Rodrigo Broggi

Análise estrutural de tubulações *Pipe-in-Pipe*

São Paulo - Brasil

2014

Rodrigo Broggi

Análise estrutural de tubulações *Pipe-in-Pipe*

Monografia apresentada à Escola Politécnica
da Universidade de São Paulo para a obtenção
do título de Engenheiro Mecânico

Universidade de São Paulo – USP

Escola Politécnica

Departamento de Engenharia Mecânica

Orientador: Prof. Dr. Roberto Ramos Jr.

Coorientador: Prof. Dr. Luca Formaggia

São Paulo - Brasil

2014

Broggi, Rodrigo
Análise estrutural de tubulações pipe-in-pipe / R. Broggi. -- São Paulo, 2014.
237 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecânica.

1.Tubulações 2.Risers (Estudo numérico) I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecânica II.t.

Este trabalho é dedicado a: meus avós

Agradecimentos

Agradeço antes de tudo aos professores que me estimularam em todo meu percurso acadêmico, em especial: ao professor Roberto Ramos pela sua seriedade, disponibilidade e dedicação nos cursos ministrados, uma tendência cada vez mais rara no mundo universitário; ao professor Sandro Salsa, pela sua capacidade de ensinar temas complicadíssimos com simplicidade e pela sua preocupação com a real instrução de seus alunos; à memória do professor Martinho que, com afeto e dedicação, exerceu a sua profissão até o último dos seus dias.

Agradeço à Escola Politécnica e seus funcionários por todo o apoio nos anos de formação e pela oportunidade a mim concedida de estudar no exterior.

Finalmente agradeço à minha família por ter estimulado e possibilitado meus estudos.

"If people do not believe that mathematics is simple,
it is only because they do not realize how complicated life is.
(John von Neumann)

Resumo

No presente trabalho são estudadas, sob o aspecto da resistência estrutural, tubulações conhecidas como *risers* na indústria do petróleo. Especificamente será enfatizado um tipo de configuração chamado *pipe-in-pipe* que vem ganhando importância e interesse no âmbito da exploração *offshore* principalmente graças à sua eficiência no isolamento térmico. Sob algumas hipóteses iniciais é possível dividir o estudo em duas etapas onde os efeitos que derivam do peso próprio imerso da estrutura e do carregamento imposto pela corrente marítima são desacoplados dos efeitos das pressões e da distribuição de temperatura: na primeira etapa, os carregamentos são abordados de modo global usando as equações de catenária e com essa técnica é possível determinar as trações resultantes em uma seção transversal genérica, desprezando a resistência à flexão da tubulação. Nessa parte se concentram as não-linearidades do problema que derivam das condições de contorno, das relações cinemáticas e do carregamento proveniente das correntes marítimas. Dada a não-linearidade do modelo, é feito um estudo sobre a sensibilidade de resposta às variações dos diversos parâmetros de cálculo para entender a importância relativa de cada um no problema geral. A segunda parte é elaborada localmente de modo a poder conservar as hipóteses de axissimetria e de pequenas deformações. Um estudo comparativo é feito para avaliar a importância relativa dos efeitos decorrentes do gradiente de temperatura, fenômeno que é frequentemente desprezado na literatura.

A abordagem matemático-numérica se baseia em modernas técnicas e métodos quais formulações fracas ou variacionais para a resolução das equações a derivadas parciais que derivam da mecânica dos sólidos e das relações cinemáticas, constitutivas e de equilíbrio. Essa abordagem é ideal para aplicação de métodos numéricos como o *método dos elementos finitos* (FEM) e o método de Newton, principais instrumentos da teoria de análise numérica usados neste estudo. Foram desenvolvidos códigos em linguagem C++ com o auxílio da biblioteca *libmesh* para elementos finitos. Essa escolha foi feita pela eficiência que deriva das linguagens compiladas de relativo baixo nível, pela versatilidade em comunicação com diversos programas e extensões e pela facilidade de paralelização e refinamento de malha (AMR - *Adaptive Mesh Refinment*). Além disso foram escritos alguns *shell scripts* e *gnuplot scripts* para o gerenciamento dos executáveis e para o pós-processamento em sistemas UNIX.

Quanto aos resultados, além da grande coerência com a literatura, foi possível atingir um ótimo rendimento do ponto de vista numérico pelo fato de concentrar as não-linearidades na etapa global (que é intrinsecamente *1D*). Ainda no âmbito do problema global, foi descoberto um método inédito e robusto que combina as características dos métodos clássico e misto do método dos elementos finitos para um melhor rendimento sob o ponto de vista da eficiência numérica e da confiabilidade de convergência. Também foi possível verificar que o gradiente térmico tem um papel importante no estudo desse tipo de estrutura sendo responsável por uma série de modificações em relação à resposta em sua ausência.

Palavras-chaves: Pipe in pipe. Catenária. Elementos Finitos. C++. Mecânica dos sólidos. EDP não lineares.

Abstract

The aim of this work is to study the structural behavior of pipe-in-pipe risers. This kind of structure is important in the offshore industry due to its improved insulation performace. Under certain hypothesis it was possible to divide the problem into two parts by uncoupling load effects derived from current and self weight from load effects derived from thermal diffusion and pressures: in the first part, current and self weight loads have been tackled from a global perspective using the catenary equations. With this technic, it was possible to reach the traction in a given section neglecting the flexural resistance. In this part some nonlinearities of the problem have been dealt, namely boundary conditions, kinematic relation and current load expression. Given those nonlinearities, a study has been made to understand the relative importance of the input parameters by performing variations on each of them and testing their response sensibility. In the second part, the pressure and temperature loads have been tackled in a local fashion as to preserve axisymmetry and small-displacement hypothesis. A comparison study has been made to evaluate the relative importance of the thermal effects since it is often neglected in literature.

The mathematical approach is based on modern technics to solve partial differential equations e.g. use of weak formulation and some functional analysis contents. The differential equations arise from kinematic, equilibrium and constitutive relations and the numerical methods as the FEM or Newton's methods for their solution are easily implemented when the mentioned mathematical tools are used. Codes have been written in C++ language together with *libmesh* library for finite elements method. This choice has been made due to the efficiency of compiled low level languages, to the great compatibility with third part softwares and extensions and to facilities on parallel implementations and grid refinement procedures (like AMR - *Adaptive Mesh Refinment*). Furthermore some shell and gnuplot scripts have been written to the management and post-processing in UNIX systems.

Results were coherent with literature data and it was possible to reach good numerical performances since nonlinearities were present just in the intrinsically 1D global analysis. Yet, a new method was developed to solve the global analysis problem by combining both classic and mixed FEM formulations and reaching a more robust performance for convergence and stability. Finally the thermal effects have been proved to have an important role in the study of this kind of structures since they were responsible for considerable changes on final stress states both in magnitude and distribution.

Key-words: Pipe-in-pipe. FEM. C++. Catenary. Solid Mechanics. Partial Differential Equations.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Na indústria do petróleo, especificamente no segmento *offshore*, estruturas chamadas risers são utilizadas no transporte do fluido extraído pela árvore de natal[1], no solo submerso, até as unidades flutuantes. Essas tubulações são submetidas a diversos tipos de solicitações de natureza estática e dinâmica como pressões internas e externas, esforços impostos pelos movimentos da plataforma, forças decorrentes das ações de correntezas, grandes gradientes de temperatura e fenômenos de vibrações induzidas por vórtices.

Um tipo de *riser* muito utilizado pela sua eficiência e custo-benefício é o SCR (*steel catenary riser*). Essa estrutura é assim chamada pois apesar de ser rígida à flexão em pequenos comprimentos, quando os comprimentos são da ordem de grandeza da profundidade do oceano essa rigidez é praticamente desprezível e seus efeitos são confinados somente às proximidades das extremidades. Dessa maneira, a forma assumida pelo SCR é ditada principalmente pelo seu peso próprio imerso e pelas forças hidrodinâmicas às quais é submetido. Esse tipo de estrutura é utilizado em plataformas flutuantes, ditas FPU (*floating production unit*) e liga diretamente o TDP (*touch down point*) na base do oceano à plataforma, sem auxílio de bóias. A figura 1 esquematiza os principais elementos mencionados.

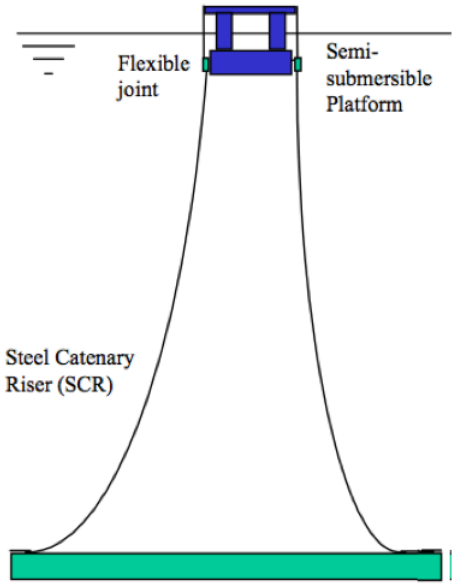


Figura 1: SCR e FPU

Fonte: (PESCE; MARTINS; CHAKRABARTI, 2005)

[1] A árvore de natal ou *Cristmas tree* é um conjunto de válvulas utilizadas na extração do petróleo do subsolo.

Muitos dos fenômenos relacionados ao problema são de caráter não linear sendo grande parte relacionada a efeitos dinâmicos. Além disso, o problema possui diversas escalas de tempo, comprometendo a eficiência e confiabilidade de métodos numéricos nesse contexto. Cabe observar que, no entanto, não são somente os efeitos dinâmicos a apresentar não linearidades: as condições de contorno do problema são do tipo contato, os deslocamentos globais possuem grandes magnitudes e os carregamentos que derivam das forças de correnteza são dependentes da forma da estrutura. Todos esses fatores são intrinsecamente não lineares e dificultam a análise já em âmbito estático.

Outro fenômeno interessante é como o fluxo interno influencia o efeito do peso próprio além de contribuir à rigidez à flexão efetiva dos cabos.

No âmbito desse tipo de estrutura, um tipo de *riser* que tem ganhado espaço nas aplicações são as chamadas tubulações *pipe-in-pipe* que consistem na sobreposição radial de múltiplas tubulações concêntricas, como ilustra a figura 2.

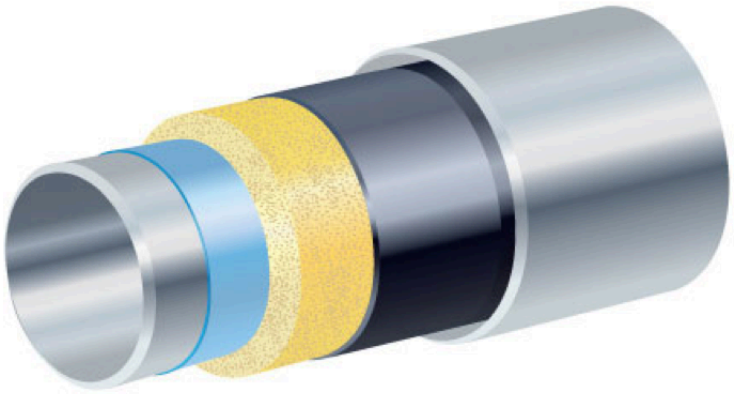


Figura 2: Estrutura *pipe-in-pipe*

Fonte: Bredero... (2014, http://www.brederoshaw.com/solutions/images/illustration_pip.jpg)

As principais partes são o tubo interno portante, o material isolante e o tubo externo protetor. O material isolante possui características de resistência mecânica inferiores aos outros e é protegido das ações de pressões hidrostáticas externas e danos mecânicos pelas outras duas camadas. Outra grande vantagem está no fato que, sendo o tubo protetor mantido a baixas temperaturas, podem ser utilizados processos convencionais e não muito onerosos para a proteção contra a corrosão, como proteção catódica e revestimentos.

Existem dois principais tipos de tubulações *pipe-in-pipe* ditas *compliant* e *non-compliant*: no primeiro tipo são presentes conexões entre os tubos interno e externo em intervalos freqüentes de comprimento e nesse caso a transferência de carregamentos é bem distribuída implicando uma expansão uniforme entre as tubulações. Para o segundo caso as conexões são feitas somente nas extremidades da tubulação ou espaçadas com distâncias

da ordem de grandeza de quilômetros. Em ambos os casos são empregados centralizadores para manter a concentricidade entre os tubos e prevenir contatos, esses são colocados na tubulação com espaços de 1-3 metros. Para uma classificação mais detalhada dos tipos de tubulações *pipe-in-pipe* e dos tipos de isolantes geralmente utilizados veja (HAUSNER; DIXON, 2002).

A principal razão para o emprego de tubulações *pipe-in-pipe* é o isolamento térmico pois, com a necessidade de instalações *offshore* cada vez mais profundas, a perda de calor do fluido tem um papel relevante pois essa é acompanhada de um aumento na sua viscosidade que acarreta a necessidade de maiores pressões para garantir seu escoamento. Esse quadro se traduz na necessidade de estruturas mais robustas. Portanto um isolamento térmico adequado pode levar a um menor custo dos *risers* pela necessidade de menor quantidade de materiais devido a necessidade de menor resistência, bem como maiores profundidades de exploração.

Claramente, dada a esbeltez da estrutura, grande peso próprio e grandes pressões hidrostáticas, outro importante critério de falha a ser levado em consideração é a flambagem. No presente trabalho esse critério não será tratado já que extensamente presente na literatura, por exemplo veja (KYRIAKIDES, 2002) e (KYRIAKIDES; VOGLER, 2002).

Neste projeto, pretende-se entender a relação complexa entre os fenômenos solicitantes descritos, delimitar a importância relativa de cada um para o estado de tensões resultante e desenvolver um método robusto que aborde o problema da forma mais completa possível. Os fenômenos considerados são os carregamentos de correnteza, de peso próprio imerso, de pressões interna e externa e de gradientes de temperatura, todos sob uma ótica estática. A abordagem dinâmica foi preterida pela maior preocupação do projeto em tratar de maneira concisa e direta os efeitos em termos de tensões equivalentes finais. Essas derivam principalmente dos efeitos estáticos mencionados.

2 Revisão da Literatura

Neste capítulo pretende-se fazer uma coleta de ferramentas e conceitos para familiarizar o leitor com o contexto no qual o trabalho será desenvolvido. Nos anexos são presentes alguns instrumentos que serão utilizados no texto e que também podem auxiliar uma leitura mais linear.

O problema estrutural, sob hipótese de pequenas deformações e deslocamentos, se resume matematicamente à solução da *equação de Navier* (A.34) acompanhada de condições de contorno. Para a solução dessa equação vetorial serão usados métodos da teoria geral de equações lineares à derivadas parciais que, além de permitir uma grande abrangência quanto às aplicações, é também ideal para a discretização e aproximação numérica, sendo essa última uma etapa fundamental dada a complexidade das equações e ausência de soluções analíticas. Nesse sentido se farão necessários alguns instrumentos de análise funcional que são descritos brevemente no anexo B.

2.1 Enquadramento na teoria geral das EDPs

A *equação de Navier* (A.34) associada às suas condições de contorno é uma equação linear a derivadas parciais de segunda ordem elíptica. Essa classe de equações é muito abrangente[1] e seu estudo é feito através da chamada *formulação fraca* ou *variacional* a qual se baseia em uma série de resultados da análise funcional.

A ideia por trás desse tipo de formulação é a de enquadrar uma classe de funções que seja *coerente* com o problema em questão e identificar univocamente, dentro dessa classe, a função que corresponde à solução do problema. O problema é identificar a classe que dê a *coerência* ao problema e que forneça contemporaneamente a existência de uma única solução.

No caso em questão, a incógnita é o campo de deslocamento $\vec{u}$. Pode-se intuir que a classe de funções que se procura deve ser tal que suas funções satisfaçam as condições de contorno do problema e que sejam, por exemplo, contínuas já que uma descontinuidade no campo de deslocamentos seria equivalente a uma fratura do material. Ao mesmo tempo, os espaços funcionais utilizados devem também possuir determinadas características para poder fruir de propriedades desejáveis que serão destacadas no decorrer do texto.

O estudo do problema estrutural é inclusive muito instrutivo para o entendimento da formulação fraca pois, como será visto, essa corresponde ao princípio do trabalho virtual

[1] Equações estacionárias como a de difusão e convecção de temperatura ou difusão de poluentes também pertencem a essa classe.

que por sua vez corresponde ao princípio da mínima energia potencial. Em outras palavras, o campo de deslocamentos procurado é aquele que minimiza a energia potencial total do sistema (note que essa inclui o trabalho realizado pelos carregamentos externos).

Na presente seção será desenvolvida a formulação fraca de modo usual sem atribuir no entanto significado físico ao procedimento que é puramente matemático. No fim da seção será feita a demonstração da equivalência com o princípio da mínima energia potencial total. Lembre-se que esta etapa se refere ao problema geral (não necessariamente o tratado no projeto) para situações onde são válidas as hipóteses de elasticidade-linear, pequenos deslocamentos e deformações e isotropia. Sucessivamente será necessária a introdução de novos desenvolvimentos pois a etapa global do problema não pode ser avaliada em pequenos deslocamentos.

2.1.1 A formulação variacional do problema geral

Considere um corpo de domínio genérico Ω submetido à ação de um campo de força de volume $\vec{b} : \Omega \to \mathbb{R}^3$ e de um campo de força de superfície $s : \Gamma_N \to \mathbb{R}^3$ com a fronteira do domínio $\partial\Omega = \Gamma_N \bigcup \Gamma_D$ tal que $\Gamma_N \bigcap \Gamma_D = \emptyset$. Considere ainda, sem perda de generalidade[2] que em Γ_D vale $\vec{u} = \vec{0}$ e dessa maneira o domínio é dividido em uma porção em que é engastado (Γ_D) e outra onde são impostos esforços.

A formulação fraca leva o problema que é funcional vetorial para um ambiente escalar através do uso de um operador linear[3], que no presente caso corresponde à multiplicação por uma *função teste* v (que pertence ao espaço funcional *coerente* mencionado anteriormente e que definiremos mais tarde) e integração no domínio. Em outras palavras partindo da equação (A.6) se obtém:

$$\begin{cases} \int_\Omega div(\mathbf{T}(\vec{x})).\vec{v}(\vec{x}) + \vec{b}(\vec{x}).\vec{v}(\vec{x})\mathrm{d}V(\vec{x}) = 0 & \forall \vec{v} \in coerente \\ \mathbf{T}(\vec{x}).\vec{n} = \vec{s}(\vec{x}) & \forall \vec{x} \in \Gamma_N \\ \vec{v}(\vec{x}) = \vec{0} & \forall \vec{x} \in \Gamma_D \end{cases} \tag{2.1}$$

Na sequência, usa-se a *fórmula de integração por partes*, também conhecida como *fórmula de Green*, que é um dos pilares da formulação fraca pois *descarrega a derivada à função teste*:

$$\int_\Omega div(\mathbf{T}(\vec{x})).\vec{v}(\vec{x})\mathrm{d}V(\vec{x}) = -\int_\Omega \mathbf{T}(\vec{x}) : \nabla\vec{v}(\vec{x})\mathrm{d}V(\vec{x}) + \int_{\partial\Omega} \mathbf{T}(\vec{x})).\vec{n}.\vec{v}(\vec{x})\mathrm{d}A(\vec{x}) \tag{2.2}$$

Note que $\mathbf{T} : \mathbf{C} = \sum_{i,j} T_{ij}C_{ij}$, que a função teste deve satisfazer a terceira equação de (2.1) e que o segundo termo a direita de (2.2) tem uma parte coincidente com a segunda

[2] É possível introduzir um campo, dito de relevo, $\vec{R}$ que assume exatamente os valores de $\vec{u}$ em Γ_D e resolver o problema para uma função $\vec{u}^* = \vec{u} - \vec{R}$ conduzindo o problema à formulação anterior com um termo adicional à direita.

[3] Ver anexo B.

equação de (2.1), obtém-se:

$$\int_\Omega \mathbf{T}(\vec{x}) : \nabla\vec{v}(\vec{x})\mathrm{d}V(\vec{x}) = \int_\Omega \vec{b}(\vec{x}).\vec{v}(\vec{x})\mathrm{d}V(\vec{x}) + \int_{\Gamma_N} \vec{s}(\vec{x})).\vec{v}(\vec{x})\mathrm{d}A(\vec{x}) \quad \forall\vec{v} \in coerente$$

Substituindo a lei de Hooke (A.27) e notando que:

- $\mathbf{T} : \nabla\vec{v} = \mathbf{T} : \mathbf{E}(\vec{v})$ [4]
- $Tr(\mathbf{E}(\vec{u}))\mathbf{I} : \nabla\vec{v} = Tr(\mathbf{E}(\vec{u}))Tr(\mathbf{E}(\vec{v}))$

Obtém-se:

$$a(\vec{u},\vec{v}) = F(\vec{v}) \quad \forall\vec{v} \in coerente$$

Com:

$$a(\vec{u},\vec{v}) = \int_\Omega \left[2\mu\mathbf{E}(\vec{u}) : \mathbf{E}(\vec{v}) + \lambda Tr(\mathbf{E}(\vec{u}))Tr(\mathbf{E}(\vec{v}))\right]\mathrm{d}V(\vec{x}) \tag{2.3a}$$

$$F(\vec{v}) = \int_\Omega \vec{b}.\vec{v}\mathrm{d}V(\vec{x}) + \int_{\partial\Omega} \vec{s}.\vec{v}\mathrm{d}A(\vec{x}) \tag{2.3b}$$

Cabe neste ponto introduzir uma discussão breve do espaço de funções *coerentes* nomeado anteriormente: a formulação variacional corresponde a um aumento da classe de funções ditas clássicas que seriam aquelas que são deriváveis ao menos duas vezes em sentido clássico (visto que o problema é de segunda ordem) e que satisfazem as condições de contorno pontualmente. Esse aumento corresponde ao espaço funcional $H^1(\Omega;\mathbb{R}^3)$ ou a subespaços desse que são *espaços de Sobolev*[5]. Para o entendimento da estrutura desse tipo de espaço é necessário um conhecimento consistente de instrumentos da análise funcional e alguns instrumentos de análise real, veja, por exemplo, (RUDIN, 1991) e (KOLMOGOROV; FOMIN, 1970).

Daqui em diante os instrumentos intrínsecos relacionados aos *espaços de Banach, Hilbert e Sobolev* serão considerados familiares ao leitor pois o tratamento desses temas foge ao escopo do presente texto. Uma descrição dos elementos essenciais é encontrada no anexo B. Para o leitor interessado ver (SALSA, 2010).

Enfim pode-se concluir que a formulação variacional do problema estrutural genérico (na ausência de dilatações térmicas) é:

Encontrar a função $\vec{u} \in H^1_{\Gamma_D}(\Omega;\mathbb{R}^3)$ tal que

$$a(\vec{u},\vec{v}) = F(\vec{v}) \quad \forall\vec{v} \in H^1_{\Gamma_D}(\Omega;\mathbb{R}^3) \tag{2.4}$$

[4] $\nabla v = \mathbf{E}(\vec{v}) + \frac{1}{2}\left\{\nabla\vec{v} - \nabla\vec{v}^T\right\}$ e o segundo termo à direita é anti-simétrico e portanto se anula quando é calculado um produto escalar com um tensor simétrico.

[5] Ver anexo B.

Tendo posto $H^1_{\Gamma_D}(\Omega;\mathbb{R}^3) = \{\vec{u} \in H^1(\Omega;\mathbb{R}^3) \quad t.q. \quad \vec{u} = 0 \quad em \quad \Gamma_D\}$.

Para o leitor que não tem familiaridade com a análise funcional, o entendimento dessa etapa pode ser comprometido e esse tipo de descrição pode parecer supérfluo, no entanto esse tipo de formulação facilita muito a etapa de aproximação numérica, pois conduz a resolução de um sistema linear, como veremos adiante.

2.1.2 A formulação variacional como princípio do trabalho virtual

Como dito anteriormente a formulação variacional ou fraca corresponde ao princípio ou teorema do trabalho virtual que vem enunciado a seguir:

Teorema 2.1. ***(Trabalho Virtual)*** *Dado um campo qualquer de deslocamento $\vec{v}$ congruente e cinematicamente admissível, ou seja, que obedece às mesmas condições de contorno na parcela de fronteira Γ_D que as do problema relacionado, então o trabalho interno W_i feito pelas tensões internas estaticamente admissíveis (que sejam conforme a A.6) sob tal campo de deslocamento, iguala o trabalho externo W_e feito pelos campos de força de volume e de superfície, respectivamente $\vec{b}$ e $\vec{s}$. Ou seja:*

$$\int_\Omega \sum_{i,j=1}^{3} (\sigma_{ij}\epsilon^*_{ij})\mathrm{d}V(\vec{x}) = \int_\Omega \sum_{i=1}^{3}(b_i v_i)\mathrm{d}V(\vec{x}) + \int_{\Gamma_N} \sum_{i=1}^{3}(s_i v_i)\mathrm{d}A(\vec{x}) \quad \forall \vec{v} \in \quad congruente \tag{2.5}$$

Com $\epsilon^*_{ij} = \frac{1}{2}\left\{\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right\}$ quando é válida a hipótese de pequenos deslocamentos.

A equivalência da formulação fraca com o PTV é quase imediata, basta notar que a formulação fraca é o PTV com a substituição da *lei de Hooke* (A.27) e que o espaço das funções congruentes é exatamente $H^1_{\Gamma_D}(\Omega;\mathbb{R}^3)$. O PTV é um importante instrumento não só no âmbito da formulação fraca de um problema mas também no estudo de problemas hiperestáticos pois permite o cálculo de reações e deslocamentos incógnitos através da imposição de deslocamentos virtuais em pontos oportunamente colocados.

