHILLIPO.jl e o executa. O conteúdo dessa pasta deve ser copiado para a pasta *...\GiD 16.1.6d \ProblemTypes*, localizada onde o próprio GID está instalado[4], para que sejam automaticamente carregados durante a inicialização do GID[5].

5.2 FLUXO DE EXECUÇÃO

O fluxo de execução é uma ferramenta de projeto que tem como objetivo descrever a ordem e as condições que determinadas seções do código são executadas. A utilização de PHILLIPO.jl segue os digramas das figuras 13 e 14. Esses diagramas não representações simplificados e parciais, e não utilizam um padrão ou simbologia formal que é própria desses diagramas.

Nesses diagramas, é possível separar a execução de uma utilização normal do software em três partes principais:

1. Pré-processamento; Parte em que ocorre a criação da geometria, a definição das propriedades dos materiais, das condições de contorno e das cargas aplicadas, assim como a geração da malha e elementos.

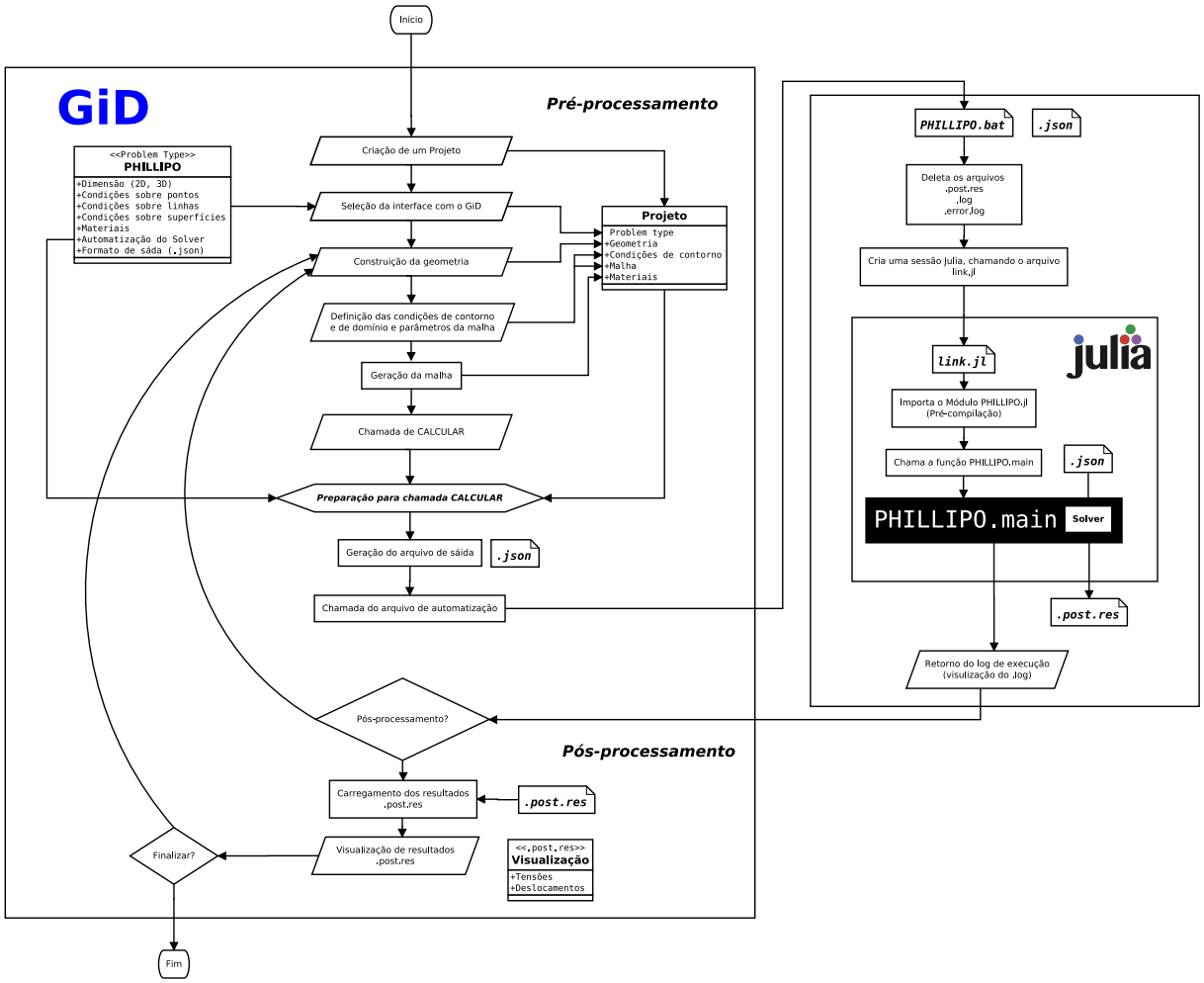
2. Processamento; Parte em que é chamada uma sessão Julia para carregar o módulo PHILLIPO.jl, que é responsável por ler os arquivos de entrada, e executar o algoritmo de elementos finitos, gerando os arquivos de saída para o GID.

[3] O versionamento é importante pois permite que a compilação do pacote seja feita utilizando exatamente os códigos dos módulos de quando foi desenvolvido, assim baixando o risco de resultados inesperados devido a uma alteração no funcionamento de um módulo exterior ao que se trabalha.

[4] O caminho para a pasta *ProblemTypes* pode variar de acordo com a versão do GID, e com o sistema operacional.

[5] Também é possível importar *Problem types* dentro da interface do GID, entretanto, desse modo, a importação não é permanente.

Figura 13 – Fluxograma de execução: GID



3. Pós-processamento; Parte em que o GID lê os arquivos de saída, e gera os gráficos de resultados.

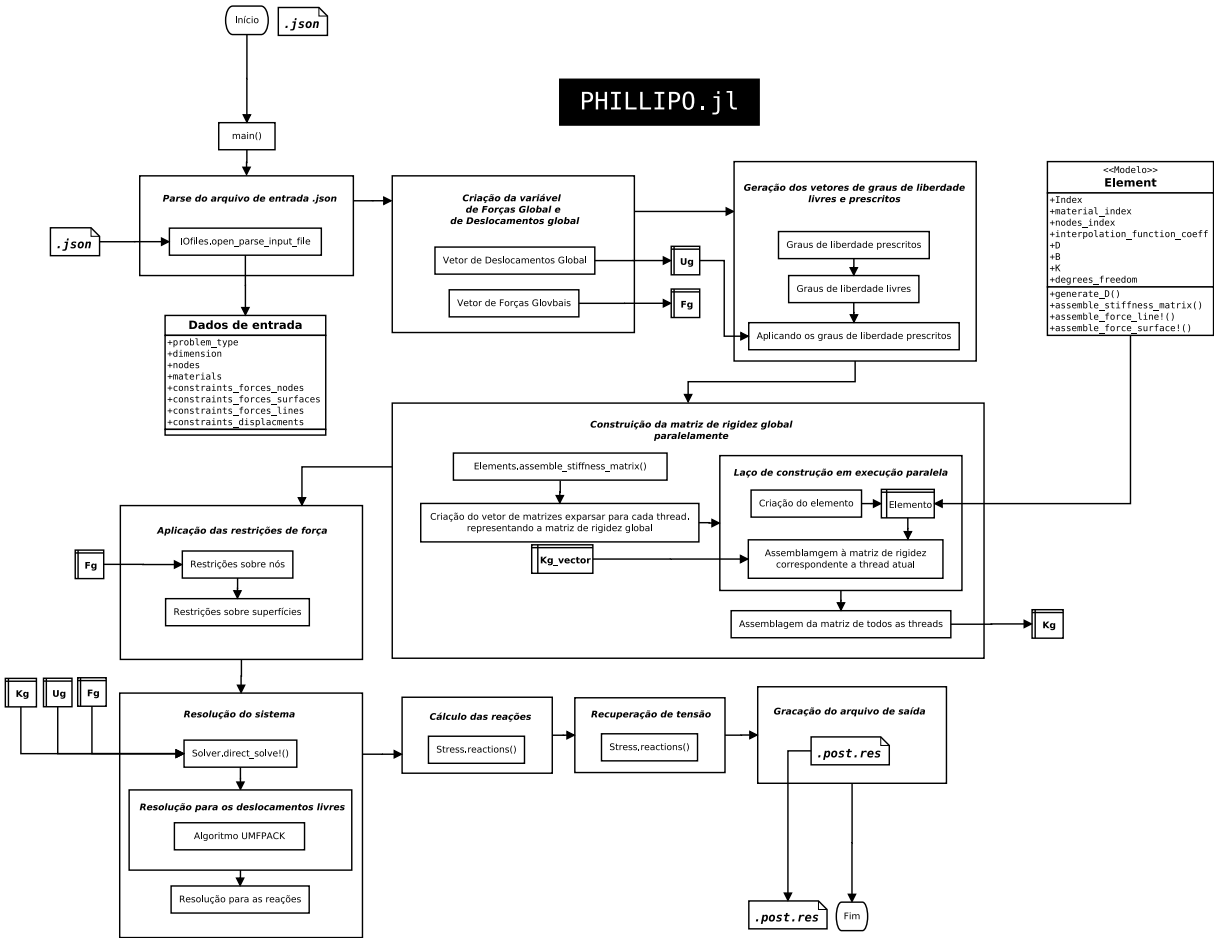
5.2.1 Pré-processamento

O pré-processamento é realizado totalmente pelo GID (figura 13), e consiste na definição da geometria, das condições de contorno, do material e na geração de malha, e segue:

1. Definição do *Problem type*. Caso o problema físico seja modelado em duas dimensões, deve-se escolher o *Problem type PHILLIPO*, caso seja modelado em três dimensões, deve-se escolher o *Problem type PHILLIPO3D*. Essa escolha define quais arquivos serão utilizados para a definição das condições de contorno, dos materiais, e como será a geração da malha. O GID cria então uma pasta onde os arquivos do problema serão salvos.

2. **Definição da geometria do problema,** que pode ser tanto construída utilizando as ferramente CAE do próprio GID, como também, importada de um arquivo externo de algum

Figura 14 – Fluxograma de execução: PHILLIPO.jl



outro software CAD cujo formato seja reconhecido pelo GID [6].

3. **Definição das características do material,** como também as condições de contorno: sejam elas deslocamentos sobre pontos, linhas e superfícies, ou carregamentos concentrados sobre pontos, ou carregamentos distribuídos sobre linhas e superfícies. [7]

4. **Geração da malha,** que pode ser amplamente configurada pelas ferramentas oferecidas pelo GID para gerar malhas estruturadas ou não estruturadas, com refinamentos em determinadas regiões, e com a possibilidade de se definir o tipo de elemento a ser utilizado.

5. **Chamada da função de CALCULAR.**

A função de CALCULAR do GID, executa o arquivo *.bat* do *problem type*: deleta possíveis arquivos de saída anteriores, e gera os arquivos de saída *.dat*[8], assim como os arquivos de *log*, e cria a sessão Julia, chamando para ser executado nela o arquivo *link.jl*.

[6] O GID reconhece uma grande variedade de arquivo de entrada (sejam de geometria, malhas, condições de contorno), inclusive arquivos próprios de programas comerciais, como ANSYS e Abaqus;

[7] Carregamentos sobre linha só estão disponíveis para o *Problem type PHILLIPO*.

[8] Embora o arquivo esteja nomeado no formato *dat* ele, na verdade, é um *.json*, pois esse é o padrão de entrada para PHILLIPO.

Na sessão Julia, o módulo PHILLIPO.jl é importado, momento em que ocorre a pré-compilação do código, e, em seguida, a chamada da função principal do módulo *PHILLIPO.main*, passando como parâmetros os caminhos para o arquivo *.json* e o caminho de saída do resultado da análise, assim iniciando o processamento.

5.2.2 Processamento

O processamento é realizado dentro da sessão Julia, executando a função principal de *PHILLIPO.main* (figura 14), e consiste nas seguintes etapas:

1. **Leitura do arquivo de entrada.** O arquivo *.json* é lido e convertido em um dicionário pelo *parser* JSON, cujos dados são distribuídos nas variáveis do problema.

2. **Definição dos graus de liberdade livres e prescritos.** Conforme sãos as restrições de deslocamento, o programa calcula a numeração dos graus de liberdade prescritos, e os armazena em um vetor.

3. **Construção da matriz global de rigidez paralelamente.** Nessa etapa, o programa chama as funções de construção de elemento paralelamente, utilizando uma macro do módulo *Threads*, para construir a matriz global de rigidez, que é salva no formato COO (um formato de matriz esparsa que armazena os valores em um vetor único cuja ordem não importa para a interpretação da matriz, o que facilita a execução paralela). Após a construção da matriz global de rigidez, é feita a conversão para o formato de matriz esparsa CSR.

4. **Aplicação das restrições de forças.** As restrições de forças são aplicadas sobre o vetor de forças prescritas. Dependendo do tipo (sobre linhas ou superfícies), são calculadas as forças nodais equivalentes.

5. **Resolução do sistema.** Com a matriz global de rigidez e o vetores de deslocamentos e forças nodais, o programa decompõe o sistema pelos graus de liberdade livres e prescritos, e o resolve diretamente, utilizando o método mais apropriado, determinado pelo módulo *LinearAlgebra*, levando em consideração as características da matriz de rigidez global.

6. **Cálculo das reações e recuperação de tensão.** O programa calcula as reações de apoio, utilizando os graus de liberdade prescritos.

7. **Realização da recuperação de tensão.** O programa realiza a recuperação de tensão, utilizando as funções de interpolação do elemento, e os deslocamentos nodais.

8. **Geração do arquivo de saída.** O programa gera o arquivo de saída, no formato *.dat*, que é lido pelo GID para gerar os gráficos de resultados.

Com o fim da execução da função principal, é encerrada a sessão Julia e, e é chamado o programa Notepad para a abrir o arquivo *.log*, onde algumas informações de debug foram impressas durante o processamento.

5.2.3 Pós-processamento

O Pós-processamento é realizado pelo GID, e consiste num conjunto de ferramentas de plotagem, com diversas formas de visualizar todo os dados impressos no arquivo de saída.

5.3 INTEGRAÇÃO COM GID

O GID é um software utilizado como pré e pós-processamento (ver figuras 15 e 16). Com ele é possível criar a geometria do problema, definir as propriedades dos materiais, as condições de contorno, as cargas aplicadas, e, principalmente, gerar a malha de elementos. Além de se ser possível a integração com um *solver* qualquer, por meio de um conjunto de arquivos de entrada e saída (ambos configurados de forma a permitir uma certa flexibilidade nessa integração), cuja execução é controlada por um *script* em Batch, o que possibilita a automatização do processo de simulação. Nesta seção é abordado como é feita e integração entre o GID e PHILLIPO, por meio das pastas *PHILLIPO.gid* e *PHILLIPO3D.gid*, sendo que, como a nomeação dos arquivos sugere, a primeira é utilizada para problemas bidimensionais, e a segunda para problemas tridimensionais.

