**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 RÍGIDAS UTILIZANDO PARADIGMAS DE PROGRAMAÇÃO PARALELA

**JOINVILLE**
**2023**

**LUCAS BUBLITZ**

# PHILLIPO: APLICAÇÃO DO MÉTODO DE ELEMENTOS FINITOS NA ANÁLISE ESTÁTICA DE ESTRUTURAS RÍGIDAS UTILIZANDO PARADIGMAS DE PROGRAMAÇÃO PARALELA

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 rígidas utilizando paradigmas de programação paralela / Lucas Bublitz. - Joinville, 2023.
65 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. Palavra-chave. 2. Palavra-chave. 3. Palavra-chave. 4. Palavra-chave. 5. Palavra-chave. 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 RÍGIDAS UTILIZANDO PARADIGMAS DE PROGRAMAÇÃO PARALELA

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 a quem, gostando de aprender, não fica cansado em fracassar.

## 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.

"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

Elemento obrigatório que contém a apresentação concisa dos pontos relevantes do trabalho, fornecendo uma visão rápida e clara do conteúdo e das conclusões do mesmo. A apresentação e a redação do resumo devem seguir os requisitos estipulados pela NBR 6028 (ABNT, 2003). Deve descrever de forma clara e sintética a natureza do trabalho, o objetivo, o método, os resultados e as conclusões, visando fornecer elementos para o leitor decidir sobre a consulta do trabalho no todo.



**Palavras-chave**: Palavra 1. Palavra 2. Palavra 3. Palavra 4. Palavra 5.

## ABSTRACT

Elemento obrigatório para todos os trabalhos de conclusão de curso. Opcional para os demais trabalhos acadêmicos, inclusive para artigo científico. Constitui a versão do resumo em português para um idioma de divulgação internacional. Deve aparecer em página distinta e seguindo a mesma formatação do resumo em português.



**Keywords**: Keyword 1. Keyword 2. Keyword 3. Keyword 4. Keyword 5.

## LISTA DE ILUSTRAÇÕES

# LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| MEF | Método dos Elementos Finitos |
| FEM | Finite Element Method |
| MVF | Método dos volumes finitos |
| SI | Sistema Internacional de Unidades |

## LISTA DE SÍMBOLOS

| | |
|---|---|
| $\boldsymbol{\sigma}$ | tensor de tensões |
| $\boldsymbol{\varepsilon}$ | tensor de deformações |
| $\mathscr{B}$ | um corpo sólido, elástico, homogêneo e isotrópico, em equilíbrio |
| $\sigma_{ij}$ | tensão na direção $i$ e sentido $j$ |
| $\varepsilon_{ij}$ | deformação na direção $i$ e sentido $j$ |
| $\Omega$ | região que compreende os pontos de um sólido |
| $\hat{\boldsymbol{n}}$ | um versor |

**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 MEF consiste, basicamente, na ideia apresentada de análise, em que o domínio de uma equação diferencial é subdividido em elementos discretos, descritos por um conjunto de nós formando uma malha. Nesse método, os elementos tem suas propriedades herdadas do domínio (características, condições de contorno etc.), entretanto, a descrição do fenômeno é simplificada por meio de funções de interpolação. Essas discretizações são, então, justapostas, de modo a garantir continuidade, formando um sistema, cuja solução é uma aproximação da solução da própria equação diferencial.

Esse procedimento é muito custoso em termos de cálculo, visto que para cada elementos 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, acompanha o desenvolvimento da programação, impulsionado pelo avanço do processamento computacional (OñATE, 2009). O poder computacional permite que se trabalhe com um volume inconcebível, para a capacidade humana, de dados e operações, como também das estruturas de dados e algoritmos que os manipulam. O algoritmo e a estrutura de dados passam a ser tão relevantes quanto a própria equação diferencial. Então, é de se esperar que uma aplicação desse método seja acompanhada de aspectos de projeto de software conciso, cujo objetivo não seja só a otimização computacional, mas a legibilidade e modularização, que são características úteis quando se espera a reutilização, aprimoramento continuado e, acima de tudo, a comunicação do código-fonte.

A escolha da linguagem de programação, então, 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, esses softwares 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, também são conhecidas pela sua prolixidade, complexidade de sintaxe de distribuição de bibliotecas, empacotamento... Elas não se mostram mais atrativas para se desenvolver, com praticidade, não só aplicações do MEF, como

também a maioria das aplicações práticas na vida dos engenheiros e cientistas (automatização de tarefas, análise de dados e até computação algébrica simbólica). Hoje, Python e suas bibliotecas: Pandas, NumPy, CoolProp... utilizando-se de sua sintaxe de sua sintaxe simplificada, tipagem dinâmica e popularidade (ocupando o TIOBE Index três vezes nos últimos cinco anos), possibilita o acesso a ferramentas muito sofisticas: manipulação de dados (estatística e filtragem com Pandas), construção de modelos físicos (interpolação de propriedades termodinâmicas com CoolProp) e automatização de tarefas. (ERNESTI, 2022) Porém, toda essa conveniência tem um preço: o desempenho.

É notável que linguagens interpretadas, como Python, tem desempenho, em termos de processamento, muito inferior ao de linguagens compiladas, como C e Fortran, cuja eficiência de execução é necessária, quando se trabalha com problemas grandes e complexos, para se obter um tempo de execução razoável, às custas de uma sintaxe prolixa, tipagem estática e gerenciamento manual de memória. Esse dilema entre a produtividade de linguagens como Python e o desempenho de linguagens como C, é conhecido como *The Two languages Problem*, ou, em tradução livre, O Problema das Duas Linguagens.

Visando unificar esses dois mundos, e diminuir a distância entre as linguagens, engenheiros do MIT desenvolveram Julia, "a programming language for the scientific community that combines features of productivity languages, such as Python or MATLAB, with characteristics of performance-oriented languages, such as C++ or 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 na 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 de 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. Entendendo como o método funciona e como é aplicado, observando aspectos tanto matemáticos e físicos, quanto de programação e de estrutura de dados, é crucial para que engenheiros possam aplicá-lo devidamente, principalmente quando se utilizam de soluções já prontas: de código-aberto ou proprietárias. Este documento aborda o desenvolvimento de um desses softwares: PHILLIPO.jl, cujo objetivo é expor e aplicar o MEF, abordando alguns aspectos de programação diferenciados daqueles vistos na graduação, como programação paralela, modularização e empacotamento, sem o intuito de concorrer com outras soluções já consolidadas ou ser referência de aplicação, mas de ser exemplificativo.

## 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 visto uma introdução ao assunto na graduação, teve o interesse de se aprofundar. Então resolveu por criar seu próprio programa,

em Julia, aplicando seus conhecimento prévios de projeto de software, 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 sobre carregamentos estáticos em regime elástico linear, utilizando para tanto, aspectos de programação funcional, processamento paralelo, focando em algumas características modulares de implementação 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 propostos

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;
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 sobre o paradigma estruturado;
5. estudar as características da linguagem Julia, e como ela pode ser uma alternativa viável para C e FORTRAN em programação científica de alta performance.

## 1.3 ORGANIZAÇÃO DO DOCUMENTO

Este documento aborda o projeto e 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 sobre carregamentos estáticos; e é 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 de programação Julia: o processamento paralelo acessível a engenheiros mecânicos;
4. PHILLIPO.jl, o módulo;
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 é parte da física que estuda o comportamento de objetos sólidos sobre carregamentos, aplicando métodos analíticos para determinar suas características de resistência, rigidez e estabilidade. Seu conteúdo é notório por ser fundamental para grande parte da vida dos engenheiros, como mecânicos, civis ou mesmo eletricistas, ao lado de outras áreas também tão fundamentais, Mecânica dos Fluidos e Termodinâmica. Sua aplicação é voltada ao projeto de estruturas a fim de que cumpram determinadas exigências, sejam tanto de deformação máxima, capacidade de carga e peso, como também de 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)

Corpos sólidos são conjuntos de matéria que resistem a forças cisalhantes, ou seja, que resistem a trações tangenciais à suas superfícies. Essa é a característica fundamental dos sólidos, e, para contextualização deste capítulo, é importante que também sejam 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, dentro de seu regime elástico. *Corpos sólidos homogêneos* são aqueles que possuem as mesmas propriedades físicas em todos os pontos de sua geometria, 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 de sua geometria.