Pode-se ainda demonstrar a equivalência da formulação fraca com o princípio da mínima energia potencial total (EPT). Seja portanto o caso especial onde são válidas as hipóteses de elasticidade-linear, pequenas deformações e isotropia (ver seção A.3.1) a EPT total é dada pela expressão:

$$EPT(\vec{v}) = \frac{1}{2}\int_\Omega \left[2\mu|\mathbf{E}(\vec{v})|^2 + \lambda(Tr(\mathbf{E}(\vec{v})))^2\right]\mathrm{d}V(\vec{x}) - \int_\Omega \vec{b}.\vec{v}\mathrm{d}V(\vec{x}) - \int_{\partial\Omega} \vec{s}.\vec{v}\mathrm{d}A(\vec{x}) \tag{2.6}$$

Ou seja:

$$EPT(\vec{v}) = \frac{1}{2}a(\vec{v},\vec{v}) - F(\vec{v}) \tag{2.7}$$

E sendo $a(.,.)$ uma forma bilinear, contínua e simétrica ($a(\vec{u},\vec{v}) = a(\vec{v},\vec{u})$) e $F(.)$ um operador linear vale o seguinte:

Teorema 2.2. ***(Energia Potencial Mínima)*** *A formulação fraca ou variacional (2.4) é equivalente à:*

$$EPT(\vec{u}) = \min_{\vec{v} \in H^1_{\Gamma_D}(\Omega;\mathbb{R}^3)} EPT(\vec{v}) \tag{2.8}$$

Demonstração: $\forall \epsilon \in \mathbb{R}$ e $\forall \vec{v} \in H^1_{\Gamma_D}(\Omega;\mathbb{R}^3)$ tem-se:

$$\begin{aligned} EPT(\vec{u}+\epsilon\vec{v}) - EPT(\vec{u}) &= \frac{1}{2}\{a(\vec{u}+\epsilon\vec{v}, \vec{u}+\epsilon\vec{v}) - a(\vec{u},\vec{v})\} - F(\vec{u}+\epsilon\vec{v}) + F(\vec{u}) \\ &= \epsilon\{a(\vec{u},\vec{v}) - F(\vec{v})\} + \frac{1}{2}\epsilon^2 a(\vec{v},\vec{v}), \end{aligned}$$

Então se $\vec{u}$ é solução de (2.4), tem-se que $a(\vec{u},\vec{v}) - F(\vec{v}) = 0$, portanto:

$$EPT(\vec{u}+\epsilon\vec{v}) - EPT(\vec{u}) = \frac{1}{2}\epsilon^2 a(\vec{v},\vec{v}) \geq 0$$

Assim $\vec{u}$ minimiza a EPT ($EPT(\vec{u}) \leq EPT(\vec{u}+\epsilon\vec{v})$). Por outro lado, se $\vec{u}$ minimiza a EPT, tem-se que $EPT(\vec{u}) \leq EPT(\vec{u}+\epsilon\vec{v})$, e portanto:

$$\epsilon\{a(\vec{u},\vec{v}) - F(\vec{v})\} + \frac{1}{2}\epsilon^2 a(\vec{v},\vec{v}) \geq 0$$

Mas essa inequação força o anulamento do termo que multiplica ϵ porque sendo ϵ genérico pode-se tomar o limite para ϵ tendendo a zero à esquerda ($lim_{\epsilon\to 0^-}$) ou o limite de ϵ tendendo a zero pela direita ($lim_{\epsilon\to 0^+}$) e em ambos os casos a magnitude do primeiro termo domina a do segundo termo e como deve ser maior ou igual a zero para ambos os limites de ϵ então se conclui a afirmação, i.é. a (2.4). $\diamondsuit$

2.1.3 Questões de existência e unicidade

Antes de qualquer procedimento de cálculo e de resolução de equações a derivadas parciais, deve-se meditar sobre a existência e unicidade de uma solução pois nem sempre equações a derivadas parciais possuem soluções e nem sempre essas são únicas. Nesse caso um procedimento de solução numérica pode dar resultados não coerentes; por isso, teoremas de unicidade e existência não são uma simples formalidade matemática, mas uma pré-avaliação da consistência de um modelo matemático que é essencial para etapas numéricas. Por mais que um modelo matemático seja coerente e fiel à realidade, a existência de uma solução experimental não comporta a existência de uma solução para o modelo matemático. Por isso cabe ao engenheiro investigar a *resolubilidade* das equações.

Para as equações a derivadas parciais lineares os teoremas principais que regulam a existência e unicidade de uma solução são: *teorema de Riez*, *teorema de Lax-Milgran* e *alternativa de Fredholm*. Para problemas não lineares, são necessários teoremas mais sofisticados como *teorema do ponto fixo* e os *teoremas de Shauder*.

No caso em questão será suficiente o uso do *teorema de Riez* (quando $\Gamma_D \neq \emptyset$) pois, como se virá a compreender, a forma bilinear $a(.,.)$ é *contínua*, *coerciva* e *simétrica* e portanto induz um produto interno ou escalar no espaço funcional $H^1_{\Gamma_D}(\Omega;\mathbb{R}^3)$. Para o caso $\Gamma_D = \emptyset$ a situação é mais complicada pois a solução não é única (se $\vec{u}_0$ é solução então qualquer sobreposição de movimento rígido do corpo $\vec{u}_0 + \vec{v}$ também é solução) e nesse caso deve-se selecionar a solução mais adequada. Um exemplo relevante dessa etapa é como a simplificação do modelo a um problema plano 2D, por exemplo axissimétrico, pode levar à ausência de engastamentos e a fronteira do domínio, nesse caso, é descrita inteiramente por condições ditas *de Neumann*, i.é. ($\Gamma_N = \partial\Omega$). Na realidade, qualquer uma das soluções possíveis é suficiente para a análise estrutural pois os critérios de falha, geralmente, não consideram o campo de deslocamentos mas sim o estado tensorial e esse último é unicamente determinado pois movimentos rígidos $\vec{v}$ possuem tensor deformação $\mathbf{E}$ nulo e portanto para todas as soluções do tipo $\vec{u}_0 + \vec{v}$ os estados de deformação e consequentemente de tensão, são os mesmos.

Os problemas estruturais são um ótimo ambiente para estudar unicidade e existência pois os teoremas abstratos mencionados frequentemente levam a conclusões de grande interesse físico e muitas vezes intuitivos: por exemplo, usando a *alternativa de Fredholm* para um problema com $\Gamma_D = \emptyset$, conclui-se que um problema estático é solúvel somente se as forças externas são auto-equilibradas.

2.2 Aproximação numérica pelo método de Galerkin

Antes de prosseguir para o método propriamente dito cabe uma discussão sobre métodos de aproximação numérica. Uma maneira de se desenvolver uma intuição sobre o método dos elementos finitos (FEM) é entender antes a ideia do método de *Rayleigh-Ritz* (RR). Uma descrição didática e breve do tema pode ser encontrada em (CORIGLIANO; TALIERCIO, 2005). Para uma descrição mais completa e consagrada veja (BATH, 1996).

2.2.1 O método de Rayleigh-Ritz e introdução ao método dos elementos finitos

A idéia do método RR é tomar a equação da energia potencial total do problema e introduzir um modelo simplificado de deslocamentos que seja coerente com as condições ditas essenciais (Γ_D). Em seguida o modelo de deslocamentos é substituído na equação da energia potencial (2.6) e é imposta a estacionariedade igualando as derivadas em relação a cada um dos coeficientes a zero. Esse procedimento conduz a um sistema linear de ordem n onde n é o número de parâmetros incógnitos.

Exemplo ilustrativo Seja um problema plano onde o domínio (Ω) coincide com um quadrado de vértices em $(1:(0,0),\ 2:(1,0),\ 3:(1,1),\ 4:(0,1))$, o lado esquerdo

é engastado e sobre o lado superior age um carregamento uniformemente distribuído p, direção vertical e sentido contrário ao do eixo y (veja a figura 3). Introduz-se, como modelo cinemático, um polinômio de segundo grau, que em 2D seria do tipo: $\overline{u}_x(x,y) = c_1 + c_2x + c_3y + c_4xy + c_5x^2 + c_6y^2$ e $\overline{u}_y(x,y) = c_7 + c_8x + c_9y + c_{10}xy + c_{11}x^2 + c_{12}y^2$, ou seja, é feita uma discretização com 12 graus de liberdade. Esses graus de liberdade serão reduzidos pelo fato de que o modelo deve satisfazer as condições de contorno em Γ_D: $\overline{u}_x(0,y) = c_1 + c_3y + c_6y^2 = 0$ e $\overline{u}_y(0,y) = c_7 + c_9y + c_{12}y^2 = 0\ \forall y$. Logo $c_1 = c_3 = c_6 = c_7 = c_9 = c_{12} = 0$.

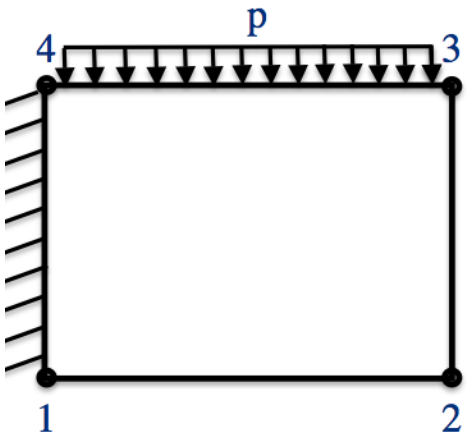


Figura 3: Exemplo ilustrativo.

Portanto o modelo congruente final a ser substituído na equação da energia potencial (2.6) é $\overline{\overline{u}} = [\overline{u}_x, \overline{u}_y, 0]$ com $\overline{u}_x(x,y) = c_2x + c_4xy + c_5x^2$ e $\overline{u}_y(x,y) = c_8x + c_{10}xy + c_{11}x^2$ i.é um modelo de 6 graus de liberdade.

Finalmente o modelo é substituído na expressão da EPT (2.6) e são impostas as equações de estacionariedade $\frac{\partial EPT(\overline{\overline{u}})}{\partial c_i} = 0 \quad i = 2, 4, 5, 8, 10, 11$ que conduzem a um sistema linear de sexto grau com incógnitas c_2, c_4, c_5, c_8, c_{10}, c_{11}. Uma vez resolvido o sistema, o campo de deslocamentos fica determinado e, consequentemente, via equações de congruência e de compatibilidade, obtêm-se as tensões. São omitidos os cálculos pois essa etapa é puramente ilustrativa.

O método dos elementos finitos é similar ao método RR porém possui três grandes vantagens:

i No método dos elementos finitos, os parâmetros do modelo cinemático são as próprias deslocamentos em pontos colocados no domínio, ditos nós;

ii No método dos elementos finitos pode-se refinar o modelo cinemático aumentando o grau dos polinômios usados ou aumentando o número de elementos disponíveis;

iii O método dos elementos finitos é facilmente automatizável;

Observe que o fato de atribuir deslocamentos em nós como parâmetros do modelo implica restrições quanto ao campo de deslocamentos: o tipo de forma e a quantidade de nós dos elementos ditam o grau do polinômio usado, veja o exemplo:

No caso anterior, faça-se uma discretização com um único elemento finito quadrado e como parâmetros do modelo cinemático as deslocamentos nos vértices do domínio. Para obter um modelo onde os parâmetros indiquem exatamente as deslocamentos em cada nó deve-se definir as *funções de forma* dos nós. Por exemplo, a função de forma do nó 3 deve ser $\Phi_3(x,y)$ tal que $\Phi_3(0,0)=0$, $\Phi_3(0,1)=0$, $\Phi_3(1,0)=0$ e $\Phi_3(1,1)=1$. Tomando $\Phi_3(x,y)=c_1+c_2x+c_3y+c_4xy$, a imposição das equações anteriores fornece: $\Phi_3(x,y)=xy$. Analogamente encontram-se as equações de $\Phi_1(x,y)$, $\Phi_2(x,y)$ e $\Phi_4(x,y)$. Com essas o modelo cinemático a ser substituído na expressão da EPT (2.6) é $\overline{\vec{u}}=[\overline{u}_x,\overline{u}_y,0]$ com:

$$\overline{u}_x(x,y)=u_1^x\Phi_1(x,y)+u_2^x\Phi_2(x,y)+u_3^x\Phi_3(x,y)+u_4^x\Phi_4(x,y)$$

$$\overline{u}_y(x,y)=u_1^y\Phi_1(x,y)+u_2^y\Phi_2(x,y)+u_3^y\Phi_3(x,y)+u_4^y\Phi_4(x,y)$$

Uma segunda possibilidade seria a de discretizar o domínio com dois elementos finitos triangulares: *elemento 1* delimitado pelos vértices $1,2,4$ do domínio, com referência local $1_l,2_l,3_l$ respectivamente, e *elemento 2* delimitado pelos vértices $3,4,2$ com referência local $1_l,2_l,3_l$ respectivamente (veja a figura 4). Nesse caso as funções de forma serão do tipo $\Phi_i(x_l,y_l)=c_1+c_2x_l+c_3y_l$ e para cada um dos elementos o modelo cinemático é descrito em relação aos graus de liberdade locais:

$$\overline{u}_x^{e_i}(x_l,y_l)=u_1^{(x,l)}\Phi_1(x_l,y_l)+u_2^{(x,l)}\Phi_2(x_l,y_l)+u_3^{(x,l)}\Phi_3(x_l,y_l)\quad i=1,2$$

$$\overline{u}_y^{e_i}(x_l,y_l)=u_1^{(y,l)}\Phi_1(x_l,y_l)+u_2^{(y,l)}\Phi_2(x_l,y_l)+u_3^{(y,l)}\Phi_3(x_l,y_l)\quad i=1,2$$

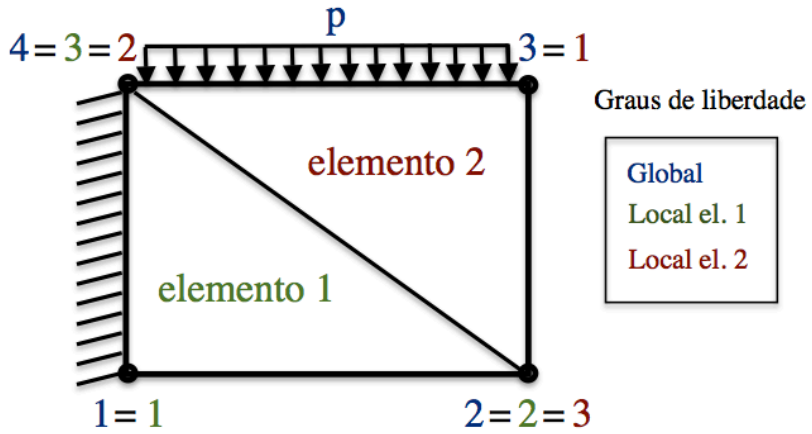


Figura 4: Exemplo ilustrativo segunda discretização.

Após a etapa de integração no domínio, cada elemento possui uma correspondente matriz de rigidez $\mathbf{K}_e$ e a equação da energia potencial, após a discretização decorrente do

modelo assume a forma:

$$EPT(\overline{\vec{u}}) = \sum_{e=1}^{n_e} \vec{U}_e^T \mathbf{K}_e \vec{U}_e - \vec{F}_e.\vec{U}_e \tag{2.11}$$

Com n_e sendo o número de elementos, que no caso é 2, e $\vec{U}_e = [u_1^{l,x}, u_2^{l,x}, u_3^{l,x}, u_1^{l,y}, u_2^{l,y}, u_3^{l,y}]^T$ o vetor de incógnitas em coordenadas locais.

Em seguida, o modelo cinemático passa por uma etapa de "montagem" onde todos os graus de liberdade locais são escritos todos em função dos graus de liberdade globais e desta maneira, são suprimidos graus de liberdade redundantes (como os graus de liberdade 3_l do elemento 1 e 2_l do elemento 2 no exemplo anterior). Essa "montagem" pode ser feita com o auxílio de um mapa de conectividade[6] (modo mais vantajoso numericamente) ou simplesmente com o auxílio de matrizes de conectividade que são matrizes esparsas $\mathbf{L}_e$ que fornecem a relação $\vec{U}_e = \mathbf{L}_e\vec{U}$, sendo $\vec{U}$ o vetor de todas as coordenadas globais.

Após essa última operação a EPT assume finalmente sua configuração final:

$$EPT(\overline{\vec{u}}) = \vec{U}^T \mathbf{K} \vec{U} - \vec{F}.\vec{U} \tag{2.12}$$

Onde a matriz $\mathbf{K}$ é a matriz de rigidez (quadrada e de ordem n) e n é o número total de graus de liberdade do domínio discretizado (incluindo os graus de liberdade colocados na fronteira Γ_D) que, no caso do exemplo anterior, são 8. Impondo estacionariedade em relação aos graus de liberdade se chega ao sistema linear:

$$\mathbf{K}\vec{U} = \vec{F} \tag{2.13}$$

A última observação é que a matriz $\mathbf{K}$ é singular e o sistema não é solúvel porque contém os graus de liberdade da fronteira Γ_D. Para a resolução a 2.13 é rescrita colocando por último todos os graus de liberdade de $\mathbf{K}$ correspondentes aos nós colocados em Γ_D (no exemplo anterior são os nós 1 e 4) e esse re-ordenamento resulta na segunte:

$$\begin{bmatrix} \mathbf{K}_{\Omega\Omega} & \mathbf{K}_{\Omega\Gamma_D} \\ \mathbf{K}_{\Omega\Gamma_D} & \mathbf{K}_{\Gamma_D\Gamma_D} \end{bmatrix} \begin{bmatrix} \vec{U}_\Omega \\ \vec{U}_{\Gamma_D} \end{bmatrix} = \begin{bmatrix} \vec{F}_\Omega \\ \vec{F}_{\Gamma_D} \end{bmatrix} \tag{2.14}$$

Pelo fato de que $\vec{U}_{\Gamma_D}$ é conhecido, pode-se remanejar o sistema e chegar a:

$$\mathbf{K}_{\Omega\Omega}\vec{U}_\Omega = \vec{F}_\Omega - \mathbf{K}_{\Omega\Gamma_D}\vec{U}_{\Gamma_D} \tag{2.15}$$

Que não é um sistema singular.

[6] Esse mapa armazena a correspondência em coordenadas globais de todas as coordenadas locais.

2.2.2 O método de Galerkin

O *método de Galerkin* é uma generalização da aplicação do método dos elementos finitos a formulações fracas de equações elípticas que não levem a uma forma bilinear simétrica. Em última análise, isto significa que a formulação variacional não equivale ao problema de mínimo de um funcional. Por exemplo, pode-se considerar a equação de difusão e convecção de temperatura estacionária: $(\Delta T(\vec{x}) + \vec{b}.\nabla T(\vec{x}) = 0 \quad \forall \vec{x} \in \Omega)$ que, quando conduzida à formulação fraca, gera um adendo a esquerda uma forma bilinear não simétrica.

Considera-se, então, a forma bilinear genérica $a(.,.)$, não necessariamente simétrica, e a formulação fraca geral:

Encontrar $u \in H^1_{\Gamma_D}(\Omega)$ tal que: $a(u,v) = F(v) \quad \forall v \in H^1_{\Gamma_D}(\Omega)$.

Com $F(.)$ um funcional sobre V.

O espaço $H^1_{\Gamma_D}(\Omega)$ é infinito dimensional e o método de Galerkin é uma espécie de projeção[7] da solução em um espaço finito dimensional. Assim, tendo uma família de espaços finito dimensionais $V_h \subset V = H^1_{\Gamma_D}(\Omega)$ que possuam a propriedade de saturação em relação a V[8], onde h é um parâmetro positivo que dita a dimensão do espaço, encontro a solução aproximada $u_h \in V_h$ que se "aproxima" ao máximo da solução $u \in V$ respeitando a equação da formulação fraca. Portanto o método de Galerkin corresponde à seguinte formulação:

Teorema 2.3 (Método de Galerkin)**.** *Encontrar $u_h \in V_h$ tal que:*

$$a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h \tag{2.16}$$

No âmbito dos elementos finitos para o caso Lagrangeano, será definido o espaço V_h, dito *dos elementos finitos*. Seja portanto o domínio discretizado em elementos geométricos $K_i,\ i = 1, \ldots, n_e$ (no caso 2D, triângulos ou quadriláteros, e 3D, tetraedros ou hexaedros, etc.) e seja $\mathbb{P}_r(K_i)$ o espaço dos polinômios de grau menor ou igual a r definido sobre o elemento genérico K_i. A família dos elementos ditos *lagrangeanos* ($\{\psi_j(\vec{x})\}_{j=1,\ldots,n_n}$) é tal que representa uma base de $\mathbb{P}_r(K_i)$ para a qual cada polinômio seu assume valor unitário em um determinado nó do elemento e valor nulo para os nós restantes. Portanto, sendo $\{\vec{\alpha}_j\}_{j=1,\ldots,n_n}$ a família de coordenadas do genérico nó j, tem-se que $\psi_j(\vec{\alpha}_i) = \delta_{ij}$. Note que, como os polinômios lagrangeanos devem interpolar n_n nós, esses terão grau dependente do número de nós n_n e da dimensão do elemento (1D, 2D, etc.). A base lagrangeana é também dita família das *funções de forma*.

[7] Tecnicamente, para poder caracterizar uma projeção, $a(.,.)$ deve induzir um produto interno em $H^1_{\Gamma_D}(\Omega)$ pois nesse caso o problema seria equivalente a encontrar o elemento u_h no espaço finito dimensional que minimiza a distância em relação a $u \in H^1_{\Gamma_D}(\Omega)$.

[8] Isto significa que $\inf_{h\to 0, u_h \in V_h} ||u_h - u||_V \to 0$, i.é. que com o refinamento da malha, o espaço V_h finito dimensional tende a ocupar todo V.

Seja por último $\mathrm{T}_h = \cup_{i=1,\ldots,n_e} K_i$ o conjunto dos elementos, pode-se definir o espaço dos elementos finitos como:

$$V_h = X_h^r = \left\{ v_h \in C^0(\overline{\Omega}) : \ v_h|_{K_i} \in \mathbb{P}_r(K_i), \ \forall K_i \in \mathrm{T}_h \right\} \tag{2.17}$$

Com $h = \max_{K_i \in \mathrm{T}_h} h_{K_i}$ e $h_{K_i} = \sup_{\vec{x},\vec{y} \in K_i} |\vec{x} - \vec{y}|$.

Uma base natural para o espaço 2.17 é a família $\{\phi\}_{i=1}^{n_{gl}}$ onde $\phi_i = \sum_{k=1}^{n_e} \psi_k$ é a soma das funções de forma associadas ao i-ésimo grau de liberdade. Tais funções serão em número iguais ao número de elementos ao qual tal grau de liberdade pertence (n_e). Desta forma qualquer função de V_h pode ser escrita como $v_h = \sum_{i=1}^{n_{gl}} V_i \phi_i'$, tendo-se numerado oportunamente, de 1 a n_{gl}, os graus de liberdade do problema. Dessa maneira, os valores do vetor $\vec{V}$ são exatamente os valores da incógnita no nó correspondente.

Escrevendo portanto u_h como acima e impondo a validade de 2.16 para todas funções de forma do espaço, pode-se obter o seguinte sistema de equações:

$$\sum_{j=1}^{n_{gl}} U_j a(\phi_j, \phi_i) = F(\phi_i) \quad \forall i = 1, \ldots, n_{gl} \tag{2.18}$$

Toda função ϕ_j assume valor nulo fora dos elementos aos quais o nó j pertence e portanto a matriz de rigidez $K_{ij} = a(\phi_j, \phi_i)$ é esparsa. É fácil demonstrar que tal matriz é simétrica se a forma bilinear é simétrica e que é definida positiva se a forma bilinear é coerciva (ver p.ex. (QUARTERONI, 2008)).

Uma última observação em relação a etapa de integração: normalmente é definido um elemento de referência $\hat{K}$ e para todo elemento $K_i \in \mathrm{T}_h$ é definida uma função Φ_i tal que $K_i = \Phi_i(\hat{K})$ (veja a figura 5). Isso é feito para dinamizar a etapa de integração numérica usando a seguinte identidade:

$$\int_{K_i} f(x)\mathrm{d}x = \int_{\hat{K}} f \circ \Phi_i(\xi) |J(\Phi_i(\xi))| \mathrm{d}\xi \tag{2.19}$$

Onde $J(\Phi_i(\xi))$ é o *Jacobiano* da transformação $\Phi_i(\xi)$.

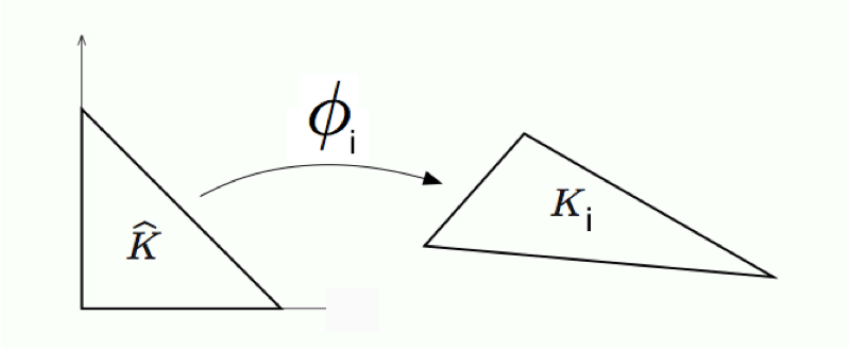


Figura 5: Mapa do elemento de referência ao elemento da malha.