Figura 15 – Exemplo de pré-processamento de geometria no GID

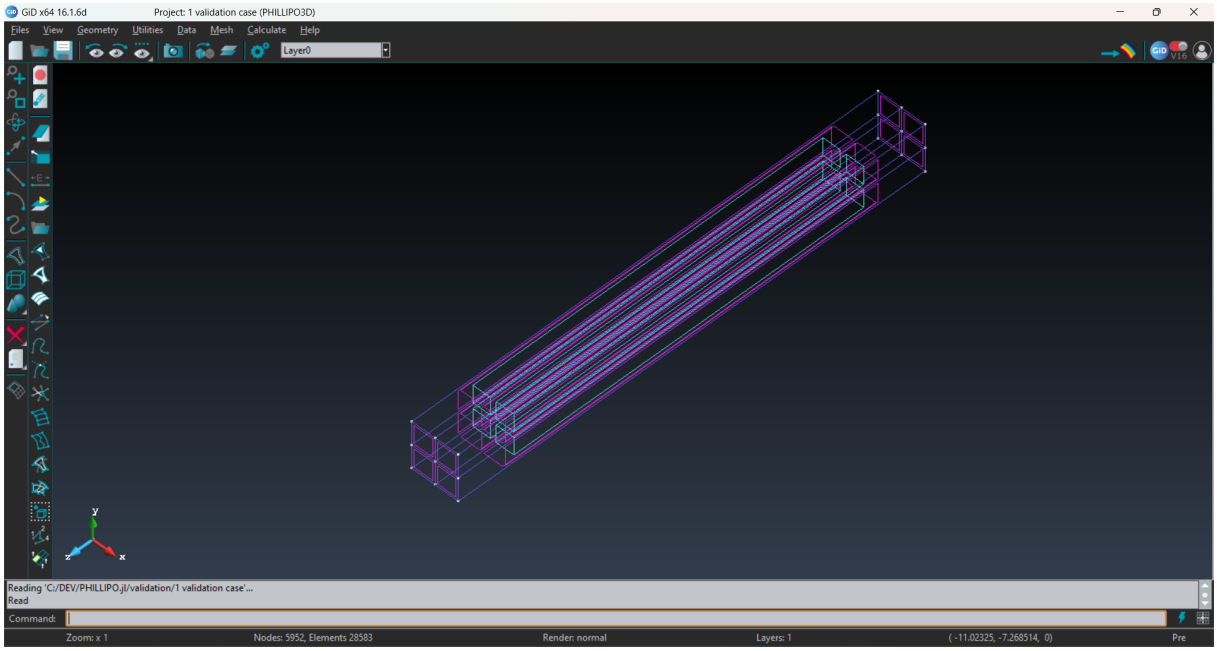
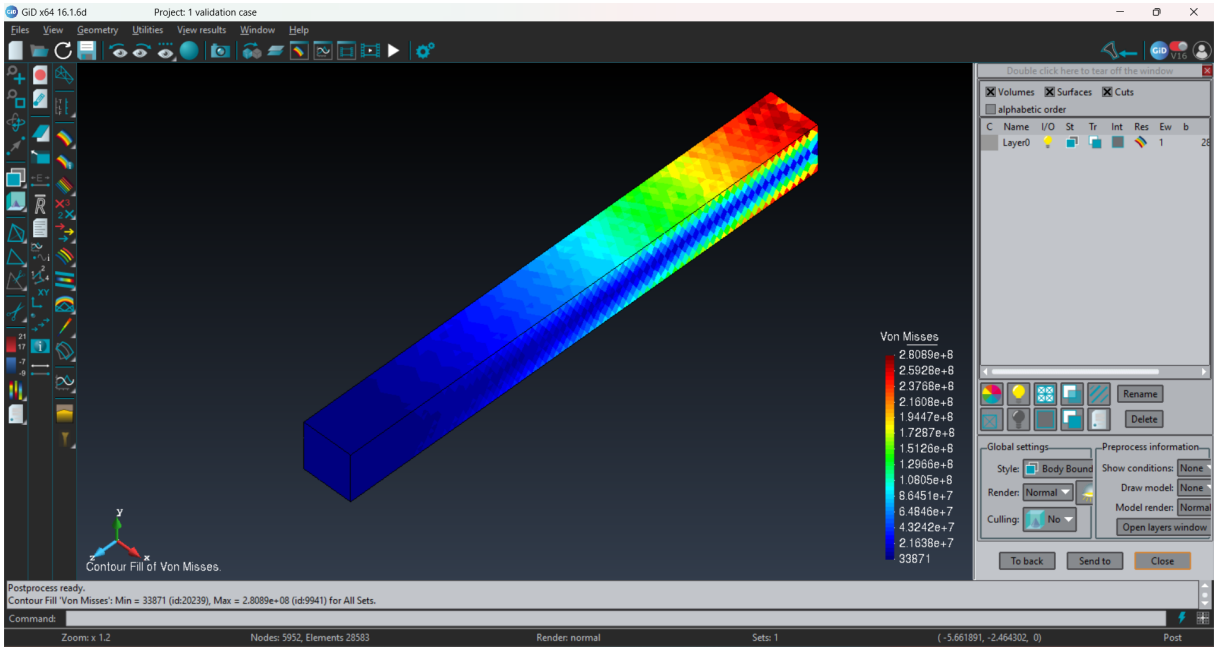


Figura 16 – Exemplo de pós-processamento no GID



5.3.1 PHILLIPO.gid

O GID pode ser configurado para operar como pré e pós-processamentos de diversos programas, como o Abaqus, o Ansys, o Calculix..., por meio de um *Problem type*, que é como o GID chama o conjunto de arquivos que configuram o formato de saída dos dados, a criação de determinadas propriedades para as condições de contorno e materiais, como também automatizar a execução da simulação, chamando o programa. Pode-se dizer que o *Problem type* é uma interface para que as informações contidas nos arquivos gerados pelo GID (geometria, malha, condições de contorno etc.) sejam salvas em um formato que o programa, o *solver*, possa interpretar, ao passo que o *script* de execução automatiza o chamamento desse, e a simulação seja iniciada.

Na pasta *PHILLIPO.gid* é possível encontrar os seguinte arquivos:

1. *PHILLIPO.cnd*: define as condições de contorno e como são aplicadas;
2. *PHILLIPO.prb*: define as entradas de informações gerais;
3. *PHILLIPO.mat*: define as características dos materiais utilizados para os elementos;
4. *PHILLIPO.bas*: configura o arquivo de saída do GID para ser interpretado por PHILLIPO.jl;
5. *PHILLIPO.bat*: *script* para chamar uma sessão Julia, chamando *link.jl*;
6. *link.jl*: importa o módulo PHILLIPO e o executa.

O primeiro arquivo do *Problem type* de PHILLIPO é *PHILLIPO.cnd*, que define as condições de contorno e sobre quais entidades, leia-se nós, elementos ou geometrias (superfícies, volumes, linhas etc.), são aplicadas, por meio de uma sintaxe específica [9], uma forma de marcação de texto, que é interpretada pelo GID.

Figura 17 – Parte do arquivo de condições de contorno: PHILLIPO.cnd

```
1  CONDITION: Constraint_displacement_point
2  CONDTYPE: over points
3  CONDMESHTYPE: over nodes
4  QUESTION: X
5  VALUE: 0.0
6  QUESTION: Y
7  VALUE: 0.0
8  QUESTION: Z
9  value: 0.0
10 END CONDITION
```

Em sua representação parcial, da figura 17, é possível notar a construção de uma condição de contorno por meio de um bloco que inicia na linha 1, com a expressão *CONDITION: Constraint_displacement_point*, que também nomeia esta condição, referente a restrição de deslocamento em pontos (entidade geométrica discretizada por um nó), e que acaba com *END CONDITION*. Dentre essas linhas, são definidas as formas e os valores que essa condição vai aplicar sobre a geometria selecionada, neste caso, os pontos. Na linha 2, é definido, justamente, sobre qual entidade geométrica se aplica essa condição: pontos. Na próxima linha, é definido como que essa informação, que foi associada à entidade geométrica se traduz na malha: essa condição é aplicada sobre os nós que cujo ponto foi discretizado. [10] As linhas seguintes, 4 a 9, se referem aos valores da condição, neste caso, aos deslocamentos prescritos sobre os nós nas direções de *X*, *Y* e *Z*.

Na condição *Constraint_force_line* (figura 18), o processo é análogo. A restrição de carregamento uniforme é aplicada sobre uma linha (o tipo de geometria), e, diferentemente da anterior, é traduzida sobre as faces. Nesse caso bidimensional, é o segmento de reta determinado pelos nós nas extremidades. Os campos *X*, *Y* e *Z*, referem-se ao vetor desse carregamento uniforme.

O arquivo *PHILLIPO.prb* (figura 19) define as entrada de informações gerais ao problema, ou seja, que não são aplicados diretamente sobre a geometria. Somente um campo foi utilizado e se refere ao tipo do estado plano (EPD ou EPT). A sintaxe da linha 3 é indica que esse campo só possa ser preenchido por essas duas opções, no formato *drop-down list*.

[9] No manual do usuário do GID, acessível em <https://gidsimulation.atlassian.net/wiki/spaces/GUM/overview>, é possível encontrar a descrição o funcionamento de toda essa sintaxe, que compreende desde esse arquivo de condições de contorno, como também, dos outros que compões a construção do *problem type*.

[10] Isso se deve porque a malha é criada sobre a geometria, posteriormente à aplicação das condições de contorno sobre aquela.

Figura 18 – Parte do arquivo de condições de contorno: PHILLIPO.cnd

```
CONDITION: Constraint_force_line
CONDTYPE: over lines
CONDMESHTYPE: over face elements
QUESTION: X
VALUE: 0.0
QUESTION: Y
VALUE: 0.0
QUESTION: Z
value: 0.0
END CONDITION
```

Figura 19 – Arquivo de dados gerais: PHILLIPO.prb

```
PROBLEM DATA

QUESTION: type#CB#(plane_stress,plane_strain)
value: plane_strain
QUESTION: thickness
VALUE: 1.0

END GENERAL DATA
```

O arquivo *PHILLIPO.mat* (figura 20) define as características dos materiais implementados. Somente um material foi definido, o aço AISI 4340, com os campos de módulo de elasticidade, coeficiente de Poisson e massa específica.

Figura 20 – Arquivo de materiais: PHILLIPO.mat

```
MATERIAL: AISI_4340_Steel
QUESTION: Young_Modulus
VALUE: 210E+9
QUESTION: Poisson_Ratio
VALUE: 0.3
QUESTION: Density
VALUE: 0.785
END MATERIAL
```

O arquivo *PHILLIPO.bas* configura o arquivo de saída do GID. Esse arquivo intercala o conteúdo explicito do arquivo de saída (as partes invariáveis), com trechos de programação, responsáveis por imprimir os valores dinâmicos, aqueles referentes ao problema em si, como as coordenadas dos nós, a conectividade dos elementos etc.[11] A separação entre texto e programa se dá pelo caractere * no início da linha, indicando que toda ela é uma instrução para ser executada.

O arquivo gerado é do formato *.json*, o que pode ser não usual para aplicações do MEF. Entretanto, esse formato é amplamente utilizado para troca de dados entre aplicações, e, por

[11] Esse modo de compor os arquivo, misturando textos estáticos e programação é similar aos arquivos em PHP. Isso numa utilização mais clássica.

isso, pode ser facilmente lido por um módulo já consolidado em Julia, o *JSON.jl*[12].

Para gerar as listas de elementos triangules, foi escrito o seguinte código, dentro do arquivo emphPHILLIPO.bas:

```
1     "nodes":[
2 *loop nodes
3 *format "%e,%e"
4         [*NodesCoord],
5 *end
6         null
7     ],
```

Em cada linha, os nós são definidos por listas, cujos valores representam as componentes *x* e *y* respectivamente. A numeração dos nós é implícita pela ordem da lista de *"nodes"*.

Na linha 1 é definido o nome do campo *"nodes"*, separado pelo características *:* inicia a declaração do valor daquele campo: uma lista *[*[13]. Na linha 2, é iniciado um laço de repetição, que percorre todos os nós, e, para cada um, é adicionado um elemento à lista, que é composto por um par de coordenadas, separadas por vírgula, e envolvidas por colchetes, pois cada nó é presentando por um vetor que contem suas coordenadas. Na linha 5, é fechado o laço de repetição, e, na linha 6, é fechado o campo *]*.

Esse padrão se repete para o elementos também, com a diferença que estes, dependendo do seu tipo (CST ou tetraedro), são listas dentro da tupla que representa os elementos. Na figura 21, é possível notar essa hierarquia dos dados na estrutura de entrada de PHILLIPO. Os elementos são separados pela forma das funções de interpolação. Para o CST, as funções são linear, então esses elementos são gravados dentro da chave *"linear"*.

Vale ressaltar também como as condições de contorno são gravadas. Como as restrições de deslocamento sempre são sobre nós, só há uma lista referente a restrições de deslocamento no arquivo *.json*. Já os carregamentos distribuídos, para serem aplicados no MEF, precisam ser convertidos para forças nodais, e, para tanto, é necessário utilizar as funções de interpolação que dependem do elemento. Por conta disso, são gravadas as condições de contorno de carregamento sobre as entidades geométricas: linhas e superfícies, por meio dos nós, de forma que o PHILLIPO se encarrega de calcular, utilizando as funções de interpolação, as forças nodais equivalentes sobre o elemento que a contém. Na linha 35, da figura 21, uma restrição de força é descrita: a primeira posição se refere a numeração que o GID deu para aquela entidade de linha, os dois valores seguintes são os identificadores dos que definem a entidade, e os valores restantes são as componentes do carregamento.

O arquivo *PHILLIPO.bat* (figura 22) é um *script* que é chamado pela função CALCULAR do GID, para iniciar a análise do problema. É um arquivo de linhas de comando (ou

[12] JSON se refere a *JavaScript Object Notation*, um formato de texto para representar objetos na linguagem JavaScript, que se tornou padrão na implementação de APIs.

[13] Essa sintaxe de listas e *literals* é próprio do JavaScript.

Figura 21 – Arquivo de saída do quarto caso de verificação: 4 verification case.dat

```
1
2 {
3     "title": "PHILLIPO: arquivo de entrada",
4     "type": "plane_stress",
5     "materials":[
6         ["AISI_4340_Steel", 210E+9, 0.3],
7         null
8     ],
9     "nodes":[
10        [0.000000e+00,0.000000e+00],
11        [1.000000e+00,0.000000e+00],
12        [0.000000e+00,1.000000e+00],
13        [1.000000e+00,1.000000e+00],
14        null
15    ],
16    "elements":{
17        "linear":{
18            "triangles":[
19                [1, 1, 1,2,4],
20                [2, 1, 4,3,1],
21                null
22            ]
23        }
24    },
25    "constraints":{
26        "displacements":[
27            [1, 0.0, 0.0, 0.0],
28            [3, 0.0, 0.0, 0.0],
29            null
30        ],
31        "forces_nodes":[
32            null
33        ],
34        "forces_lines":[
35            [2, 4,3, 0.000000e+00, -1.000000e+06, 0.000000e+00],
36            null
37        ],
38        "forces_surfaces":[
39            null
40        ]
41    }
42 }
```

arquivo de lote)[14] para o sistema operacional Windows[15].

Nas linhas 1 a 3, são feitas as exclusões de possíveis arquivos restantes de análises passadas. Na linha 4, é chamada a sessão Julia, passando como parâmetro o arquivo *link.jl*, o caminho absoluto do arquivo de entrada *.dat* e o local em que o arquivo de saída deve ser gravado. No final da linha, os operadores right-shift » redirecionam o debug da execução de PHILLIPO para o arquivos *.log* e *.error.log*. Na linha 5, o arquivo log de erro é aglutinado ao arquivo de log para que, quando a linha 5 for executada, o aplicativo Notepad exiba os dois conteúdos juntos[16].

A flag *-t* com o parâmetro *auto* na linha 4, indica que a sessão Julia deve ser iniciado utilizado um número de threads disponibilizados pelo sistema [17]. Isso é importante para que ocorra a montagem da matriz global de rigidez paralelamente.

Figura 22 – Arquivo de execução: PHILLIPO.bat

```
1 del %1.post.res
2 del %1.log
3 del %1.error.log
4 call julia -t auto %3\link.jl "%1.dat" "%1.post.res" >> %1.log 2>> %1.
     error.log
5 type  %1.error.log  >> %1.log
6 notepad %1.log
```

O arquivo *link.jl* (figura 23) é o arquivo que executado dentro da sessão Julia. Ele recebe esse nome porque é o arquivo que faz a ligação entre a execução do GID e o módulo PHILLIPO.jl, e, por não estar previsto dentro da estrutura de um *problem type*, recebe um nome diferenciado. Nele é apenas importado o módulo PHILLIPO.jl (linha 1) e chamado a função principal do módulo (linha 2), passando como parâmetros o caminho para o arquivo de entrada e o caminho para a gravação do arquivo de saída [18]. A última linha é responsável por garantir o encerramento da sessão Julia.

A macro *@time* é responsável por fornecer algumas informações sobre o tempo de execuções do módulo (como também memória alocada e tempo dedicado à compilação ao *Garbage Collector*). É uma ferramenta de debug, e seu retorno é impresso no arquivo *.log*.