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 sobre carregamentos contantes:

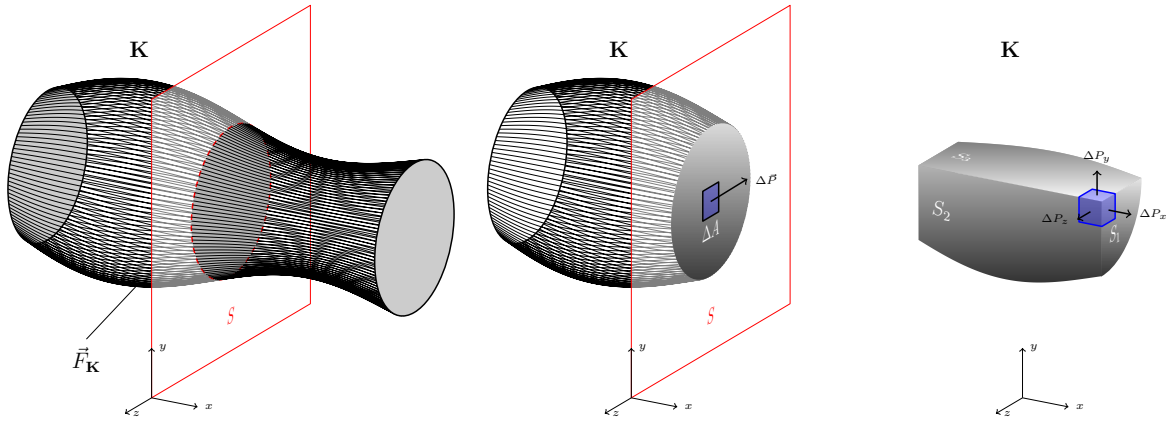
1. tensão;
2. deslocamento e deformação;
3. a lei de Hooke generalizada;
4. tensão de von mises.

## 2.1 TENSÃO

Um corpo sólido se deforma quando submetido a carregamentos externos[1], distribuindo essas cargas ao longo de sua geometria. Tensão, no contexto de cargas mecânicas, define a grandeza dessa distribuição agindo sobre áreas infinitesimais, de modo que qualquer secção do corpo revele forças internas que estejam em equilíbrio entre si, e que sejam balanceadas pelos carregamentos externos. Essas forças, geralmente, variam ao longo do corpo, como também,

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

Figura 1 – Forças internas: seção em um sólido qualquer




(a) (b) (c)

dependem do plano de seção. E, devido a sua forma vetorial, é conveniente que sejam decompostas em parcelas tangenciais e normais à seção. (POPOV, 1990, pág. 60)

Sejam um corpo $\mathscr{B}$, sólido, em equilíbrio e de geometria qualquer, situado sobre um sistema de referência $(x, y, z)$, submetido a forças externas na forma do carregamento $\boldsymbol{F}$, e 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 1a. $\Delta\boldsymbol{P}$ é a resultante de forças que atuam sobre uma área $\Delta A$ (centrada em um certo ponto $p$), pertencente a $S$. O limite da razão entre cada componente de $\Delta\boldsymbol{P}$ (tangenciais e normais) e a área $\Delta A$, quando $\Delta A \to 0$, define as componentes da tração agindo sobre o ponto de análise do corpo, de forma que

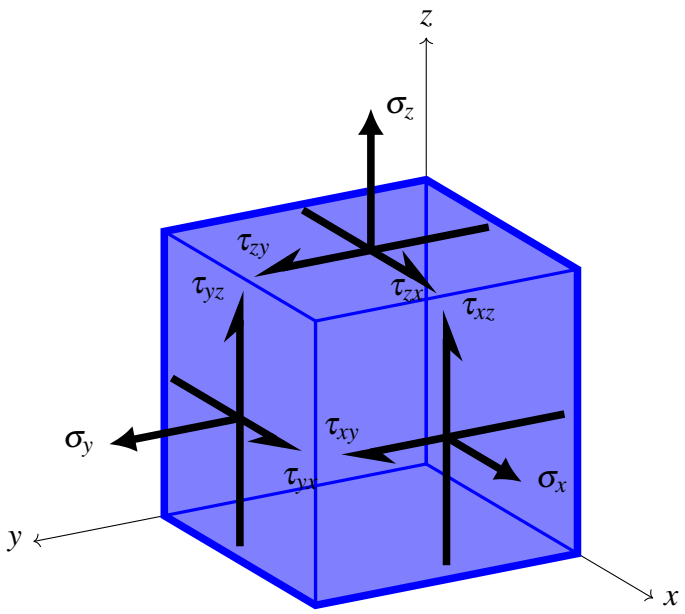
$$\tau_{xx} = \lim_{\Delta A \to 0} \frac{\Delta P_x}{\Delta A}, \qquad \tau_{xy} = \lim_{\Delta A \to 0} \frac{\Delta P_y}{\Delta A}, \qquad \tau_{xz} = \lim_{\Delta A \to 0} \frac{\Delta P_z}{\Delta A}, \tag{1}$$

em que os índices de $\tau$ 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 $\sigma$, ao invés de se utilizar $\tau$ com índices repetidos ($\tau_{xx} \equiv \sigma_x$). O símbolo *tau*, então, é reservado às tensões de cisalhamento, que atuam tangencialmente ao plano infinitesimal. No SI, a tensão é mensurada em Pascal ([Pa] = [N/mš]).

Se o mesmo procedimento for realizado para cada face de o elemento cúbico, formado por mais três seções paralelas e equidistantes 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 $\mathscr{K}$, 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

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

Figura 2 – Estado de tensão




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 torna com que o tensor de tensões possua apenas seis componentes independentes, ao invés de nove. Aproveitando-se disso, a notação de Voigt reduz a ordem do tensor, distribuindo as componentes em um vetor coluna de seis elementos, tal que, de acordo com Roylance (2000),

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

### 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 $\Omega$ 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 eletromagnéticas. (ROYLANCE, 2000, pág. 4, The Equilibrium Equations)

A integração é sobre uma região fechada, fronteira de um subdomínio de $\Omega$, e portanto, como a tensão foi assumida contínua e diferenciável em todo $\Omega$, podemos aplicar o teorema da divergência[2] à equação 6, obtendo que

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

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 $\Omega$ 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 tanto a distribuição de tensão sobre o sólido, desde que sejam conhecidas as condições de contorno. Comumente 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

[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 $\Omega$.

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

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

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

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 tem componentes tangenciais, ou seja, têm cisalhamento nulo. Essas tensões são chamadas, então, tensões principais.

O tensor de tensões é um transformação linear que recebe um vetor unitário $\hat{n}$ e retorna o vetor da tensão resultante, $\boldsymbol{\sigma}_{\hat{\boldsymbol{n}}}$, agindo sobre um plano normal a $\hat{\boldsymbol{n}}$.[3] Caso exista uma tensão resultante que tenha a mesma direção $\hat{\boldsymbol{n}}$, 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 $\boldsymbol{\sigma}_{\hat{\boldsymbol{n}}} = \sigma\hat{\boldsymbol{n}}$[4], ou, aplicando a transformação linear $\boldsymbol{\sigma}$,

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

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

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

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

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

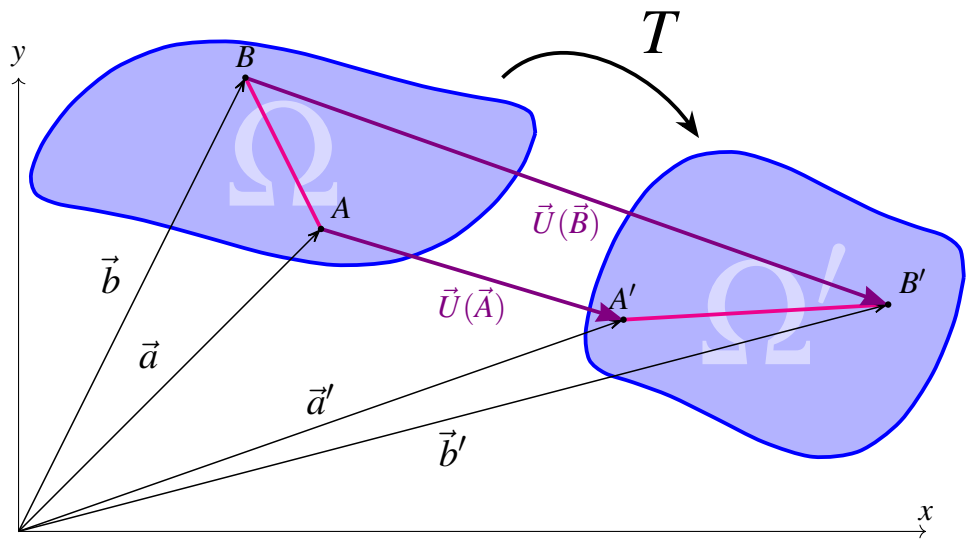
## 2.2 DESLOCAMENTO E DEFORMAÇÃO

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

[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 tensão resultante, uma vez que $\hat{\boldsymbol{n}}$ é unitário e adimensional. $|\boldsymbol{\sigma}_{\hat{\boldsymbol{n}}}| = |\sigma\hat{\boldsymbol{n}}| = |\sigma||\hat{\boldsymbol{n}}| = |\sigma|$

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




Seja um corpo $\mathscr{B}$ definido sobre uma região $\Omega$, e a função $\boldsymbol{U}(\boldsymbol{x})$, a representação de seu deslocamento, que descreve a transformação da posição original em deformada de cada ponto, mapeando $\Omega$ para $\Omega'$.

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

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.

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 $\boldsymbol{K}$, e como o segmento de reta $AB$ é mapeado para sua nova configuração sobre $\Omega'$.