2.3 O método de Newton para solução de sistemas não lineares

O método de Newton é um dos métodos numéricos mais conhecidos para a solução de sistemas não lineares. No caso discreto, o sistema de n equações não lineares em n incógnitas pode ser escrito da seguinte maneira: seja $\vec{F}(\vec{u}) : \mathbb{R}^n \to R^n$ tal que todos os termos presentes estão do lado esquerdo, i.é, de modo que seja válida $\vec{F}(\vec{u}) = \vec{0}$. O método de Newton se escreve:

Dada uma aproximação inicial $\vec{u}_0$, resolve-se iterativamente a 2.20 enquanto $k < MAXIT$ e $||\vec{r}(\vec{u_k})||_{\mathbb{R}^n} > Tol$.

$$\nabla\vec{F}(\vec{u}_k).\vec{\delta u} = -\vec{F}(\vec{u}_k) \tag{2.20a}$$

$$\vec{u}_{k+1} = \vec{u}_k + \vec{\delta u} \tag{2.20b}$$

Onde Tol é uma tolerância[9] pré-estabelecida, $MAXIT$ um limite para o número de iterações e $||\vec{F}(\vec{u_k})||_{\mathbb{R}^n}$ uma norma no espaço $\mathbb{R}^n$.

Note-se que $(\nabla\vec{F}(\vec{u}_k))_{ij} = \frac{\partial F_i}{\partial u_j}(\vec{u}_k)$ é o Jacobiano da função e pode-se afirmar que caso esse seja não singular ($\det(\nabla\vec{F}(\vec{u}_k)) \neq 0$) existe uma vizinhança da solução do problema, $\vec{u}$, para a qual a convergência é quadrática, i.é.:

$$||\vec{u} - \vec{u}_k||_{\mathbb{R}^n} \leq C||\vec{u} - \vec{u}_{k-1}||^2_{\mathbb{R}^n} \tag{2.21}$$

Com C uma constante que dependerá de uma série de parâmetros do problema.

O método de Newton para o caso discreto pode ser estendido para o caso contínuo da seguinte forma:

Sejam $u \in V$ e $T \in Q$ duas funções que satisfazem um sistema de equações diferenciais não lineares $\mathcal{L}(u, T) = 0$ e denotando por $D\mathcal{L}_{(u_k,T_k)}(\delta u, \delta T)$ a derivada de Gâteaux[10] avaliada no ponto (u_k, T_k) na direção $(\delta u, \delta T)$, o método de Newton se escreve:

Dada uma aproximação inicial $(u_0, T_0) \in V \times Q$, resolve-se iterativamente a 2.22 enquanto $k < MAXIT$ e $\frac{||(u_{k+1},T_{k+1})-(u_k,T_k)||_{V\times Q}}{||(u_k,T_k)||_{V\times Q}} > Tol$.

$$D\mathcal{L}_{(u_k,T_k)}(\delta u, \delta T) = -\mathcal{L}(u_k, T_k) \tag{2.22a}$$

$$\begin{bmatrix} u_{k+1} \\ T_{k+1} \end{bmatrix} = \begin{bmatrix} u_k \\ T_k \end{bmatrix} + \begin{bmatrix} \delta u \\ \delta T \end{bmatrix} \tag{2.22b}$$

É importante notar que a 2.22a é um sistema de equações a derivadas parciais lineares e para a sua resolução são usados métodos de aproximação como o usado na seção 2.2.2.

[9] Muitas vezes é conveniente estabelecer uma tolerância para o resíduo relativo pois é um modo de normalizar a convergência, por exemplo: $||r_k|| = \frac{||(\vec{u}_k)-(\vec{u}_{k-1})||_{\mathbb{R}^n}}{||(\vec{u}_{k-1})||_{\mathbb{R}^n}}$.

[10] Veja a seção B.3 na página 114 no anexo B.

2.4 O problema: análise global

Nesta seção, serão introduzidos alguns conceitos e discussões para o caso do problema da análise global. Nesta etapa algumas das não-linearidades do problema serão evidenciadas. A formulação geral discutida anteriormente não será válida pois foi desenvolvida nas hipóteses de carregamentos e relação de congruência lineares.

Numa segunda etapa, denominada "local", as hipóteses de linearidade serão retomadas e será possível fruir dos resultados até aqui desenvolvidos.

Como dito anteriormente, a esbeltez da tubulação considerada determina a pequena relevância da sua resistência à flexão e por isso essas estruturas podem ser analisadas globalmente como cabos flexíveis. Matematicamente isso significa que as equações que regem o comportamento estrutural são as de catenária (cabos flexíveis).

As equações da estática da catenária derivam de uma formulação de equilíbrio diferencial geometricamente muito simples (veja, p.ex., (IRVINE, 1981)). Nesse caso o único esforço solicitante considerado é a tração na seção transversal já que a estrutura é considerada flexível. Na figura 6 pode-se observar a situação que se procura descrever.

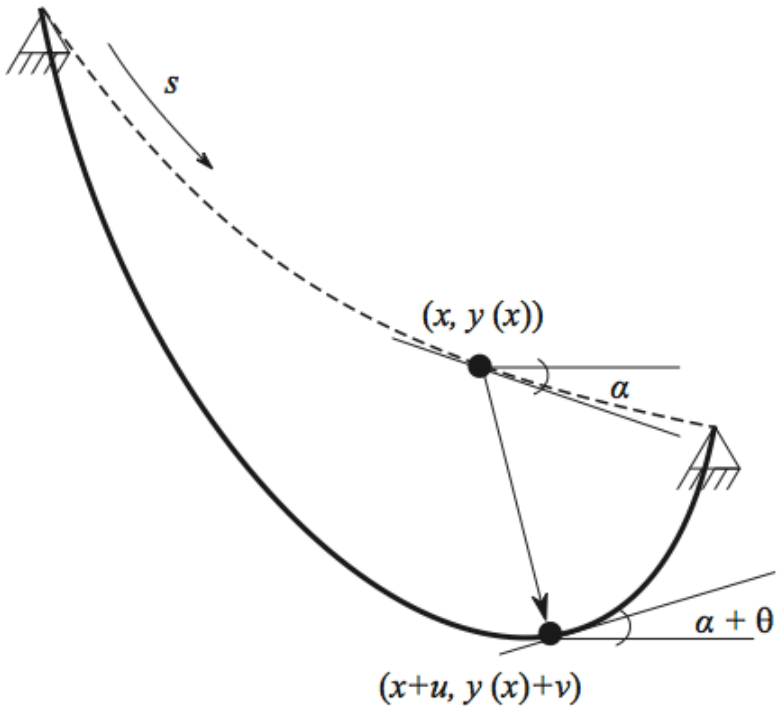


Figura 6: Cabo flexível deformado.

Fonte: (SANTOS; ALMEIDA, 2011)

2.4.1 Equações de equilíbrio

Sendo s a coordenada lagrangeana, H e V as forças horizontal e vertical internas do cabo, componentes da normal N que é o único esforço solicitante, as equações de equilíbrio de força e momento fornecem:

$$\frac{\mathrm{d}H(s)}{\mathrm{d}s} = -f_x(s) \tag{2.23a}$$

$$\frac{\mathrm{d}V(s)}{\mathrm{d}s} = -f_y(s) \tag{2.23b}$$

$$H(y+v) - V(x+u) + \int_0^s f_x(y(s)+v(s))\mathrm{d}s - \int_0^s f_y(x(s)+u(s))\mathrm{d}s = 0 \tag{2.23c}$$

Sendo $F_x(s) = \int_0^s f_x(t)\mathrm{d}t$ e $F_y(s) = \int_0^s f_y(t)\mathrm{d}t$ os carregamentos globais impostos na catenária nos primeiros s metros de comprimento e f_x e f_y a distribuição de carregamento nas direções x e y que derivam do peso submerso e da ação da corrente:

$$f_x = \frac{1}{2}c_d D \rho_a V_c^2 f(y)(\sin(\alpha+\theta))^2 |f(y)\sin(\alpha+\theta)| \tag{2.24a}$$

$$f_y = q - \frac{1}{2}c_d D \rho_a V_c^2 f(y)\sin(\alpha+\theta)\cos(\alpha+\theta)|f(y)\sin(\alpha+\theta)| \tag{2.24b}$$

Outra maneira de escrever as 2.24 é:

$$f_x = C_d(f(y))^2(\sin(\alpha+\theta))^3 \mathrm{sgn}(f(y)\sin(\alpha+\theta)) \tag{2.25a}$$

$$f_y = q - C_d(f(y))^2(\sin(\alpha+\theta))^2\cos(\alpha+\theta)\mathrm{sgn}(f(y)\sin(\alpha+\theta)) \tag{2.25b}$$

Onde "sgn" é a função sinal[11], c_d é o coeficiente de arrasto do cilindro, D seu diâmetro, ρ_a a densidade da água e q o peso submerso da tubulação. Uma outra observação é que com considerações de caráter físico, se a distribuição da velocidade é monotonicamente crescente e positiva pode-se afirmar que na configuração estática é lícito desprezar a função módulo pois os ângulos finais serão compreendidos entre $0\,^\circ$ e $90\,^\circ$ e $f(y)$ será sempre positiva.

A equação 2.23c pode ser simplificada em sua versão diferencial substituindo as relações 2.23a e 2.23b:

$$H(y_0' + v') - V(x_0' + u') = 0 \tag{2.26}$$

[11]

$$\mathrm{sgn}(f(x)) = \begin{cases} -1 & : f(x) < 0 \\ 0 & : f(x) = 0 \\ 1 & : f(x) > 0 \end{cases}$$

A relação entre a normal e suas componentes é dada por:

$$H = N\cos(\alpha+\theta) \tag{2.27a}$$

$$V = N\sin(\alpha+\theta) \tag{2.27b}$$

Sendo um cabo, a relação $N(s) > 0$ vale em todo seu comprimento.

2.4.2 Relações Cinemáticas

Os ângulos da figura 6 podem ser descritos como funções das coordenadas de referência e deslocamentos conforme se segue:

$$\cos(\alpha+\theta) = \frac{x_0' + u'}{\sqrt{(x_0'+u')^2 + (y_0'+v')^2}} \tag{2.28a}$$

$$\sin(\alpha+\theta) = \frac{y_0' + v'}{\sqrt{(x_0'+u')^2 + (y_0'+v')^2}} \tag{2.28b}$$

Introduzindo os espaços funcionais:

$$\mathcal{U} = \{(u,v) \in H^1(\Omega)\times H^1(\Omega) | u(s) = \overline{u}, v(s) = \overline{v}\ \forall s \in \Gamma_D\} \tag{2.29a}$$

$$\mathcal{V} = \{(\delta u,\delta v) \in H^1(\Omega)\times H^1(\Omega) | \delta u(s) = \overline{u}, \delta v(s) = \overline{v}\ \forall s \in \Gamma_D\} \tag{2.29b}$$

Esses espaços são ditos respectivamente *cinematicamente admissível* e *cinematicamente homogêneo admissível*.

Pelo princípio do trabalho virtual, se $(u,v) \in \mathcal{U}$ é o campo de deslocamento de equilíbrio, então para qualquer campo virtual $(\delta u, \delta v) \in \mathcal{V}$ o trabalho virtual interno é igual ao trabalho virtual externo, i.é.:

$$\int_0^L N\delta\epsilon(u,v)\mathrm{d}s = \int_0^L \left[f_x(s)\delta u + f_y(s)\delta v\right]\mathrm{d}s \quad \forall(\delta u,\delta v)\in\mathcal{V} \tag{2.30}$$

Na equação anterior, substituindo as relações 2.23a e 2.23b, integrando por partes, usando a relação 2.27 e por fim a relação 2.28:

$$\begin{aligned}
\int_0^L \left[f_x(s)\delta u + f_y(s)\delta v\right]\mathrm{d}s &= -\int_0^L \left[\frac{\mathrm{d}H}{\mathrm{d}s}\delta u + \frac{\mathrm{d}V}{\mathrm{d}s}\delta v\right]\mathrm{d}s = \\
&= \int_0^L H\left[\delta u' + V\delta v'\right]\mathrm{d}s = \\
&= \int_0^L N\left[\cos(\alpha+\theta)\delta u' + N\sin(\alpha+\theta)\delta v'\right]\mathrm{d}s = \\
&= \int_0^L N\frac{(x_0'+u')\delta u' + (y_0'+v')\delta v'}{\sqrt{(x_0'+u')^2+(y_0'+v')^2}}\mathrm{d}s
\end{aligned}$$

Em outras palavras:

$$\delta\epsilon(u,v) = \frac{(x_0' + u')\delta u' + (y_0' + v')\delta v'}{\sqrt{(x_0' + u')^2 + (y_0' + v')^2}} \tag{2.31}$$

Integrando a relação anterior, chega-se na equação diferencial cinemática, que é exata e não linear:

$$\epsilon(u,v) = \sqrt{(x_0' + u')^2 + (y_0' + v')^2} - 1 \tag{2.32}$$

Note que ϵ deve ser sempre positivo e que as relações cinemáticas obtidas, assim como as de equilíbrio, são válidas para deslocamentos arbitrariamente grandes.

2.4.3 Relação Constitutiva

Como se sabe, na região elástica, para o caso de deformação uniforme na seção transversal (sem escorregamento entre as camadas), a *densidade de energia potencial de deformação* é dada pela seguinte função convexa:

$$W(\epsilon) = \frac{1}{2}EA\epsilon^2 \tag{2.33}$$

Onde EA é a rigidez axial equivalente da seção transversal, dada pela soma de rigidez das diversas camadas (conforme será visto, a rigidez equivalente do isolante é praticamente desprezível). Daqui em diante, qualquer menção ao termo EA refere-se à essa rigidez equivalente. Dessa maneira pode-se definir a *densidade de energia complementar de deformação* que, para o caso de densidade de energia potencial de deformação convexa, pode ser obtida através da seguinte transformação de Legendre:

$$W_c(N) = N\epsilon - W(\epsilon) \tag{2.34}$$

Assim:

$$W_c(N) = \frac{1}{2}\frac{N^2}{EA} \tag{2.35}$$

Ou em outras palavras:

$$N = EA\epsilon \tag{2.36}$$

2.4.4 Princípio da mínima energia potencial total e simplificações possíveis

A energia potencial total associada ao estado de deformação correspondente ao campo de deslocamentos (u, v) é o funcional $\Pi_T : \mathcal{U} \to \mathbb{R}$ dado por:

$$\Pi_T(u,v) = U(\epsilon(u,v)) - F(u,v) \tag{2.37}$$

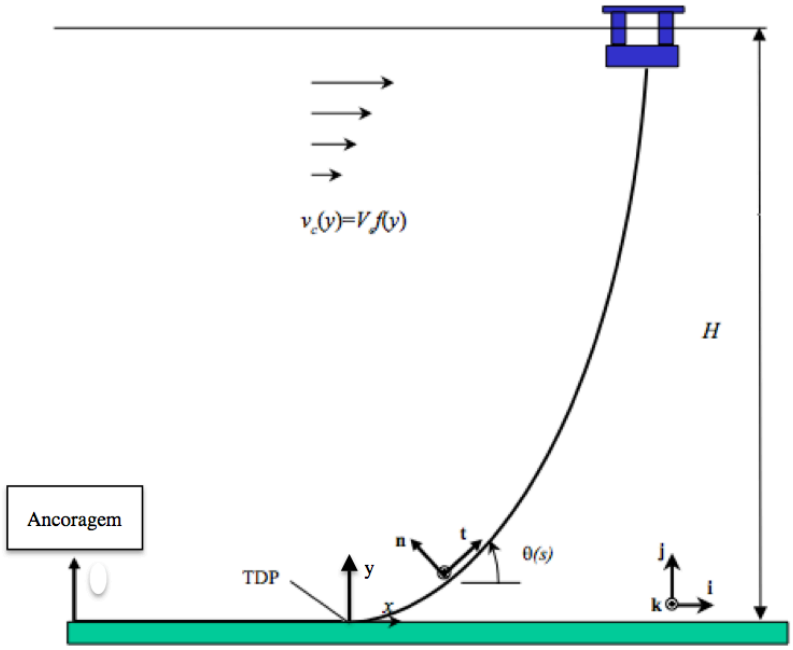


Figura 7: Cabo flexível suspenso sob ação de correnteza estática.

Fonte: (PESCE; MARTINS; CHAKRABARTI, 2005)

Onde U é a energia interna de deformação e F o trabalho exercido pelos esforços externos. Ou seja:

$$U(\epsilon(u,v)) = \int_0^L W(\epsilon)\mathrm{ds} = \frac{EA}{2}\int_0^L (\epsilon(u,v))^2\mathrm{ds} \tag{2.38}$$

$$F(u,v) = \int_0^L \left[f_x u + f_y v\right]\mathrm{ds} \tag{2.39}$$

A expressão anterior é altamente não linear: substituindo a equação 2.25, com a notação $C_d(y) = \frac{1}{2}c_d D\rho_a V_c^2 f(y)$ e usando a relação 2.28 pode-se obter a seguinte equação:

$$\begin{aligned} F(u,v) = \int_0^L C_d(y)\Bigg\{ \left(\frac{(y_0'+v')^2}{(x_0'+u')^2+(y_0'+v')^2}\right)\left| f(y)\frac{y_0'+v'}{\sqrt{(x_0'+u')^2+(y_0'+v')^2}}\right| u + \\ + \left(\frac{(x_0'+u')(y_0'+v')}{(x_0'+u')^2+(y_0'+v')^2}\right)\left| f(y)\frac{y_0'+v'}{\sqrt{(x_0'+u')^2+(y_0'+v')^2}}\right| v \Bigg\} + qv\mathrm{ds} \end{aligned} \tag{2.40}$$

Outra fonte de não linearidade é a própria relação 2.32.

Pelo princípio da mínima energia potencial total, o(s) equilíbrio(s) do sistema é(são) dado(s) pelo(s) campo(s) de deslocamento $(u,v) \in \mathcal{U}$ que configura(m) estacionariedade do funcional Π_T. Em outras palavras a derivada de Gâteaux[12] de primeira ordem definida

[12] A derivada de Gâteaux é a generalização do conceito de derivada direcional para o caso de funcionais i.é. onde as variáveis de diferenciação são funções. Ver a seção B.3 do anexo B na página 114.

em $(u, v) \in \mathcal{U}$ em relação a qualquer "direção" $(\delta u, \delta v) \in \mathcal{V}$ se anula:

$$\delta\Pi_T = 0 \quad \forall(\delta u, \delta v) \in \mathcal{V} \tag{2.41}$$

A real expressão da variação do funcional energia potencial total é:

$$\delta\Pi_T = \int_0^L \left[\frac{\mathrm{d}W}{\mathrm{d}\epsilon}\delta\epsilon - f_x\delta u - u\delta f_x - f_y\delta v - v\delta f_y\right]\mathrm{d}s \tag{2.42}$$

Com:

$$\delta f_x = \left.\frac{\mathrm{d}f_x(u + t\delta u, v + t\delta v)}{\mathrm{d}t}\right|_{t=0} \tag{2.43a}$$

$$\delta f_y = \left.\frac{\mathrm{d}f_y(u + t\delta u, v + t\delta v)}{\mathrm{d}t}\right|_{t=0} \tag{2.43b}$$

A expressão acima assume uma forma suficientemente complicada:

$$\begin{aligned}\delta f_x = C_d(f(y))^2\mathrm{sgn}(f(y)(y_0' + v'))\Bigg\{&\frac{-3(x_0' + u')(y_0' + v')^3}{((x_0' + u')^2 + (y_0' + v')^2)^{\frac{5}{2}}}\delta u' + \\ &+ \frac{3(x_0' + u')^2(y_0' + v')^2}{((x_0' + u')^2 + (y_0' + v')^2)^{\frac{5}{2}}}\delta v'\Bigg\}\end{aligned} \tag{2.44a}$$

$$\begin{aligned}\delta f_y = C_d(f(y))^2\mathrm{sgn}(f(y)(y_0' + v'))\Bigg\{&\frac{2(x_0' + u')^2(y_0' + v')^2 - (y_0' + v')^4}{((x_0' + u')^2 + (y_0' + v')^2)^{\frac{5}{2}}}\delta u' + \\ &+ \frac{(x_0' + u')(y_0' + v')^3 - 2(x_0' + u')^3(y_0' + v')}{((x_0' + u')^2 + (y_0' + v')^2)^{\frac{5}{2}}}\delta v'\Bigg\}\end{aligned} \tag{2.44b}$$

Tendo posto $C_d = \frac{1}{2}c_d D\rho_a V_c^2$.

Assim pode-se escrever a formulação variacional:

Encontar $(u, v) \in \mathcal{U}$ tal que seja válida a relação:

$$\begin{aligned}&\int_0^L EA(\sqrt{(x_0' + u')^2 + (y_0' + v')^2} - 1)\frac{(x_0' + u')\delta u' + (y_0' + v')\delta v'}{\sqrt{(x_0' + u')^2 + (y_0' + v')^2}}\mathrm{d}s = \\ &= \int_0^L \frac{C_d(f(y))^2\mathrm{sgn}(f(y)(y_0' + v'))}{((x_0' + u')^2 + (y_0' + v')^2)^{\frac{5}{2}}}\Big\{(x_0' + u')^2(y_0' + v')^3\delta u + \\ &+ (y_0' + v')^5\delta u - 3(x_0' + u')(y_0' + v')^3 u\delta u' + 3(x_0' + u')^2(y_0' + v')^2 u\delta v' + \\ &- (x_0' + u')^3(y_0' + v')^2\delta v - (x_0' + u')(y_0' + v')^4\delta v + 2(x_0' + u')^2(y_0' + v')^2 v\delta u' + \\ &- (y_0' + v')^4 v\delta u' + (x_0' + u')(y_0' + v')^3 v\delta v' - 2(x_0' + u')^3(y_0' + v')v\delta v'\Big\} + \\ &+ q\delta v\mathrm{d}s \quad \forall(\delta u, \delta v) \in \mathcal{V}\end{aligned} \tag{2.45}$$

Uma simplificação possível é a de considerar a força que deriva da interação fluido-estrutura como proporcional à posição inicial da tubulação, simplificação essa que é válida

para o caso em que a configuração inicial (não deformada) é muito próxima da configuração final (deformada) e pode ser eficiente para perfis de corrente $f(y)V_c$ não muito complexos. Deve-se observar que essa é uma hipótese diferente da de pequenos deslocamentos pois os deslocamentos podem ser relevantes sem alterar significativamente o perfil geométrico. Nesse caso valem as equações:

$$f_x = \frac{1}{2} c_d D \rho_a V_c^2 f(y) (\sin(\alpha))^2 |f(y)\sin(\alpha)| \tag{2.46a}$$

$$f_y = q - \frac{1}{2} c_d D \rho_a V_c^2 f(y) \sin(\alpha)\cos(\alpha) |f(y)\sin(\alpha)| \tag{2.46b}$$

Com:

$$\cos(\alpha) = \frac{x_0'}{\sqrt{(x_0')^2 + (y_0')^2}} \tag{2.47a}$$

$$\sin(\alpha) = \frac{y_0'}{\sqrt{(x_0')^2 + (y_0')^2}} \tag{2.47b}$$

A minimização da energia potencial total nesse caso leva ao próprio princípio do trabalho virtual ($\delta\Pi_T = \int_0^L (N\delta\epsilon - f_x\delta u - f_y\delta v)\mathrm{d}s = 0 \ \forall(\delta u, \delta v) \in \mathcal{V}$). Portanto, conclui-se que o sistema está em equilíbrio se, e somente se, a sua energia potencial total assume um valor estacionário para qualquer deslocamento cinematicamente admissível, i.é. $\forall(\delta u, \delta v) \in \mathcal{V}$.

Por último, cabe notar que, para esse último caso, a segunda derivada de Gâteaux assume a seguinte forma:

$$\delta^2\Pi_T = \int_0^L \left(\frac{\mathrm{d}^2 W}{\mathrm{d}\epsilon^2} \delta^2\epsilon \right) \mathrm{d}s \tag{2.48}$$

Com:

$$\delta^2\epsilon = \left. \frac{\mathrm{d}\delta\epsilon(u + t\delta u, v + t\delta v)}{\mathrm{d}t} \right|_{t=0} = \frac{((y_0' + v')\delta u' - (x_0' + u')\delta v')^2}{((x_0' + u')^2 + (y_0' + v')^2)^{\frac{3}{2}}} \geq 0 \tag{2.49}$$

Sendo a densidade de energia potencial de deformação $W(\epsilon)$ uma função convexa e sendo $\delta^2\epsilon \geq 0$ pode-se concluir que o funcional Π_T é convexo $\forall(u, v) \in \mathcal{U}$ e portanto pode-se demonstrar a existência e unicidade do ponto de mínimo[13].