Figura 23 – Arquivo de execução: link.jl

```
1 import PHILLIPO
2 @time PHILLIPO.main(ARGS[1], ARGS[2])
3 exit(0)
```

[14] Comandos utilizados dentro do terminal do sistema: o CMD.
[15] Por conta disso, o *PHILLIPO.gid* só funciona em sistemas Windows.
[16] Assim foi implementado porque se o mesmo arquivo receber o fluxo de texto, o conteúdo dos erros sobrescrevem aos normais
[17] O parâmetro pode ser alterado para forçar a sessão Júlia a usar mais threads.
[18] ARGS é um vetor presente em toda a sessão Julia, que recebe os parâmetros passados para o executável que a iniciou.

5.4 ESTRUTURA DO MÓDULO PHILLIPO.JL

O código-fonte de PHILLIPO foi organizado em seis módulos, ao longo de sete arquivos *.jl*, agrupando as funções em categorias conforme sua aplicação. O início da definição do módulo de PHILLIPO é o arquivo homônimo, localizado na raiz da pasta *src* do repositório. São os arquivos:

1. ***PHILLIPO.jl***: define o módulo PHILLIPO e a função principal;
2. ***includes.jl***: juntas as importações dos módulos internos;
3. ***IOfiles.jl***: módulo interno que define as funções de leitura e escrita de arquivos;
4. ***Elements.jl***: módulo interno que define as estruturas dos elementos e suas funções;
5. ***Matrices.jl***: módulo interno que define a estrutura das matrizes COO;
6. ***Solver.jl***: módulo interno que executa a resolução do sistema;
7. ***Stress.jl***: módulo de recuperação de tensões.

5.4.1 PHILLIPO.jl

O arquivo *PHILLIPO.jl* é o que define o próprio módulo, e por contada disso é homônimo. Nele é se encontra a função principal *main*, que vai conduzir toda a parte do processamento da análise [19]. Na figura 24, uma parte desse arquivo é reproduzida.

Figura 24 – Parte do arquivo principal do módulo: PHILLIPO.jl

```
1  module PHILLIPO
2      # Módulo do escopo principal
3      include("./modules/includes.jl") # Módulos internos
4      # MÓDULOS EXTERNOS
5      import LinearAlgebra
6      import SparseArrays
7
8      # MÓDULOS INTERNOS
9      import .IOfiles
10     import .Elements
11     import .Solver
12     import .Matrices
13     import .Stress
14
15     # PONTO DE PARTIDA (aqui inicia a execução)
16     function main(
17             input_path::String, # Arquivo de entrada (.json)
18             output_path::String # Arquivo de saida (.post.res, formato
       do GiD)
19         )
```

[19] Pode-se pensar na função principal como um ponto de partida do software, um *int main()* como em C.

A primeira linha é a declaração da abertura do módulo, cujo nome é precedido pela keyword *module*. As linhas 3 a 10 são responsáveis por importar os módulos utilizadas. Nessa parte é importante salientar o seguinte: os módulos internos, aqueles definidos como bibliotecas explícitas dentro de PHILLIPO são incluídos aninhadamente pela inclusão do arquivo *includes.jl*, e para serem chamados à pré-compilação devem ser importados diretamente, conforme as linhas 9 a 13[20]. As Inclusões são a forma de se particionar os arquivos do módulo, fazendo com que sejam aglutinados naquela linha em que são chamados. Para o *Pkg.jl*, todo o código se torna apenas um arquivo contínuo.

Na linha 16, a função *main* é definida, e recebe dos valores: o caminho para o arquivo *.dat* de dados de entrada, o caminho para o arquivo de saída. Após essas linha, o chamament das funções segue o descrito no fluxo de execução do processamento, na subseção anterior. Vale ressaltar que nessa função estão presentes uma série de impressões, *prints*. Esses são os *debugs* já mencionados, que vão compor o arquivo *.log*[21].

A parte posterior da função principal é a leitura dos dados de entrada, chamando a função *IOfiles.open_parse_input_file*, na linha 5 da figura 25, retornando um dicionário na variável *input_dict*, cujos dados, nas linhas seguintes, são distribuídos às variáveis principais do problema:

1. ***problem_type***: referente ao tipo do problema (EPT, EPD ou tridimensional);
2. ***nodes***: vetor de vetores das coordenadas dos nós ordenados;
3. ***materials***: vetor de materiais declarados;
4. ***constraints_forces_nodes***: vetor dos vetores de restrições de força sobre nós;
5. ***constraints_forces_lines***: vetor dos vetores de restrições de força sobre linhas;
6. ***constraints_forces_surfaces***: vetor dos vetores de restrições de força sobre superfícies;
7. ***constraints_forces_displacements*** : vetor dos vetores de restrições de deslocamento sobre nós.

Também são definidas, com espaço previamente alocado, os vetores deslocamento nodais global e forças nodais globais, respectivamente *Ug* e *Fg* (linhas 36 e 37). As linhas de chamando a função *pop!* servem para retirar os valores *null* dos fins dos vetores[22].

[20] Os nomes do módulos internos devem ser seguidos de um ponto quando forem importados. Isso é devido a localização dos arquivos do sistema de Julia. Sem essa notação, a sessão Julia vai tentar buscar nos módulos instaladas na aquele ambiente, e não os declarados explicitamente no código.

[21] Existem outros modos de realizar o debug, utilizando a macro *@debug*.

[22] Isso se dá pela forma como a programação no arquivo *.bas* foi realizada. Nos arquivos *.json* não são permitidos os *Trailing commas*, e, para contornar essa situação, em cada loop foi adicionado um valor *null* que é retirado justamente nesta parte do programa.

Figura 25 – Parte do arquivo principal do módulo: PHILLIPO.jl

```
function main(
        input_path::String, # Arquivo de entrada (.json)
        output_path::String # Arquivo de saida (.post.res, formato do GiD)
    )
    IOfiles.header_prompt()
    println("Número de threads: $(Threads.nthreads())")
    print("Lendo arquivo JSON...
           ")

    @time input_dict = string(input_path) |> IOfiles.open_parse_input_file

    problem_type = input_dict["type"]
    nodes = input_dict["nodes"]
    materials = input_dict["materials"]
    constraints_forces_nodes    = input_dict["constraints"]["forces_nodes"]
    constraints_forces_lines    = input_dict["constraints"]["forces_lines"]
    constraints_forces_surfaces = input_dict["constraints"]["forces_surfaces"]
    constraints_displacements    = input_dict["constraints"]["displacements"]

    println("Tipo de problema: $(problem_type)")

    # REMOVENDO ELEMENTOS NÃO UTILIZADOS
    # esses elementos nulos são gerados pelo modo que o arquivo JSON é criado pelo GiD
    # É uma falha que deve ser corrigida, mas que não é urgente.
    pop!(nodes)
    pop!(materials)
    pop!(constraints_forces_nodes)
    pop!(constraints_forces_lines)
    pop!(constraints_forces_surfaces)
    pop!(constraints_displacements)

    if isempty(materials) error("Não há nenhum material definido!") end

    # VARIÁVEIS do PROBLEMA
    dimensions = input_dict["type"] == "3D" ? 3 : 2
    nodes_length = length(nodes)
    Fg = zeros(Float64, dimensions * nodes_length)
    Ug = zeros(Float64, dimensions * nodes_length)
```

A próxima parte do arquivo cria os vetores de graus de liberdade prescritos e livres, respectivamente *dof_prescribe* e *dof_free*, e, em seguida, já aplicada as restrições de deslocamentos nodais sobre os graus respectivos no vetor *Ug*, como está na figura 26.

Os graus de liberdade são numerados de acordo com a numeração dos nós sequencialmente, de modo que o nó de número 1 tenha os graus 1, 2 e 3, e um certo nó n, tenha os graus $3n-2$, $3n-1$ e $3n$, para um problema tridimensional. Para o caso bidimensional, a regra é similar: o nó n tem o graus $2n-1$ e $2n$. Essas regras estão implementadas na forma de funções anônimas (linhas 4 e 12).

A função *map* mapeia cada elemento do vetor de restrições de deslocamento para um vetor de graus de liberdade restritos. Para unir esses vetores, a função *reduce* aplica uma função binária, no caso a função *vcat*, entre os elementos do vetor, e retorna um único vetor com todos os graus de liberdade [23].

Figura 26 – Parte do arquivo principal do módulo: PHILLIPO.jl

```
1         # GRAUS DE LIBERDADE: LIVRES E PRESCRITOS
2         if problem_type == "3D"
3             dof_prescribe = reduce(vcat, map(
4                     (x) -> [3 * x[1] - 2, 3 * x[1] - 1, 3 * x[1]],
5                     constraints_displacements
6                 ))
7             dof_free = filter(x -> x  dof_prescribe, 1:dimensions*
    nodes_length)
8             # RESTRIÇÃO DE DESLOCAMENTO
9             Ug[dof_prescribe] = reduce(vcat, map((x) -> [x[2], x[3], x
    [4]], constraints_displacements))
10        else
11            dof_prescribe = reduce(vcat, map(
12                (x) -> [2 * x[1] - 1, 2 * x[1]],
13                constraints_displacements)
14            )
15            dof_free = filter(x -> x  dof_prescribe, 1:dimensions*
    nodes_length)
16            # RESTRIÇÃO DE DESLOCAMENTO
17            Ug[dof_prescribe] = reduce(vcat, map((x) -> [x[2], x[3]],
    constraints_displacements))
18        end
```

A próxima parte do arquivo, figura 27, é o chamamento da função da montagem da matriz de rigidez global *Elements.assemble_stiffness_matrix*, que a retorna no formato CSR para a variável *Kg*.

Em seguida, a função principal aplica as restrições de força(figura 28) de dois modos: diretamente sobre o vetor de forças nodais globais, e, chamando as funções que calculam as forças nodais equivalentes, para os casos de carregamentos sobre linhas e superfícies.

Na aplicação de forças nodais prescritas, é primeiro calculado os graus de liberdade as forças serão aplicadas, num procedimento idêntico ao realizado para as restrições de desloca-

[23] Essa forma de construir essa parte do código é um exemplo de um aspecto de programação funcional.

Figura 27 – Parte do arquivo principal do módulo: PHILLIPO.jl

```
1
2       # CONSTRUÇÃO DOS ELEMENTOS
3       print("Construindo os elementos e a matrix de rigidez global
    paralelamente...")
4       @time Kg = Elements.assemble_stiffness_matrix(input_dict["
    elements"]["linear"], materials, nodes, problem_type)
```

mento. Em seguida, o vetor *Fg* é acessado sobre essas posições dos graus de liberdade para receber todas as forças prescritas na forma de um vetor único, por uma concatenação de vertical de vetores.

As outras formas de restrições de força são aplicadas por meio das funções *Elements.assemble_force_* e *Elements.assemble_force_line!*.

Figura 28 – Parte do arquivo principal do módulo: PHILLIPO.jl

```
1
2       @time if problem_type == "3D"
3           # RESTRIÇÕES DE FORÇA SOBRE NÓS
4           if !isempty(constraints_forces_nodes)
5               dof_constraints_forces_nodes = reduce(vcat, map((x) ->
    [3 * x[1] - 2, 3 * x[1] - 1, 3 * x[1]], constraints_forces_nodes))
6               Fg[dof_constraints_forces_nodes] = reduce(vcat, map((x)
    -> [x[2], x[3], x[4]], constraints_forces_nodes))
7           end
8           # RESTRIÇÃO DE FORÇAS SOBRE SUPERFÍCIES (somente
    TetrahedronLinear)
9           if !isempty(constraints_forces_surfaces)
10              Elements.assemble_force_surface!(Fg, nodes,
    constraints_forces_surfaces)
11          end
12      else
13          # RESTRIÇÕES DE FORÇA SOBRE NÓS
14          if !isempty(constraints_forces_nodes)
15              dof_constraints_forces_nodes = reduce(vcat, map((x) ->
    [2 * x[1] - 1, 2 * x[1]], constraints_forces_nodes))
16              Fg[dof_constraints_forces_nodes] = reduce(vcat, map((x)
    -> [x[2], x[3]], constraints_forces_nodes))
17          end
18          # RESTRIÇÃO DE FORÇAS SOBRE LINHAS (somente TriangleLinear)
19          if !isempty(constraints_forces_lines)
20              Elements.assemble_force_line!(Fg, nodes,
    constraints_forces_lines)
21          end
22      end
```

A próxima parte do arquivo, figura 29, é a resolução do sistema, chamando a função *Solver.direct_solve!*. Como essa função recebe os vetores de deslocamentos e forças nodais globais, não retorna nada, pois já aplicada a resolução do sistema sobre esses vetores. Em seguida, ocorre o cálculo das reações pela função *Stress.reactions*, que retorna o vetor de reações *R* e seu somatórios *Re_sum*, cujo propósito é verificar se o resultado do sistema está em equilíbrio.

O cálculo das tensões é feito pela função *Stress.recovery*, para as variáveis σ e σ_vm, respectivamente, o vetor de tensões na notação de Voigt e o vetor de tensões de von Mises.

Figura 29 – Parte do arquivo principal do módulo: PHILLIPO.jl

```
1
2         println("Resolvendo o sistema de $(size(Kg)) ")
3         @time Solver.direct_solve!(Kg, Ug, Fg, dof_free, dof_prescribe)
4
5         print("Calculando as reações...
                 ")
6         @time Re, Re_sum = Stress.reactions(Kg, Ug, dimensions)
7
8         println("Somatório das reações: $(Re_sum)")
9
10        print("Recuperando as tensões...
                 ")
11        @time , vm = Stress.recovery(input_dict["elements"]["linear"],
    Ug, materials, nodes, problem_type)
```

A última parte do arquivo *PHILLIPO.jl* é a escrita do próprio arquivo de saída *.dat*, por meio do módulo interno *IOfiles*. Nessa etapa é preciso inserir uma série de cabeçalhos no arquivo para que o GID possa interpretá-los corretamente.

Finalizada a escrita sobre o arquivo de saída, a função principal é encerrada.

5.4.2 Elements.jl e paralelismo na montagem da matriz de rigidez global

O módulo *Elements.jl* é divido em duas seções: a definição dos *structs* do tipos de elementos, compostos pelos dados necessários para montar a matriz de rigidez local e uma função de criação, e as funções relativas à montagem da matriz de rigidez e da aplicação dos carregamentos sobre o sistema.

O elemento CST, por exemplo, é definido pelo *struct TriangleLinear*, e contém os seguintes dados na forma de variáveis:

1. ***nodes***: vetor de vetores das coordenadas dos nós ordenados;
2. ***index***: número de identificação do elemento;
3. ***material_index***: número de identificação do material;
4. ***nodes_index***: conectividade;
5. ***interpolation_function_coeff***: coeficientes das funções de interpolação na forma da matriz $\boldsymbol{X}^{-1}$, equação 73;
6. ***D***: matriz constitutiva do material $\boldsymbol{C}$;
7. ***B***: matriz deformação-deslocamento B, equação 86;

8. $\boldsymbol{K}$: matriz de rigidez local;

9. ***degrees_freedom***: graus de liberdade do elemento;

A função homônima *TriangleLinear* é responsável por construir o elemento CST, conforme as relações estabelecidas no capitulo de Elementos Finitos. Ela é uma abstração superior da função *new*, que de fato vai preencher as variáveis listadas acima e retornar o próprio *struc* do elemento específico.

Primeiramente, a função de construção do CST remaneja os dados de entrada para as variáveis do *Struc*: *index*, *material_index* e *nodes_index*, e preenche os as variáveis i, j e m os vetores de coordenadas dos respectivos nós, para, em seguida, construir a matriz $\boldsymbol{X}$ na variável *position_nodes_matrix*. Em seguida, calcula a matriz $\boldsymbol{X}^{-1}$, e a área do elemento, variável Δ. A matriz B é calculada com os os temos da inversa matriz $\boldsymbol{X}^{-1}$, cujas colunas foram distribuídas nas variáveis a, b e c, que representam os termos α, β e γ da equação 73.