Nesse sentido, são duas as possibilidades:

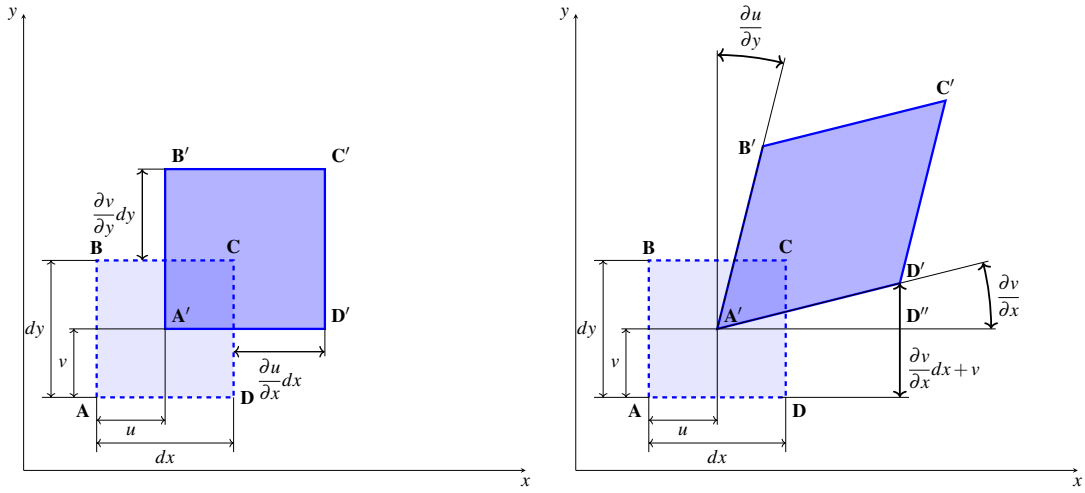
1. As distâncias entre os pontos permanece a mesma; Nesse caso, podemos dizer que a transformação é uma translação[5], e que o corpo não sofreu de deformação, pois sua geometria foi conservada. Em termos matemáticos,

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

2. As distâncias entre os pontos não se conservam; Quando isso ocorre, a geometria do corpo é alterada, deformando-se; não significa, entretanto, que o corpo não passou por uma translação. A deformação de sua geometria são as variações das distâncias entre os

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

Figura 4 – Deformação de um elemento quadrado infinitesimal




(a) deformação linear (b) deformação angular

pontos antes e após a transformação, e nada diz respeito à mudança de posição do corpo. 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 é esse alongamento, ou encurtamento, sofrido pelas linhas entre pontos do corpo, é descrita razão entre a variação do comprimento do segmento e o comprimento original, ou seja, a variação relativa do comprimento.

Essa definição, entretanto, está atrelada a uma curva no interior do sólido, e não descreve como que a deformação se manifesta ao longo de toda sua geometria, de forma a descrever continuamente ao longo de todo do domínio do corpo. Similarmente à tensão, define-se a deformação por um tensor de segunda ordem, de modo que suas componentes sejam determinadas pelo efeito que têm sobre um elemento infinitesimal (figura 4). (LUBLINER, 2017, pág. 230)

A figura 4a mostra um elemento infinitesimal, em que o segmento $AD$ sofreu tanto uma translação quanto uma deformação, dado pelo campo de deslocamento $\vec{U}$, de modo a se tornar $A'D'$. Portanto,

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

O comprimento do segmento é o próprio infinitesimal, $|AD| = dx$, já o deformado, é dado pela diferença das posições dos pontos deslocados, de modo, e sabendo da diferenciabilidade do campo de deslocamentos[6], é

$$||A'D'|| = (u + A_x + dx + \frac{\partial u}{\partial x}dx) - (u + A_x) \implies ||A'D'|| = \frac{\partial u}{\partial x}dx, \tag{19}$$

[6] Isso é importante pois o deslocamento de $D'$ é descrito em função do deslocamento em $A$, de forma que $u$, ao ser expandido em uma série de Taylor, ao redor de $A_x$, seja, determinando se a deformação de $D'$, $u(A_x + dx) = u(A_x) + \frac{\partial u}{\partial x}(A_x)dx$, em que os termos $O(x^3)$ foram desconsiderado, visto que $dx^2 << dx$.

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

$$\varepsilon_x = \frac{\partial u}{\partial x}. \tag{20}$$

O mesmo procedimento pode ser feito na direção de $y$, com o segmento $AB$, como 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 linear (POPOV, 1990):

$$\varepsilon_x = \frac{\partial u}{\partial x}, \qquad \varepsilon_y = \frac{\partial u}{\partial y}, \qquad \varepsilon_z = \frac{\partial u}{\partial z}. \tag{21}$$

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 na direção tanto em $x$, quanto em $y$, e é definida pela redução do ângulo reto $\angle BAD$, determinada, em termos do campo de deslocamentos (tal como na deformação linear) analisando o triângulo $A'D'D''$.

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,

$$\angle D'A'D'' = \tan\frac{\partial v}{\partial x} \approx \frac{\partial v}{\partial x}.^{7} \tag{22}$$

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,

$$|A'D''| = \sqrt{dx^2 - \left(\frac{\partial v}{\partial x}dx\right)^2}, \text{Teorema de Pitágoras} \tag{23}$$

$$= \sqrt{dx^2}, \left(\frac{\partial v}{\partial x}dx\right)^2 << dx, \tag{24}$$

$$|A'D''| = dx, \tag{25}$$

$$\tan\angle D'A'D'' = \frac{|D'D''|}{|A'D''|}, \tag{26}$$

$$= \frac{\partial v}{\partial x}dx\frac{1}{dx}, \tag{27}$$

$$= \frac{\partial v}{\partial x}. \tag{28}$$

$$\tag{29}$$

[7] 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 $x = 0$.

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.

A deformação, portanto, de cisalhamento do elemento infinitesimal é dada, em termos do deslocamento, por

$$\gamma_{xy} = \gamma_{yx} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \tag{30}$$

$$\gamma_{xz} = \gamma_{zx} = \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \tag{31}$$

$$\gamma_{yz} = \gamma_{zy} = \frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \tag{32}$$

$$\tag{33}$$

Por convenção[8], $\gamma_{ij} = 2\varepsilon_{ij}, i \neq j$.(ROYLANCE, 2000)

O tensor de deformações é

$$[\varepsilon] = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{zz} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) & \frac{1}{2}\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\ \frac{1}{2}\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) & \frac{\partial v}{\partial y} & \frac{1}{2}\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \frac{1}{2}\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) & \frac{1}{2}\left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right) & \frac{\partial w}{\partial z} \end{bmatrix}, \tag{34}$$

ou, em notação indicial

$$\varepsilon_{ij} = \frac{1}{2}\left(\frac{\partial \vec{U}_j}{\partial i} + \frac{\partial \vec{U}_i}{\partial j}\right), \tag{35}$$

em que vale o mesmo tipo de notação que as tesões, $\varepsilon_{ii} = \varepsilon_i$, e que $\vec{U}_x = u, \vec{U}_y = v, \vec{U}_z = w$.

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 (2000),

$$\{\varepsilon\} = \begin{bmatrix} \varepsilon_x & \varepsilon_y & \varepsilon_z & \gamma_{xy} & \gamma_{xz} & \gamma_{yz} \end{bmatrix}^t \tag{36}$$

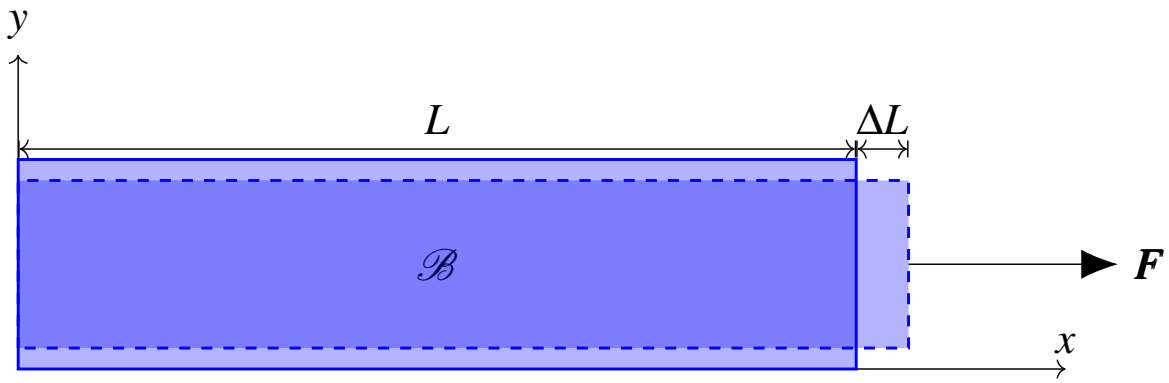
## 2.3 A LEI DE HOOKE

Em corpos sólidos e elásticos, 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