2.4.5 Formulação Dual

Para o caso em que pode-se admitir que as forças externas são independentes dos deslocamentos u e v, pode-se formular o problema de modo que os esforços sejam as

[13] O teorema do método direto do Cálculo das variações garante a existência. Veja (GELFAND; FOMIN, 2000) para detalhes.

incógnitas. Para isso tomando as equações 2.37 e 2.34 e adicionando o termo $\int_0^L u(H' + f_x) + v(V' + f_y)\mathrm{d}s$ (que é nulo) chega-se ao seguinte funcional Lagrangeano:

$$\begin{aligned} L(H,V,u,v) = &\int_0^L \left[N(H,V)\epsilon(u,v) - W_c(N(H,V))\right]\mathrm{d}s - \\ &- \int_0^L \left[N(H,V)f_x u + f_y v\right]\mathrm{d}s + \int_0^L \left[u(H' + f_x) + v(V' + f_y)\right]\mathrm{d}s \end{aligned}$$

Integrando por partes:

$$L(H,V,u,v) = \int_0^L \left[N(H,V)\epsilon(u,v) - W_c(N(H,V))\right]\mathrm{d}s - \int_0^L \left[u'H + v'V\right]\mathrm{d}s$$

Usando as relações 2.27, 2.28 e 2.32 note que:

$$\begin{aligned} \int_0^L [u'H + v'V - N(H,V)\epsilon(u,v)\mathrm{d}s &= \int_0^L N\{u'\cos\alpha + \theta + v'\sin\alpha + \theta - \epsilon(u,v)\}\mathrm{d}s \\ &= \int_0^L N\left\{1 + \frac{u'(x_0' + u') + v'(y_0' + v') - (x_0' + u')^2 - (y_0' + v')^2}{\sqrt{(x_0' + u')^2 + (y_0' + v')^2}}\right\}\mathrm{d}s \\ &= \int_0^L N\left\{1 - \frac{x_0'(x_0' + u') + y_0'(y_0' + v')}{\sqrt{(x_0' + u')^2 + (y_0' + v')^2}}\right\}\mathrm{d}s \\ &= \int_0^L \left[\sqrt{H^2 + v^2} - x_0'H - y_0'V\right]\mathrm{d}s \end{aligned}$$

Assim o Lagrangeano assume a forma da energia complementar total $\Pi_c : \mathcal{U}_s \to \mathbb{R}$:

$$\Pi_c(H,V) = -U_c(N(H,V)) - Gap(H,V) \tag{2.50}$$

Sendo os funcionais da energia complementar interna e de *gap* definidos conforme segue:

$$U_c(N(H,V)) := \int_0^L W_c(N)\mathrm{d}s \tag{2.51a}$$

$$Gap(H,V) := \int_0^L \sqrt{H^2 + V^2} - x_0'H - y_0'V\mathrm{d}s \tag{2.51b}$$

O espaço funcional $\mathcal{U}_s$ é o espaço das funções estaticamente admissíveis e $\mathcal{V}_s$ o espaço homogêneo associado, definidos conforme se segue:

$$\mathcal{U}_s = \{(H,V) \in H^1(0,L) \times H^1(0,L) :\ H'(s) + f_x(s) = 0,\ V'(s) + f_y(s) = 0\ \forall s \in (0,L)\} \tag{2.52a}$$

$$\mathcal{V}_s = \{(\delta H, \delta V) \in H^1(0,L) \times H^1(0,L) :\ \delta H'(s) = 0,\ \delta V'(s) = 0\ \forall s \in (0,L)\} \tag{2.52b}$$

O ponto de equilíbrio, como se sabe, corresponde ao ponto que torna a energia complementar total estacionária, i.é. (H,V) é o campo de esforços de equilíbrio se, e somente se, nesse ponto se anula a primeira variação do funcional:

$$\delta\Pi_c((U,V);(\delta U,\delta V)) = 0 \quad \forall(\delta U,\delta V) \in \mathcal{V}_s \tag{2.53}$$

Especificando os termos:

$$\int_0^L \left[-\frac{\mathrm{d}W_c}{\mathrm{d}N}\delta N(H,V) + x_0'\delta H + y_0'\delta V - \delta N(H,V) \right] \mathrm{d}s = 0 \quad \forall(\delta U, \delta V) \in \mathcal{V}_s \tag{2.54}$$

Com:

$$\delta N(H,V) = \left.\frac{\mathrm{d}N(H + t\delta H, V + t\delta V)}{\mathrm{d}t}\right|_{t=0} = \frac{H\delta H + V\delta V}{\sqrt{H^2 + V^2}} \tag{2.55}$$

Analogamente ao que foi feito na seção anterior, analisando a segunda variação da energia potencial complementar total conclui-se que essa é côncava e, portanto, possui somente um ponto de máximo:

$$\delta^2\Pi_c = -\delta^2 U_c - \delta^2 Gap = -\int_0^L \frac{1}{EA}\frac{(H\delta V - V\delta H)^2}{(H^2+V^2)^{\frac{3}{2}}}\mathrm{d}s - \int_0^L \frac{(H\delta V - V\delta H)^2}{(H^2+V^2)^{\frac{3}{2}}}\mathrm{d}s \leq 0 \tag{2.56}$$

A unicidade concluída tanto para o caso da formulação variacional quanto para o caso da formulação dual é interessante pois como a formulação dual deriva basicamente da própria formulação variacional pode-se concluir que:

$$\inf_{(u,v)\in\mathcal{U}} \Pi_T = \sup_{(H,V)\in\mathcal{U}_s} \Pi_c \tag{2.57}$$

Essa informação ajuda a avaliar a qualidade e eficiência de ambas as abordagens comparando, após a solução, a energia potencial do sistema, energia que será superestimada pela abordagem primária e subestimada pela abordagem dual.

3 Materiais e métodos

Neste capítulo a primeira parte é dedicada à aplicação dos métodos e das formulações desenvolvidos no capítulo 2. Em seguida os materiais e instrumentos utilizados no estudo são brevemente descritos enfatizando as vantagens e potencialidades da abordagem adotada.

3.1 Análise e formulação do problema

No presente texto a análise estrutural é dividida em duas etapas. Na primeira, é estudado o comportamento global do *riser* primando pela avaliação dos esforços solicitantes resultantes e pela forma assumida pela estrutura. Na segunda, é conduzido um estudo local da relevância dos esforços como pressões interna e externa e efeitos anelásticos que derivam da difusão de temperatura.

Cabe observar que, na primeira parte, a teoria se refere ao problema global mencionado no capítulo 2 onde discussões e intuições foram desenvolvidas para o problema da catenária. Na segunda parte, são retomados os conceitos do problema geral discutido no início do capítulo 2, e no anexo A pois são válidas as hipóteses introduzidas.

Uma consideração importante é que os efeitos axissimétricos, dado que são autoequilibrados, não contribuem de maneira decisiva à forma assumida pela tubulação e por isso são desprezados na análise global. Reciprocamente, tendo em mente que as curvaturas assumidas pelo *riser* são de pequena magnitude, pode-se assumir que localmente (em segmentos de pequenos comprimentos) a geometria continua sendo axissimétrica.

3.1.1 Formulação completa do problema global

Nas seções anteriores foram apresentados alguns conceitos relativos ao problema da catenária elástica. Nesta seção será considerada uma formulação mais eficiente e compacta que inclui também as não linearidades do problema. Seja, como visto anteriormente, a equação de equilíbrio da catenária elástica em forma vetorial:

$$\frac{\mathrm{d}}{\mathrm{d}s}\left(\frac{N}{1+\epsilon}\frac{\mathrm{d}\vec{r}}{\mathrm{d}s}\right)+\vec{f}(1+\epsilon)=0 \tag{3.1}$$

Sendo $\vec{r}=(x_0+u, y_0+v)=(x,y)$ e $\vec{f}=(f_x, f_y)$.

Duas formulações distintas podem ser consideradas, que terão vantagens e desvantagens na etapa numérica.

3.1.1.1 Formulação fraca clássica

Antes de proceder formalmente à formulação fraca do problema, substitui-se as relações cinemática 2.32 e constitutiva 2.36 na equação 3.1 obtendo a equação 3.2.

$$\frac{\mathrm{d}}{\mathrm{d}s}\left\{\frac{EA\left(\sqrt{x'^2+y'^2}-1\right)}{\sqrt{x'^2+y'^2}}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}x'\\ y'\end{bmatrix}\right\}+\sqrt{x'^2+y'^2}\begin{bmatrix}f_x(\vec{r},s)\\ f_y(\vec{r},s)\end{bmatrix}=\begin{bmatrix}0\\ 0\end{bmatrix} \tag{3.2}$$

Onde $x'=\frac{\mathrm{d}x}{\mathrm{d}s}$ e $y'=\frac{\mathrm{d}y}{\mathrm{d}s}$ e conforme visto:

$$f_x(\vec{r},s)=C_d(f(y))^2\frac{y'^3}{(x'^2+y'^2)^{\frac{3}{2}}}\mathrm{sgn}(f(y)y') \tag{3.3a}$$

$$f_y(\vec{r},s)=q-C_d(f(y))^2\frac{x'y'^2}{(x'^2+y'^2)^{\frac{3}{2}}}\mathrm{sgn}(f(y)y') \tag{3.3b}$$

A equação 3.2 é não linear e para sua resolução recorre-se ao método de Newton, que é especificado na seção 2.3 (página 36). Denotando o sistema de equações 3.2 com $\mathcal{L}(x,y)=0$ e omitindo os cálculos, a derivada de Gâteaux $D\mathcal{L}_{(\Delta x,\Delta y)}(\overline{x},\overline{y})$ de $\mathcal{L}$ avaliada no ponto $(\overline{x},\overline{y})$ na direção $(\Delta x,\Delta y)$ é:

$$D\mathcal{L}_{(\Delta x,\Delta y)}(\overline{x},\overline{y})=\begin{bmatrix}D\mathcal{L}^1_{(\Delta x,\Delta y)}(\overline{x},\overline{y})\\ D\mathcal{L}^2_{(\Delta x,\Delta y)}(\overline{x},\overline{y})\end{bmatrix} \tag{3.4}$$

Com:

$$\begin{aligned}D\mathcal{L}^1_{(\Delta x,\Delta y)}(\overline{x},\overline{y})=&\frac{\mathrm{d}}{\mathrm{d}s}\left\{\frac{EA\left((\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}-\overline{y}'^2\right)}{(\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}}\Delta x'+\frac{EA\overline{x}'\overline{y}'}{(\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}}\Delta y'\right\}+\\&+C_d\mathrm{sgn}(f(\overline{y})\overline{y}')\left\{-\frac{2(f(\overline{y}))^2\overline{x}'\overline{y}'^3}{(\overline{x}'^2+\overline{y}'^2)^2}\Delta x'+\frac{2f(\overline{y})f'(\overline{y})\overline{y}'^3}{(\overline{x}'^2+\overline{y}'^2)}\Delta y+\right.\\&\left.+\frac{(f(\overline{y}))^2\left(3\overline{x}'^2\overline{y}'^2+\overline{y}'^4\right)}{(\overline{x}'^2+\overline{y}'^2)^2}\Delta y'\right\}\end{aligned} \tag{3.5a}$$

$$\begin{aligned}D\mathcal{L}^2_{(\Delta x,\Delta y)}(\overline{x},\overline{y})=&\frac{\mathrm{d}}{\mathrm{d}s}\left\{\frac{EA\overline{x}'\overline{y}'}{(\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}}\Delta x'+\frac{EA\left((\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}-\overline{x}'^2\right)}{(\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}}\Delta y'\right\}+\\&-C_d\mathrm{sgn}(f(\overline{y})\overline{y}')\left\{\frac{(f(\overline{y}))^2\left(-\overline{x}'^2\overline{y}'^2+\overline{y}'^4\right)}{(\overline{x}'^2+\overline{y}'^2)^2}\Delta x'+\frac{2f(\overline{y})f'(\overline{y})\overline{x}'\overline{y}'^2}{(\overline{x}'^2+\overline{y}'^2)}\Delta y+\right.\\&\left.+\frac{2(f(\overline{y}))^2\overline{x}'^3\overline{y}'}{(\overline{x}'^2+\overline{y}'^2)^2}\Delta y'\right\}+\frac{q\overline{x}'}{(\overline{x}'^2+\overline{y}'^2)^{\frac{1}{2}}}\Delta x'+\frac{q\overline{y}'}{(\overline{x}'^2+\overline{y}'^2)^{\frac{1}{2}}}\Delta y'\end{aligned} \tag{3.5b}$$

O método de Newton consiste na resolução iterativa (as incógnitas são as funções $(\Delta x,\Delta y)$ a cada iteração) das equações lineares a derivadas parciais dada uma função de aproximação inicial (x_0,y_0):

$$D\mathcal{L}_{(\Delta x,\Delta y)}(x_k,y_k)=-\mathcal{L}(x_k,y_k) \tag{3.6a}$$

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{3.6b}$$

O procedimento iterativo deve continuar até que se satisfaça um critério de convergência ou até que seja atingido um número máximo de iterações pré-estabelecido. Como se sabe, não é garantida a convergência do método para qualquer valor inicial (x_0, y_0). Computacionalmente o procedimento é potencialmente oneroso pois, para cada iteração do método de Newton, a resolução da equação linear parcial associada conduz a um sistema linear não simétrico.

Introduzindo uma notação mais compacta que engloba as diversas parcelas referentes às equações 3.5 e 3.2 e rescrevendo a equação 3.6a tem-se:

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}s}\left\{ \begin{bmatrix} \mathcal{A}_{11}(x_k,y_k) & \mathcal{A}_{12}(x_k,y_k) \\ \mathcal{A}_{21}(x_k,y_k) & \mathcal{A}_{22}(x_k,y_k) \end{bmatrix} \begin{bmatrix} \Delta x' \\ \Delta y' \end{bmatrix} \right\} + \begin{bmatrix} \mathcal{B}_{11}(x_k,y_k) & \mathcal{B}_{12}(x_k,y_k) \\ \mathcal{B}_{21}(x_k,y_k) & \mathcal{B}_{22}(x_k,y_k) \end{bmatrix} \begin{bmatrix} \Delta x' \\ \Delta y' \end{bmatrix} + \\ + \begin{bmatrix} 0 & \mathcal{C}_{12}(x_k,y_k) \\ 0 & \mathcal{C}_{22}(x_k,y_k) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} \mathcal{L}^1(x_k,y_k) \\ \mathcal{L}^2(x_k,y_k) \end{bmatrix} \end{aligned} \tag{3.7}$$

Ou em forma vetorial:

$$\frac{\mathrm{d}}{\mathrm{d}s}\left\{ \mathbf{A}(\vec{x_k}) \frac{\mathrm{d}\vec{\Delta x}}{\mathrm{d}s} \right\} + \mathbf{B}(\vec{x_k}) \frac{\mathrm{d}\vec{\Delta x}}{\mathrm{d}s} + \mathbf{C}(\vec{x_k})\vec{\Delta x} = -\mathcal{L}(\vec{x_k}) \tag{3.8}$$

Onde:

$$\mathcal{A}_{11}(x_k,y_k) = \frac{EA\left((x_k'^2 + y_k'^2)^{\frac{3}{2}} - y_k'^2\right)}{(x_k'^2 + y_k'^2)^{\frac{3}{2}}} \tag{3.9a}$$

$$\mathcal{A}_{12}(x_k,y_k) = \mathcal{A}_{21}(x_k,y_k) = \frac{EAx_k'y_k'}{(x_k'^2 + y_k'^2)^{\frac{3}{2}}} \tag{3.9b}$$

$$\mathcal{A}_{22}(x_k,y_k) = \frac{EA\left((x_k'^2 + y_k'^2)^{\frac{3}{2}} - x_k'^2\right)}{(x_k'^2 + y_k'^2)^{\frac{3}{2}}} \tag{3.9c}$$

$$\mathcal{B}_{11}(x_k,y_k) = -2C_d \mathrm{sgn}(f(y_k)y_k') \frac{(f(y_k))^2 x_k' y_k'^3}{(x_k'^2 + y_k'^2)^2} \tag{3.9d}$$

$$\mathcal{B}_{12}(x_k,y_k) = C_d \mathrm{sgn}(f(y_k)y_k') \frac{(f(y_k))^2\left(3x_k'^2y_k'^2 + y_k'^4\right)}{(x_k'^2 + y_k'^2)^2} \tag{3.9e}$$

$$\mathcal{B}_{21}(x_k,y_k) = \frac{qx_k'}{(x_k'^2 + y_k'^2)^{\frac{1}{2}}} + C_d \mathrm{sgn}(f(y_k)y_k') \frac{(f(y_k))^2\left(x_k'^2y_k'^2 - y_k'^4\right)}{(x_k'^2 + y_k'^2)^2} \tag{3.9f}$$

$$\mathcal{B}_{22}(x_k,y_k) = \frac{qy_k'}{(x_k'^2 + y_k'^2)^{\frac{1}{2}}} - 2C_d \mathrm{sgn}(f(y_k)y_k') \frac{(f(y_k))^2 x_k'^3 y_k'}{(x_k'^2 + y_k'^2)^2} \tag{3.9g}$$

$$\mathcal{C}_{11}(x_k, y_k) = \mathcal{C}_{21}(x_k, y_k) = 0 \tag{3.9h}$$

$$\mathcal{C}_{12}(x_k, y_k) = 2C_d \text{sgn}(f(y_k)y'_k)\frac{f(y_k)f'(y_k)y'^3_k}{(x'^2_k + y'^2_k)} \tag{3.9i}$$

$$\mathcal{C}_{22}(x_k, y_k) = -2C_d \text{sgn}(f(y_k)y'_k)\frac{f(y_k)f'(y_k)x_k y'^2_k}{(x'^2_k + y'^2_k)} \tag{3.9j}$$

A formulação fraca ou variacional se obtém multiplicando a equação linearizada em forma forte 3.8 por uma função (na realidade um campo de funções) test, integrando e finalmente usando a fórmula de Green para o primeiro adendo. Cabe notar que as condições de contorno são de Dirichlet no TDP, onde é colocado o sistema de referência ($x(0) = y(0) = 0$) e mistas no vínculo com a plataforma onde se tem ($y(L) = p$ e $H(L) = 0$) sendo p a profundidade na qual é colocado o TDP. Na realidade, a condição $H(L) = 0$ é pouco realista para o caso de uma plataforma de petróleo pois essa tem uma dinâmica própria e impõe forças ou deslocamentos na extremidade. O estudo acoplado da dinâmica da plataforma e do cabo vai além do escopo do presente trabalho, para o leitor interessado veja (MONTANO; RESTELLI; SACCO, 2007).

Sendo portanto os espaços das funções incógnitas e funções teste respectivamente:

$$\mathcal{U} = \Big\{(\Delta x(s), \Delta y(s)) \in H^1(0, L) \times H^1(0, L) : \Delta x(0) = \Delta y(0) = 0 = \Delta y(L) = 0\Big\} \tag{3.10a}$$

$$\mathcal{V} = \Big\{(\delta x(s), \delta y(s)) \in H^1(0, L) \times H^1(0, L) : \delta x(0) = \delta y(0) = \delta y(L) = 0\Big\} \tag{3.10b}$$

A formulação fraca do problema não linear e do passo de iteração do método de Newton é:

Encontrar $\vec{\Delta x} = (\Delta x, \Delta y) \in \mathcal{U}$ de modo que a seguinte relação seja válida:

$$\int_0^L \left[-\mathbf{A}(\vec{x_k})\frac{\mathrm{d}\vec{\Delta x}}{\mathrm{d}s}\frac{\mathrm{d}\vec{\delta x}}{\mathrm{d}s} + \mathbf{B}(\vec{x_k})\frac{\mathrm{d}\vec{\Delta x}}{\mathrm{d}s}\vec{\delta x} + \mathbf{C}(\vec{x_k})\vec{\Delta x}\vec{\delta x}\right]\mathrm{d}s = -\int_0^L \mathcal{L}(\vec{x_k})\vec{\delta x}\mathrm{d}s \quad \forall \vec{\delta x} \in \mathcal{V} \tag{3.11}$$

Cabe observar que o termo a direita também é integrado por partes e por isso convém distinguir quatro parcelas de $\mathcal{L}(\vec{x_k})$ que são:

$$\mathcal{L}^1_A(x_k, y_k, N_k) = \frac{EA\Big(\sqrt{x'^2_k + y'^2_k} - 1\Big)}{\sqrt{x'^2_k + y'^2_k}}x'_k \tag{3.12a}$$

$$\mathcal{L}^1_B(x_k, y_k, N_k) = \sqrt{x'^2_k + y'^2_k}\, f_x(x_k, y_k) \tag{3.12b}$$

$$\mathcal{L}^2_A(x_k, y_k, N_k) = \frac{EA\Big(\sqrt{x'^2_k + y'^2_k} - 1\Big)}{\sqrt{x'^2_k + y'^2_k}}y'_k \tag{3.12c}$$

$$\mathcal{L}_B^2(x_k, y_k, N_k) = \sqrt{x_k'^2 + y_k'^2} f_y(x_k, y_k) \tag{3.12d}$$

Onde os termos designados com o índice A são integrados por parte e os designados com B não, ou seja:

$$-\int_0^L \mathcal{L}(\vec{x_k})\vec{\delta x}\mathrm{d}s = \int_0^L \begin{bmatrix} \mathcal{L}_A^1(x_k, y_k, N_k) \\ \mathcal{L}_A^2(x_k, y_k, N_k) \end{bmatrix} \begin{bmatrix} \delta x' \\ \delta y' \end{bmatrix} - \begin{bmatrix} \mathcal{L}_B^1(x_k, y_k, N_k) \\ \mathcal{L}_B^2(x_k, y_k, N_k) \end{bmatrix} \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \mathrm{d}s \tag{3.13}$$

A equação 3.11 é pronta para a fase de discretização pelo método de Galerkin.

3.1.1.2 A formulação fraca mista

Nesta seção é apresentado um método alternativo para a formulação do problema. Cabe introduzir uma pequena discussão do porquê dessa nova abordagem:

Note que a formulação fraca 3.11 equivale à imposição em maneira fraca da equação de equilíbrio. Desta forma, quando se procede à discretização, o resultado aproximado não satisfaz em todo o domínio a equação de equilíbrio: isso ocorre pois nas fronteiras dos elementos não é satisfeito o princípio de ação e reação dada a descontinuidade das deformações (descontinuidade da derivada dos deslocamentos). O que ocorre é que com o refinamento da malha, essas descontinuidades (de salto) tendem à zero e por isso é garantida a convergência, no entanto se a malha não é suficientemente fina, podem existir erros relevantes na estima da tração.

Uma solução para o problema exposto é uma formulação alternativa onde a tração passa a ser uma incógnita junto ao campo de deslocamentos e a formulação fraca é feita em duas equações, a de equilibrio e a de congruência. Dessa maneira a continuidade da tração e dos deslocamentos é imposta implicitamente nos espaços funcionais considerados. O preço a ser pago é que nesse caso, na solução aproximada, não somente a equação de equilíbrio, mas também a de congruência não serão satisfeitas em todo o domínio. Além disso o fato de adicionar uma variável ao problema aumenta a dimensão do sistema linear a ser resolvido a cada passo do método de Newton e requer uma aproximação inicial também para a tração.

Note que na seção 2.4.5 na página 43, a formulação foi conduzida a um problema puro em esforços, o que corresponde à extensão da idéia mencionada no parágrafo anterior a um caso onde a equação de equilíbrio é imposta implicitamente no espaço funcional e onde a formulação fraca corresponde à imposição da equação de congruência em modo fraco (e não da equação de equilíbrio como no caso da formulação primária). É importante notar que esse procedimento foi possível somente para um caso simplificado do problema real e que nem sempre é possível obter uma formulação pura em esforços.

A idéia exposta é a base do chamado *mixed or hybrid finite element method*. Para o leitor interessado veja (SACCO, 2007).

Nesse contexto a formulação forte do problema será um conjunto de três equações geradas com a substituição da lei de Hooke 2.36 nas equações de equilíbrio 3.1 e de congruência 2.32:

$$\frac{\mathrm{d}}{\mathrm{d}s}\left(\frac{EAN}{EA+N}\frac{\mathrm{d}\vec{r}}{\mathrm{d}s}\right)+\vec{f}\left(1+\frac{N}{EA}\right)=0 \tag{3.14a}$$

$$\frac{1}{2}\left(\frac{\mathrm{d}\vec{r}}{\mathrm{d}s}\right)^2-\frac{1}{2}\left(1+\frac{N}{EA}\right)^2=0 \tag{3.14b}$$

Conforme feito na seção anterior, o sistema não linear de equações a derivadas parciais 3.14 denotado com $\mathcal{L}(x,y,N)$ deve ser linearizado, e para isso se usa a derivada de Gâteaux. Mais uma vez omitimos os cálculos pois são tediosos.

$$\begin{aligned}D\mathcal{L}^1_{(\Delta x,\Delta y,\Delta N)}(\overline{x},\overline{y},\overline{N}) &= \frac{\mathrm{d}}{\mathrm{d}s}\left\{\frac{EA\overline{N}}{EA+\overline{N}}\Delta x'+\left(\frac{EA}{EA+\overline{N}}\right)^2\overline{x}'\Delta N\right\}+\\ &+\left(1+\frac{\overline{N}}{EA}\right)C_d\mathrm{sgn}(f(\overline{y})\overline{y}')\left\{-\frac{3(f(\overline{y}))^2\overline{x}'\overline{y}'^3}{(\overline{x}'^2+\overline{y}'^2)^{\frac{5}{2}}}\Delta x'+\frac{2f(\overline{y})f'(\overline{y})\overline{y}'^3}{(\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}}\Delta y+\right.\\ &\left.+\frac{3(f(\overline{y}))^2\overline{x}'^2\overline{y}'^2}{(\overline{x}'^2+\overline{y}'^2)^{\frac{5}{2}}}\Delta y'\right\}+\frac{C_d\mathrm{sgn}(f(\overline{y})\overline{y}')(f(\overline{y}))^2\overline{y}'^3}{EA(\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}}\Delta N\end{aligned} \tag{3.15a}$$

$$\begin{aligned}D\mathcal{L}^2_{(\Delta x,\Delta y,\Delta N)}(\overline{x},\overline{y},\overline{N}) &= \frac{\mathrm{d}}{\mathrm{d}s}\left\{\frac{EA\overline{N}}{EA+\overline{N}}\Delta y'+\left(\frac{EA}{EA+\overline{N}}\right)^2\overline{y}'\Delta N\right\}+\\ &-\left(1+\frac{\overline{N}}{EA}\right)C_d\mathrm{sgn}(f(\overline{y})\overline{y}')\left\{\frac{(f(\overline{y}))^2\left(-2\overline{x}'^2\overline{y}'^2+\overline{y}'^4\right)}{(\overline{x}'^2+\overline{y}'^2)^{\frac{5}{2}}}\Delta x'+\frac{2f(\overline{y})f'(\overline{y})\overline{x}'\overline{y}'^2}{(\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}}\Delta y+\right.\\ &\left.+\frac{(f(\overline{y}))^2\left(2\overline{x}'^3\overline{y}'-\overline{x}'\overline{y}'^3\right)}{(\overline{x}'^2+\overline{y}'^2)^{\frac{5}{2}}}\Delta y'\right\}+\left(q-C_d\mathrm{sgn}(f(\overline{y})\overline{y}')\frac{(f(\overline{y}))^2\overline{x}'\overline{y}'^2}{(\overline{x}'^2+\overline{y}'^2)^{\frac{3}{2}}}\right)\frac{1}{EA}\Delta N\end{aligned} \tag{3.15b}$$

$$D\mathcal{L}^3_{(\Delta x,\Delta y,\Delta N)}(\overline{x},\overline{y},\overline{N})=\overline{x}'\Delta x'+\overline{y}'\Delta y'-\frac{1}{EA}\left(1+\frac{\overline{N}}{EA}\right)\Delta N \tag{3.15c}$$

De maneira analoga à seção anterior o método de Newton em forma vetorial se escreve:

Sendo $\vec{\Delta u}=(\Delta x,\Delta y,\Delta N)$ a incógnita e $\vec{u}_0=(x_0,y_0,N_0)$ uma aproximação inicial da solução, deve-se resolver iterativamente o problema 3.16 até que seja satisfeito um critério de convergência ou se chegue a um número máximo de iterações.

$$D\mathcal{L}_{\vec{\Delta u}}(\vec{u}_k)=-\mathcal{L}(\vec{u}_k) \tag{3.16a}$$

$$\vec{u}_{k+1}=\vec{u}_k+\vec{\Delta u} \tag{3.16b}$$

Para cada iteração do método de Newton é feita a formulação fraca de 3.16a que se obtém de modo ligeiramente diferente do usado na seção anterior: para as duas primeiras equações o procedimento é o mesmo e através da multiplicação escalar seguida de integração e integração por partes se gera uma única equação integral. A segunda equação integral se gera de modo análogo. Observe a equação 3.17 para entender o procedimento:

$$\int_0^L \begin{bmatrix} D\mathcal{L}^1_{\vec{\Delta u}}(\vec{u}_k) \\ D\mathcal{L}^2_{\vec{\Delta u}}(\vec{u}_k) \end{bmatrix} . \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \mathrm{d}s = \int_0^L \begin{bmatrix} \mathcal{L}^1(\vec{u}_k) \\ \mathcal{L}^2(\vec{u}_k) \end{bmatrix} . \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \mathrm{d}s \quad \forall (\delta x, \delta y) \in \mathcal{V} \tag{3.17a}$$

$$\int_0^L D\mathcal{L}^3_{\vec{\Delta u}}(\vec{u}_k).\delta N \mathrm{d}s = \int_0^L \mathcal{L}^3(\vec{u}_k).\delta N \mathrm{d}s \quad \forall \delta N \in \mathcal{Q} \tag{3.17b}$$

Sendo que $\mathcal{Q} = L^2(0, L)$[1].