Por fim, a função de construção calcula a matriz de rigidez local, K(pela fórmula da equação 96), determina os graus de liberdade do nós que compõe o elemento, na vetor *degrees_freedom*, e, por meio da função *new*, retorna o elemento.

A função *assemble_stiffness_matrix*, figura 31, é responsável pela montagem da matriz de rigidez global paralelamente, utilizando a estrutura de dados COO, para separar o processo em *threads* diferentes.

Na linha 6, a variável *Kg_vector* é criada como um vetor de tamanho equivalente ao número de *threads* disponíveis na sessão Julia. Cada posição desse vetor armazena matrizes no formato COO, que são utilizadas para representar a matriz de rigidez global. O formato COO é particularmente eficiente para matrizes esparsas neste caso porque inserções em posições nulas não fazem com que a matriz precise ser reescrita e realocada na memória, como ocorre com o formato CSR[24], tornando o formato COO uma escolha razoável para a montagem da matriz de rigidez global.

A macro *Threads.@threads* na linha 12 transforma a execução do loop em uma abordagem assíncrona. Isso significa que as iterações do loop são distribuídas dinamicamente entre as *threads* disponíveis em tempo de execução. Cada iteração é colocada em uma fila, ou *pipe*, e as *threads* disponíveis retiram iterações da fila para execução. Esse paralelismo oferece um aproveitamento melhor de sistemas com múltiplos núcleos de processamento.

Dentro do loop, cada elemento é construído pela sua função correspondente, utilizando a estrutura de *strucs*. A função *Matrices.add!* é então utilizada para somar a matriz local do elemento aos graus de liberdade correspondentes na matriz global, armazenada na posição do vetor *Kg_vector* associada à *thread* em execução. É importante destacar que a função *Matrices.add!* não realiza a soma efetiva dos valores, mas sim concatena os novos valores nos vetores que compõem o formato COO implementado. A soma real ocorre posteriormente, após o encerramento

[24] No formato CST, o vetor de valores é ordenados conforme as posições não nulas da matriz.

Figura 30 – *Struct* do elemento CST: TriangleLinear

```
    function TriangleLinear(triangle_element_vector::Vector{Any}, materials::Vector{Any}, nodes::Vector{Any}, problem_type::String)

        index          = Integer(triangle_element_vector[1])
        material_index = Integer(triangle_element_vector[2])
        nodes_index    = Vector{Integer}(triangle_element_vector[3:5])


        i = Vector{Real}(nodes[nodes_index[1]])
        j = Vector{Real}(nodes[nodes_index[2]])
        m = Vector{Real}(nodes[nodes_index[3]])

        position_nodes_matrix = [
            1  i[1]  i[2];
            1  j[1]  j[2];
            1  m[1]  m[2]
        ]

        interpolation_function_coeff = LinearAlgebra.inv(position_nodes_matrix)

         = 1/2 *  LinearAlgebra.det(position_nodes_matrix)

        a = interpolation_function_coeff[1,:]
        b = interpolation_function_coeff[2,:]
        c = interpolation_function_coeff[3,:]


        B = [
            b[1] 0    b[2] 0    b[3] 0   ;
            0    c[1] 0    c[2] 0    c[3];
            c[1] b[1] c[2] b[2] c[3] b[3]
        ]

        try
            materials[material_index]
        catch
            error("Material não definido no elemento de índice: $(index)")
        end

        D = generate_D(problem_type, materials[material_index])

        K = B' * D * B *  * 1

        degrees_freedom = reduce(vcat, map((x) -> [2 * x - 1, 2 * x], nodes_index))

        new(index, material_index, nodes_index, interpolation_function_coeff, D, B, K, degrees_freedom)
    end
end
```

Figura 31 – Função assemble_stiffness_matrix do arquivo: Elements.jl

```
1
2     function assemble_stiffness_matrix(input_elements, materials, nodes,
      problem_type)
3         # Realiza a criação dos elementos e já aplica os valores de
    rigez sobre a matriz global
4
5         # O paralelismo é realizado reservando para cada thread uma
    matriz separada
6         Kg_vector = [Matrices.SparseMatrixCOO() for i = 1:Threads.
    nthreads()]
7
8         if problem_type == "3D"
9             if "tetrahedrons" in keys(input_elements)
10                pop!(input_elements["tetrahedrons"])
11                elements_length = length(input_elements["tetrahedrons"])
12                Threads.@threads for j in 1:elements_length
13                    element = TetrahedronLinear(input_elements["
    tetrahedrons"][j], materials, nodes)
14                    Matrices.add!(
15                        Kg_vector[Threads.threadid()],
16                        element.degrees_freedom,
17                        element.K
18                    )
19                end
20            end
21        else
22            if "triangles" in keys(input_elements)
23                pop!(input_elements["triangles"])
24                elements_length = length(input_elements["triangles"])
25                Threads.@threads for j in 1:elements_length
26                    element = TriangleLinear(input_elements["triangles
    "][j], materials, nodes, problem_type)
27                    Matrices.add!(
28                        Kg_vector[Threads.threadid()],
29                        element.degrees_freedom,
30                        element.K
31                    )
32                end
33            end
34        end
35
36        # A matriz global de rigidez é a soma das matrizes globais
    calculadas em cada thread
37        Kg = Matrices.sum(Kg_vector)
38
39        return Kg
40    end
```

do loop, pela função *Matrices.sum*[25].

O uso de *Threads.threadid* dentro do loop garante que a soma da matriz local seja realizada sequencialmente em cada posição do vetor *Kg_vector* por vez, evitando assim que iterações acessem a mesma posição simultaneamente. Isso é crucial para evitar conflitos de acesso que poderiam resultar em erros na montagem da matriz de rigidez global[26].

Ao final do loop, as matrizes armazenadas no vetor *Kg_vector* são somadas pela função *Matrices.sum*, que retorna a matriz global de rigidez no formato CSR, já sob o tipo *LinearAlgebra.Symmetric*. O formato CSR é utilizado para a matriz global pois é mais eficiente para a resolução do sistema, que é realizada pela função *Solver.direct_solve!*.

5.4.3 Solver.jl

Neste arquivo é feita a divisão da matriz de rigidez global de acordo com os graus de liberdade livres e prescritos, conforme a equação 107. O que vale comentar aqui é a forma como a solução é encontrada, utilizando \. A função \, embora tenha outras utilidades devido ao emprego dos despachos múltiplos em Julia, define o chamamento da resolução de sistemas lineares pelo módulo *LinearAlgebra.jl*, que, analisando as características da matriz, escolhe o método mais adequado para a resolução. Por conta dessa seleção de método que o módulo executa, a matriz global de rigidez é convertida para o formato CSR, que é mais eficiente na resolução de sistemas lineares.

Com ferramentas de debug empregadas, porém, não foi possível determinar com exatidão a forma como a função \ escolhe o método de resolução. Observando o código fonte de *LinearAlgebra.jl*, e a documentação da própria linguagem, é possível concluir que o método aplicado é a decomposição de Cholesky, conforme a descrição da função *LinearAlgebra.cholesky*.

[25] O formato COO não é ordenado, isto é, os valores podem ser inseridos em qualquer ordem e gravados em qualquer ordem pois cada inserção é vinculada a sua posição individualmente.

[26] Existem outras formas de evitar conflitos de acesso simultâneo, como utilizar variáveis *Atomic*.

Figura 32 – Arquivo: Solver.jl

```
# PHILLIPO
# Módulos: funções para executar o método de soluação

module Solver

    using SparseArrays
    using LinearAlgebra
    using InteractiveUtils

    function direct_solve!(
            Kg::SparseMatrixCSC,
            Ug::Vector{<:Real},
            Fg::Vector{<:Real},
            dof_free::Vector{<:Integer},
            dof_prescribe::Vector{<:Integer}
        )
        # Realiza a soluação direta para o sistema
        Ug[dof_free]        = Kg[dof_free, dof_free] \ (Fg[dof_free] - Kg[dof_free, dof_prescribe] * Ug[dof_prescribe])
        Fg[dof_prescribe]   = Kg[dof_prescribe, dof_free] * Ug[dof_free] + Kg[dof_prescribe, dof_prescribe] * Ug[dof_prescribe]

    end

end
```

6 VALIDAÇÃO & VERIFICAÇÃO

O desenvolvimento de softwares de simulação, seja utilizando o MEF ou não, é sempre acompanhado de uma bateria de testes, além de um procedimento de validação e verificação (V & V), que garante, dentro de uma margem de abrangência do que se propõe, sua capacidade de reproduzir resultados concisos, sendo, então, um espelho da realidade.

Verificação, dentro desse contexto, é o procedimento pelo qual se evidencia a exata implementação do modelo matemático no próprio software, ou seja, a verificação que a modelagem programada é equivalente ao algoritmo matemático, dentro dos limites impostos pela aritmética computacional em relação às operações em ponto flutuante[1]. Já a validação, por sua vez, é o processo para determinar a acurácia que um modelo computacional possui de presentar a realidade, dentro dos limites que se propõe. Essas definições estão de acordo com o documento *An Overview of the Guide for Verification and Validation in Computational Solid Mechanics*, referente a norma ASME respectiva.

Por questões de simplificação, a verificação foi realizada por comparação com outro programa que realiza a mesma análise, o Abaqus, e a validação, por comparação com resultados analíticos, já consolidados experimentalmente.

6.1 VERIFICAÇÃO

Para verificar os resultados foram utilizados seis sequências de comparação (três com o elemento tetraédrico, e três triangulares), nas quais aplicaram-se todas as condições de contorno (deslocamentos prescritos, carregamentos pontuais, em linhas e superficiais). Os resultados foram comparados com os obtidos no Abaqus, nas mesmas condições de contorno e geometrias, para deslocamentos nodais e tensões sobre os elementos. As geometrias empregadas foram simples, com malhas pouco refinadas, a fim de facilitar a comparação sobre todos os nós e elementos.

Os arquivos de verificação estão disponíveis no repositório do projeto, no pasta *verification*, na qual constam os arquivos de entrada do Abaqus (*.cae*) e do GID (*.gid*), além dos arquivos respectivos arquivos de saída, (*.rpt* e *.post.res*)

Em todos os casos, definiu-se o aço ASIS 4340 como material, cujas características físicas empregadas aqui foram: $E = 210\text{GPa}$ (módulo de elasticidade) e $\nu = 0.3$ (coeficiente de Poisson).

O carregamento distribuído $\boldsymbol{t}$

Primeiramente, o problema é definido no Abaqus, para depois ser exportado para o GID, a fim de se manter tanto a geometria como a malha e, principalmente, as identificações dos nós e elementos, o que facilitada a comparação dos resultados. O procedimento empregado em todos os casos foi o seguinte:

[1] No computador, os números ditos Reais ($\mathbb{R}$) são representandos por um ponto flutuante, que é uma forma discreta, pois os computadores são desenvolvidos em lógica booleana

1. Definição do problema no Abaqus (*.cae*);
    a) Construção da geometria;
    b) Definição do material (AISI 4340 STEEL);
    c) Construção da malha;
    d) Definição das condições de contorno;
2. Execução da análise;
3. Exportação dos resultados para um arquivo de texto (*.rpt*);
    a) utilização da ferramenta *probe* sobre os nós (deslocamentos nodais);
    b) utilização da ferramenta *probe* sobre os elementos (componentes das tensões e tensão de von Mises);
4. Importação do arquivo de *input* no GID (*.inp*);
5. Definição do problema no GID;
    a) Definição do material (AISI 4340 STEEL);
    b) Definição das condições de contorno;
6. Execução da análise;
7. Exportação do arquivo de resultados (*.post.res*);
8. Comparação dos resultados.

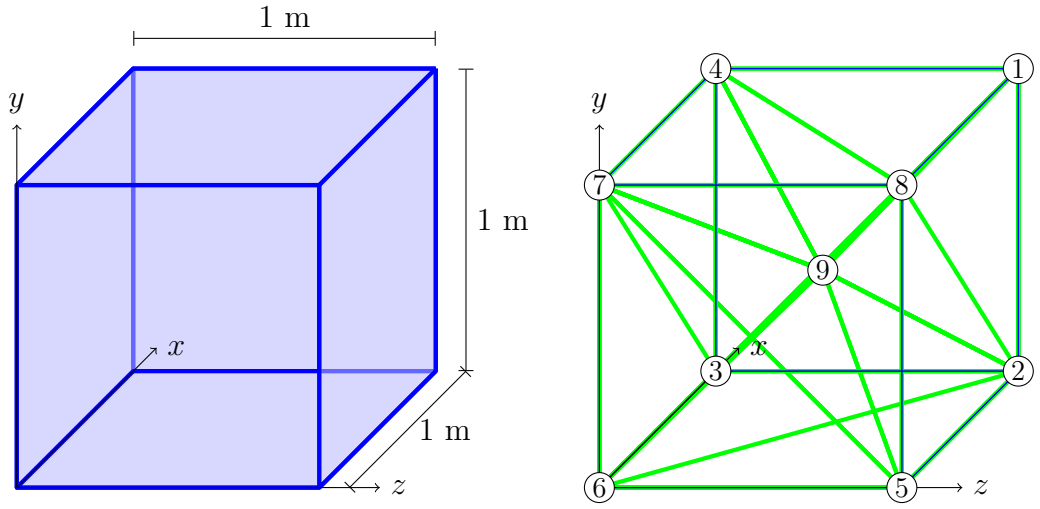
As análises podem ser repetidas diretamente utilizando os arquivos *.inp*, para dar entrada com dados do problema no Abaqus, e os arquivos *.dat* (dentro da pasta *.gid*), para dar entrada com dados do problema em PHILLIPO. No Abaqus é necessário importar o modelo, e no PHILLIPO é necessário chamar a função principal do módulo indicando a localização do arquivo *.dat*, para que não seja preciso utilizar a interface do GID. A numeração dos casos de verificação segue a mesma sequência de apresentação deste capítulo.

6.1.1 Casos tridimensionais (elemento tetraédrico)

Nos casos tridimensionais foi empregado a geometria de um cubo unitário, conforme a figura 33a, com a malha descrita na tabela 1 (figura 33b) e conectividade na tabela 2. A face formada pelos nós 3, 4, 7 e 6 foi engastada, e as demais condições de contorno foram aplicadas nas seguintes sequências:

1. Carregamento superficial (figura 34a), resultados nas tabelas 3 e 4;

Figura 33 – O cubo unitário.



(a) Geometria (b) Malha

Tabela 1 – Descrição dos nós para a malha do cubo unitário.

Nó	**x** [m]	**y** [m]	**z** [m]
1	1.00	1.00	1.00
2	1.00	0.00	1.00
3	1.00	0.00	0.00
4	1.00	1.00	0.00
5	0.00	0.00	1.00
6	0.00	0.00	0.00
7	0.00	1.00	0.00
8	0.00	1.00	1.00
9	4.929074e-01	5.337674e-01	5.471772e-01

2. Carregamento pontual (figura 34b), resultados nas tabelas 5 e 6;

3. Deslocamento prescrito (figura 34c), resultados nas tabelas 7 e 8.

Em todas as sequências de comparação para o caso tridimensionais, os resultados exportados do Abaqus convergiram com os produzidos por PHILLIPO, com exceção dos deslocamentos prescritos, que não são definidos exatamente zero nos engastes, o que pode ser devido ao procedimento de solução do sistema no Abaqus, por algum método iterativo ou de penalização, porém essa diferença é desprezível.

Destes modo, fica demonstrado que algoritmo matemático do ME foi implementado corretamente nessas condições.

Tabela 2 – conectividade dos elementos na malha do cubo unitário.