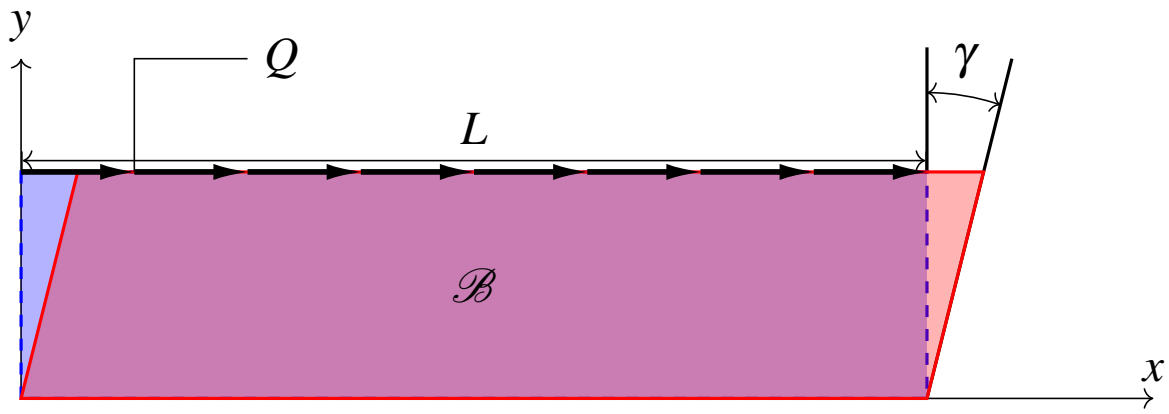
$$\{\sigma\} = \boldsymbol{C}\{\varepsilon\}. \tag{37}$$

[8] Essa convenção não é mero simbolismo, mas faz com que o tensor de deformações tenha propriedades interessantes; é possível, porém, intuir uma razão para tanto, observado que, para um mesmo elemento, a deformação por cisalhamento em $x$ já tem a parcela da deformação na direção de $y$, e por conta disso, são divididas. (POPOV, 1990)

Figura 5 – quadrilátero deformado




(a) por tração




(b) por cisalhamento

$\boldsymbol{C}$ é denominada *matriz de constitutiva*[9], definida em termos das características do material do corpo, como Módulo de Elasticidade e Coeficiente de Poisson, utilizando o *Princípio de Sobreposição*

### 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 $A$, e $F$, 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 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, $\varepsilon_x$, atuando na mesma direção da força. Essa relação, denominada *Lei de Hooke Uniaxial*, é linear da forma

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

$$\sigma_x = E\varepsilon_x. \tag{38}$$

A unidade de $E$ é a mesma de $\sigma_x$, o que é coerente, pois $\varepsilon_x$ é adimensional, ou seja, o módulo de elasticidade é medido, no SI, em Pascal. O módulo de Young 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 $\boldsymbol{K}$. Aqui $E$ é tratado como constante.

Essa relação desconsidera outros efeitos de deformação no interior do sólido, como a deformação transversal, $epsilon_y$ (que pode ser observada como o encurtamento da altura da barra na figura 5a), e a por cisalhamento, $\gamma_{xy}$. É, comumente, empregado em casos que essas não são relevantes, como molas, barras e vigas. Vale lembrar que a Lei de Hooke é válida apenas para deformações elásticas, ou seja, que não ultrapassem o limite de elasticidade do material. [10].

### 2.3.2 A Lei de Hooke em Cisalhamento

Sejam a quadrilátero $\mathscr{B}$, um corpo sólido, homogêneo, em equilíbrio, de comprimento $L$, engastado em sua face inferior, e de seção transversal $A$, e $Q$, uma 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 $\gamma$, tal como na figura 5b.

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

$$\tau_{xy} = G\gamma_{xy}, \qquad = \tau_{xy} = 2G\varepsilon_{xy}. \tag{39}$$

O módulo de cisalhamento tem a mesma unidade de tensão, e, assim como o módulo de Young (a final, $\gamma$ é adimensional), 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 $G$ é tratado como constante.

### 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 em outras direções, perpendiculares a aquela, de forma que existe,

[10] 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.

dentro de determinados limites do regime elástico, uma relação entre essa deformações. Observa-se, por meio da experiência prática, que a deformação transversal geralmente é negativa, o corpo tende a se contrair quando submetido a uma deformação axial. Se o corpo se deforma ao longo de $x$ um $\varepsilon_x > 0$, ele se contrai em $y$, ou seja, desenvolve uma deformação $\varepsilon_y < 0$, de forma que, de acordo com Lubliner (2017)

$$\nu = -\frac{\varepsilon_y}{\varepsilon_x}, \tag{40}$$

em $\nu$ representa o coeficiente de Poisson, a razão entre a deformação transversal e a deformação axial, 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 $\nu$ 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 desmonstrar a linearidade da transformação $T(*) = \nabla \cdot (*)$,

$$\nabla \cdot (k_1[\boldsymbol{\sigma}]_1 + k_2[\boldsymbol{\sigma}]_2) = 0 \implies \nabla \cdot (k_1[\boldsymbol{\sigma}]_1) + \nabla \cdot (k_2[\boldsymbol{\sigma}]_2) = 0 \implies k_1 \nabla \cdot [\boldsymbol{\sigma}]_1 + k_2 \nabla \cdot [\boldsymbol{\sigma}]_2 = 0, \tag{41}$$

   em que $k_1, k_2$ são constantes reais, e $[\boldsymbol{\sigma}]_1, [\boldsymbol{\sigma}]_2$ são conjuntos de tensões arbitrários, que satisfazem as equações de equilíbrio.

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 38 e 39, como também a relação entre deformações, equação 40, é possível determinar o efeito de deformação causado por cada componente de tensão, tal que, devido à tensão $\sigma_x$, 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, $\sigma_y$ e $\sigma_z$, 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, $\tau_{xy}$, $\tau_{xz}$ e $\tau_{yz}$, 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 relaçãoes 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[11]

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

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

Em forma matricial, a lei de Hooke generalizada pode ser escrita como, utilizando a notação de Voigt para $\varepsilon$ e $\sigma$, como que $\frac{\gamma}{2} = \varepsilon$,

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

Invertendo esse sistema, obtemos a matriz constitutiva da equação 37,

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

Para que essa matriz exista, é necessário que $-1 < \nu < \frac{1}{2}$.[12]

### 2.3.5 Estado Plano de Deformação e de Tensão

Quando se analisa um problema plano, é possível simplificar as relações descritras pela matriz $\boldsymbol{C}$ em dois casos, observando como a tensão é distribuída.

O Estado Plano de Tensão (EPT) é quando o corpo é suficientemente flexível em uma direção (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, [13]

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

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

$\varepsilon_z$ passou a ser uma variável dependende, pois é determinada, totalmente, pelas outras deformações. (ZIENKIEWICZ, 2000, pág. 90)

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

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

As Relações que definem a matriz constitutiva para o estado plano de tensão:

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

O Estado de Plano de Deformação (EPD), por sua vez, é quando o corpo é suficientemente rígido em uma direção (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

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

$$\varepsilon_{xz} = -\nu\frac{\sigma_x}{E} - \nu\frac{\sigma_y}{E}. \tag{59}$$

$\gamma_{xz}$ passou a ser uma variável dependente, pois é determinada, totalmente, pelas outras deformações. (ZIENKIEWICZ, 2000, pág. 91)

# 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 utilizado para encontrar soluções de

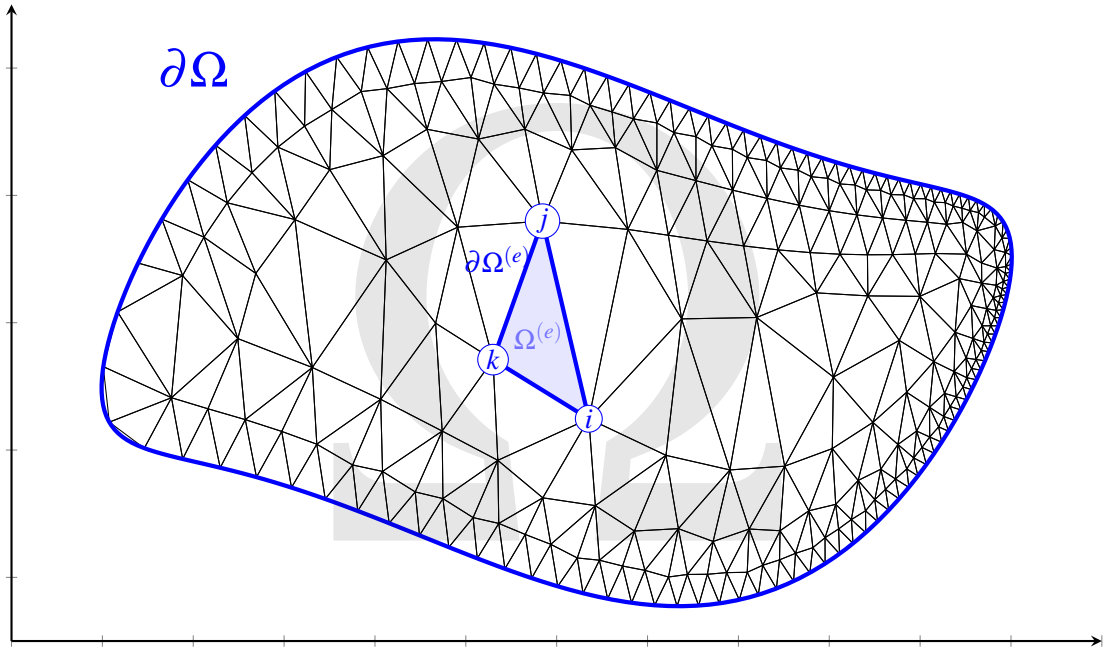
O Método dos Elementos Finitos (MEF), *Finite Element Method* (FEM), é um método numérico, e tem como finalidade aproximar a solução de funções de campo numericamente, o domínios que são difíceis de se obter repostas diretas algebricamente. Para tanto, esse domínio é discretizado em vários elementos, ou sub-domínios (ver Figura 6), de tamanho finito, cujos comportamentos já são conhecidos da aplicação de leis físicas. A função de campo desconhecida é aproximada em cada elemento por meio de funções interpoladoras polinomiais, calculadas sobre o valor de campo em cada nó, que são os pontos do domínio sobre os quais os elementos são construídos (o campo, portanto, passa a ser definido não mais pelo conjunto de valores do contínuo, mais sim por essas variáveis desconhecidas discretizadas). Para cada elemento, são definidas equações, por meio das quais eles se relacionam entre si e com o campo. Isso leva à formação de um grande sistema linear, que pode ser resolvido facilmente, e obter-se, dessa forma, a aproximação da função de campo. (OñATE, 2009) Em sua, o método de elementos finitos segue o seguinte procedimento:

1. definição do domínio ($\Omega$), e das condições de contorno ($\partial\Omega$);
2. discretização do domínio em uma malha formada por nós que constituem os elementos ($\Omega^{(e)}$);
3. aplicação da equação de governo sobre cada elemento;
4. assemblagem dessas equações em um único grande sistema linear global ($K^{(g)}$);
5. resolução do sistema, encontrando os valores nodais do campo ($U^{(g)}$).

No âmbito da análise estrutural aqui aplicada, a função de campo é o deslocamento sobre a estrutura, cuja geometria é o próprio domínio. A equação que rege os elementos é derivada do princípios dos trabalhos virtuais, em que há o balanço de energia entre o trabalho realizado pela deformação do elemento e pelas forças que atuam sobre ele. O sistema formado pelo conjunto dessas equações, aplicadas previamente em cada elemento, tem três termos, que se relacionam assim:

$$K_g \delta u = F \tag{60}$$

Figura 6 – Domínio discretizado em elementos triangulares.




Fonte: Elaborado pelo autor (2022).

O primeiro termo, $[K_g]$, relaciona as deformações dos nós com as forças externas aplicadas sobre a estrura. $\{U\}$ e $\{F\}$, por sua vez, são, respectivamento, o pseuvetor das deformações dos nós e o pseudovetor das forças externas.

São dois os tipos de elementos tratados aqui: o triângulo de tensão constante, ou *Constant Strain Triangle* (CST) na aplicação bidimensional, e o tetraedro linear, na aplicação tridimensional.

## 3.1 ANÁLISE TRIDIMENSIONAL SOBRE O TETRAEDRO

### 3.1.1 Relação entre tensão, deformação e deslocamento

Quando sólidos são postos sobre carregamentos, eles deforamam, criando tensões internas. A relaçaõ entre a tensão e a deformação

As componentes de tensão de sólidos em qualquer ponto são definidas sobre a superfície de um cubo infinitesimal, como mostrado na figura. Em cada superfície opera uma componente normal e duas tangenciais, denominadas, respectivamente, por tensão normal e de cisalhamento (o primeiro termo do índice subescrito indica a o plano de atuação da tensão, o segundo, a sua direção). O conjunto dessas tensões, de acordo com Popov (1990), quando posto em forma matricial, é chamado de tensor de Couchy, em que as linhas indicam a superfície de atuação da tensão, e a coluna, a direção. Aplicando o somatório de momento no interior desse cubo, é fácil mostrar que, segundo Quek (2003):

$$\sigma_{xy} = \sigma_{yx} \qquad \sigma_{xz} = \sigma_{zx} \qquad \sigma_{zy} = \sigma_{yz} \tag{61}$$

Portanto, há somente um total de seis componentes distintas de tensão, permitindo que o tensor de Cauchy, possa ser reescrito de forma vetorial, mais compacta:

$$\sigma^t = \begin{bmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{zz} & \sigma_{yz} & \sigma_{xz} & \sigma_{xy} \end{bmatrix} \tag{62}$$

A mesma estratégia pode, também, ser aplicada à matriz de deformação. Para cada componente do vector de tensão, em todo ponto do sólido, existe uma componenete do vector de deformação:

$$\varepsilon^t = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{yy} & \varepsilon_{zz} & \varepsilon_{yz} & \varepsilon_{xz} & \varepsilon_{xy} \end{bmatrix} \tag{63}$$

A deformação é a variação da função de deslocamento por unidade de comprimento, desse modo, os componentes do vetor de deformação podem definidos em função das derivadas da do desclocamento, da seguinte mandeira, de acordo com Quek (2003):

$$\begin{array}{ccc} \varepsilon_{xx} = \frac{\partial u}{\partial x} & \varepsilon_{yy} = \frac{\partial v}{\partial y} & \varepsilon_{zz} = \frac{\partial w}{\partial z} \\ \varepsilon_{xy} = \frac{\partial u}{\partial y} + \frac{\partial u}{\partial y} & \varepsilon_{xz} = \frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} & \varepsilon_{yz} = \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \end{array} \tag{64}$$

Nessas expressões, as funções $u$, $v$ e $w$ correspondem às componentes nas direções de $x$, $y$ e $z$, respectivamento, do vector de deslocamento sobre o sólido, que é definido como:

$$\varphi = \begin{Bmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{Bmatrix} \tag{65}$$

O cojunto de equaçãoes 64 pode ser reescrito em termos do vector de deformações e do vetor de deslocamentos do sólido, obtendo assim a relação deslocamento-deformação em forma matricial, como consta na expressão 66.

$$\varepsilon = \mathbf{L}\mathbf{U} \tag{66}$$

em que a matrix $\mathbf{L}$ é composta pelos operadores diferenciais parciais, da seguinte maneira:

$$\mathbf{L} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \end{bmatrix} \tag{67}$$

A relação entre tensão e deformação, ou equações constitutivas, é comumente denominada Lei de Hook, e é dada de acordo com Logan (2022), para um material isotrópico, em termos do módulo de Young ($E$) e o coeficiente de Poisson ($\nu$), ambos obtidos experimentalmente. De forma similar à relação dentre deformação e deslocamento, a relação tensão-deformação é expressa por:

$$\sigma = \mathbf{D}\varepsilon \tag{68}$$

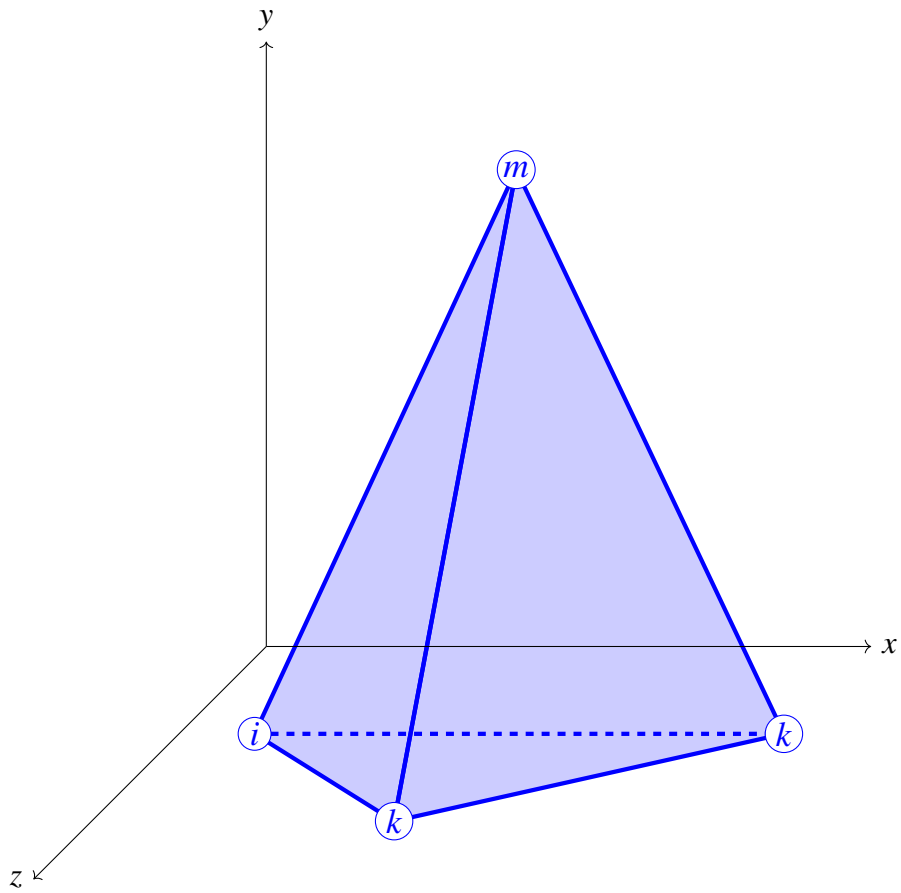
$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{69}$$

em que o termo $D$ é denominado matriz constitutiva, que é constante ao longo de todo o sólido.