A equação 3.17 é pronta para a fase de discretização e implementação do método de Galerkin. Como para a seção anterior convém adotar uma notação mais compacta para a formulação fraca, conforme a seguinte:

$$\begin{aligned} \int_0^L & - \begin{bmatrix} \mathcal{A}^m_{11}(x_k, y_k, N_k) & \mathcal{A}^m_{12}(x_k, y_k, N_k) & \mathcal{A}^m_{13}(x_k, y_k, N_k) \\ \mathcal{A}^m_{21}(x_k, y_k, N_k) & \mathcal{A}^m_{22}(x_k, y_k, N_k) & \mathcal{A}^m_{23}(x_k, y_k, N_k) \end{bmatrix} \begin{bmatrix} \Delta x' \\ \Delta y' \\ \Delta N \end{bmatrix} . \begin{bmatrix} \delta x' \\ \delta y' \end{bmatrix} + \\ & + \begin{bmatrix} \mathcal{B}^m_{11}(x_k, y_k, N_k) & \mathcal{B}^m_{12}(x_k, y_k, N_k) \\ \mathcal{B}^m_{21}(x_k, y_k, N_k) & \mathcal{B}^m_{22}(x_k, y_k, N_k) \end{bmatrix} \begin{bmatrix} \Delta x' \\ \Delta y' \end{bmatrix} . \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} + \\ & + \begin{bmatrix} \mathcal{C}^m_{11}(x_k, y_k, N_k) & \mathcal{C}^m_{12}(x_k, y_k, N_k) & \mathcal{C}^m_{13}(x_k, y_k, N_k) \\ \mathcal{C}^m_{21}(x_k, y_k, N_k) & \mathcal{C}^m_{22}(x_k, y_k, N_k) & \mathcal{C}^m_{23}(x_k, y_k, N_k) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta N \end{bmatrix} . \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \mathrm{d}s = \\ & = \int_0^L \begin{bmatrix} \mathcal{L}^{m1}_A(x_k, y_k, N_k) \\ \mathcal{L}^{m2}_A(x_k, y_k, N_k) \end{bmatrix} \begin{bmatrix} \delta x' \\ \delta y' \end{bmatrix} - \begin{bmatrix} \mathcal{L}^{m1}_B(x_k, y_k, N_k) \\ \mathcal{L}^{m2}_B(x_k, y_k, N_k) \end{bmatrix} \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \mathrm{d}s \end{aligned} \tag{3.18a}$$

$$\int_0^L \left[\; x'_k y'_k - \tfrac{1}{EA}\left(1 + \tfrac{N_k}{EA}\right) \; \right] . \begin{bmatrix} \Delta x' \\ \Delta y' \\ \Delta N \end{bmatrix} \delta N \mathrm{d}s = -\int_0^L \left[\frac{1}{2}\left(x'^2_k + y'^2_k\right) - \frac{1}{2}\left(1 + \frac{N_k}{EA}\right)^2 \right] \delta N \mathrm{d}s \tag{3.18b}$$

Onde:

$$\mathcal{A}^m_{11}(x_k, y_k, N_k) = \mathcal{A}^m_{22}(x_k, y_k, N_k) = \frac{EAN_k}{EA + N_k} \tag{3.19a}$$

$$\mathcal{A}^m_{12}(x_k, y_k, N_k) = \mathcal{A}^m_{21}(x_k, y_k, N_k) = 0 \tag{3.19b}$$

$$\mathcal{A}^m_{13}(x_k, y_k, N_k) = \left(\frac{EA}{EA + N_k}\right)^2 x_k \tag{3.19c}$$

[1] Veja o anexo B para a definição do espaço funcional $L^2(\Omega)$

$$\mathcal{A}^m_{23}(x_k, y_k, N_k) = \left(\frac{EA}{EA + N_k}\right)^2 y_k \tag{3.19d}$$

$$\mathcal{B}^m_{11}(x_k, y_k, N_k) = -3\left(1 + \frac{N_k}{EA}\right) C_d \text{sgn}(f(y_k)y'_k) \frac{(f(y_k))^2 x'_k y'^3_k}{(x'^2_k + y'^2_k)^{\frac{5}{2}}} \tag{3.19e}$$

$$\mathcal{B}^m_{12}(x_k, y_k, N_k) = 3\left(1 + \frac{N_k}{EA}\right) C_d \text{sgn}(f(y_k)y'_k) \frac{(f(y_k))^2 x'^2_k y'^2_k}{(x'^2_k + y'^2_k)^{\frac{5}{2}}} \tag{3.19f}$$

$$\mathcal{B}^m_{21}(x_k, y_k, N_k) = \left(1 + \frac{N_k}{EA}\right) C_d \text{sgn}(f(y_k)y'_k) \frac{(f(y_k))^2 \left(2x'^2_k y'^2_k - y'^2_k\right)}{(x'^2_k + y'^2_k)^{\frac{5}{2}}} \tag{3.19g}$$

$$\mathcal{B}^m_{22}(x_k, y_k, N_k) = -\left(1 + \frac{N_k}{EA}\right) C_d \text{sgn}(f(y_k)y'_k) \frac{(f(y_k))^2 \left(2x'^3_k y'_k - x_k y'^3_k\right)}{(x'^2_k + y'^2_k)^{\frac{5}{2}}} \tag{3.19h}$$

$$\mathcal{C}^m_{11}(x_k, y_k, N_k) = \mathcal{C}^m_{21}(x_k, y_k, N_k) = 0 \tag{3.19i}$$

$$\mathcal{C}^m_{12}(x_k, y_k, N_k) = 2\left(1 + \frac{N_k}{EA}\right) C_d \text{sgn}(f(y_k)y'_k) \frac{f(y_k)f'(y_k)y'^3_k}{(x'^2_k + y'^2_k)^{\frac{3}{2}}} \tag{3.19j}$$

$$\mathcal{C}^m_{13}(x_k, y_k, N_k) = \frac{f_x(x_k, y_k)}{EA} = C_d \text{sgn}(f(y_k)y'_k) \frac{(f(y_k))^2 y'^3_k}{EA(x'^2_k + y'^2_k)^{\frac{3}{2}}} \tag{3.19k}$$

$$\mathcal{C}^m_{22}(x_k, y_k, N_k) = -2\left(1 + \frac{N_k}{EA}\right) C_d \text{sgn}(f(y_k)y'_k) \frac{f(y_k)f'(y_k)x_k y'^2_k}{(x'^2_k + y'^2_k)^{\frac{3}{2}}} \tag{3.19l}$$

$$\mathcal{C}^m_{13}(x_k, y_k, N_k) = \frac{f_y(x_k, y_k)}{EA} = \frac{1}{EA}\left(q - C_d \text{sgn}(f(y_k)y'_k) \frac{(f(y_k))^2 x_k y'^2_k}{(x'^2_k + y'^2_k)^{\frac{3}{2}}}\right) \tag{3.19m}$$

$$\mathcal{L}^{m1}_A(x_k, y_k, N_k) = \frac{EAN_k}{EA + N_k} x'_k \tag{3.19n}$$

$$\mathcal{L}^{m2}_A(x_k, y_k, N_k) = \frac{EAN_k}{EA + N_k} y'_k \tag{3.19o}$$

$$\mathcal{L}^{m1}_B(x_k, y_k, N_k) = \left(1 + \frac{N}{EA}\right) f_x(x_k, y_k) \tag{3.19p}$$

$$\mathcal{L}^{m2}_B(x_k, y_k, N_k) = \left(1 + \frac{N}{EA}\right) f_y(x_k, y_k) \tag{3.19q}$$

3.1.2 Discretização e aproximação com o método de Galerkin

Como observado anteriormente, a resolução completa do problema é dada pela resolução iterativa das equações 3.11 e 3.18 respectivamente para os casos de formulação clássica e formulação mista. Nesta seção é descrita uma iteração do método, isto é a resolução das equações 3.11 e 3.18 pelo método de Galerkin.

3.1.2.1 Método de Galerkin - formulação clássica

Seja a partição T_h do domínio $[0, L]$ sendo h o parâmetro de dimensão dos elementos (quanto menor h menor o comprimento dos segmentos) e seja o conjunto de elementos $\left\{K_e\right\}_{e=1}^{n_e}$ tal que $\cup_{e=1}^{n_e} K_e = \mathrm{T}_h$. Introduzindo o espaço dos elementos finitos $\mathcal{U}_h \subset \mathcal{U}$ onde a deflexão interna de cada elemento é a interpolação polinomial dos deslocamentos de seus nós, i.é.:

$$\mathcal{U}_h = \left\{[\Delta x_h, \Delta y_h] \in C^0([0,L]) \times C^0([0,L]) : \ \Delta x_h\Big|_{K_e},\ \Delta y_h\Big|_{K_e} \in \mathbb{P}_r(K_e)\ \forall K_e \in \mathrm{T}_h\right\} \tag{3.20}$$

Dois parâmetros caracterizam a aproximação: a quantidade de elementos da partição (inversamente proporcional a h) e o grau de aproximação polinomial r. Como se trata de um domínio 1D, a quantidade de nós requerida para cada elemento é $n_n^e = r + 1$, suficiente para a interpolação do correspondente polinômio local.

Como já dito, a idéia do método é encontrar o elemento do espaço 3.20 que minimize a energia potencial total de deformação. A validade do método se dá pelo fato do espaço $\mathcal{U}_h$ possuir a propriedade de saturação[2] em relação a $\mathcal{U}$.

Seja n_e o número de elementos da malha e para o caso de malha uniforme $K_e = (eh, eh + h)$ $e = 0, 1, \ldots, n_e - 1$ onde $h = \frac{L}{n_e}$. Tomando um elemento qualquer considera-se o campo de deflexões (ou de deslocamentos):

$$\Delta x_h(s) = \sum_{e=0}^{n_e-1} \sum_{i=0}^{n_n^e-1} \Delta X_i^e \phi_i^e \tag{3.21a}$$

$$\Delta y_h(s) = \sum_{e=0}^{n_e-1} \sum_{i=0}^{n_n^e-1} \Delta Y_i^e \phi_i^e \tag{3.21b}$$

Sendo ϕ_i^e a função de forma do e-ésimo elemento relacionada ao i-ésimo nó ($i = 0, 1, \ldots, n_n^e - 1$).

Com essa discretização, um número finito de parâmetros (deslocamentos em cada nó do domínio) é suficiente a caracterizar todo o campo de deslocamentos. Observe-se que, excluindo os nós de fronteira, o último nó de cada elemento coincide com o primeiro do elemento seguinte, isso significa que para o caso de um espaço de aproximação com polinômios de primeiro grau (dois nós por elemento) são presentes $n_n = 2n_e - n_e + 1$ nós e para polinômios de segundo grau $n_n = 2n_e + 1$ pois aos nós anteriores se adiciona um nó interno para cada elemento. Assim, como as incógnitas são duas, são necessários $2n_n$ parâmetros para a descrição completa do estado de deflexão do problema.

Considerando que cada nó possui uma ou duas funções de forma (uma para nós internos e de fronteira do domínio e duas para os restantes) e considerando $\psi_n(s) =$

[2] Ver a página 34 para a definição.

$\sum_{i=1}^{n_{ec}(n)} \phi_e^i(s)$ a soma das funções de forma do n-ésimo nó que possui $n_{ec} = n_{ec}(n)$ elementos em comum, a 3.21 pode ser rescrita da seguinte maneira:

$$\Delta x_h(s) = \sum_{n=0}^{n_n-1} \Delta X_n \psi_n(s) \tag{3.22a}$$

$$\Delta y_h(s) = \sum_{n=0}^{n_n-1} \Delta Y_n \psi_n(s) \tag{3.22b}$$

O sistema linear correspondente se obtém substituindo o campo 3.22 na equação 3.11 e tomando $2n_n$ deslocamentos virtuais linearmente independentes iguais exatamente a $[\psi_n(s), 0]$ ou $[0, \psi_n(s)]$ onde $n = 0, 1, \ldots, n_n$. Veja a 3.23 para o sistema linear equivalente:

$$\begin{bmatrix} \mathbf{K}_{XX}(x_k, y_k) & \mathbf{K}_{XY}(x_k, y_k) \\ \mathbf{K}_{YX}(x_k, y_k) & \mathbf{K}_{YY}(x_k, y_k) \end{bmatrix} \begin{bmatrix} \Delta\vec{X} \\ \Delta\vec{Y} \end{bmatrix} = \begin{bmatrix} \vec{F}_x(x_k, y_k) \\ \vec{F}_y(x_k, y_k) \end{bmatrix} \tag{3.23}$$

Onde, tomando os termos das equações 3.9 e 3.12:

$$\begin{aligned} \mathbf{K}_{XX,ij}(x_k, y_k) = &\sum_{e=1}^{n_{ec}} \int_{K_e} -\mathcal{A}_{11}(x_k, y_k) \frac{\mathrm{d}\psi_j}{\mathrm{d}s} \frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{B}_{11}(x_k, y_k) \frac{\mathrm{d}\psi_j}{\mathrm{d}s} \psi_i + \\ &+ \mathcal{C}_{11}(x_k, y_k) \psi_j \psi_i \mathrm{d}s \quad i, j = 1, 2, \ldots, n_n \end{aligned} \tag{3.24a}$$

$$\begin{aligned} \mathbf{K}_{XY,ij}(x_k, y_k) = &\sum_{e=1}^{n_{ec}} \int_{K_e} -\mathcal{A}_{12}(x_k, y_k) \frac{\mathrm{d}\psi_j}{\mathrm{d}s} \frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{B}_{12}(x_k, y_k) \frac{\mathrm{d}\psi_j}{\mathrm{d}s} \psi_i + \\ &+ \mathcal{C}_{12}(x_k, y_k) \psi_j \psi_i \mathrm{d}s \quad i, j = 1, 2, \ldots, n_n \end{aligned} \tag{3.24b}$$

$$\begin{aligned} \mathbf{K}_{YX,ij}(x_k, y_k) = &\sum_{e=1}^{n_{ec}} \int_{K_e} -\mathcal{A}_{21}(x_k, y_k) \frac{\mathrm{d}\psi_j}{\mathrm{d}s} \frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{B}_{21}(x_k, y_k) \frac{\mathrm{d}\psi_j}{\mathrm{d}s} \psi_i + \\ &+ \mathcal{C}_{21}(x_k, y_k) \psi_j \psi_i \mathrm{d}s \quad i, j = 1, 2, \ldots, n_n \end{aligned} \tag{3.24c}$$

$$\begin{aligned} \mathbf{K}_{YY,ij}(x_k, y_k) = &\sum_{e=1}^{n_{ec}} \int_{K_e} -\mathcal{A}_{22}(x_k, y_k) \frac{\mathrm{d}\psi_j}{\mathrm{d}s} \frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{B}_{22}(x_k, y_k) \frac{\mathrm{d}\psi_j}{\mathrm{d}s} \psi_i + \\ &+ \mathcal{C}_{22}(x_k, y_k) \psi_j \psi_i \mathrm{d}s \quad i, j = 1, 2, \ldots, n_n \end{aligned} \tag{3.24d}$$

$$\vec{F}_{Xi}(x_k, y_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} \mathcal{L}_A^1(x_k, y_k) \frac{\mathrm{d}\psi_i}{\mathrm{d}s} - \mathcal{L}_B^1(x_k, y_k) \psi_i \mathrm{d}s \quad i = 1, 2, \ldots, n_n \tag{3.24e}$$

$$\vec{F}_{Yi}(x_k, y_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} \mathcal{L}_A^2(x_k, y_k) \frac{\mathrm{d}\psi_i}{\mathrm{d}s} - \mathcal{L}_B^2(x_k, y_k) \psi_i \mathrm{d}s \quad i = 1, 2, \ldots, n_n \tag{3.24f}$$

A quantidade $n_{ec} = n_{ec}(ij)$ indica o número de elementos em comum entre os nós i e j. Esse número pode ser zero para nós que pertencem a elementos diferentes, um para nós diferentes pertencentes ao mesmo elemento e dois para nós nas extremidades de elementos onde vale também $i = j$.

Na realidade a organização da matriz é um pouco diferente pois os nós de Dirichlet são colocados por último por uma questão de implementação, mas essa discussão será abordada numa parte posterior do texto para evitar repetições.

3.1.2.2 Método de Galerkin - formulação mista

A aproximação com o método misto é muito similar à formulação clássica contudo uma diferença importante é o fato de adotar espaços polinomiais diferentes entre variáreis de deslocamento e de força. Essa abordagem é sugerida pelo próprio fato que a distribuição de tração possui regularidade[3] menor que a distribuição de deslocamento pois é proporcional à derivada dessa última. Assim, os espaços funcionais de aproximação serão do seguinte tipo:

$$\mathcal{U}_h = \left\{ [\Delta x_h, \Delta y_h] \in C^0([0,L]) \times C^0([0,L]) : \ \Delta x_h\Big|_{K_e}, \ \Delta y_h\Big|_{K_e} \in \mathbb{P}_r(K_e) \ \forall K_e \in \mathrm{T}_h \right\} \tag{3.25a}$$

$$\mathcal{Q}_h = \left\{ [\Delta N_h] \in L^2([0,L]) : \ \Delta N_h\Big|_{K_e} \in \mathbb{P}_t(K_e) \ \forall K_e \in \mathrm{T}_h \right\} \tag{3.25b}$$

Com $t < r$. Note que $\mathbb{P}_0(K_e)$ é o espaço onde os valores são constantes no interior de K_e.

Com esse fato, se conclui que o número de nós do problema difere em deslocamentos e trações. Seja $\varphi_n(s)$ $n = 1, 2, \ldots, n_{nt}$ a soma das funções de forma relacionadas ao n-ésimo nó da variável tração, onde n_{n_t} é o número total de nós relacionados ao campo de trações.

Como feito na seção anterior, após a substituição dos campos de deslocamento e de tração aproximados na 3.18 e impondo os deslocamentos virtuais como sendo as $2n_n + n_{nt}$ somas de funções de forma associadas a um mesmo nó, se chega ao seguinte sistema linear:

$$\begin{bmatrix} \mathbf{K}^m_{XX}(x_k,y_k,N_k) & \mathbf{K}^m_{XY}(x_k,y_k,N_k) & \mathbf{K}^m_{XN}(x_k,y_k,N_k) \\ \mathbf{K}^m_{YX}(x_k,y_k,N_k) & \mathbf{K}^m_{YY}(x_k,y_k,N_k) & \mathbf{K}^m_{YN}(x_k,y_k,N_k) \\ \mathbf{K}^m_{NX}(x_k,y_k,N_k) & \mathbf{K}^m_{NY}(x_k,y_k,N_k) & \mathbf{K}^m_{NN}(x_k,y_k,N_k) \end{bmatrix} \begin{bmatrix} \Delta\vec{X} \\ \Delta\vec{Y} \\ \Delta\vec{N} \end{bmatrix} = \begin{bmatrix} \vec{F_X}^m(x_k,y_k,N_k) \\ \vec{F_Y}^m(x_k,y_k,N_k) \\ \vec{F_N}^m(x_k,y_k,N_k) \end{bmatrix} \tag{3.26}$$

Onde, tomando os termos das equações 3.19:

$$\begin{aligned} \mathbf{K}^m_{XX,ij}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} & -\mathcal{A}^m_{11}(x_k,y_k,N_k)\frac{\mathrm{d}\psi_j}{\mathrm{d}s}\frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{B}^m_{11}(x_k,y_k,N_k)\frac{\mathrm{d}\psi_j}{\mathrm{d}s}\psi_i + \\ & + \mathcal{C}^m_{11}(x_k,y_k,N_k)\psi_j\psi_i \mathrm{d}s \quad i,j = 1,2,\ldots,n_n \end{aligned} \tag{3.27a}$$

$$\begin{aligned} \mathbf{K}^m_{XY,ij}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} & -\mathcal{A}^m_{12}(x_k,y_k,N_k)\frac{\mathrm{d}\psi_j}{\mathrm{d}s}\frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{B}^m_{12}(x_k,y_k,N_k)\frac{\mathrm{d}\psi_j}{\mathrm{d}s}\psi_i + \\ & + \mathcal{C}^m_{12}(x_k,y_k,N_k)\psi_j\psi_i \mathrm{d}s \quad i,j = 1,2,\ldots,n_n \end{aligned} \tag{3.27b}$$

[3] A regularidade de uma função é uma noção que avalia pontos críticos como ângolos, assíntotas e cúspedes. Para quantificar essa noção se estuda a continuidade das derivadas da função: por exemplo uma função em $C^1(\Omega)$ é uma função contínua com derivada contínua e uma função em $C^2(\Omega)$ é uma função contínua com todas derivadas até a segunda ordem contínuas. Assim pode-se afirmar que $C^2(\Omega) \subset C^1(\Omega)$ e que uma função genérica de $C^2(\Omega)$ possui regularidade maior ou igual a uma função de $C^1(\Omega)$.

$$\mathbf{K}^m_{XN,ij}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} -\mathcal{A}^m_{13}(x_k,y_k,N_k)\varphi_j \frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{C}^m_{13}(x_k,y_k,N_k)\varphi_j\psi_i \mathrm{d}s + \\ i=1,2,\ldots,n_n \; j=1,2,\ldots,n_{nt} \tag{3.27c}$$

$$\mathbf{K}_{YX,ij}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} -\mathcal{A}^m_{21}(x_k,y_k,N_k)\frac{\mathrm{d}\psi_j}{\mathrm{d}s}\frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{B}^m_{21}(x_k,y_k,N_k)\frac{\mathrm{d}\psi_j}{\mathrm{d}s}\psi_i + \\ + \mathcal{C}^m_{21}(x_k,y_k,N_k)\psi_j\psi_i \mathrm{d}s \quad i,j=1,2,\ldots,n_n \tag{3.27d}$$

$$\mathbf{K}^m_{YY,ij}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} -\mathcal{A}^m_{22}(x_k,y_k,N_k)\frac{\mathrm{d}\psi_j}{\mathrm{d}s}\frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{B}^m_{22}(x_k,y_k,N_k)\frac{\mathrm{d}\psi_j}{\mathrm{d}s}\psi_i + \\ + \mathcal{C}^m_{22}(x_k,y_k,N_k)\psi_j\psi_i \mathrm{d}s \quad i,j=1,2,\ldots,n_n \tag{3.27e}$$

$$\mathbf{K}_{YN,ij}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} -\mathcal{A}^m_{23}(x_k,y_k,N_k)\varphi_j \frac{\mathrm{d}\psi_i}{\mathrm{d}s} + \mathcal{C}^m_{23}(x_k,y_k,N_k)\varphi_j\psi_i \mathrm{d}s + \\ i=1,2,\ldots,n_n \; j=1,2,\ldots,n_{nt} \tag{3.27f}$$

$$\mathbf{K}^m_{NX,ij}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} x'_k \frac{\mathrm{d}\psi_j}{\mathrm{d}s}\varphi_i \mathrm{d}s \quad i=1,2,\ldots,n_{nt} \; j=1,2,\ldots,n_n \tag{3.27g}$$

$$\mathbf{K}^m_{NY,ij}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} y'_k \frac{\mathrm{d}\psi_j}{\mathrm{d}s}\varphi_i \mathrm{d}s \quad i=1,2,\ldots,n_{nt} \; j=1,2,\ldots,n_n \tag{3.27h}$$

$$\mathbf{K}^m_{NN,ij}(x_k,y_k,N_k) = -\sum_{e=1}^{n_{ec}} \int_{K_e} \frac{1}{EA}\left(1+\frac{N_k}{EA}\right)\varphi_j\varphi_i \mathrm{d}s \quad i,j=1,2,\ldots,n_{nt} \tag{3.27i}$$

$$\vec{F_{X}}_i^{\,m}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} \mathcal{L}^{m1}_A(x_k,y_k,N_k)\frac{\mathrm{d}\psi_i}{\mathrm{d}s} - \mathcal{L}^{m1}_B(x_k,y_k,N_k)\psi_i \mathrm{d}s \quad i=1,2,\ldots,n_n \tag{3.27j}$$

$$\vec{F_{Y}}_i^{\,m}(x_k,y_k,N_k) = \sum_{e=1}^{n_{ec}} \int_{K_e} \mathcal{L}^{m2}_A(x_k,y_k,N_k)\frac{\mathrm{d}\psi_i}{\mathrm{d}s} - \mathcal{L}^{m2}_B(x_k,y_k,N_k)\psi_i \mathrm{d}s \quad i=1,2,\ldots,n_n \tag{3.27k}$$

$$\vec{F_{N}}_i^{\,m}(x_k,y_k,N_k) = -\sum_{e=1}^{n_{ec}} \int_{K_e} \frac{1}{2}\left(x'^2_k + y'^2_k - \left(1+\frac{N_k}{EA}\right)^2\right)\varphi_i \mathrm{d}s \quad i=1,2,\ldots,n_{nt} \tag{3.27l}$$

3.1.3 O problema local em sua formulação axissimétrica

Nesta seção será feito um estudo dos efeitos axissimétricos relacionados ao problema. É importante notar que os efeitos axissimétricos não influenciam a forma que a estrutura assume pois suas componentes são equilibradas em todas as direções. Uma outra consideração importante é que esses efeitos podem ser estudados *localmente*[4] pois os gradientes dos esforços (pressões e mudanças de temperatura) aos quais o corpo é submetido são ínfimos e dessa maneira é possível estudar somente uma porção pequena do comprimento da tubulação.