Elemento	**Nó 1**	**Nó 2**	**Nó 3**	**Nó 4**
1	2	6	5	9
2	9	6	5	7
3	7	9	8	5
4	6	7	9	3
5	4	9	1	8
6	3	4	9	1
7	3	7	9	4
8	3	9	2	1
9	5	9	8	2
10	9	2	1	8
11	6	9	2	3
12	7	9	4	8

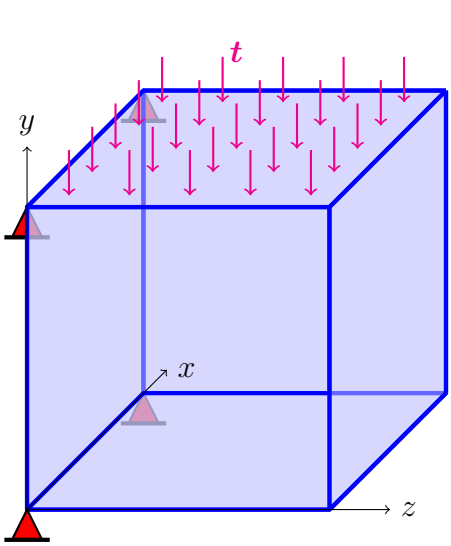
Tabela 3 – Deslocamentos nodais para o primeiro caso tridimensional.

Nó	**PHILLIPO.jl**		
	x [m]	**y** [m]	**z** [m]
1	8.65925×10^{-8}	-8.92024×10^{-6}	1.71092×10^{-6}
2	2.34872×10^{-7}	-7.50586×10^{-6}	-2.45344×10^{-6}
3	0.0	0.0	0.0
4	0.0	0.0	0.0
5	-4.97641×10^{-7}	-7.52208×10^{-6}	-1.51181×10^{-6}
6	0.0	0.0	0.0
7	0.0	0.0	0.0
8	-2.34375×10^{-7}	-1.04911×10^{-5}	3.58197×10^{-6}
9	-1.43153×10^{-7}	-4.38813×10^{-6}	1.65793×10^{-7}

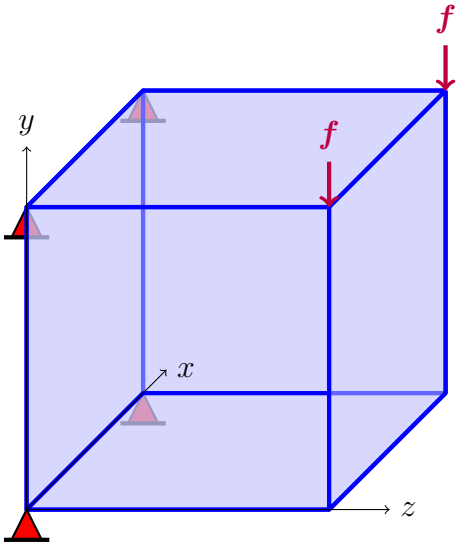
Tabela 4 – Tensões sobre os elementos para o primeiro caso tridimensional.

ID	**PHILLIPO.jl** [Pa]					
	σ_x	σ_y	σ_z	σ_{xy}	σ_{xz}	σ_{zy}
1	-3.96885×10^{4}	-2.42821×10^{5}	-4.02234×10^{5}	-3.37838×10^{4}	-3.87057×10^{5}	-1.16248×10^{5}
2	-1.09096×10^{5}	-1.51419×10^{5}	-3.95636×10^{5}	-4.46066×10^{4}	-6.07553×10^{5}	1.22526×10^{5}
3	1.36097×10^{5}	-3.78867×10^{5}	6.79384×10^{5}	1.60388×10^{4}	-4.35943×10^{5}	7.62264×10^{4}
4	3.67093×10^{4}	3.67093×10^{4}	8.56550×10^{4}	0.00000	-6.47735×10^{5}	-2.11311×10^{4}
5	-3.70490×10^{4}	-5.35656×10^{5}	1.87482×10^{5}	1.31695×10^{5}	-4.22653×10^{5}	-1.44130×10^{5}
6	3.13504×10^{5}	2.52807×10^{5}	5.29187×10^{5}	-7.84963×10^{4}	-7.20481×10^{5}	1.29700×10^{5}
7	3.67093×10^{4}	3.67093×10^{4}	8.56550×10^{4}	0.00000	-6.47735×10^{5}	-2.11311×10^{4}
8	-3.61274×10^{5}	-6.51080×10^{5}	-8.18929×10^{5}	-8.74518×10^{4}	-2.69891×10^{5}	1.32782×10^{5}
9	1.77784×10^{3}	-5.96175×10^{5}	8.93419×10^{4}	2.25736×10^{4}	-4.28830×10^{5}	-4.98169×10^{4}
10	6.71645×10^{4}	-2.13160×10^{5}	2.12365×10^{5}	1.14907×10^{5}	-4.96502×10^{5}	-1.26844×10^{5}
11	-3.61046×10^{5}	-4.46115×10^{5}	-7.57371×10^{5}	-4.11090×10^{4}	-3.78014×10^{5}	1.89704×10^{4}
12	8.25382×10^{4}	-3.86038×10^{5}	6.61165×10^{5}	2.58282×10^{3}	-5.36544×10^{5}	-1.89303×10^{4}

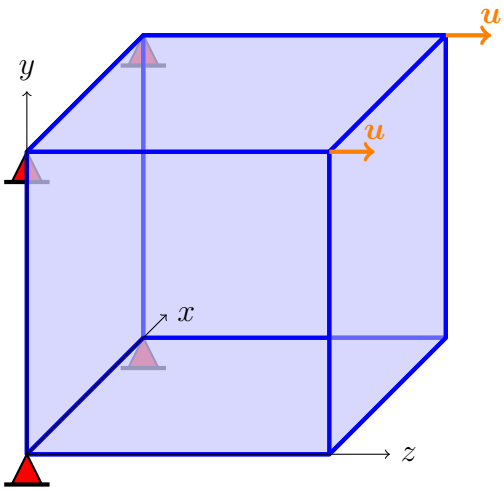
Figura 34 – Condições de contorno para os casos tridimensionais.



(a) Carregamento superficial



(b) Carga concentrada



(c) Deslocamento prescrito

Tabela 5 – Deslocamentos nodais para o segundo caso tridimensional.

Nó	**PHILLIPO.jl**		
	x [m]	**y** [m]	**z** [m]
1	1.02738×10^{-6}	-2.02588×10^{-5}	3.00251×10^{-6}
2	-4.34358×10^{-7}	-1.56996×10^{-5}	-4.85199×10^{-6}
3	0.0	0.0	0.0
4	0.0	0.0	0.0
5	-1.19159×10^{-6}	-1.38869×10^{-5}	-2.90123×10^{-6}
6	0.0	0.0	0.0
7	0.0	0.0	0.0
8	4.4546×10^{-7}	-1.89878×10^{-5}	6.81349×10^{-6}
9	-3.03331×10^{-7}	-8.53913×10^{-6}	1.49066×10^{-7}

Tabela 6 – Tensões sobre os elementos para o segundo caso tridimensional.

ID	**PHILLIPO.jl** [Pa]					
	σ_x	σ_y	σ_z	σ_{xy}	σ_{xz}	σ_{zy}
1	-1.48107×10^5	-2.84665×10^5	-7.39090×10^5	-1.50128×10^5	-7.13362×10^5	-2.53806×10^5
2	-1.51519×10^5	-2.65791×10^5	-7.34452×10^5	-1.54118×10^5	-1.12164×10^6	1.88313×10^5
3	3.31474×10^5	-5.63353×10^5	1.36127×10^6	4.57622×10^4	-7.48975×10^5	1.91682×10^5
4	3.30055×10^4	3.30055×10^4	7.70128×10^4	0.0	-1.26047×10^6	-4.47749×10^4
5	3.41582×10^4	-7.18663×10^5	4.25177×10^5	-3.84442×10^3	-1.04271×10^6	-2.24829×10^5
6	8.46258×10^5	5.70548×10^5	1.05557×10^6	-4.05526×10^5	-1.63629×10^6	3.20919×10^5
7	3.30055×10^4	3.30055×10^4	7.70128×10^4	0.0	-1.26047×10^6	-4.47749×10^4
8	-6.68634×10^5	-1.67458×10^6	-1.72188×10^6	-2.77724×10^5	-6.33643×10^5	1.86079×10^5
9	-8.99318×10^4	-1.03624×10^6	2.06398×10^5	-1.41868×10^4	-8.14231×10^5	-9.35646×10^4
10	-8.71417×10^4	-9.17625×10^5	2.19806×10^5	1.54039×10^4	-9.61902×10^5	-2.44647×10^5
11	-5.76185×10^5	-5.60648×10^5	-1.35997×10^6	-9.93562×10^3	-8.43752×10^5	-3.50827×10^4
12	3.44607×10^5	-2.96558×10^5	1.44525×10^6	9.47744×10^4	-9.13586×10^5	3.59795×10^4

Tabela 7 – Deslocamentos nodais para o terceiro caso tridimensional.

Nó	**PHILLIPO.jl**		
	x [m]	**y** [m]	**z** [m]
1	0.00000×10^0	0.00000×10^0	0.10000×10^0
2	-5.32004×10^{-3}	-1.33670×10^{-3}	-1.85191×10^{-3}
3	0.00000×10^0	0.00000×10^0	0.00000×10^0
4	0.00000×10^0	0.00000×10^0	0.00000×10^0
5	1.45540×10^{-2}	1.22280×10^{-2}	2.39641×10^{-2}
6	0.00000×10^0	0.00000×10^0	0.00000×10^0
7	0.00000×10^0	0.00000×10^0	0.00000×10^0
8	0.00000×10^0	0.00000×10^0	0.10000×10^0
9	1.19228×10^{-3}	1.15650×10^{-2}	2.80404×10^{-2}

Tabela 8 – Tensões sobre os elementos para o terceiro caso tridimensional.

ID	**PHILLIPO.jl** [Pa]					
	σ_x	σ_y	σ_z	σ_{xy}	σ_{xz}	σ_{zy}
1	-9.09572×10^7	6.61804×10^9	6.99059×10^9	-6.37915×10^8	5.17203×10^9	-9.09626×10^8
2	-9.80149×10^8	1.239×10^9	5.11012×10^9	7.98683×10^8	9.87649×10^8	3.62163×10^9
3	7.42606×10^9	7.28382×10^9	2.5413×10^{10}	-2.14641×10^8	6.14136×10^9	1.43759×10^9
4	6.2086×10^9	6.2086×10^9	1.44867×10^{10}	0.0	1.70712×10^9	1.75993×10^8
5	9.11014×10^9	5.10314×10^9	2.5264×10^{10}	-2.06548×10^8	4.62152×10^9	0.0
6	1.14507×10^{10}	1.18305×10^{10}	2.79844×10^{10}	-1.84206×10^9	0.0	4.24913×10^9
7	6.2086×10^9	6.2086×10^9	1.44867×10^{10}	0.0	1.70712×10^9	1.75993×10^8
8	-7.66859×10^8	-1.48395×10^8	-6.63477×10^8	-1.41522×10^9	8.11854×10^9	3.60189×10^9
9	-7.36171×10^8	4.98949×10^8	1.0959×10^{10}	-2.27113×10^9	3.90283×10^9	-2.83479×10^9
10	6.70973×10^9	6.92566×10^9	1.54401×10^{10}	4.29695×10^8	6.05251×10^9	-6.55085×10^8
11	2.56666×10^9	6.28802×10^9	2.2675×10^9	6.20905×10^8	4.28842×10^9	-4.29695×10^8
12	9.11014×10^9	5.10314×10^9	2.5264×10^{10}	-2.06548×10^8	4.62152×10^9	0.0

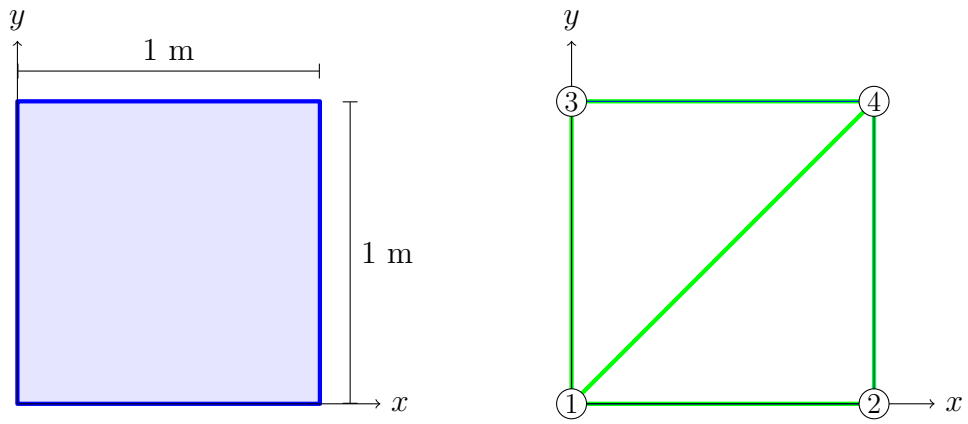
Tabela 9 – Descrição dos nós para a malha do quadrado unitário.

Nó	**x** [m]	**y** [m]
1	0.000000e+00	0.000000e+00
2	1.000000e+00	0.000000e+00
3	0.000000e+00	1.000000e+00
4	1.000000e+00	1.000000e+00

Tabela 10 – conectividade dos elementos na malha do quadrado unitário.

Elemento	**Nó 1**	**Nó 2**	**Nó 3**
1	1	2	4
2	4	3	1

Figura 35 – O quadrado unitário.



(a) Geometria (b) Malha

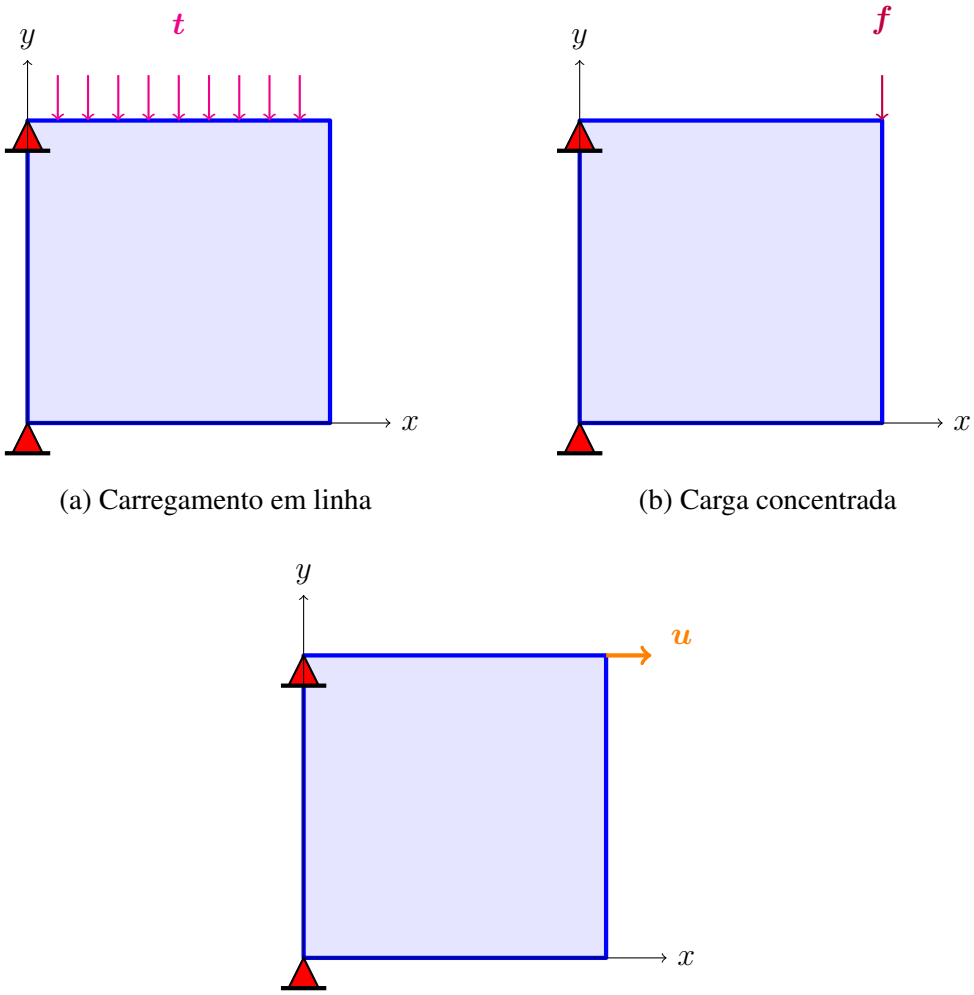
6.1.2 Casos bidimensionais (elemento triangular)

Nos casos bidimensionais foi empregado a geometria de um quadrado unitário, conforme a figura 35a, com a malha descrita na tabela 9 (figura 33b) e conectividade na tabela 10. A fronteira formada pelos nós 1 e 3, foi engastada, e as demais condições de contorno foram aplicadas nas seguintes sequências, dentro da análise do EPT:

1. Carregamento em linha (figura 36a), resultados nas tabelas 11 e 12;
2. Carregamento pontual (figura 36b), resultados nas tabelas 13 e 14;
3. Deslocamento prescrito (figura 34c) resultados nas tabelas 15 e 16.