### 3.1.2 As funções de interpolação

Como já mencionado, a função de campo tratada aqui é o deslocamento sobre o sólido. Para cada elemento discretizado, essa função é interpolada por um polinômio, definido pelo valor do próprio campo nos nós do elemento. Em um elemento tetraédico, como o da figura 7, há quatro nós ($i$, $j$, $k$ e $m$), e em cada um o campo de deslocamento tem três componentes. Essa liberdade do deslocamento que o campo tem nos nós é chamada de grau de liberdade. As funções de deslocamento, expressão 65, como ditas anteriormes, são defnidas como lineares, o que garate a compatibilidade entre cada elemento, fazendo com que não existam desconitnuidades no campo de deslocamentos de acordo com Logan (2022). A função $\varphi$, portanto, pode ser decomposta em termos de suas variáveis da seguinte forma:

Figura 7 – Elemento tetraédrico




Fonte: Elaborado pelo autor (().2022)

$$\varphi = \begin{Bmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{Bmatrix} = \begin{Bmatrix} a_1 + a_2x + a_3y + a_4z \\ b_1 + b_2x + b_3y + b_4z \\ c_1 + c_2x + c_3y + c_4z \end{Bmatrix} \tag{70}$$

Para encotrar esses fatores, basta aplicar as funções em cada nó.

$$\begin{cases} u_i = a_1 + a_2x_i + a_3y_i + a_4z_i \\ u_j = a_1 + a_2x_j + a_3y_j + a_4z_j \\ u_k = a_1 + a_2x_k + a_3y_k + a_4z_k \\ u_m = a_1 + a_2x_m + a_3y_m + a_4z_i \end{cases} \tag{71}$$

Esse sistema pode ser rearranjado na seguinte forma matricial:

$$\{\mathbf{u}\} = \begin{Bmatrix} u_i \\ u_j \\ u_k \\ u_m \end{Bmatrix} = \begin{bmatrix} 1 & x_i & y_i & z_i \\ 1 & x_j & y_j & z_j \\ 1 & x_k & y_k & z_k \\ 1 & x_m & y_m & z_m \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \tag{72}$$

Portanto,

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 1 & x_i & y_i & z_i \\ 1 & x_j & y_j & z_j \\ 1 & x_k & y_k & z_k \\ 1 & x_m & y_m & z_m \end{bmatrix}^{-1} \{\mathbf{u}\} \tag{73}$$

O mesmo procedimento pode ser aplicado às outras funções de deslocamento ($v$ e $w$).

# 4 PHILLIPO

PHILLIPO é um *solver* para campos de deformação elástica em estruturas discretizadas por elementos finitos, e segue a simbologia e o padrão dos algoritmos descritos por Zienkiewicz em sua obra intitulada *The Finite Element Method*, com algumas otimizações computacionais relacionadas a paralelismo e matrizes esparsas, e que visa constituir-se como referência didática 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 deste documento.

PHILLIPO foi idealizado, a princípio, como um projeto de aplicação do método de elementos finitos em um contexto de programação estruturada, porém, observou-se que essa abordagem é, senão obsoleta, já muito utilizada em pesquisas científicas. Portanto, optou-se em trazer uma visão de projeto de software, alterando o paradigma para a programação em despachos múltiplos (um forma alternativa à orientação a objetos), uma vez que tópicos envolvendo esses assuntos não são muito discutidos nas cadeiras dos cursos de engenharia (menos a de software, é claro), e que as vantagens desse tipo de abordagem vão desde a legibilidade do código, até o reaproveitamento de estruturas de dados e funções.

Um *solver*, ou em melhor português, um solucionador em MEF não é uma novidade no mundo acadêmico, nem no comercial. Softwares como Calculix (que é distribuído integrado com o FreeCAD) e o FreeFEM, que já conta com 7 mil commits em seu repositório, são continuamente produzidos e aprimorados desde antes da virada do milênio, um trabalho que demanda tempo e uma comunidade bem ativa.

Destarte, a pretensão de PHILLIPO não é fornecer uma alternativa a esses softwares, muito menos servir de módulo ou biblioteca para agregar algum deles, além do mais, a elaboração de programas robustos e confiáveis é um trabalho demorado e de muitas pessoas. O próprio FreeFEM já conta com mais de 7 mil commits em seu repositório, com a participação de 41 desenvolvedores.

A pretenção de PHILLIPO é construir uma aplicação simples utilizando o MEF, que possa aproveitar algumas ferramentas de contrução de código em Julia, como paralelismo e despachos múltiplos, para apresentar mais uma referência de programação em engenharia para os alunos do campus CCT, da UDESC, e, deste modo, evidenciar que é possível construir programas de engenharia de forma simples e legível, e que, por meio de uma linguagem de programação moderna, como Julia.

---

[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.

Neste capítulo é apresentado como é feita a distribuição é 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.

## 4.1 DISTRIBUIÇÃO PELO PKG.JL

O Pkg.jl é o gerenciador de pacotes anexado à Julia, tal como PIP é anexado ao Pyhton. Ele é reponsável por distruibuir, 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, utilizando-se de uma função do gerenciador de pacotes: a função *add*.

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. Isso é devido ao fato de que o versionamento é mantido

## 4.2 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 8 e 9.

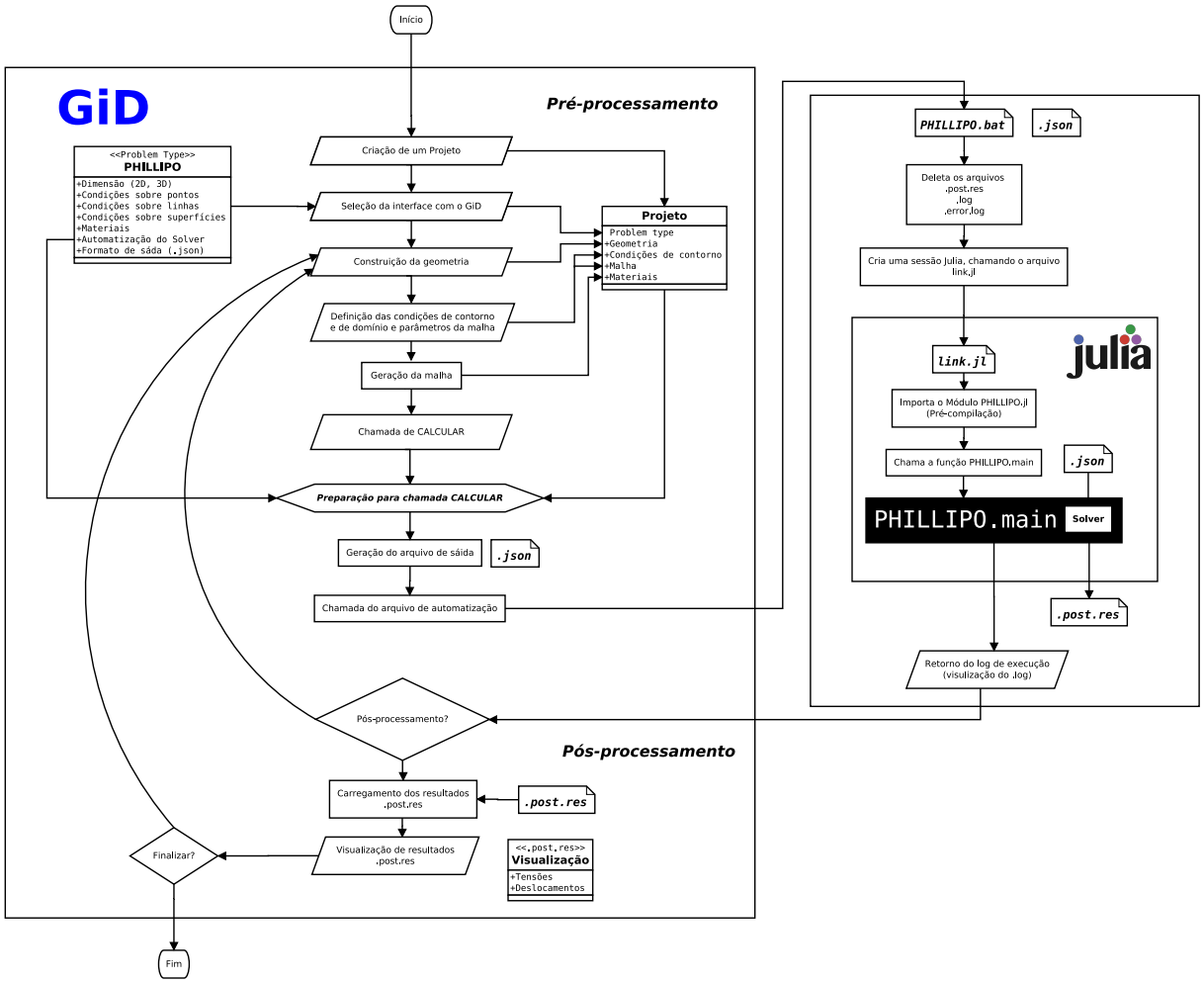
Nesses diagramas, é possível separar a execução de uma utilização normal do software em três partes principais:

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.

[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.