O problema estrutural é, na realidade, acoplado ao problema térmico pois as constantes de elasticidade (E e G) são dependentes da temperatura assim como a constante de difusão térmica (α) é dependente do estado de tensão local. Esse efeito será desprezado pois é pouco relevante e a abordagem utilizada para a componente axissimétrica e estática do problema será a de considerar um isolamento perfeito, i.é.: a temperatura no interior da parede do tubo interno será considerada igual à do óleo que flui e a temperatura externa à parede do tubo externo será considerada igual à do oceano. Nessa ótica a solução da parte axissimétrica do problema é composta de duas etapas:

i Solução do modelo de difusão térmica que é, no máximo, bidimensional, dada a axissimetria do problema;

ii Inclusão do efeito térmico no modelo estrutural axissimétrico através dos contributos forçantes que derivam da variação de temperatura e consequente expansão/contração da estrutura.

3.1.3.1 Difusão térmica

A temperatura do fluido, dada a hipótese de isolamento perfeito, permanecerá inalterada por todo o comprimento do *riser*, já a temperatura do oceano será considerada em uma primeira análise constante e posteriormente será incluído o efeito da variação da temperatura externa usando como parâmetro uma curva de temperatura por profundidade, como na figura 8.

Da equação de *Fourier* e do balanço energético pode-se obter a equação que rege a difusão de temperatura no caso de material isotropo mas não homogêneo (pela presença de mais de um material isotropo):

$$\begin{cases} -div(c(r)\nabla T(\vec{x})) = 0 & \forall \vec{x} \in \Omega \\ T(R_i) = \overline{T}_i(z) \\ T(R_e) = \overline{T}_e(z) & , \end{cases} \tag{3.28}$$

[4] Com a expressão *localmente* refere-se ao fato de tomar um pequeno comprimento da tubulação e não ao significado que a palavra assume usualmente no contexto do estudo de vigas (i.é. estudo das tensões a partir dos esforços solicitantes de uma seção transversal).

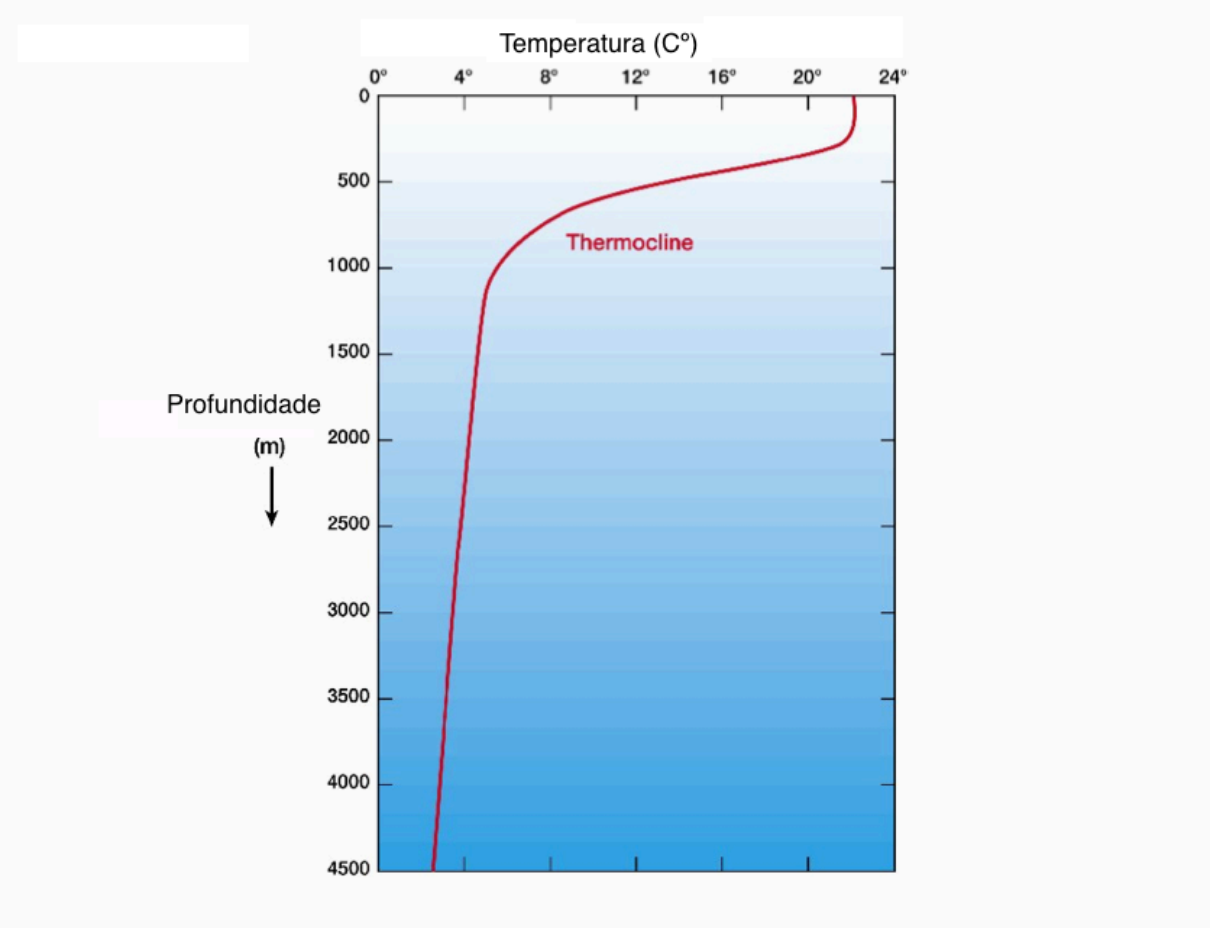


Figura 8: Temperatura da água do oceano em relação à profundidade.

Fonte: http://www.windows2universe.org/earth/Water/temp.html (BERGMAN,)

Com $c(r)$ o coeficiente de difusão térmica que é função do material e portanto do raio r dada a geometria da tubulação. Se os materiais não apresentassem isotropia c seria uma matriz simétrica e definida positiva.

Finalmente se procede à formulação variacional: nesse caso o problema é *full-Dirichlet* com condições não homogêneas e assim sendo se usa uma mudança de variável com auxílio de uma função dita de *relevo* $R = R(r, z)$ construída ad hoc em modo a satisfazer as seguintes relações:

$$\begin{cases} R(R_i, z) = T_i(z) \\ R(R_e, z) = T_e(z), \end{cases} \tag{3.29}$$

É suficiente tomar uma função que interpole linearmente os dados na fronteira. Em seguida o problema se constrói com a variável $\hat{T}(r, z) = T(r, z) - R(r, z)$ que assume, por definição, valores nulos na fronteira de Dirichlet ($r = R_i$ e $r = R_e$). Substituindo a nova variável, multiplicando a equação por uma função de teste de $H_0^1(\Omega)$, integrando a direita e a esquerda a EDP e usando a fórmula de Green se chega à seguinte formulação variacional:

Encontrar $\hat{T} \in H_0^1(\Omega)$ de modo que:

$$\int_\Omega c(r)\nabla\hat{T}(\vec{x})\nabla\phi(\vec{x})\mathrm{d}V(\vec{x}) = -\int_\Omega c(r)\nabla R(\vec{x})\nabla\phi(\vec{x})\mathrm{d}V(\vec{x}) \quad \forall\phi \in H_0^1(\Omega) \tag{3.30}$$

Sendo um problema axissimétrico se reduz a um problema *2D* e, no caso de temperaturas externas constantes, se reduz adicionalmente a um problema *1D*. Para o caso *2D* deve-se impor condições de contorno também na parte superior e inferior da superfície, que será uma condição de *Newmann* de fluxo de calor nulo ($\frac{\partial T}{\partial \vec{n}} = 0 \in \Gamma_N$).

Finalmente, especificando os extremos de integração a formulação, pronta para a fase de discretização é:

Encontrar $\hat{T} \in H_0^1(\Omega)$ tal que:

$$\begin{aligned}&\int_0^L \int_{R_i}^{R_e} c(r)\left\{\frac{\partial\hat{T}(r,z)}{\partial r}\frac{\partial\phi(r,z)}{\partial r} + \frac{\partial\hat{T}(r,z)}{\partial z}\frac{\partial\phi(r,z)}{\partial z}\right\}r\mathrm{d}r\mathrm{d}z = \\ &= -\int_0^L \int_{R_i}^{R_e} c(r)\left\{\frac{\partial R(r,z)}{\partial r}\frac{\partial\phi(r,z)}{\partial r} + \frac{\partial R(r,z)}{\partial z}\frac{\partial\phi(r,z)}{\partial z}\right\}r\mathrm{d}r\mathrm{d}z \quad \forall\phi \in H^1_{\Gamma_D}(\Omega)\end{aligned} \tag{3.31}$$

Uma vez resolvido o problema 3.31 resta fazer a substituição $T(r,z) = \hat{T}(r,z) + R(r,z)$.

Operacionalmente o que se faz é construir o sistema linear com uma base de um subespaço de $H^1(\Omega)$ (não de $H^1_{\Gamma_D}(\Omega)$) e sucessivamente passar a direita a parcela correspondente ao contributo de Dirichlet uma vez conformado o sistema. Sendo uma aproximação de segundo grau, o espaço usado é $X_h^2 = \left\{v_h \in C^0(\overline{\Omega}) : \ v_h|_{K_i} \in \mathbb{P}_2(K_i),\ \forall K_i \in \mathrm{T}_h\right\}$. Usando uma base $\psi_i(\vec{x})$ con $i = 1, 2, \ldots, n_n$ do espaço X_h^2 com n_n o número de nós (ou graus de liberdade pois coincidem nesse caso). Tendo posto $\psi_i(\vec{x}) = \sum_{e=1}^{n_e} \phi_e^i(\vec{x})$ onde $\phi_e^i(\vec{x})$ são as funções lagrangeanas associadas ao i-ésimo nó no e-ésimo elemento ao qual tal nó pertence. Em outras palavras, $\psi_i(\vec{x})$ é a soma das funções lagrangeanas as quais um mesmo nó é relacionado, veja a seguir a distribuição do número de funções de forma para cada tipo de nó:

- quatro para nós no interior do domínio nos vértices dos elementos;
- duas para nós no interior do domínio e nas faces dos elementos;
- uma para nós no interior do domínio e interiores aos elementos;
- duas para nós no contorno do domínio e nos vértices dos elementos (excluídos os nós de vértice de domínio);
- uma para nós no contorno do domínio e nas faces dos elementos;
- uma para nós nos vértices do domínio.

Com essa notação a aproximação da temperatura pode ser escrita como $T_h(\vec{(x)}) = \sum_{j=1}^{n_n} T_j \psi_j(\vec{x})$.

Deixando por último o contributo dos nós Dirichlet o sistema linear pode ser escrito como segue:

$$\begin{bmatrix} \mathbf{K}_{\Omega\Omega} & \mathbf{K}_{\Omega\Gamma_D} \\ \mathbf{K}_{\Gamma_D\Omega} & \mathbf{K}_{\Gamma_D\Gamma_D} \end{bmatrix} \begin{bmatrix} \vec{T}_\Omega \\ \vec{T}_{\Gamma_D} \end{bmatrix} = \begin{bmatrix} \vec{0} \\ \vec{0} \end{bmatrix} \tag{3.32}$$

Com:

$$\mathbf{K}^{ij}_{\Omega\Omega} = \sum_{e=1}^{n_{ec}} \int_{K_e} c(r) r \nabla\psi_i(r,z).\nabla\psi_j(r,z)\mathrm{d}(r)\mathrm{d}(z) \quad i,j = 1,2,\ldots,n_\Omega \tag{3.33a}$$

$$\mathbf{K}^{ij}_{\Omega\Gamma_D} = \sum_{e=1}^{n_{ec}} \int_{K_e} c(r) r \nabla\psi_i(r,z).\nabla\psi_j(r,z)\mathrm{d}(r)\mathrm{d}(z) \quad i = 1,\ldots,n_\Omega;\ j = n_\Omega+1,\ldots,n_n \tag{3.33b}$$

$$\mathbf{K}^{ij}_{\Omega\Gamma_D} = \mathbf{K}^{ji}_{\Gamma_D\Omega} \tag{3.33c}$$

$$\mathbf{K}^{ij}_{\Gamma_D\Gamma_D} = \sum_{e=1}^{n_{ec}} \int_{K_e} c(r) r \nabla\psi_i(r,z).\nabla\psi_j(r,z)\mathrm{d}(r)\mathrm{d}(z) \quad i,j = n_\Omega+1,\ldots,n_n \tag{3.33d}$$

Com n_{ec} o número de elementos que os nós i e j possuem em comum, n_Ω o número de nós internos ao domínio mais o número de nós presentes na fronteira de Neumann e n_n o número total de nós. Note que os últimos $n_n - n_\Omega$ são os nós que pertencem a fronteirade Dirichlet.

O sistema linear pode portanto ser resolvido como na equação 2.15 na página 33 em ausência de vetor forçante.

3.1.3.2 Inclusão do efeito térmico no problema axissimétrico

Uma vez resolvido o sistema de difusão térmica e dispondo da distribuição de temperatura no corpo, a expansão/contração gerada pela variação de temperatura local produz, por sua vez, tensões. Considerando nulas as autotensões provenientes de processos produtivos, a relação A.27 (lei de Hooke) na página 107 em presença de efeitos anelásticos térmicos, ainda sob hipótese de isotropia se escreve:

$$\sigma_{ij} = 2\mu e_{ij} + \lambda\delta_{ij}e_{kk} \tag{3.34}$$

Com $e_{ij} = \epsilon_{ij} - \delta_{ij}\alpha\Delta T$ a parte elástica do tensor de Cauchy, α *coeficiente de expansão térmica* e $\Delta T = T - T_0$ com T_0 uma temperatura de referência na qual as deformações anelásticas se anulam.

A relação constitutiva final fica conforme a equação 3.35:

$$\sigma_{ij} = 2\mu\epsilon_{ij} + \delta_{ij}\left[\lambda e_{kk} - (2\mu + 3\lambda)\alpha\Delta T\right] \tag{3.35}$$

Com o mesmo procedimento usado na formulação variacional 2.4 mas usando a nova relação entre tensões e deformações 3.34 aparece uma nova parcela a direita da formulação:

Encontrar $\vec{u}(\vec{x}) \in H^1_{\Gamma_D}(\Omega;\mathbb{R}^3)$ tal que:

$$\begin{aligned}&\int_\Omega \left[2\mu(r)\mathbf{E}(\vec{u}) : \mathbf{E}(\vec{v}) + \lambda(r)Tr(\mathbf{E}(\vec{u}))Tr(\mathbf{E}(\vec{v}))\right]\mathrm{d}\Omega = \\ &\quad = \int_\Omega \left[\vec{b}.\vec{v} + (2\mu(r) + 3\lambda(r))\Delta T\alpha(r)Tr(\mathbf{E}(\vec{v}))\right]\mathrm{d}\Omega + \int_{\Gamma_N} \vec{p}.\vec{v}\mathrm{d}\Gamma \\ &\quad \forall \vec{v} \in H^1_{\Gamma_D}(\Omega;\mathbb{R}^3)\end{aligned} \tag{3.36}$$

Note que $Tr(\mathbf{E}(\vec{u})) = div(\vec{u})$.

Com a axisimetria do problema e a solicitação somente no plano meridional, pode-se afirmar que as derivadas parciais em relação a θ são nulas e que $u_\theta = 0$. As equações de congruência interna em coordenadas cilíndricas se simplificam como segue:

$$\begin{cases} \epsilon_r = \frac{\partial u_r}{\partial r} \\ \epsilon_\theta = \frac{1}{r}\frac{\partial u_\theta}{\partial \theta} + \frac{u_r}{r} = \frac{u_r}{r} \\ \epsilon_z = \frac{\partial u_z}{\partial z}, \end{cases} \quad \begin{cases} \gamma_{rz} = \frac{\partial u_r}{\partial z} + \frac{\partial u_z}{\partial r} \\ \gamma_{r\theta} = \frac{1}{r}\frac{\partial u_r}{\partial \theta} + \frac{\partial u_\theta}{\partial r} - \frac{u_\theta}{r} = -\frac{u_\theta}{r} = 0 \\ \gamma_{\theta z} = \frac{1}{r}\frac{\partial u_z}{\partial \theta} + \frac{\partial u_\theta}{\partial z} = 0, \end{cases} \tag{3.37}$$

Com essas podem ser definidos os extremos de integração e a formulação variacional do problema específico:

Encontrar $\vec{u}(r,z) = (u_r(r,z), u_z(r,z)) \in H^1_{\Gamma_D}([R_i,R_e]\times[0,L];\mathbb{R}^2)$ tal que:

$$\begin{aligned}&\int_0^L \int_{R_i}^{R_e} 2\mu(r)r\left\{\frac{\partial u_r}{\partial r}\frac{\partial v_r}{\partial r} + \frac{u_r}{r}\frac{v_r}{r} + \frac{\partial u_z}{\partial z}\frac{\partial v_z}{\partial z} + \left(\frac{\partial u_r}{\partial z} + \frac{\partial u_z}{\partial r}\right)\left(\frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r}\right)\right\} + \\ &\quad + \lambda(r)r\left\{\left(\frac{\partial u_r}{\partial r} + \frac{u_r}{r} + \frac{\partial u_z}{\partial z}\right)\left(\frac{\partial v_r}{\partial r} + \frac{v_r}{r} + \frac{\partial v_z}{\partial z}\right)\right\}\mathrm{d}r\mathrm{d}z = \\ &\quad = \int_0^L \int_{R_i}^{R_e} gr(\rho_f - \rho_m(r)).v_z + \\ &\quad + (2\mu(r) + 3\lambda(r))r\alpha(r)\Delta T(r,z)\left(\frac{\partial v_r}{\partial r} + \frac{v_r}{r} + \frac{\partial v_z}{\partial z}\right)\mathrm{d}r\mathrm{d}z + \\ &\quad + \int_0^L R_i p_i(z).v_r\mathrm{d}z - \int_0^L R_e p_e(z).v_r\mathrm{d}z \\ &\quad \forall \vec{v}(r,z) = (v_r,v_z) \in H^1_{\Gamma_D}([R_i,R_e]\times[0,L];\mathbb{R}^2),\end{aligned} \tag{3.38}$$

Onde R_i e R_e são os raios interno e externo, ρ_f e $\rho_m(r)$ são as densidades do fluido movido (água) e do material (por isso a dependência em relação a r). Por último L é o comprimento do *riser* e g a aceleração da gravidade. Note também a dependência

dos parâmetros μ, λ e α em relação a r e a dependência de p_i e p_e em relação a z. A temperatura T é fornecida pela resolução do sistema de difusão.

A essa altura é pertinente discutir as hipóteses adotadas e esclarecer alguns pontos: na formulação desenvolvida, alguns fenômenos foram desprezados como o efeito do fluxo interno e outros não incluídos pois já estudados na etapa global como o carregamento de correntes. Os efeitos do empuxo e do peso próprio foram incluídos na última equação somente para ilustrar como incluir carregamentos de volume porém na etapa de cálculo não serão considerados pois já abordados na etapa global. A tubulação, nessa etapa é considerada engastada em uma extremidade onde, portanto, é usada condição de contorno de Dirichlet homogênea (i.é.: $\vec{u} = \vec{0}$ para $z = 0$) e livre para translações axiais na outra (por isso não aparecem forçastes de superfície para o extremo superior).

Sob as observações anteriores a equação 3.38 é pronta para a fase de discretização através do *método de Galerkin* descrito na seção 2.2.2. O procedimento é análogo ao utilizado no desenvolvimento do problema da difusão de temperatura mas com o espaço funcional bidimensional $V_h = X_h^2 \times X_h^2 = \Big\{[v_r, v_z] \in C^0(\overline{\Omega}) \times C^0(\overline{\Omega}) : \ v_r|_{K_i}, v_z|_{K_i} \in \mathbb{P}_2(K_i), \ \forall K_i \in \mathrm{T}_h\Big\}$.

Omitindo o procedimento formal (suficientemente tedioso dada a complexidade de notação e do termo a esquerda da 3.38) se observa que o sistema final será do tipo:

$$\begin{bmatrix} \mathbf{K}_{\Omega\Omega}^{UU} & \mathbf{K}_{\Omega\Omega}^{UV} & \mathbf{K}_{\Omega\Gamma_D}^{UU} & \mathbf{K}_{\Omega\Gamma_D}^{UV} \\ \mathbf{K}_{\Omega\Omega}^{VU} & \mathbf{K}_{\Omega\Omega}^{VV} & \mathbf{K}_{\Omega\Gamma_D}^{VU} & \mathbf{K}_{\Omega\Gamma_D}^{VV} \\ \mathbf{K}_{\Gamma_D\Omega}^{UU} & \mathbf{K}_{\Gamma_D\Omega}^{UV} & \mathbf{K}_{\Gamma_D\Gamma_D}^{UU} & \mathbf{K}_{\Gamma_D\Gamma_D}^{UV} \\ \mathbf{K}_{\Gamma_D\Omega}^{VU} & \mathbf{K}_{\Gamma_D\Omega}^{VV} & \mathbf{K}_{\Gamma_D\Gamma_D}^{VU} & \mathbf{K}_{\Gamma_D\Gamma_D}^{VV} \end{bmatrix} \begin{bmatrix} \vec{U}_\Omega \\ \vec{V}_\Omega \\ \vec{U}_{\Gamma_D} \\ \vec{V}_{\Gamma_D} \end{bmatrix} = \begin{bmatrix} \vec{F}_\Omega^U \\ \vec{F}_\Omega^V \\ \vec{F}_{\Gamma_D}^U \\ \vec{F}_{\Gamma_D}^V \end{bmatrix} \tag{3.39}$$

Tendo posto $\vec{U}$ como vetor de incógnitas de u_r e $\vec{V}$ vetor de incógnitas de u_z.

$$\begin{aligned} \mathbf{K}_{\Omega\Omega}^{UU,ij} = \sum_{e=1}^{n_{ec}} \int_{K_e} & 2\mu(r) r \left\{ \frac{\partial \psi_j}{\partial r}\frac{\partial \psi_i}{\partial r} + \frac{\psi_j \psi_i}{r^2} + \frac{\partial \psi_j}{\partial z}\frac{\partial \psi_i}{\partial z} \right\} + \\ & + \lambda(r) r \left\{ \frac{\partial \psi_j}{\partial r}\frac{\partial \psi_i}{\partial r} + \frac{\psi_j}{r}\frac{\partial \psi_i}{\partial r} + \frac{\partial \psi_j}{\partial r}\frac{\psi_i}{r} + \frac{\psi_j \psi_i}{r^2} \right\} \mathrm{d}(r)\mathrm{d}(z) \end{aligned} \tag{3.40}$$

Com $i, j = 1, \ldots, n_\Omega$ e n_Ω o número de nós internos mais o número de nós sobre o contorno de Neumann.

$$\mathbf{K}_{\Omega\Omega}^{UV,ij} = \sum_{e=1}^{n_{ec}} \int_{K_e} 2\mu(r) r \left\{ \frac{\partial \psi_j}{\partial r}\frac{\partial \psi_i}{\partial z} \right\} + \lambda(r) r \left\{ \frac{\partial \psi_j}{\partial z}\frac{\partial \psi_i}{\partial r} + \frac{\partial \psi_j}{\partial z}\frac{\psi_i}{r} \right\} \mathrm{d}(r)\mathrm{d}(z) \tag{3.41}$$

Com $i, j = 1, \ldots, n_\Omega$.

$$\mathbf{K}_{\Omega\Omega}^{VV,ij} = \sum_{e=1}^{n_{ec}} \int_{K_e} 2\mu(r) r \left\{ \frac{\partial \psi_j}{\partial z}\frac{\partial \psi_i}{\partial z} + \frac{\partial \psi_j}{\partial r}\frac{\partial \psi_i}{\partial r} \right\} + \lambda(r) r \left\{ \frac{\partial \psi_j}{\partial z}\frac{\partial \psi_i}{\partial z} \right\} \mathrm{d}(r)\mathrm{d}(z) \tag{3.42}$$

Com $i, j = 1, \ldots, n_\Omega$.