Em todas as sequências de comparação para o caso bidimensionais, os resultados exportados do Abaqus convergiram com os produzidos por PHILLIPO, com exceção, tal qual nos casos tridimensionais, dos deslocamentos prescritos, que não são definidos exatamente zero nos engastes, e, da mesma forma, essa diferença é desprezível.

Figura 36 – Condições de contorno para os casos bidimensionais.



(a) Carregamento em linha

(b) Carga concentrada

(c) Deslocamento prescrito

Tabela 11 – Deslocamentos nodais para o primeiro caso bidimensionais.

Nó	**PHILLIPO.jl**	
	x [m]	**y** [m]
1	0.0	0.0
2	-1.15402×10^{-6}	-6.94059×10^{-6}
3	0.0	0.0
4	1.50022×10^{-6}	-8.09460×10^{-6}

Tabela 12 – Tensões sobre os elementos para o primeiro caso tridimensional.

ID	**PHILLIPO.jl** [Pa]		
	σ_x	σ_y	σ_{xy}
1	-3.46205×10^5	-3.46205×10^5	-3.46205×10^5
2	3.46205×10^5	1.03862×10^5	-6.53794×10^5

Tabela 13 – Deslocamentos nodais para o segundo caso bidimensionais.

Nó	**PHILLIPO.jl**	
	x [m]	**y** [m]
1	0.0	0.0
2	-1.15402×10^{-6}	-6.94059×10^{-6}
3	0.0	0.0
4	1.50022×10^{-6}	-8.09460×10^{-6}

Tabela 14 – Tensões sobre os elementos para o segundo caso tridimensional.

ID	**PHILLIPO.jl** [Pa]		
	σ_x	σ_y	σ_{xy}
1	-3.46205×10^5	-3.46205×10^5	-3.46205×10^5
2	3.46205×10^5	1.03862×10^5	-6.53794×10^5

Tabela 15 – Deslocamentos nodais para o terceiro caso bidimensionais.

Nó	**PHILLIPO.jl**	
	x [m]	**y** [m]
1	0.0	0.0
2	1.75000×10^{-2}	-1.75000×10^{-2}
3	0.0	0.0
4	1.00000×10^{-1}	0.0

Destes modo, fica demonstrado que algoritmo matemático do MEF foi implementado corretamente nessas condições.

Tabela 16 – Tensões sobre os elementos para o terceiro caso tridimensional.

ID	**PHILLIPO.jl** [Pa]		
	σ_x	σ_y	σ_{xy}
1	5.25000×10^9	5.25000×10^9	5.25000×10^9
2	2.30769×10^{10}	6.92308×10^9	0.0

6.2 VALIDAÇÃO

Para validar os resultados obtidos pelo PHILLIPO.jl, foi utilizado a comparação com a solução analítica um problema de viga longa, na teoria de Euler-Bernoulli, analisando o deslocamento vertical sobre a linha elástica, que passa no centro da viga.

A viga longa é um problema clássico, que consiste em simplificações da teoria da elasticidade aplicada diretamente sobre vigas. Nesse contexto, os deslocamentos da linha elástica da viga é modelado pela equação diferencial de Euler-Bernoulli, dada por:

$$EIv^{(4)}(x) = q(x), \tag{131}$$

em que E é o módulo de elasticidade do material, I é o momento de inércia da seção transversal da viga, $v(x)$ é o deslocamento vertical da linha elástica, $q(x)$ é a carga normal a sua superfície, x é a posição ao longo da viga. Para determinar a distribuição $v(x)$ é necessário conhecer as condições de contorno que descrevem a liberdade da viga. (HIBBELER, 2010)

Seja uma viga $\mathscr{B}$, como na figura 37a, composta pelo aço AISI, de $E = 210$GPa (módulo de elasticidade) e $\nu = 0.3$, e que está engastada em uma extremidade e livre na outra, e sujeita a um carregamento distribuído $\boldsymbol{t} = -100$ kPa $\hat{\boldsymbol{y}}$, conforme a figura 37b. A viga possui uma seção transversal quadrada, logo seu momento de inércia é dado por $I = \frac{b^4}{12}$, em que b é a largura da seção transversal da viga.

Para essas condições de contorno, a solução analítica para o deslocamento vertical $v(x)$ é dada por, de acordo com Hibbeler (2010):

$$v(x) = -\frac{t_0 x^2}{24EI}\left(x^2 - 4Lx + 6L^2\right), \tag{132}$$

em que L é o comprimento da viga.

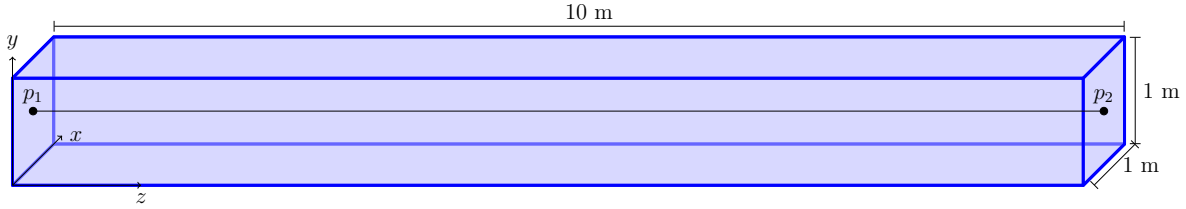
No GID, a geometria da viga foi construída utilizando quatro paralelepípedos idênticos, conforme a figura 37c, para que fosse possível extrair o deslocamento sobre a linha elástica de forma direta. Foram geradas malhas não estruturas com quantidade de nós diferentes. Os resultados de deslocamento da linha elástico foram plotados na figura 38, como também a solução analítica.

É possível notar que os resultados obtidos pelo PHILLIPO.jl, no caso tridimensional, convergem para a solução analítica, conforme a malha é refinada, o que demonstra que o algoritmo matemático implementado é válido para representar o problema físico.

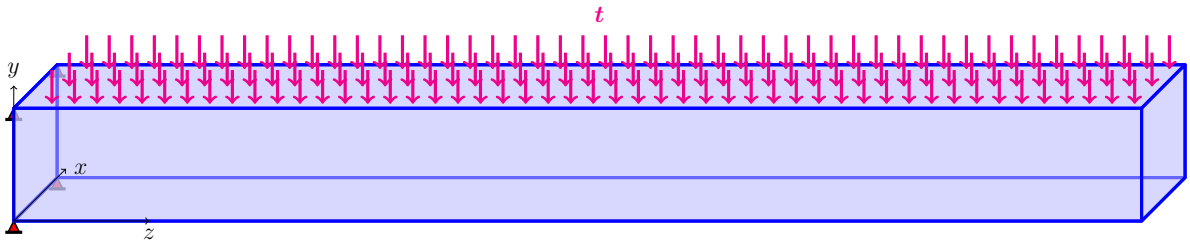
A mesma análise foi realizada utilizando elementos CST, modelando a viga por um EPT, cujo resultado consta na figura 39.

É possível notar que a modelagem por CST converge mais rápido, para o total de nós utilizado em cada análise, que a feita por elementos tetraédricos. Isso é porque no caso tridimensional os nós são postos sobre toda a geometria da viga, e no caso bidimensional, os nós

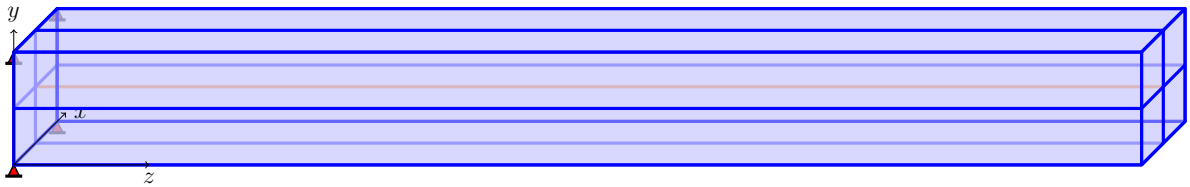
Figura 37 – Uma viga longa.



(a) Geometria



(b) Condições de contorno



(c) Geometria no GID

Figura 38 – Comparação dos deslocamentos da linha elástica da viga longa com elementos tetraédricos.

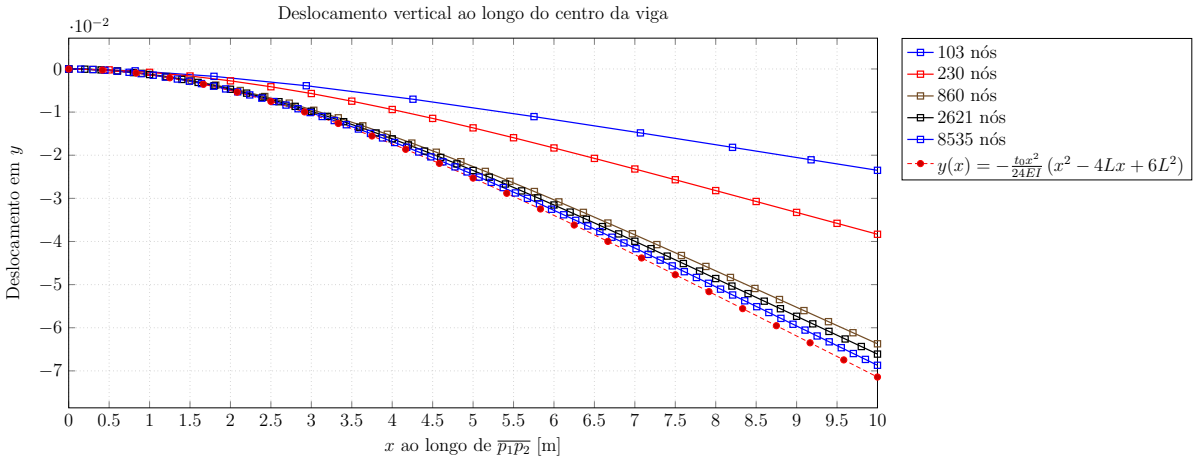
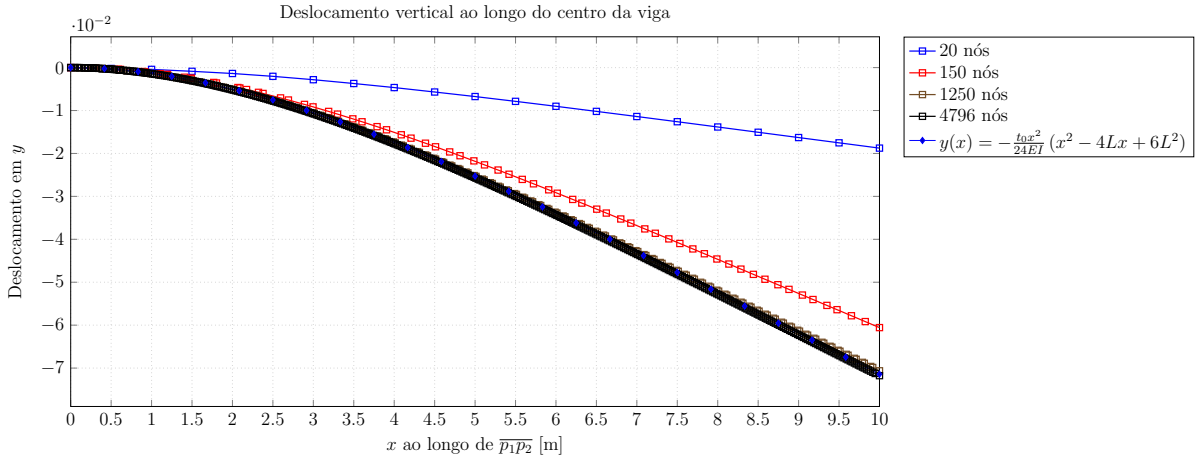


Figura 39 – Comparação dos deslocamentos da linha elástica da viga longa com elementos CST.



são postos apenas sobre um plano e os efeitos transversais são definidos pelo EPT. Assim, a estrutura é melhor discretizada no plano com menos nós, obtendo resultado mais próximo da solução analítica.

7 CONCLUSÃO E PROPOSTAS DE MELHORIAS EM PROJETOS FUTUROS

Ao longo do desenvolvimento do projeto, todos os objetivos específicos foram alcançados. O MEF foi devidamente compreendido e aplicado na análise de estruturas sólidas elásticas sob carregamentos contantes, por meio dos elemento triangulares e tetraédricos lineares. Os algoritmos do MEF foram implementados com sucesso na linguagem Julia, em um programa, nomeado PHILLIPO.jl, distribuível, e empacotado pelo gerenciador *Pkg.jl*. Nesse programa, foi aplicado o processamento paralelo na montagem da matriz de rigidez global. E, por fim, os resultados obtidos foram verificados com o software comercial Abaqus, e validados com o modelo de viga Euler-Bernoulli. Com exceção da função de resolução do sistema não reconhecer a simetria da matriz de rigidez global, todos os outros algoritmos foram implementados como planejado.

O programa, entretanto, tem grande margem para melhorias, tanto em termos de desempenho e funcionalidades, quanto em legibilidade. A seguir, são listadas algumas propostas de melhorias para projetos futuros no desenvolvimento de PHILLIPO.jl:

1. **Estudar a estrutura de implementações já consolidadas do MEF.** Estudar outras implementações do MEF, especialmente em Julia, é um modo produtivo de se aprimorar o programa, observando os pontos fortes e fracos de suas estruturas. Módulos como o *JuliaFEM.jl* vem sendo desenvolvidos há anos, já possuem uma comunidade ativa considerável.

2. **Aprimorar o empacotamento e a estrutura do programa.** O arquivo principal de PHILIPO.jl ainda se encontra com várias etapas do MEF aplicadas explicitamente, o que prejudica a leitura do código. Cálculos de graus de liberdade, estruturas condicionais para distinguir entre casos bidimensionais e tridimensionais são alguns exemplos de procedimentos que poderiam estar em funções separadas, para serem apenas chamadas na função principal, o que facilitaria a leitura e a manutenção do código. A estrutura do programa também pode ser revista, com o intuito de facilitar a implementação de novos elementos e, até, novas equações de governo e novos tipos de análise.

3. **Implementar melhores formas de debug e mensagens de erro.** O debug e as mensagens de erro foram implementados utilizando impressões no terminal, concatenadas no arquivo *.log*, o que é não é um prática recomendável em Julia. Na própria documentação da linguagem, é indicado um conjunto de macros para essas finalidades: *@warn* e *@debug*, justamente para para facilitar a leitura e manutenção do código.

4. **Aplicar mais o paradigma de despachos múltiplos.** Embora esse é um ponto notável na linguagem Julia, o programa ainda não faz uso de todo o potencial dessa característica. A função de montagem da matriz global de rigidez poderia ser implementada sem a utilização de estruturas condicionais, o que tornaria o código mais legível e reaproveitável,

ao passo que a inclusão de outros tipos de elementos não necessitaria de alterações nessa função.

5.

6. **Implementar outros procedimento de resolução de sistemas lineares.** Outras formas de resolução de sistemas lineares, principalmente procedimento iterativos, poderiam ser implementados, como também a utilização de bibliotecas especializadas (*IterativeSolvers.jl*).

7. **Implementar integração com outras interfaces.** O GID, embora muito poderoso para gerar malhas e plotar resultados, é limitado pela sua licença (não é um projeto open-source), e pela sua comunidade que não é muito ativa. O GMSH e o Salome se mostram como alternativas viáveis e gratuitas.