Figura 8 – Fluxograma de execução: GID

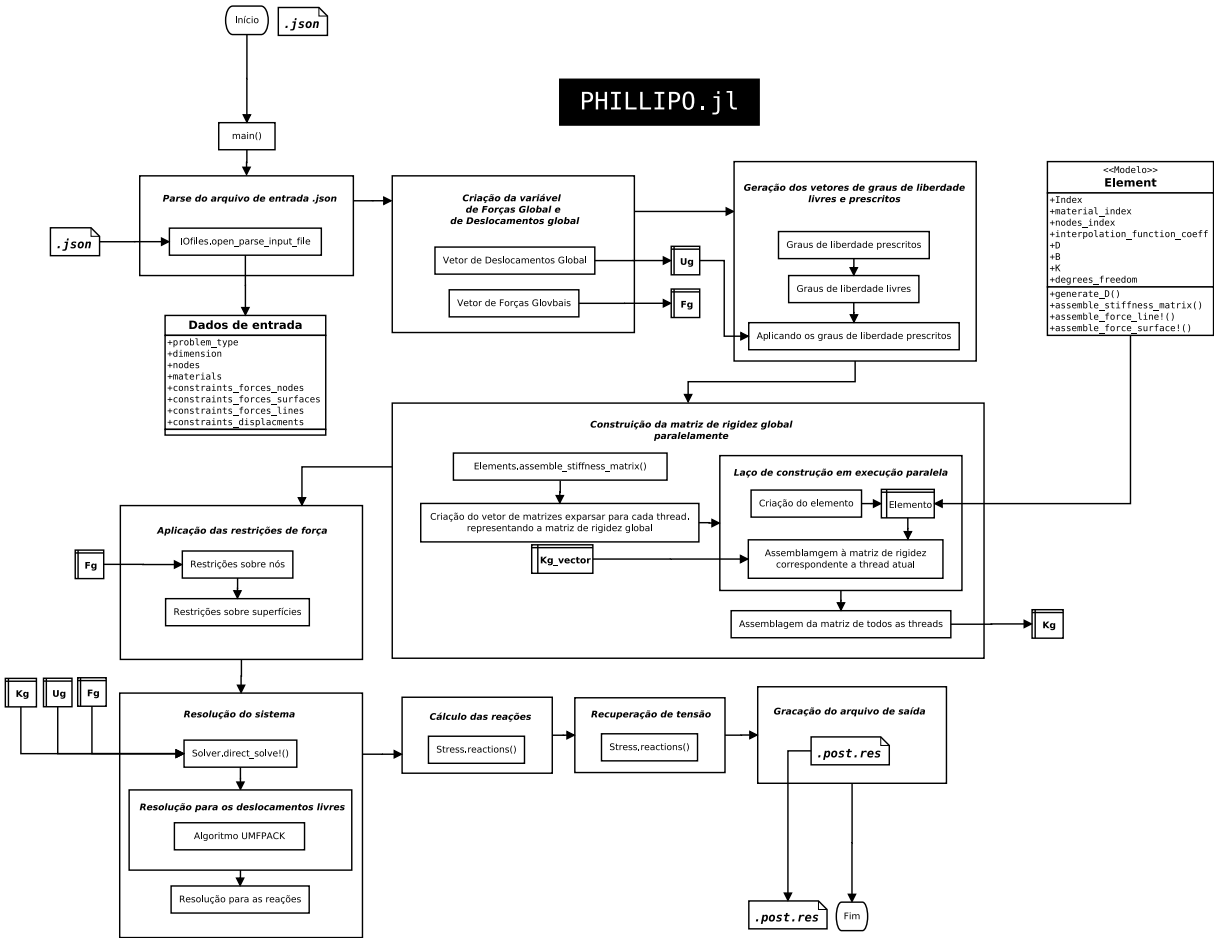



3. Pós-processamento; Parte em que o GID lê os arquivos de saída, e gera os gráficos de resultados.

### 4.2.1 Pré-processamento

O pré-processamento é realizado totalmente pelo GID, e consisti na definição da geometria, das condições de contorno e do material.

Figura 9 – Fluxograma de execução: PHILLIPO.jl




## 4.3 GID

O GID é um software utilizado como pré e pós-processamento, neste caso, para PHILLIPO. Nele 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 sua 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.

### 4.3.1 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

```
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
```

Figura 10 – Parte do arquivo de condições de contorno: PHILLIPO.cnd

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 no arquivo gerado pelo GID (geometria, malha, condições de contorno etc.) sejam salvas em um formato que o programa, o *solver* no caso, possa interpretar, ao passo que o *script* de execução automatiza o chamamento deste, 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 [2], uma forma de marcação de texto, que é interpretada pelo GID.

As linhas seguintes, 4 a 9, se referem aos valores da condição, neste caso, aos deslocamentos dos nós nas direções de *X*, *Y* e *Z*, e seus valores padrão, nesse caso,

[2] 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*.

# 5 JULIA

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, Capítulo: The scope of Julia)

## 5.1 ORIGEM

> "In short, because we are greedy."(Jeff Bezanson, Stefan Karpinski, Viral B. Shah e Alan Edelman, em *Why We Created Julia*)

Os criados de Julia eram usuários de várias linguagens, cada uma utilizada para uma tarefa específica. C, C++, Fortran, Python, MATLAB, R, Perl, Ruby, Lisp, Clojure, Mathematica, e até mesmo Java, eram algumas das linguagens que eles utilizavam em seu dia a dia. Cada uma delas tinha suas vantagens e desvantagens, mas nenhuma delas era capaz de suprir todas as suas necessidades. A

[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. Os componentes do módulo Base e as bibliotecas e ferramentas externas, que têm licença diferente, assim como a da própria Julia, podem ser consultados diretamente no repositório da linguagem: <https://github.com/JuliaLang/julia>.

*asdasd*

# 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, que garante, dentro de uma margem de abrangência do que se propõe o software, 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].

Para que um programa seja considerado válido, é preciso que passe tanto por uma bateria de testes, quanto por um processo de verificação & validação (V&V). O que ocorre, também, em programas sobre MEF. Verificação é o processo pelo qual se determina se um modelo computacional tem acurácia suficiente para representar o modelo matemático em que é embasado. Validação, por sua vez, é o processo para determinar a acurácia que um modelo computacional possui de representar a realidade, dentro dos limites que se propõe. (ASME)

[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

# REFERÊNCIAS

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 16.

ERNESTI, Peter Kaiser Johannes. **Python 3: The Comprehensive Guide to Hands-On Python Programming**. 1. ed. Rheinwerk Computing, 2022. ISBN 149322302X,9781493223022. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=D3E79CF42FF64C30BCBD1365173C229B>. Citado na página 16.

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 na página 37.

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 2 vezes nas páginas 25 e 30.

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 2 vezes nas páginas 15 e 34.

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 6 vezes nas páginas 19, 20, 21, 26, 27 e 35.

QUEK, G.R. Liu S. S. **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 35 e 36.

ROYLANCE, D. **Mechanics of Materials: Introduction to Elasticity**. Massachusetts Institute of Technology, 2000. Disponível em: <https://books.google.com.br/books?id=nP54tAEACAAJ>. Citado 3 vezes nas páginas 21, 22 e 27.

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 45.

ZIENKIEWICZ, O.C. **the Finite Element Method**: Volume 1: The basis. [S.l.]: Butterworth-Heinemann, 2000. Citado 3 vezes nas páginas 15, 32 e 33.

# ANEXO A – CÓDIGO FONTE DE PHILLIPO.JL

./src/PHILLIPO.jl

```
module PHILLIPO
    # Módulo do escopo principal
    include("./modules/includes.jl") # Módulos internos
    # 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_displacments    = input_dict["constraints"]["displacements"]

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

        # REMOVENDO ELEMENTOS NÃO UTILIZADOS
```

```
37      # esses elementos nulos são gerados pelo modo que o arquivo JSON
    é criado pelo GiD
38      # É uma falha que deve ser corrigida, mas que não é urgente.
39      pop!(nodes)
40      pop!(materials)
41      pop!(constraints_forces_nodes)
42      pop!(constraints_forces_lines)
43      pop!(constraints_forces_surfaces)
44      pop!(constraints_displacments)
45
46      if isempty(materials) error("Não há nenhum material definido!")
    end
47
48      # VARIÁVEIS do PROBLEMA
49      dimensions = input_dict["type"] == "3D" ? 3 : 2
50      nodes_length = length(nodes)
51      Fg = zeros(Float64, dimensions * nodes_length)
52      Ug = zeros(Float64, dimensions * nodes_length)
53
54      # GRAUS DE LIBERDADE: LIVRES E PRESCRITOS
55      if problem_type == "3D"
56          dof_prescribe = reduce(vcat, map(
57                  (x) -> [3 * x[1] - 2, 3 * x[1] - 1, 3 * x[1]],
58                  constraints_displacments
59              ))
60          dof_free = filter(x -> x  dof_prescribe, 1:dimensions*
    nodes_length)
61          # RESTRIÇÃO DE DESLOCAMENTO
62          Ug[dof_prescribe] = reduce(vcat, map((x) -> [x[2], x[3], x
    [4]], constraints_displacments))
63      else
64          dof_prescribe = reduce(vcat, map((x) -> [2 * x[1] - 1, 2 * x
    [1]], constraints_displacments))
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_displacments))
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
154             )
155
156             close(output_file)
157
158         end
159         print("Tempo total de execução: ")
160     end
161 end
```