As componentes da matriz $\mathbf{K}^{UU}_{\Omega\Gamma_D}$, $\mathbf{K}^{UV}_{\Omega\Gamma_D}$, $\mathbf{K}^{VV}_{\Omega\Gamma_D}$, $\mathbf{K}^{UV}_{\Gamma_D\Gamma_D}$, $\mathbf{K}^{UU}_{\Gamma_D\Gamma_D}$ e $\mathbf{K}^{VV}_{\Gamma_D\Gamma_D}$ possuem expressões similares às já obtidas ($\mathbf{K}^{UU}_{\Omega\Gamma_D}$ similar a $\mathbf{K}^{UU}_{\Omega\Omega}$, $\mathbf{K}^{UV}_{\Omega\Gamma_D}$ similar a $\mathbf{K}^{UV}_{\Omega\Omega}$ e assim por diante) mas com os índices i e j percorrendo $1,\ldots,n_\Omega$ ou $n_\Omega+1,\ldots,n_n$ dependendo se correspondem a funções de nós de Dirichlet ou não. Para as restantes componentes observa-se que a matriz é simétrica.

$$\begin{aligned}\mathbf{F}^{U,i}_{\Omega} = &\sum_{e=1}^{n_{ec}} \int_{K_e} (2\mu(r)+3\lambda(r))r\alpha(r)\Delta T(r,z)\left\{\frac{\partial\psi_i}{\partial r}+\frac{\psi_i}{r}\right\}\mathrm{d}(r)\mathrm{d}(z)+\\ &+\int_{\partial Ke\cap\Gamma_N} p(r,z)r\psi_i\mathrm{d}(z)\end{aligned} \tag{3.43}$$

Com $i=1,\ldots,n_\Omega$, $p(R_i,z)=p_i(z)$ e $p(R_e,z)=-p_e(z)$.

$$\begin{aligned}\mathbf{F}^{V,i}_{\Omega} = &\sum_{e=1}^{n_{ec}} \int_{K_e} (2\mu(r)+3\lambda(r))r\alpha(r)\Delta T(r,z)\left\{\frac{\partial\psi_i}{\partial z}\right\}+\\ &+gr(\rho_f-\rho_m(r))\psi_i\mathrm{d}(r)\mathrm{d}(z)\end{aligned} \tag{3.44}$$

Com $i=1,\ldots,n_\Omega$.

Ainda uma vez observa-se que as parcelas $\mathbf{F}^{U}_{\Gamma_D}$ e $\mathbf{F}^{V}_{\Gamma_D}$ são iguais às anteriores mudando somente o caminho dos índices $i=1,\ldots,n_\Omega$ a $i=n_\Omega+1,\ldots,n_n$.

No apêndice A na página 1 são presentes os códigos onde pode-se observar os detalhes implementativos.

3.2 Os instrumentos utilizados

Com o avanço tecnológico no ramo da projetação de softwares, um fenômeno recorrente é o da automatização de procedimentos de cálculo e avaliação numérica de problemas complexos. Esse fenômeno gera, por um lado, uma acessibilidade maior à procedimentos de cálculo numérico e um maior dinamismo na formulação de tais problemas. Por outro lado, aumenta a tendência à erros por ausência de conhecimento do usuário em relação aos algoritmos utilizados na resolução. Muitas vezes softwares com interfaces gráficas muito desenvolvidas simplificam a fase de configuração omitindo grande parte dos parâmetros sobre os quais é possível agir. Por mais robustos que sejam os métodos, a análise numérica de problemas complexos não pode ser resumida a simples etapas de configurações pois com o tempo o analista se distancia dos potenciais pontos críticos de uma simulação numérica e do modelo que está por traz da implementação. Ainda mais delicado são os casos nos quais são presentes não-linearidades pois tornam quase impossíveis automatizações confiáveis e eficientes.

Evidentemente a direção contrária também oferece riscos já que seria inútil a cada novo problema reiniciar desde à implementação da resolução de sistemas lineares quando já existem implementações com pouca margem para melhorias. Nesse sentido, a etapa

numérica do presente texto será conduzida em linguagem *C++* com auxílio da biblioteca de elementos finitos *libmesh*.

3.2.1 A linguagem C++ e a biblioteca libmesh

A linguagem *C++* é uma das mais utilizadas na atualidade em problemas do cálculo científico pois combina a versatilidade da programação orientada a objetos (*Object-oriented programming*) com a eficiência de uma linguagem de baixo nível[5]. A própria biblioteca *libmesh* é um ótimo exemplo do potencial da linguagem *C++* pois permite, através de *polimorfismos*, *heranças* e outros instrumentos intrínsecos da linguagem, uma grande abstração e ligação entre conceitos. Veja a figura 9 que mostra a popularidade da linguagem no âmbito da programação. Sem dúvidas essa popularidade, aumenta se se considera o âmbito do cálculo científico. Outras linguagens populares no cálculo científico são: Python, Fortran, C e MATLAB.

Finalmente, o fato de ser uma *linguagem compilada* torna o *C++* mais eficiente que linguagens interpretadas como por exemplo MATLAB e Python. Para o leitor interessado na linguagem *C++* veja (CPLUSPLUS..., 2014) e (PRATA, 2012).

A *libmesh* é uma biblioteca escrita em *C++* onde é implementada uma estrutura sólida para o utilizo do método dos elementos finitos. Essa biblioteca possui dentre seus principais instrumentos:

- Classes e métodos para a geração e leitura em vários formatos de malhas computacionais (*Mesh generation*);
- Classes e métodos para integração numérica/quadraturas;
- Estrutura organizada para a aplicação do método dos elementos finitos em diversos problemas no âmbito das equações diferenciais parciais;
- Compatibilidade com os melhores pacotes para solução de sistemas lineares, como por exemplo PETSc[6].
- Capacidade de execução de código em paralelo, principalmente através MPI[7].

A organização da implementação cabe ao projetista que deve selecionar os mínimos detalhes do método: desde o tipo de elemento e da dimensão da malha até os métodos de integração numérica e de resolução de sistemas lineares. Além disso, o usuário também

[5] O nível de uma linguagem é uma escala fictícia que mede a distância da linguagem em relação à linguagem das máquinas e em relação à linguagem humana. A linguagem é de baixo nível se é próxima à linguagem das máquinas e de alto nível se é próxima à linguagem humana.

[6] Para informação veja (PETSC..., 2014).

[7] MPI - *Message Passing Interface*. Para mais informações veja (OPEN..., 2014).

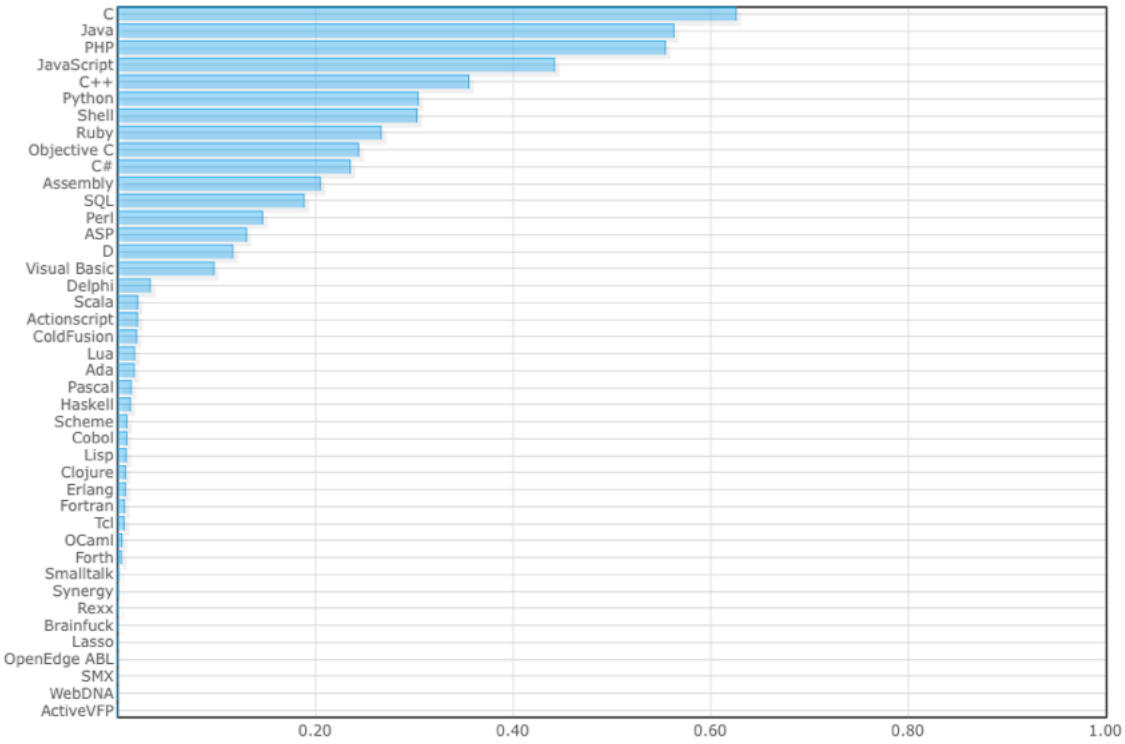


Figura 9: Popularidade linguagens de programação - 2013

Fonte: LangPop.com (2014, http://langpop.com)

é responsável por gerir a construção dos sistemas lineares e para problemas não lineares definir o *loop* do método de resolução.

Libmesh é construída com uma concepção de colaboração com outros softwares e bibliotecas sendo que é capaz de fruir de softwares terceiros com altas prestações e de prover *outputs* nos mais variados formatos para a etapa de *post processing*. Por último, é presente um suporte extensivo para refinamento adaptativo de malha (*adaptive mesh refinement* ou AMR) para plataformas seriais ou paralelas. Para detalhes e informações sobre a biblioteca *libmesh* veja (DEVELOPERS, 2014) e para uma introdução geral veja (VIEIRA, 2009).

3.2.2 O método de refinamento cooperativo

O problema global possui uma série de dificuldades de implementação. Dentre as principais pode-se destacar:

i Necessidade de aproximação inicial relativamente próxima à aproximação final para convergência do método de Newton (principalmente para o método misto);

ii Não-linearidades muito dependentes da escala e malha adotadas (*scale-dependent*

nonlinearities);

iii Sistemas lineares mal condicionados;

iv Imprecisões e instabilidade no cálculo das trações para o método clássico;

As dificuldades encontradas fizeram da análise numérica uma etapa delicada e difícil porém graças ao longo tempo de estudo foi possível descobrir como os métodos clássico e misto podem ser usados de maneira cooperativa combinando suas vantagens: maior eficiência e capacidade de convergência do método clássico[8] e melhor previsão das trações equivalentes pelo método misto são algumas das vantagens que podem ser combinadas.

Antes de tudo cabe notar que o método misto precisa ser inicializado em deslocamentos e em trações e é mais difícil intuir a priori o comportamento da distribuição de trações em relação ao campo de deslocamentos. Esse fato dificulta a utilização do método misto visto que esse converge somente para inicializações muito próximas à solução final.

Ambos os métodos clássico e misto possuem problemáticas no refinamento de malha pois as não-linearidades do problema se amplificam conforme aumentam os graus de liberdade gerando, em última análise, sistemas lineares mal condicionados e divergência para malhas pouco finas.

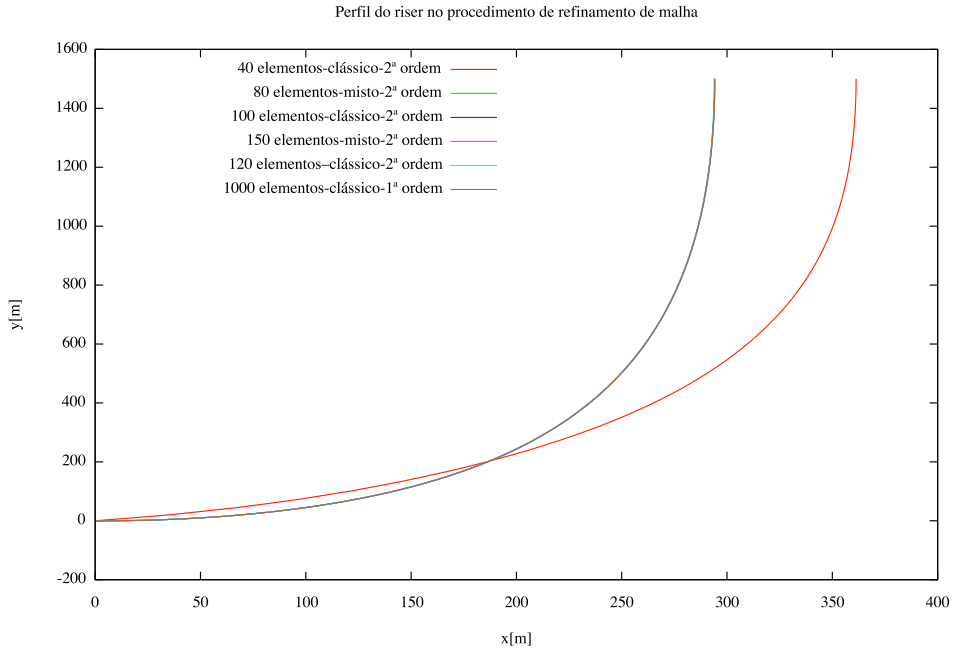
Para responder às dificuldades mencionadas, foi usado um procedimento de resolução alternada entre os métodos clássico e misto que associa em modo ótimo as características de ambos os métodos. A seguir é exposto o procedimento:

Proposição 3.1 (Refinamento de malha combinado)**.** *Antes de tudo se resolve o problema com o método clássico usando uma malha pouco refinada em modo a evitar as instabilidades que derivam das não-linearidades e a obter uma aproximação inicial para a distribuição de trações que possa inicializar o método misto. Em seguida se usa o método misto com uma malha mais refinada obtendo uma aproximação melhor para trações e deslocamentos. Voltando ao método clássico é possível usar malhas ainda mais finas dada a maior proximidade da solução inicial à solução final.*

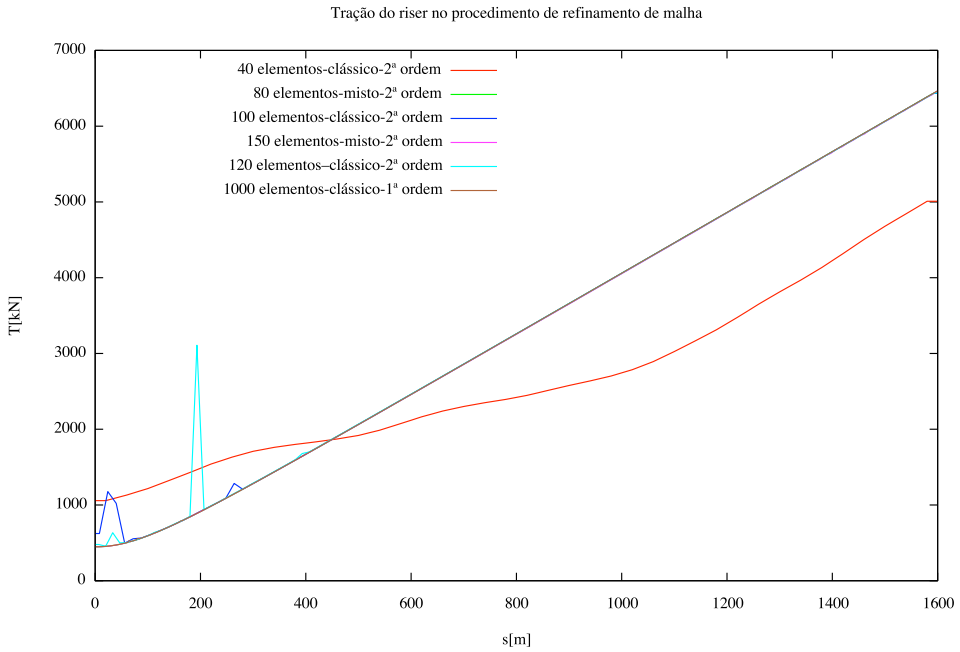
Continuando esse procedimento é possível evitar instabilidades para as trações no método clássico e divergência para o método misto.

Na figura 10 na página 69 pode-se observar o procedimento de refinamento de malha combinado. É interessante notar como os deslocamentos convergem rapidamente enquanto as trações apresentam instabilidades para malhas muito refinadas para o método clássico.

[8] Com aproximações iniciais equivalentes o método clássico converge para casos que o método misto não converge.



(a) Perfis no procedimento de refinamento de malha combinado.



(b) Trações no procedimento de refinamento de malha combinado.

Figura 10: Procedimento de refinamento de malha combinado.

Fisicamente, os perfis possuem curvaturas ligeiramente superiores ao perfil parabólico que é usado como inicializador do método.

3.2.3 Complexidade e eficiência do código para o problema axissimétrico

Uma análise sobre a complexidade dessa porção do código foi feita pelo seu maior custo computacional em relação ao problema global. Apesar do problema global ser não linear e portanto incorrer em inúmeras soluções de sistemas lineares, o fato de possuir um domínio *1D* ($[0, L]$) garante que a cada passo o sistema linear relacionado seja pequeno em relação à dimensão do sistema linear do problema axissimétrico (possui um domínio *2D*).

Para analisar a complexidade e a eficiência do algoritmo, foi usada uma abordagem empírica com auxílio da *utility PerfLog* que gere a medição do tempo de execução de porções de código oportunamente selecionadas. Dessa maneira é possível identificar gargalos do algoritmo, isto é, as porções que apresentam maior complexidade numérica[9] e portanto que representam os pontos críticos no procedimento de refinamento de malha.

Como esperado, a parte crítica do código é a resolução dos sistemas lineares dos problemas implícitos sendo que o problema da elasticidade é responsável por ao menos 90% do custo computacional (proporção que aumenta com o aumento de graus de liberdade). Na tabela 1 são apresentados os tempos de execução da solução dos sistemas implícitos (difusão e elasticidade) e explícito (cálculo das tensões) para diversas malhas.

Tabela 1: Tempo de execução para diversas malhas

malha: $5n \times 10n$	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
tempo Total $[s]$:	0.0405	0.4891	2.6789	14.4052	61.5752	282.5269
% tempo resolução elasticidade:	88.18	96.44	97.21	97.06	93.46	88, 15
% tempo resolução difuão:	6.39	1.64	1.55	1.97	5.62	11.08
% tempo resolução tensões:	5.43	1.92	1.25	0.97	0.92	0.77

Com esses dados pode-se prever a complexidade do algoritmo: considerando que a relação de proporção entre tempo de execução t e parâmetro da malha n seja do tipo $t = kn^a$, a fração $\frac{\log(\frac{t_i}{t_{i-1}})}{\log(\frac{n_i}{n_{i-1}})}$ (sendo i o número da simulação) se aproxima de a com o aumento de n. Como $\frac{n_i}{n_{i-1}} = 2$ para todo i tem-se que $a = \log_2(\frac{t_i}{t_{i-1}})$. Usando a tabela 1 pode-se notar que o expoente é próximo a 2. Eventuais imprecisões são devidas à capacidade de processamento instantânea da máquina que é variável e ao fato que o modelo de proporção não é perfeito visto que pode conter outros termos menos importantes como na seguinte relação: $t = k_1 n^a + k_2 n^b + \ldots$.

Com o método descrito, o expoente que se distingue é o mais alto e mais importante. Dessa forma pode-se extrapolar o tempo requerido para simulações com mais graus de liberdade usando o parâmetro de malha n correspondente, o coeficiênte a obtido e uma

[9] A complexidade numérica é a relação de proporcionalidade entre uma dimensão característica do INPUT do problema e o tempo de execução. As constantes são desprezadas pois são dependentes da máquina e/ou arquitetura no entanto é interessante entender como o aumento de uma dimensão de INPUT altera o tempo de execução de um programa.

constante k que é normalmente dependente da máquina em questão mas que pode-se obter facilmente com uma das simulações já efetuadas.

Por exemplo, uma simulação com uma malha de 200x1000 (200000 elementos) seria equivalente (em número de elementos) a uma malha com $n = 64$. Usando um expoente ligeiramente maior que 2 para incluir efeitos dos termos não modelados e usando a constante de proporcionalidade obtida para $n = 32$ que vale $k = \frac{t_{n=32}}{n^{2.1}} \approx 0.195$, o tempo necessário seria de $t_{n=64} \approx 0.195 \times 64^{2.1} \approx 1211.22\ s \approx 20\ min$.

4 Resultados

Para a análise do problema, pretende-se sobrepor os efeitos da análise global, que derivam exclusivamente das correntes marítimas e do peso próprio, e os da análise axissimétrica, que derivam das pressões e das variações de temperatura. Nesse sentido, a partir da distribuição de esforço normal que se obtém na análise global, pode-se chegar, com algumas hipóteses, às tensões normais na seção transversal. Dessa maneira, pode-se sobrepor essa tensão normal ao estado de esforço que se obtém no problema axissimétrico de modo a entender o estado de esforço geral.

A rigor, a hipótese de sobreposição dos efeitos não é válida para problemas não lineares como no caso da análise global, no entanto com considerações físicas pode-se intuir que as não linearidades do problema global incidem muito pouco no problema *local* e, viceversa, o problema axissimétrico praticamente não contribui ao problema global.

4.1 Resultados do problema global

Nesta seção pretende-se discutir os resultados obtidos na análise global do problema estático da catenária exposto no capítulo 3.1. O estudo estático foi considerado pela preocupação predominante quanto aos esforços aos quais a estrutura deve resistir e não quanto à forma exata que estrutura a assume num cenário tempo-dependente. Além do mais, as possíveis variações de corrente em um cenário não estacionário são inúmeras e no trabalho presa-se por uma abordagem concisa e prática no procedimento de avaliação dos esforços resultantes e na compreensão da ordem de grandeza dos diversos fenômenos envolvidos, a qual não muda para eventuais cenários dinâmicos.

No projeto, duas abordagens diferentes foram consideradas conforme apresentado na seção 3.1.1 do capítulo 3.1 e ambas demonstraram coerência e eficiência.

Os dados físicos utilizados no problema, em geral, são expostos na tabela 2. As análises foram feitas variando esses parâmetros para o entendimento do comportamento da estrutura e para verificar se o modelo responde de acordo com o esperado. Em seguida considera-se a resposta do modelo à variação de um parâmetro por vez.

Os comentários e resultados foram feitos em relação a análises com parâmetros numéricos gerais apresentados na tabela 3 e usando o método de refinamento de malha cooperativo mencionado na seção 3.2.2 na página 67.

Tabela 2: Dados do problema global

Comprimento indeformado do cabo (L) - [m]:	1600
Profundidade (H) - [m]:	1500
Rigidez axial do cabo (EA) - [N]:	11.3×10^9
Peso imerso do cabo (q) - [$\frac{N}{m}$]:	-4012
Velocidade de corrente (V_c) - [$\frac{m}{s}$]:	1.5
Perfil da corrente ($f(y)$) - adimensional:	1
Coeficiente de arrasto (*drag*) cabo (c_d) - adimensional:	0.47
Densidade da água (ρ_a) [$\frac{kg}{m^3}$]:	1000
Diâmetro do riser (D) [m]:	0.6

Tabela 3: Dados numéricos do problema global

Parâmetros	Método clássico	Método híbrido
Número de elementos:	100	150
Ordem de aproximação:	2^a	2^a
Número de nós da malha:	201	201
Graus de liberdade implícitos:	401	603
Graus de liberdade explícitos:	201	0
Máximo nº de iterações (sistema linear):	1000	1000
Máximo nº de iterações (Newton):	150	150
Tolerância (sistema linear):	10^{-10}	10^{-10}
Tolerância (Newton) em norma L^2:	1	10

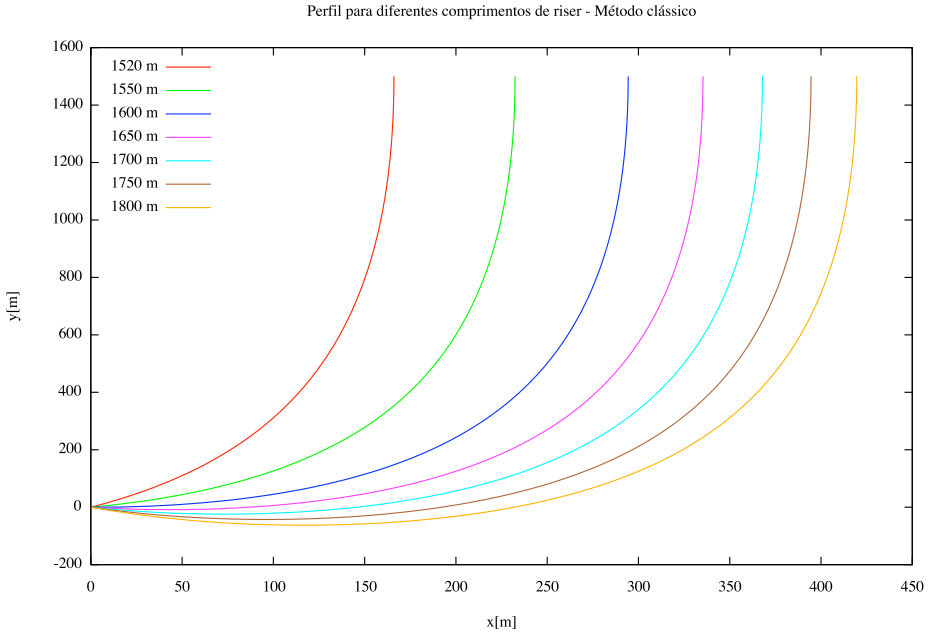
4.1.1 Variação do comprimento

A figura 11 na página 75 apresenta a resposta do modelo com os dados das tabelas 2 e 3 ao variar do comprimento da tubulação para os dois métodos utilizados.