8. **Realizar testes de desempenho.** Embora o programa tenha aplicado matrizes esparsas e processamento paralelo, não foi realizado nenhum teste formal de desempenho, comparando resultados com outros programas similares.

9. **Implementar de outros tipos de elementos.** Este projeto se ateve a implementar elementos simples, mas, com algumas alterações, é possível adicionais elementos mais complexos em PHILLIPO.jl, o que implicaria, dentre outras coisas, na criação de um novos módulos internos para abarcar novas funcionalidade, como transformação de sistemas de referência e integração numérica.

10. **Implementar de outros tipos de análise.** Não somente análises de deformação e tensão podem ser feitas com o MEF, mas virtualmente de qualquer fenômeno físico que possa ser descrito por equações diferenciais, como condução de calor e vibrações.

11. **Criar uma documentação mais didática e completa.** Como o objetivo deste trabalho foi expor o MEF em Julia, não foi abordado minuciosamente todos os aspectos do programa, como poderia ter sido realizado em um projeto de maior escopo. Uma proposta futura interessante é a criação de um documento mais completo que, além de explicar o funcionamento detalhado do programa, traria tópicos da própria linguagem Julia.

REFERÊNCIAS

BALBAERT, Ivo. **Getting Started with Julia Programming: Enter the exciting world of Julia, a high-performance language for technical computing**. Packt Publishing, 2015. ISBN 978-1-78328-479-5. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=1b9193371c3e5b470d94a7e1a77eb769>. Citado na página 60.

BEZANSON, Jeff; CHEN, Jiahao; CHUNG, Benjamin; KARPINSKI, Stefan; SHAH, Viral B.; VITEK, Jan; ZOUBRITZKY, Lionel. Julia: Dynamism and performance reconciled by design. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 2, n. OOPSLA, oct 2018. Disponível em: <https://doi.org/10.1145/3276490>. Citado na página 18.

DUFF A. M. ERISMAN, J. K. Reid I. S. **Direct Methods for Sparse Matrices**. First paperback edition. Oxford University Press, USA, 1989. (Numerical Mathematics and Scientific Computation). ISBN 9780198534211,0198534213. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=406256717d0534af510d4ea17d8b57e7>. Citado na página 58.

HIBBELER, Russell Charles. **Resistências dos Materiais**. São Paulo: Pearson Prentice Hall, 2010. Citado 3 vezes nas páginas 34, 35 e 93.

LOGAN, Daryl L. **A First Course in the Finite Element Method, Enhanced Edition, SI Version**. 6. ed. Cengage Learning, 2022. ISBN 0357676432,9780357676431. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=C987600444DEED4576ED20233CC49A72>. Citado 7 vezes nas páginas 38, 41, 42, 43, 45, 49 e 58.

LUBLINER, Panayiotis Papadopoulos (auth.) Jacob. **Introduction to Solid Mechanics: An Integrated Approach**. Springer International Publishing, 2017. ISBN 978-3-319-18878-2,978-3-319-18877-5. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=43b5224440ffc99f4b40d894dcbb2bdc>. Citado na página 31.

MUFTU, Sinan. **Finite Element Method: Physics and Solution Methods**. 1. ed. Academic Press, 2022. ISBN 012821127X,9780128211274. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=19BB81D50D51BD23CC315B2F8E43F4C0>. Citado na página 49.

OñATE, Eugenio. **Structural Analysis with the Finite Element Method. Linear Statics: Volume 1: Basis and Solids (Lecture Notes on Numerical Methods in Engineering and Sciences) (v. 1)**. 1. ed. [s.n.], 2009. ISBN 1402087322,9781402087325. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=589bba29f786a93857f01ff9d12136cc>. Citado 4 vezes nas páginas 17, 37, 50 e 56.

POPOV, Egor P. **Engineering Mechanics of Solids**. Prentice Hall, 1990. (Prentice-Hall International Series in Civil Engineering and Engineering Mechanics). ISBN 0-13-279258-3,9780132792585. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=dab102c9ca4bd8e45556ebcd62b53b57>. Citado 4 vezes nas páginas 20, 21, 26 e 27.

QUEK, S. S.; LIU, G.R. **The Finite Element Method: A Practical Course**. Butterworth-Heinemann, 2003. ISBN 9780750658669,9781417505593,0750658665. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=85760afdc4189ab75d846ee5fd53d6aa>. Citado 2 vezes nas páginas 17 e 37.

RAO, Singiresu S. **The finite element method in engineering**. 6. ed. ed. Elsevier, 2018. ISBN 978-0-12-811768-2,0128117680. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=e7ef3750d9f7ed6b4cb908c1215ba393>. Citado 2 vezes nas páginas 50 e 51.

ROYLANCE, D. **Mechanics of Materials: Introduction to Elasticity**. Massachusetts Institute of Technology, 1995. Disponível em: <https://books.google.com.br/books?id=nP54tAEACAAJ>. Citado 3 vezes nas páginas 21, 23 e 29.

SHERRINGTON, Malcolm; BALBEART, Ivo; SENGUPTA, Avik. **Mastering Julia: Develop your analytical and programming skills further in Julia to solve complex data processing problems**. Packt Publishing, 2015. ISBN 978-1-78355-331-0. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=de5338c3a3b90bba52c20f544bd71456>. Citado na página 57.

TROBEC, Roman; SLIVNIK, Botjan; BULI, Patricio. **Introduction to Parallel Computing. From Algorithms to Programming on State-of-the-Art Platforms**. Springer, 2018. ISBN 978-3-319-98833-7. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=544b285fceb1933b62105e31fcc412fe>. Citado na página 59.

ZIENKIEWICZ, O.C. **the Finite Element Method**: Volume 1: The basis. [S.l.]: Butterworth-Heinemann, 2000. Citado 4 vezes nas páginas 17, 34, 43 e 49.

ANEXO A – CÓDIGO FONTE DE PHILLIPO.JL

./src/PHILLIPO.jl

```
# MÓDULOS EXTERNOS
import LinearAlgebra
import SparseArrays

# MÓDULOS INTERNOS
import .IOfiles
import .Elements
import .Solver
import .Matrices
import .Stress

# PONTO DE PARTIDA (aqui inicia a execução)
function main(
        input_path::String, # Arquivo de entrada (.json)
        output_path::String # Arquivo de saida (.post.res, formato do GiD)
    )
    IOfiles.header_prompt()
    println("Número de threads: $(Threads.nthreads())")
    print("Lendo arquivo JSON...
            ")

    @time input_dict = string(input_path) |> IOfiles.open_parse_input_file

    problem_type = input_dict["type"]
    nodes = input_dict["nodes"]
    materials = input_dict["materials"]
    constraints_forces_nodes    = input_dict["constraints"]["forces_nodes"]
    constraints_forces_lines    = input_dict["constraints"]["forces_lines"]
    constraints_forces_surfaces = input_dict["constraints"]["forces_surfaces"]
    constraints_displacements    = input_dict["constraints"]["displacements"]

    println("Tipo de problema: $(problem_type)")

    # REMOVENDO ELEMENTOS NÃO UTILIZADOS
    # esses elementos nulos são gerados pelo modo que o arquivo JSON é criado pelo GiD
    # É uma falha que deve ser corrigida, mas que não é urgente.
    pop!(nodes)
```

```
37      pop!(materials)
38      pop!(constraints_forces_nodes)
39      pop!(constraints_forces_lines)
40      pop!(constraints_forces_surfaces)
41      pop!(constraints_displacements)
42
43      if isempty(materials) error("Não há nenhum material definido!")
    end
44
45      # VARIÁVEIS do PROBLEMA
46      dimensions = input_dict["type"] == "3D" ? 3 : 2
47      nodes_length = length(nodes)
48      Fg = zeros(Float64, dimensions * nodes_length)
49      Ug = zeros(Float64, dimensions * nodes_length)
50
51      # GRAUS DE LIBERDADE: LIVRES E PRESCRITOS
52      if problem_type == "3D"
53          dof_prescribe = reduce(vcat, map(
54                  (x) -> [3 * x[1] - 2, 3 * x[1] - 1, 3 * x[1]],
55                  constraints_displacements
56              ))
57          dof_free = filter(x -> x  dof_prescribe, 1:dimensions*
    nodes_length)
58          # RESTRIÇÃO DE DESLOCAMENTO
59          Ug[dof_prescribe] = reduce(vcat, map((x) -> [x[2], x[3], x
    [4]], constraints_displacements))
60      else
61          dof_prescribe = reduce(vcat, map(
62              (x) -> [2 * x[1] - 1, 2 * x[1]],
63              constraints_displacements)
64          )
65          dof_free = filter(x -> x  dof_prescribe, 1:dimensions*
    nodes_length)
66          # RESTRIÇÃO DE DESLOCAMENTO
67          Ug[dof_prescribe] = reduce(vcat, map((x) -> [x[2], x[3]],
    constraints_displacements))
68      end
69
70
71      # CONSTRUÇÃO DOS ELEMENTOS
72      print("Construindo os elementos e a matrix de rigidez global
    paralelamente...")
73      @time Kg = Elements.assemble_stiffness_matrix(input_dict["
    elements"]["linear"], materials, nodes, problem_type)
74
75      print("Aplicando as restrições de força...
                ")
```

```
76        @time if problem_type == "3D"
77            # RESTRIÇÕES DE FORÇA SOBRE NÓS
78            if !isempty(constraints_forces_nodes)
79                dof_constraints_forces_nodes = reduce(vcat, map((x) ->
    [3 * x[1] - 2, 3 * x[1] - 1, 3 * x[1]], constraints_forces_nodes))
80                Fg[dof_constraints_forces_nodes] = reduce(vcat, map((x)
    -> [x[2], x[3], x[4]], constraints_forces_nodes))
81            end
82            # RESTRIÇÃO DE FORÇAS SOBRE SUPERFÍCIES (somente
    TetrahedronLinear)
83            if !isempty(constraints_forces_surfaces)
84                Elements.assemble_force_surface!(Fg, nodes,
    constraints_forces_surfaces)
85            end
86        else
87            # RESTRIÇÕES DE FORÇA SOBRE NÓS
88            if !isempty(constraints_forces_nodes)
89                dof_constraints_forces_nodes = reduce(vcat, map((x) ->
    [2 * x[1] - 1, 2 * x[1]], constraints_forces_nodes))
90                Fg[dof_constraints_forces_nodes] = reduce(vcat, map((x)
    -> [x[2], x[3]], constraints_forces_nodes))
91            end
92            # RESTRIÇÃO DE FORÇAS SOBRE LINHAS (somente TriangleLinear)
93            if !isempty(constraints_forces_lines)
94                Elements.assemble_force_line!(Fg, nodes,
    constraints_forces_lines)
95            end
96        end
97
98
99        println("Resolvendo o sistema de $(size(Kg)) ")
100       @time Solver.direct_solve!(Kg, Ug, Fg, dof_free, dof_prescribe)
101
102       print("Calculando as reações...
                ")
103       @time Re, Re_sum = Stress.reactions(Kg, Ug, dimensions)
104
105       println("Somatório das reações: $(Re_sum)")
106
107       print("Recuperando as tensões...
                ")
108       @time , vm = Stress.recovery(input_dict["elements"]["linear"],
    Ug, materials, nodes, problem_type)
109
110       print("Imprimindo o arquivo de saída...
                ")
111       @time begin
```

```
112             output_file = open(string(output_path), "w")
113             IOfiles.write_header(output_file)
114 
115             # Pontos gaussianos
116             if "tetrahedrons" in keys(input_dict["elements"]["linear"])
117                 write(output_file,
118                     "GaussPoints \"gpoints\" ElemType Tetrahedra \n",
119                     " Number Of Gauss Points: 1 \n",
120                     " Natural Coordinates: internal \n",
121                     "end gausspoints \n",
122                 )
123             end
124             if "triangles" in keys(input_dict["elements"]["linear"])
125                 write(output_file,
126                     "GaussPoints \"gpoints\" ElemType Triangle \n",
127                     " Number Of Gauss Points: 1 \n",
128                     " Natural Coordinates: internal \n",
129                     "end gausspoints \n",
130                 )
131             end
132 
133             # DESLOCAMENTOS
134             IOfiles.write_result_nodes(output_file,
135                 "Result \"Displacements\" \"Load Analysis\" 0 Vector
    OnNodes",
136                 dimensions, Ug
137             )
138 
139             # ESTADO TENSÃO
140             IOfiles.write_result_gauss_center(output_file,
141                 "Result \"Stress\" \"Load Analysis\" 0 $( problem_type
    == "3D" ? "matrix" : "PlainDeformationMatrix") OnGaussPoints \"
    gpoints\"",
142 
143             )
144 
145             # REAÇÕES
146             IOfiles.write_result_nodes(output_file,
147                 "Result \"Reactions\" \"Load Analysis\" 0 Vector OnNodes
    ",
148                 dimensions, Re
149             )
150             # VON MISSES
151             IOfiles.write_result_gauss_center(output_file,
152                 "Result \"Von Misses\" \"Load Analysis\" 0 scalar
    OnGaussPoints \"gpoints\"",
153                 vm
```

```
            )

            close(output_file)

        end
        print("Tempo total de execução: ")
    end
end
```

./src/modules/includes.jl

```

# PHILLIP
# Script para adicionar todos os arquivos que contêm os módulos locais

include("IOfiles.jl")
include("Matrices.jl")
include("Elements.jl")
include("Solver.jl")
include("Stress.jl")
```

./src/modules/IOfiles.jl

```

# PHILLIPO
# Módulo: controle de entradas e saídas


module IOfiles

    # MÓDULOS EXTERNOS
    import JSON

    # texto de cabeçalho (salvando durate a compilação)
    header_msg_file = open(string(@__DIR__ ,"/header_msg.txt"), "r")
    header_msg_text = read(header_msg_file, String)

    function open_parse_input_file(file_name::String)::Dict
        # Carrega e interpreta o arquivo de entrada
        # Retorna um dicionário
        JSON.parsefile(file_name, dicttype=Dict, use_mmap = true)
    end

    function header_prompt()
        # Imprime o cabeçalho do prompt de execução do programa
        # header_msg_file = open(string(@__DIR__,"header_msg.txt"), "r")
        # header_msg_text::String = read(header_msg_file, String)
        println(header_msg_text)
    end
```

```
27
28      function write_header(file::IOStream)
29          write(file, "GiD Post Results File 1.0", "\n")
30      end
31
32      function write_result_nodes(
33              file::IOStream,
34              header::String,
35              d::Integer,
36              vector::Vector{<:Real}
37          )
38          write(file, header, "\n")
39          vector_length = length(vector) ÷ d
40
41          write(file, "Values", "\n")
42          for i = 1:vector_length
43              write(file, " $(i)", " ",
44                  join((vector[d * i - j] for j = (d - 1):-1:0), " "),
45                  "\n"
46              )
47          end
48          write(file, "End Values", "\n")
49      end
50
51      function write_result_gauss_center(
52              file::IOStream,
53              header::String,
54              vector::Vector
55          )
56          write(file, header, "\n")
57          vector_length = length(vector)
58
59          write(file, "Values", "\n")
60          for i = 1:vector_length
61              write(file, " $(i)", " ",
62                  join(vector[i], " "),
63                  "\n"
64              )
65          end
66          write(file, "End Values", "\n")
67      end
68
69  end
```