./src/modules/includes.jl

```
1
2 # PHILLIP
3 # Script para adicionar todos os arquivos que contêm os módulos locais
4
5 include("IOfiles.jl")
6 include("Matrices.jl")
7 include("Elements.jl")
8 include("Solver.jl")
9 include("Stress.jl")
```

./src/modules/IOfiles.jl

```
1
2 # PHILLIPO
3 # Módulo: controle de entradas e saídas
4
5
6 module IOfiles
7
8     # MÓDULOS EXTERNOS
9     import JSON
10
11    # texto de cabeçalho (salvando durate a compilação)
12    header_msg_file = open(string(@__DIR__ ,"/header_msg.txt"), "r")
13    header_msg_text = read(header_msg_file, String)
14
15    function open_parse_input_file(file_name::String)::Dict
16        # Carrega e interpreta o arquivo de entrada
17        # Retorna um dicionário
18        JSON.parsefile(file_name, dicttype=Dict, use_mmap = true)
19    end
20
21    function header_prompt()
22        # Imprime o cabeçalho do prompt de execução do programa
23        # header_msg_file = open(string(@__DIR__,"header_msg.txt"), "r")
```

```
24          # header_msg_text::String = read(header_msg_file, String)
25          println(header_msg_text)
26      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
3 # Módulo: definição dos elementos e funções relacionadas
4
5 module Elements
6
7     #MÓDULOS EXTERNOS
8     import LinearAlgebra
9     using SparseArrays
10    import ..Matrices
11
12    abstract type Element end
13
14    struct TriangleLinear <: Element
15
16        index::Integer
17        material_index::Integer
18        nodes_index::Vector{Integer}
19        interpolation_function_coeff::Matrix{Real}
20        D::Matrix{Real}
21        B::Matrix{Real}
22        K::Matrix{Real}
23        degrees_freedom::Vector{Integer}
24
25        function TriangleLinear(triangle_element_vector::Vector{Any},
    materials::Vector{Any}, nodes::Vector{Any}, problem_type::String)
26
27            index          = Integer(triangle_element_vector[1])
28            material_index = Integer(triangle_element_vector[2])
29            nodes_index    = Vector{Integer}(triangle_element_vector
    [3:5])
30
31
32            i = Vector{Real}(nodes[nodes_index[1]])
33            j = Vector{Real}(nodes[nodes_index[2]])
34            m = Vector{Real}(nodes[nodes_index[3]])
35
36            position_nodes_matrix = [
37                1  i[1]  i[2];
38                1  j[1]  j[2];
39                1  m[1]  m[2]
40            ]
41
42            interpolation_function_coeff = LinearAlgebra.inv(
    position_nodes_matrix)
43
```

```
44              = 1/2 *  LinearAlgebra.det(position_nodes_matrix)
45
46          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)))
128
129             new(index, material_index, nodes_index,
    interpolation_function_coeff, D, B, K, degrees_freedom)
130         end
131     end
132
133
134     function generate_D(problem_type, material)::Matrix{<:Real}
135         # Gera a matrix constitutiva
136         E::Float64 = material[2] # Módulo de young
137         ::Float64 = material[3] # Coeficiente de Poisson
138
139         if problem_type == "plane_strain"
140             return E / ((1 + ) * (1 - 2)) * [
141                 (1 - )        0           ;
142                       (1 - ) 0            ;
143                 0       0       (1 - 2) / 2
144             ]
145         end
146
147         if problem_type == "plane_stress"
148             return E / (1 - ^2) * [
149                 1              0           ;
150                       1       0           ;
151                 0       0       (1 - ) / 2
152             ]
153         end
154
155         if problem_type == "3D"
156             return E / ((1 + ) * (1 - 2)) * [
157                 (1 - )                 0            0            0
        ;
158                       (1 - )           0            0            0
        ;
159                             (1 - )     0            0            0
        ;
160                 0       0       0          (1 - 2) / 2 0              0
          ;
161                 0       0       0          0             (1 - 2) / 2 0
          ;
162                 0       0       0          0             0             (1 -
    2) / 2
163             ]
```

```
164         end
165 
166         error("PHILLIPO: 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]
```

```
            nodes_index    = force[2:4]
            forces_vector  = force[5:7]

            dof_i = mapreduce(el -> [3 * el - i for i in 2:-1:0], vcat, nodes_index[1])
            dof_j = mapreduce(el -> [3 * el - i for i in 2:-1:0], vcat, nodes_index[2])
            dof_k = mapreduce(el -> [3 * el - i for i in 2:-1:0], vcat, nodes_index[3])

            node_i = nodes[nodes_index[1]]
            node_j = nodes[nodes_index[2]]
            node_k = nodes[nodes_index[3]]

            vector_ij = node_j .- node_i
            vector_ik = node_k .- node_i

             =  1/2 * LinearAlgebra.norm(LinearAlgebra.cross(vector_ij, vector_ik))
            F =  1/3 *  .* forces_vector

            Fg[dof_i] += F
            Fg[dof_j] += F
            Fg[dof_k] += F
        end
    end

end
```

./src/modules/Matrices.jl

```

# PHILLIPO
# Módulo: construção de matrizes esparsas baseada em coordenadas
# Este arquivo é construído sobre o FEMSparse.jl (módulo utilizado no JuliaFEM.jl)

module Matrices

    using SparseArrays
    import Base.sum
    export SparseMatrixCOO, spCOO, sum, add!

    mutable struct SparseMatrixCOO{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
        I :: Vector{Ti}
        J :: Vector{Ti}
        V :: Vector{Tv}
```

```
16  end
17
18  spCOO(A::Matrix{<:Number}) = SparseMatrixCOO(A)
19  SparseMatrixCOO() = SparseMatrixCOO(Int[], Int[], Float64[])
20  SparseMatrixCOO(A::SparseMatrixCSC{Tv,Ti}) where {Tv, Ti<:Integer} =
    SparseMatrixCOO(findnz(A)...)
21  SparseMatrixCOO(A::Matrix{<:Real}) = SparseMatrixCOO(sparse(A))
22  SparseArrays.SparseMatrixCSC(A::SparseMatrixCOO) = sparse(A.I, A.J,
    A.V)
23  Base.isempty(A::SparseMatrixCOO) = isempty(A.I) && isempty(A.J) &&
    isempty(A.V)
24  Base.size(A::SparseMatrixCOO) = isempty(A) ? (0, 0) : (maximum(A.I),
    maximum(A.J))
25  Base.size(A::SparseMatrixCOO, idx::Int) = size(A)[idx]
26  Base.Matrix(A::SparseMatrixCOO) = Matrix(SparseMatrixCSC(A))
27
28  get_nonzero_rows(A::SparseMatrixCOO) = unique(A.I[findall(!iszero, A
    .V)])
29  get_nonzero_columns(A::SparseMatrixCOO) = unique(A.J[findall(!iszero
    , A.V)])
30
31  function Base.getindex(A::SparseMatrixCOO{Tv, Ti}, i::Ti, j::Ti)
    where {Tv, Ti}
32      if length(A.V) > 1_000_000
33          @warn("Performance warning: indexing of COO sparse matrix is
     slow.")
34      end
35      p = (A.I .== i) .& (A.J .== j)
36      return sum(A.V[p])
37  end
38
39  """
40      add!(A, i, j, v)
41  Add new value to sparse matrix `A` to location (`i`,`j`).
42  """
43  function add!(A::SparseMatrixCOO, i, j, v)
44      push!(A.I, i)
45      push!(A.J, j)
46      push!(A.V, v)
47      return nothing
48  end
49
50  function Base.empty!(A::SparseMatrixCOO)
51      empty!(A.I)
52      empty!(A.J)
53      empty!(A.V)
54      return nothing
```

```
    end

    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))
        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

    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"
                pop!(input_elements["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"
                pop!(input_elements["triangles"])
                map_function = e -> begin
                    el = Elements.TriangleLinear(e, materials, nodes, problem_type)
                    el.D * el.B * Ug[el.degrees_freedom]
                end
            end
        end
```

```
30
31              = Vector{Vector{Float64}}(map(
32               map_function,
33               input_elements[type]
34           ))
35           vm = von_misses.()
36           , vm
37       end
38
39       von_misses(::Vector{<:Real}) = length() == 3 ? von_misses_2D() :
     von_misses_3D()
40
41       function von_misses_2D(::Vector{<:Real})
42           ([1]^2 - [1] * [2] + [2]^2 + 3 * [3]^2)
43       end
44
45       function von_misses_3D(::Vector{<:Real})
46           ((([1] - [2])^2 + ([2] - [3])^2 + ([3] - [1])^2 + 6 * ([4]^2 +
     [5]^2 + [6]^2)) / 2)
47       end
48
49       function reactions(Kg::SparseMatrixCSC, Ug::Vector{<:Real}, d::
     Integer)
50           nodes_length = length(Ug)
51           Re = Kg * Ug
52           Re_sum = sum.([Re[i:d:nodes_length] for i in 1:d])
53           return Re, Re_sum
54       end
55
56   end
```