./src/modules/Elements.jl

```
1
2   # PHILLIPO
```

```
# Módulo: definição dos elementos e funções relacionadas

module Elements

    #MÓDULOS EXTERNOS
    import LinearAlgebra
    using SparseArrays
    import ..Matrices

    abstract type Element end

    struct TriangleLinear <: Element

        index::Integer
        material_index::Integer
        nodes_index::Vector{Integer}
        interpolation_function_coeff::Matrix{Real}
        D::Matrix{Real}
        B::Matrix{Real}
        K::Matrix{Real}
        degrees_freedom::Vector{Integer}

        function TriangleLinear(triangle_element_vector::Vector{Any},
    materials::Vector{Any}, nodes::Vector{Any}, problem_type::String)

            index          = Integer(triangle_element_vector[1])
            material_index = Integer(triangle_element_vector[2])
            nodes_index    = Vector{Integer}(triangle_element_vector
    [3:5])


            i = Vector{Real}(nodes[nodes_index[1]])
            j = Vector{Real}(nodes[nodes_index[2]])
            m = Vector{Real}(nodes[nodes_index[3]])

            position_nodes_matrix = [
                1  i[1]  i[2];
                1  j[1]  j[2];
                1  m[1]  m[2]
            ]

            interpolation_function_coeff = LinearAlgebra.inv(
    position_nodes_matrix)

             = 1/2 *  LinearAlgebra.det(position_nodes_matrix)

            a = interpolation_function_coeff[1,:]
```

```
47            b = interpolation_function_coeff[2,:]
48            c = interpolation_function_coeff[3,:]
49
50
51            B = [
52                b[1] 0    b[2] 0    b[3] 0   ;
53                0    c[1] 0    c[2] 0    c[3];
54                c[1] b[1] c[2] b[2] c[3] b[3]
55            ]
56
57            try
58                materials[material_index]
59            catch
60                error("Material não definido no elemento de índice: $(
    index)")
61            end
62
63            D = generate_D(problem_type, materials[material_index])
64
65            K = B' * D * B *  * 1
66
67            degrees_freedom = reduce(vcat, map((x) -> [2 * x - 1, 2 * x
    ], nodes_index))
68
69            new(index, material_index, nodes_index,
    interpolation_function_coeff, D, B, K, degrees_freedom)
70        end
71    end
72
73    struct TetrahedronLinear <: Element
74        index::Integer
75        material_index::Integer
76        nodes_index::Vector{<:Integer}
77        interpolation_function_coeff::Matrix{<:Real}
78        D::Matrix{<:Real}
79        B::Matrix{<:Real}
80        K::Matrix{<:Real}
81        degrees_freedom::Vector{<:Integer}
82        function TetrahedronLinear(tetrahedron_element_vector::Vector{<:
    Any}, materials::Vector{<:Any}, nodes::Vector{<:Any})
83
84            index          = Integer(tetrahedron_element_vector[1])
85            material_index = Integer(tetrahedron_element_vector[2])
86            nodes_index    = Vector{Integer}(tetrahedron_element_vector
    [3:6])
87
88
```

```
89          i = Vector{Real}(nodes[nodes_index[1]])
90          j = Vector{Real}(nodes[nodes_index[2]])
91          m = Vector{Real}(nodes[nodes_index[3]])
92          p = Vector{Real}(nodes[nodes_index[4]])
93
94          position_nodes_matrix = [
95              1 i[1] i[2] i[3];
96              1 j[1] j[2] j[3];
97              1 m[1] m[2] m[3];
98              1 p[1] p[2] p[3]
99          ]
100         interpolation_function_coeff = LinearAlgebra.inv(
    position_nodes_matrix)
101         V = 1/6 * LinearAlgebra.det(position_nodes_matrix)
102
103         a = interpolation_function_coeff[1,:]
104         b = interpolation_function_coeff[2,:]
105         c = interpolation_function_coeff[3,:]
106         d = interpolation_function_coeff[4,:]
107
108         B = [
109             b[1]  0      0    b[2]  0      0    b[3]  0      0    b
    [4]  0      0   ;
110             0     c[1]   0    0     c[2]   0    0     c[3]   0    0
    c[4]   0   ;
111             0     0      d[1] 0     0      d[2] 0     0      d[3] 0
    0      d[4];
112             c[1]  b[1]   0    c[2]  b[2]   0    c[3]  b[3]   0    c
    [4]  b[4]   0   ;
113             0     d[1]   c[1] 0     d[2]   c[2] 0     d[3]   c[3] 0
    d[4]   c[4];
114             d[1]  0      b[1] d[2]  0      b[2] d[3]  0      b[3] d
    [4]  0      b[4]
115         ]
116
117         try
118             materials[material_index]
119         catch
120             error("Material não definido no elemento de índice: $(
    index)")
121         end
122
123         D = generate_D("3D", materials[material_index])
124
125         K =  B' * D * B * V
126
127         degrees_freedom = Vector{Integer}(reduce(vcat, map((x) -> [3
```

```
    * x - 2, 3 * x - 1, 3 * x], nodes_index)))

            new(index, material_index, nodes_index,
    interpolation_function_coeff, D, B, K, degrees_freedom)
        end
    end


    function generate_D(problem_type, material)::Matrix{<:Real}
        # Gera a matrix constitutiva
        E::Float64 = material[2] # Módulo de young
        ::Float64 = material[3] # Coeficiente de Poisson

        if problem_type == "plane_strain"
            return E / ((1 + ) * (1 - 2)) * [
                (1 - )        0           ;
                       (1 - ) 0           ;
                0       0       (1 - 2) / 2
            ]
        end

        if problem_type == "plane_stress"
            return E / (1 - ^2) * [
                1              0           ;
                       1       0           ;
                0       0       (1 - ) / 2
            ]
        end

        if problem_type == "3D"
            return E / ((1 + ) * (1 - 2)) * [
                (1 - )                 0            0            0
        ;
                       (1 - )          0            0            0
        ;
                              (1 - )   0            0            0
        ;
                0       0       0         (1 - 2) / 2 0              0
          ;
                0       0       0         0             (1 - 2) / 2 0
          ;
                0       0       0         0             0             (1 -
    2) / 2
            ]
        end

        error("PHILLIP0: Tipo de problema desconhecido!")
```

```
167
168     end
169
170     function assemble_stiffness_matrix(input_elements, materials, nodes,
        problem_type)
171         # Realiza a criação dos elementos e já aplica os valores de
    rigez sobre a matriz global
172
173         # O paralelismo é realizado reservando para cada thread uma
    matriz separada
174         Kg_vector = [Matrices.SparseMatrixCOO() for i = 1:Threads.
    nthreads()]
175
176         if problem_type == "3D"
177             if "tetrahedrons" in keys(input_elements)
178                 pop!(input_elements["tetrahedrons"])
179                 elements_length = length(input_elements["tetrahedrons"])
180                 Threads.@threads for j in 1:elements_length
181                     element = TetrahedronLinear(input_elements["
    tetrahedrons"][j], materials, nodes)
182                     Matrices.add!(
183                         Kg_vector[Threads.threadid()],
184                         element.degrees_freedom,
185                         element.K
186                     )
187                 end
188             end
189         else
190             if "triangles" in keys(input_elements)
191                 pop!(input_elements["triangles"])
192                 elements_length = length(input_elements["triangles"])
193                 Threads.@threads for j in 1:elements_length
194                     element = TriangleLinear(input_elements["triangles
    "][j], materials, nodes, problem_type)
195                     Matrices.add!(
196                         Kg_vector[Threads.threadid()],
197                         element.degrees_freedom,
198                         element.K
199                     )
200                 end
201             end
202         end
203
204         # A matriz global de rigidez é a soma das matrizes globais
    calculadas em cada thread
205         Kg = Matrices.sum(Kg_vector)
206
```

```
207         return Kg
208     end
209
210     function assemble_force_line!(
211             Fg::Vector{<:Real},
212             nodes::Vector,
213             forces::Vector,
214         )
215         # Aplica a força equivalente nos nós de linha que sofre um
    carregamento constante.
216         # Por enquanto, só funciona para problemas com elementos do tipo
     TriangleLinear
217         for force in forces
218             elements_index = force[1]
219             nodes_index    = force[2:3]
220             forces_vector  = force[4:5]
221
222             dof_i = mapreduce(el -> [2 * el - i for i in 1:-1:0], vcat,
    nodes_index[1])
223             dof_j = mapreduce(el -> [2 * el - i for i in 1:-1:0], vcat,
    nodes_index[2])
224
225             node_i = nodes[nodes_index[1]]
226             node_j = nodes[nodes_index[2]]
227
228              =  LinearAlgebra.norm(node_i .- node_j)
229             F =  1/2 *  .* forces_vector
230
231             Fg[dof_i] += F
232             Fg[dof_j] += F
233         end
234     end
235
236     function assemble_force_surface!(
237             Fg::Vector{<:Real},
238             nodes::Vector,
239             forces::Vector
240         )
241         # Aplica a força equivalente nos nós de superfícies que sofre um
     carregamento constante.
242         # Por enquanto, só funciona para problemas com elementos do tipo
     TetrahedronLinear
243         for force in forces
244             elements_index = force[1]
245             nodes_index    = force[2:4]
246             forces_vector  = force[5:7]
247
```

```
248             dof_i = mapreduce(el -> [3 * el - i for i in 2:-1:0], vcat,
    nodes_index[1])
249             dof_j = mapreduce(el -> [3 * el - i for i in 2:-1:0], vcat,
    nodes_index[2])
250             dof_k = mapreduce(el -> [3 * el - i for i in 2:-1:0], vcat,
    nodes_index[3])
251 
252             node_i = nodes[nodes_index[1]]
253             node_j = nodes[nodes_index[2]]
254             node_k = nodes[nodes_index[3]]
255 
256             vector_ij = node_j .- node_i
257             vector_ik = node_k .- node_i
258 
259              =  1/2 * LinearAlgebra.norm(LinearAlgebra.cross(vector_ij,
    vector_ik))
260             F =  1/3 *  .* forces_vector
261 
262             Fg[dof_i] += F
263             Fg[dof_j] += F
264             Fg[dof_k] += F
265         end
266     end
267 
268 end
```

./src/modules/Matrices.jl

```
1 
2 # PHILLIPO
3 # Módulo: construção de matrizes esparsas baseada em coordenadas
4 # Este arquivo é construído com fork indireto o FEMSparse.jl (módulo
    utilizado no JuliaFEM.jl)
5 
6 module Matrices
7 
8     using SparseArrays
9     import Base.sum
10    export SparseMatrixCOO, spCOO, sum, add!
11    using LinearAlgebra
12 
13    mutable struct SparseMatrixCOO{Tv,Ti<:Integer} <:
    AbstractSparseMatrix{Tv,Ti}
14        I :: Vector{Ti}
15        J :: Vector{Ti}
16        V :: Vector{Tv}
17    end
18 
```

```
19     spCOO(A::Matrix{<:Number}) = SparseMatrixCOO(A)
20     SparseMatrixCOO() = SparseMatrixCOO(Int[], Int[], Float64[])
21     SparseMatrixCOO(A::SparseMatrixCSC{Tv,Ti}) where {Tv, Ti<:Integer} =
       SparseMatrixCOO(findnz(A)...)
22     SparseMatrixCOO(A::Matrix{<:Real}) = SparseMatrixCOO(sparse(A))
23     SparseArrays.SparseMatrixCSC(A::SparseMatrixCOO) = sparse(A.I, A.J,
    A.V)
24     Base.isempty(A::SparseMatrixCOO) = isempty(A.I) && isempty(A.J) &&
    isempty(A.V)
25     Base.size(A::SparseMatrixCOO) = isempty(A) ? (0, 0) : (maximum(A.I),
       maximum(A.J))
26     Base.size(A::SparseMatrixCOO, idx::Int) = size(A)[idx]
27     Base.Matrix(A::SparseMatrixCOO) = Matrix(SparseMatrixCSC(A))
28
29     get_nonzero_rows(A::SparseMatrixCOO) = unique(A.I[findall(!iszero, A
    .V)])
30     get_nonzero_columns(A::SparseMatrixCOO) = unique(A.J[findall(!iszero
    , A.V)])
31
32     function Base.getindex(A::SparseMatrixCOO{Tv, Ti}, i::Ti, j::Ti)
    where {Tv, Ti}
33         if length(A.V) > 1_000_000
34             @warn("Performance warning: indexing of COO sparse matrix is
       slow.")
35         end
36         p = (A.I .== i) .& (A.J .== j)
37         return sum(A.V[p])
38     end
39
40     """
41         add!(A, i, j, v)
42     Add new value to sparse matrix `A` to location (`i`,`j`).
43     """
44     function add!(A::SparseMatrixCOO, i, j, v)
45         push!(A.I, i)
46         push!(A.J, j)
47         push!(A.V, v)
48         return nothing
49     end
50
51     function Base.empty!(A::SparseMatrixCOO)
52         empty!(A.I)
53         empty!(A.J)
54         empty!(A.V)
55         return nothing
56     end
57
```

```
    function assemble_local_matrix!(A::SparseMatrixCOO, dofs1::Vector{<:Integer}, dofs2::Vector{<:Integer}, data)
        n, m = length(dofs1), length(dofs2)
        @assert length(data) == n*m
        k = 1
        for j=1:m
            for i=1:n
                add!(A, dofs1[i], dofs2[j], data[k])
                k += 1
            end
        end
        return nothing
    end

    function add!(A::SparseMatrixCOO, dof1::Vector{<:Integer}, dof2::Vector{<:Integer}, data)
        assemble_local_matrix!(A, dof1, dof2, data)
    end

    function sum(A::Vector{<:SparseMatrixCOO})::SparseMatrixCSC
        # Retorna uma matriz em CSC a partir de um vetor formado por matrizes em COO
        I = reduce(vcat, getfield.(A, :I))
        J = reduce(vcat, getfield.(A, :J))
        V = reduce(vcat, getfield.(A, :V))
        LinearAlgebra.Symmetric(sparse(I,J,V))
    end

    function add!(A::SparseMatrixCOO, dof::Vector{<:Integer}, data)
        assemble_local_matrix!(A, dof, dof, data)
    end

end
```

./src/modules/Solver.jl

```
# PHILLIPO
# Módulos: funções para executar o método de soluação

module Solver

    using SparseArrays
    using LinearAlgebra
    using InteractiveUtils

    function direct_solve!(
            Kg::SparseMatrixCSC,
            Ug::Vector{<:Real},
```

```
            Fg::Vector{<:Real},
            dof_free::Vector{<:Integer},
            dof_prescribe::Vector{<:Integer}
        )
        # Realiza a soluação direta para o sistema
        Ug[dof_free]        = Kg[dof_free, dof_free] \ (Fg[dof_free] - Kg[dof_free, dof_prescribe] * Ug[dof_prescribe])
        Fg[dof_prescribe]   = Kg[dof_prescribe, dof_free] * Ug[dof_free] + Kg[dof_prescribe, dof_prescribe] * Ug[dof_prescribe]

    end

end
```

./src/modules/Stress.jl

```
# PHILLIPO
# Módulo: recuperação de tensão

module Stress

    import ..Elements
    using SparseArrays

    function recovery(input_elements, Ug::Vector{<:Real}, materials::Vector{Any}, nodes::Vector{Any}, problem_type)
        map_function = e -> nothing
        if problem_type == "3D"
            if "tetrahedrons" in keys(input_elements)
                type = "tetrahedrons"
                map_function = e -> begin
                    el = Elements.TetrahedronLinear(e, materials, nodes)
                    el.D * el.B * Ug[el.degrees_freedom]
                end
            end
        else
            if "triangles" in keys(input_elements)
                type = "triangles"
                map_function = e -> begin
                    el = Elements.TriangleLinear(e, materials, nodes, problem_type)
                    el.D * el.B * Ug[el.degrees_freedom]
                end
            end
        end
        println("TESTE: ", size(input_elements[type]))
           = Vector{Vector{Float64}}(map(
            map_function,
```

```
            input_elements[type]
        ))
        vm = von_misses.()
        , vm
    end

    von_misses(::Vector{<:Real}) = length() == 3 ? von_misses_2D() : von_misses_3D()

    function von_misses_2D(::Vector{<:Real})
        ([1]^2 - [1] * [2] + [2]^2 + 3 * [3]^2)
    end

    function von_misses_3D(::Vector{<:Real})
        ((([1] - [2])^2 + ([2] - [3])^2 + ([3] - [1])^2 + 6 * ([4]^2 + [5]^2 + [6]^2)) / 2)
    end

    function reactions(Kg::SparseMatrixCSC, Ug::Vector{<:Real}, d::Integer)
        nodes_length = length(Ug)
        Re = Kg * Ug
        Re_sum = sum.([Re[i:d:nodes_length] for i in 1:d])
        return Re, Re_sum
    end

end
```