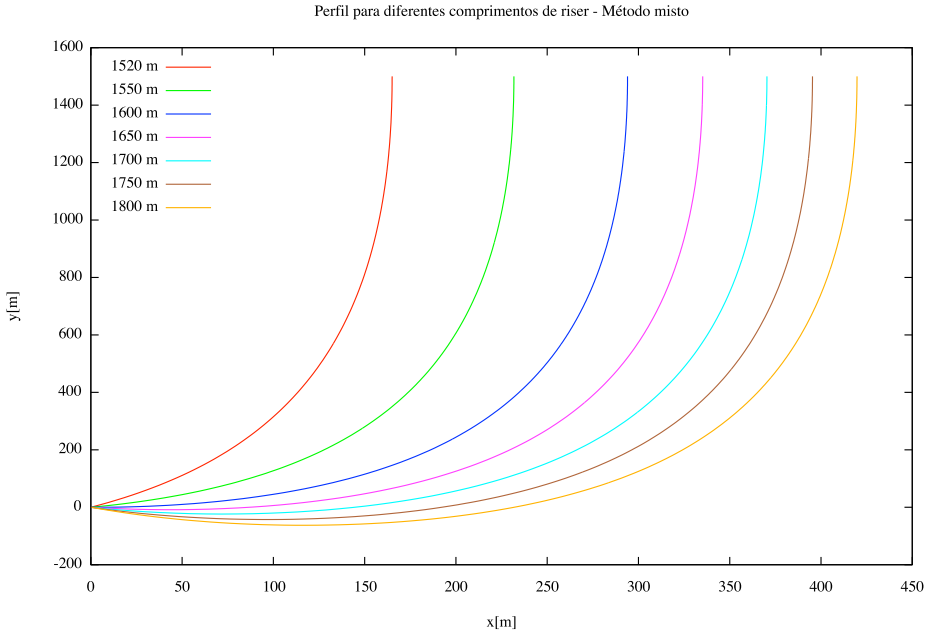
É possível perceber que ambos os métodos conseguem colher o efeito do aumento de comprimento. Os gráficos confirmam a validade dos modelos pois ambos os métodos convergem precisamente à mesma solução.

Usando essa abordagem é possível impor a condição de contorno de contato variando o comprimento da tubulação até obter aquele que prevê curvatura nula na origem (note como o comprimento de 1600 aproxima bem essa condição). Esse método iterativo foi sugerido por exemplo em (PESCE; MARTINS; CHAKRABARTI, 2005).

Quanto às trações resultantes, a figuras 12 na página 76 fornece a relação entre aumento de comprimento e trações. Note que com o aumento de comprimento, como para uma catenária pendurada, o cabo tende a assumir uma configuração que apresenta valores negativos para ordenada. Fisicamente, esses casos não podem ser verificados pois foi assumido que o o ponto ($y = 0$) corresponde ao fundo do oceano. Se, no entanto, $y = 0$ fosse uma bóia, a configuração teria sentido.



(a) Perfil do riser para diversos comprimentos com método clássico.

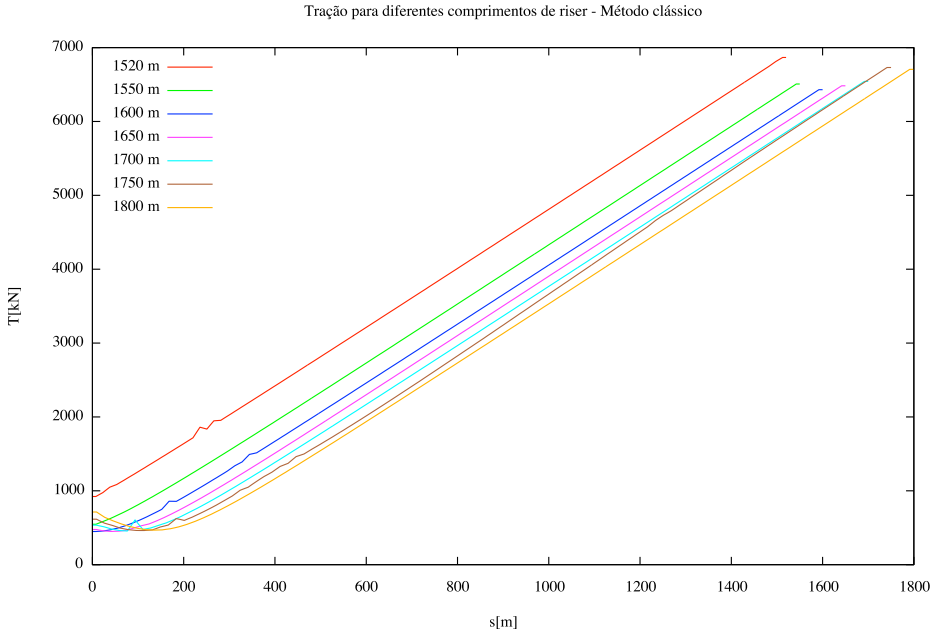


(b) Perfil do riser para diversos comprimentos com método mixed.

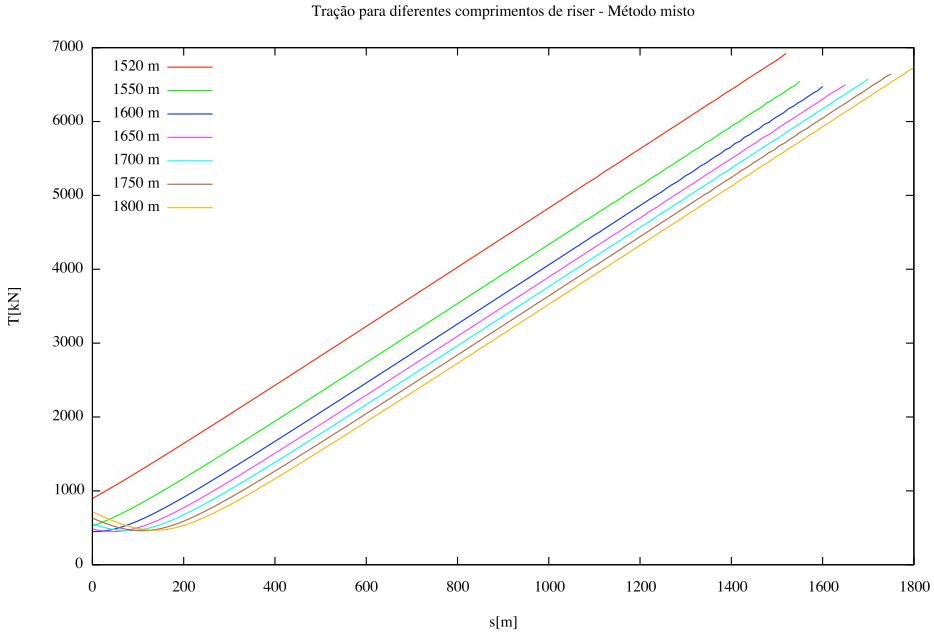
Figura 11: Perfil do riser para diversos comprimentos

Da análise do gráfico percebe-se que, como era de se esperar, o método misto é mais robusto e eficiente na previsão das trações. Para o método clássico o aumento do comprimento implica uma rápida flutuação das trações que provem do cálculo explícito das mesmas (no cálculo explícito flutuações locais podem ser facilmente amplificadas em operações de derivação).

Outra importante conclusão é que as condições de contorno do problema incidem



(a) Tração no riser para diversos comprimentos com método clássico.



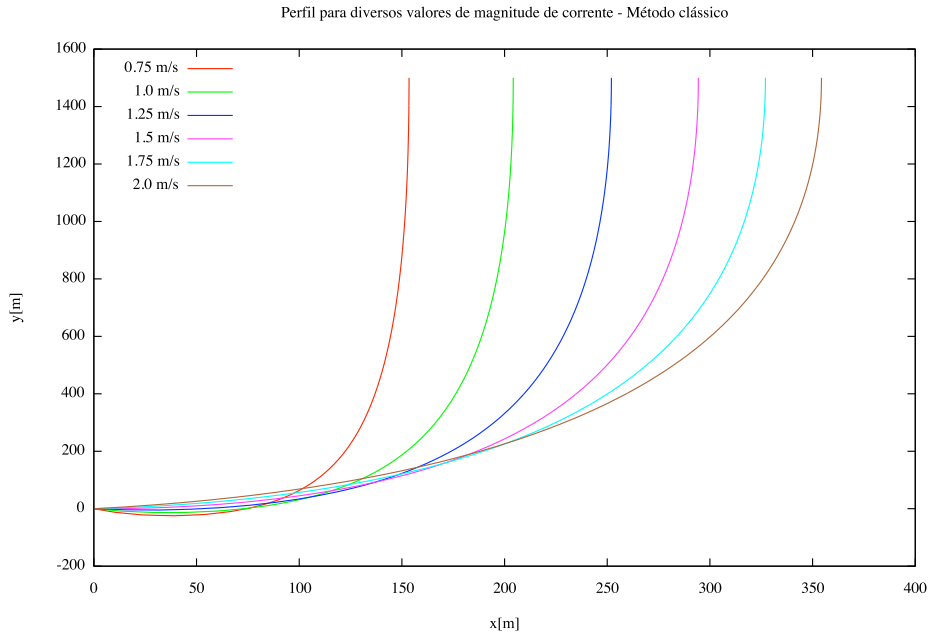
(b) Tração no riser para diversos comprimentos com método misto.

Figura 12: Tração no riser para diversos comprimentos.

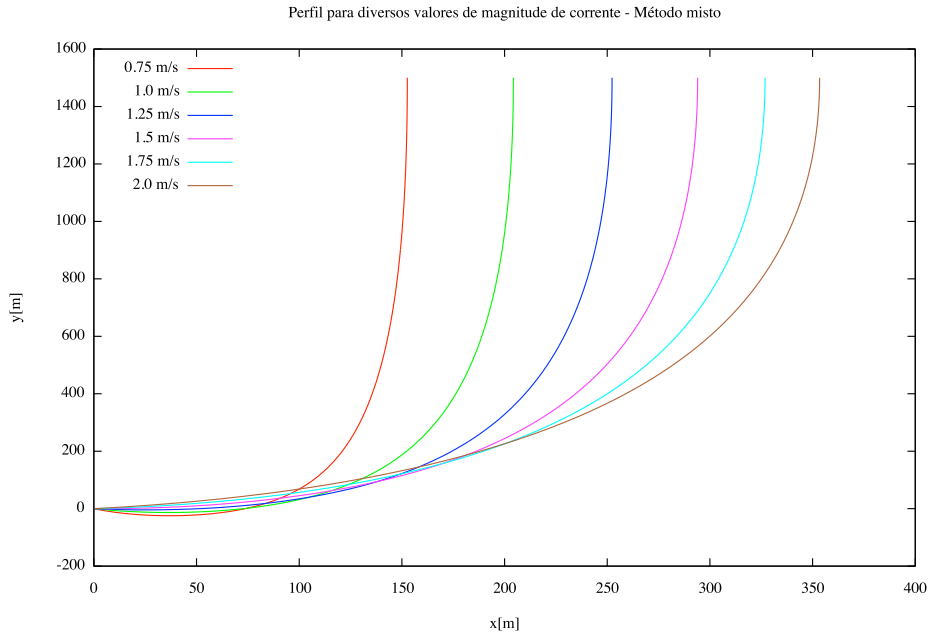
de modo decisivo nas trações resultantes já que, nesse intervalo de variação de comprimento, apesar da maior magnitude global do peso próprio da estrutura, um aumento no comprimento acarreta diminuição das trações globais do sistema. Um olhar macroscópico permite entender esse fato: um comprimento maior, permite à estrutura de se posicionar em maneira “mais paralela” à corrente por uma maior amplitude de profundidade gerando um carregamento equivalente, devido à corrente, menor.

4.1.2 Variação da magnitude de corrente

As figuras 13 na página 77 apresentam os resultados da resposta estrutural ao variar da magnitude de corrente. Os parâmetros das simulações são encontrados nas tabelas 2 e 3.



(a) Perfil do riser para diversos valores de magnitude de corrente com método clássico.



(b) Perfil do riser para diversos valores de magnitude de corrente com método misto.

Figura 13: Perfil do riser para diversas magnitudes de corrente.

Dos gráficos pode-se notar a mesma tendência vista no caso da variação de compri-

mentos onde os modelos concordam amplamente na previsão do resultado. Note que, para os casos expostos, uma corrente de $0.75\frac{m}{s}$ ainda não é suficientemente forte para garantir que todo o riser esteja acima de $y = 0$, situação que não é permitida fisicamente já que no modelo em questão o ponto $y = 0$ é o solo.

Quanto às trações pode-se verificar como o aumento de corrente tensiona progressivamente o cabo implicando em maiores diferenças de trações na parte inferior do *riser*. Note que uma corrente de $0.5\frac{m}{s}$ já é suficiente para que o cabo não esteja completamente tensionado, situação que o modelo desenvolvido não colhe mas sugere graças a uma análise assintótica[1]. Veja a figura 14 na página 79 à esse respeito.

Cabe ressaltar que o modelo misto é mais estável quanto às trações e menos estável quanto aos deslocamentos, situação que, além de ser prevista pela teoria, pode ser observada para todos os casos de estudo comparativo feitos.

4.1.3 Variação da magnitude do peso imerso

A figura 15 na página 80 ilustra o efeito do aumento de peso imerso da estrutura. Mais uma vez, os restantes parâmetros das simulações são encontrados nas tabelas 2 e 3.

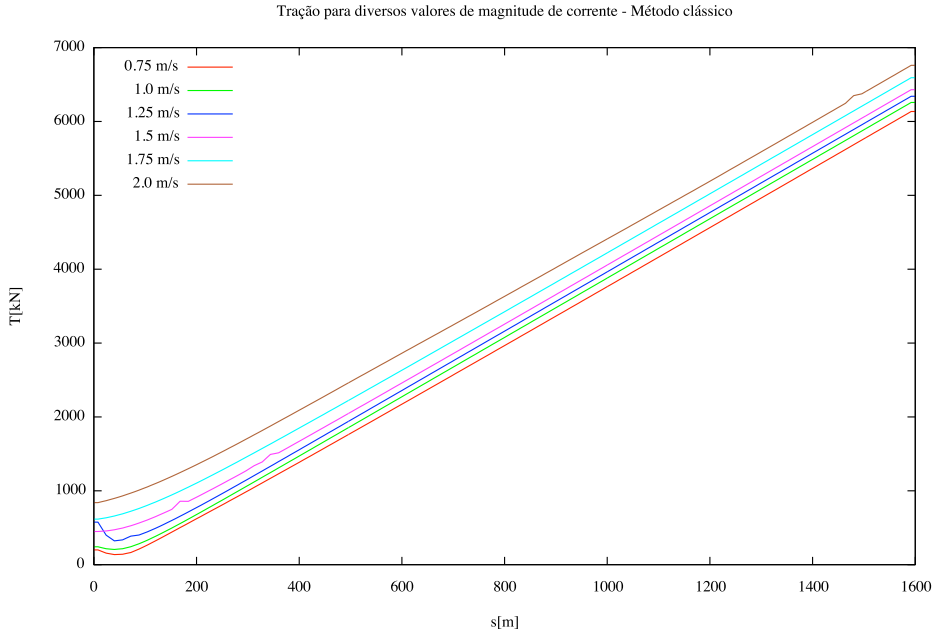
É possível perceber que o aumento de peso imerso incorre num abaixamento da porção inicial de cabo e em última análise na aproximação horizontal da FPU em relação ao TDP. Essa aproximação sugere que as deformações do cabo são de pequena magnitude já que não suficientes a compensar esse efeito. Nas figuras 16 na página 81 são expostos os resultados das trações ao variar dos valores de peso imerso do riser.

O aumento de peso próprio imerso incorre num aumento das trações que se verificam ao longo do cabo. Esse é um fenômeno côngruo com o que se intui porém pode-se notar que além disso o modelo prevê uma curva de trações que cresce mais rapidamente para pesos maiores. Essa última constatação pode ser intuída pelo fato que o peso imerso é distribuído e portanto cada seção subsequente deve suportar um peso sempre maior que é integrado ao longo do comprimento do cabo.

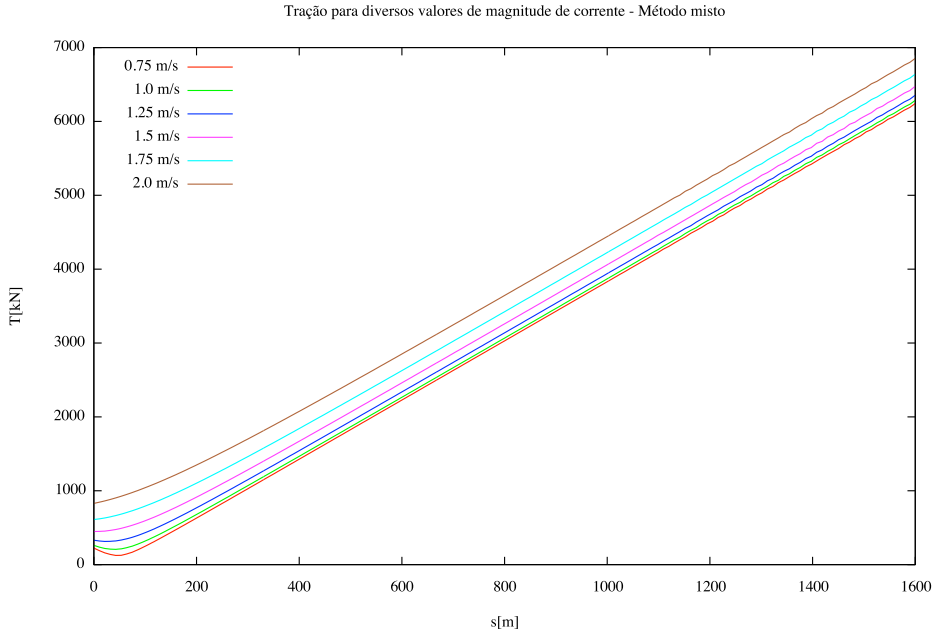
4.1.4 Importância relativa das variações

Além das análises já conduzidas também foi feito um estudo variando a rigidez axial do cabo. Essa etapa não foi digna de uma seção pois os efeitos macroscópicos da resposta são minoritários se comparados àqueles das variações dos restantes fatores. Essa relativa pouca importância tem como causa a pequena deformabilidade do cabo para

[1] Diminuindo gradualmente a corrente os valores de tração se aproximam do valor nulo na porção inicial da tubulação, diminuindo ulteriormente os cálculos se tornam instáveis e não se verifica convergência pois o modelo introduzido da catenária não prevê compressões.



(a) Tração no riser para diversos valores de magnitude de corrente com método clássico.

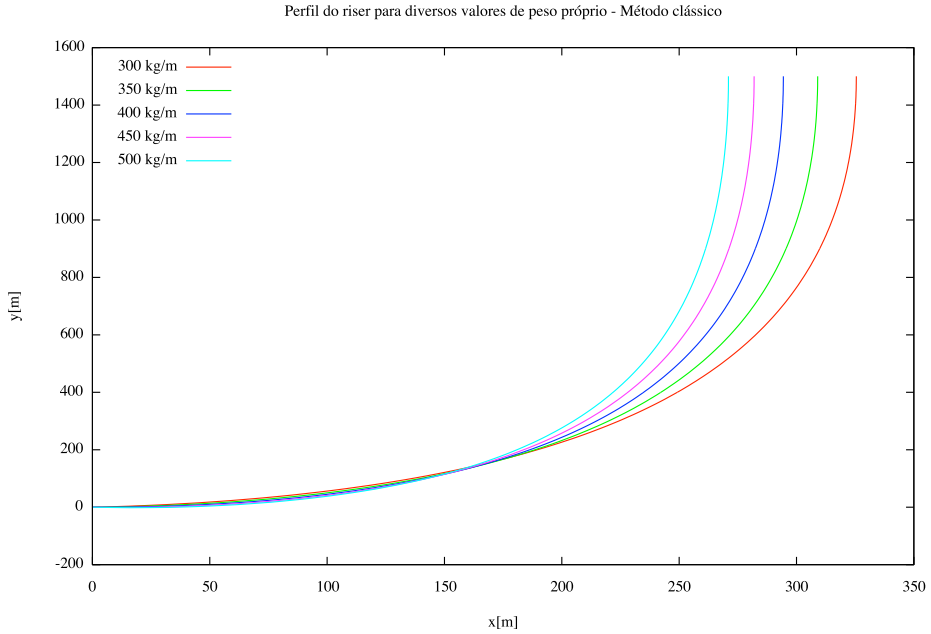


(b) Tração no riser para diversos valores de magnitude de corrente com método misto.

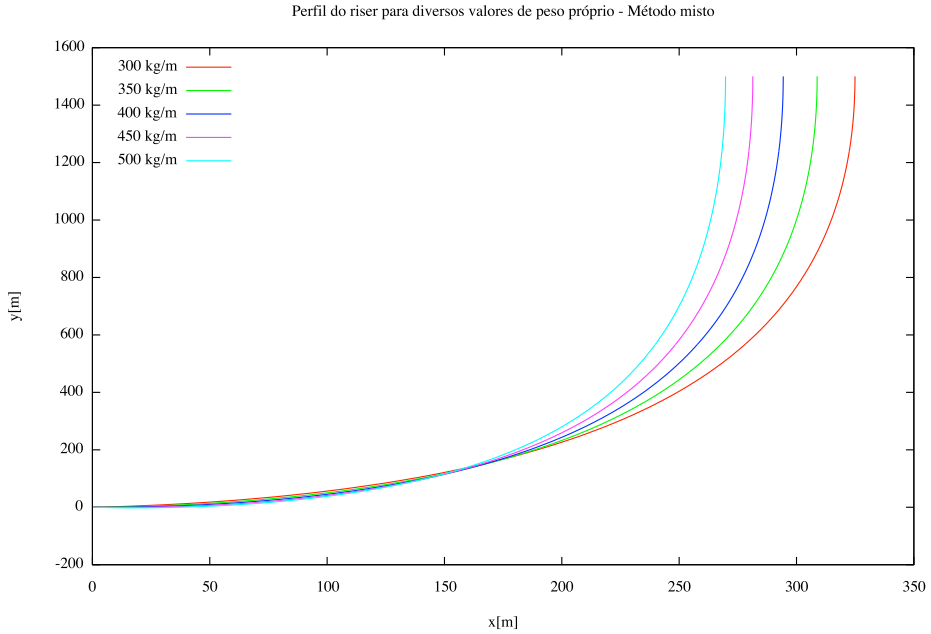
Figura 14: Tração no riser para diversas magnitudes de corrente.

valores de rigidez axial até 70% inferiores ao utilizado. Dessa maneira não ocorre uma redistribuição dos deslocamentos ou uma mudança nas trações relevantes na resposta.

Com as análises feitas foi possível notar a maior sensibilidade das trações resultantes em relação à variações de corrente na parte inferior do cabo e em relação à variações de peso imerso na parte superior. O fato que na parte inicial de cabo as trações possuem direção



(a) Perfil do riser para diversos valores de peso imerso com método clássico.



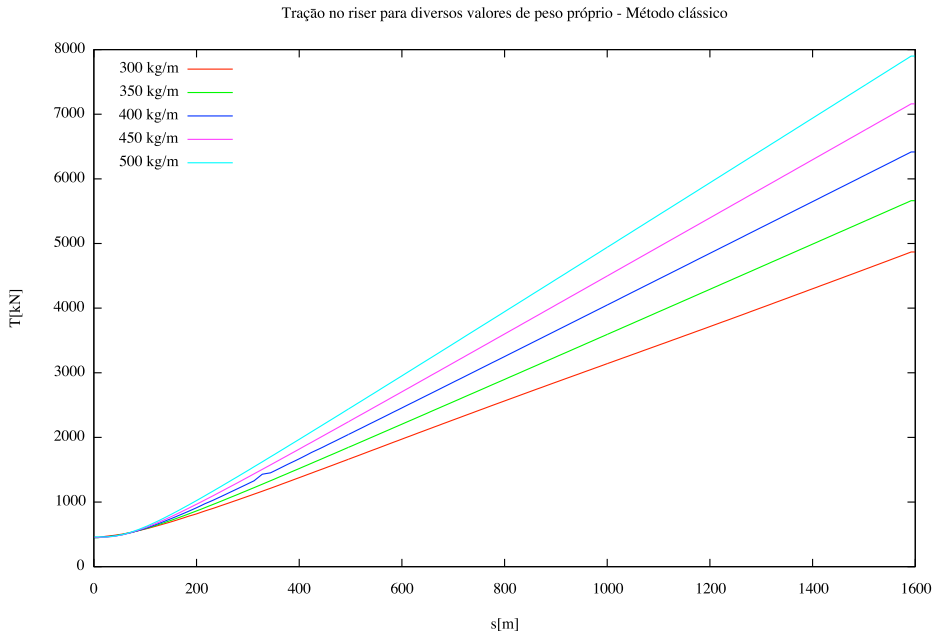
(b) Perfil do riser para diversos valores de peso imerso com método misto.

Figura 15: Perfil do riser para diversos valores de peso imerso.

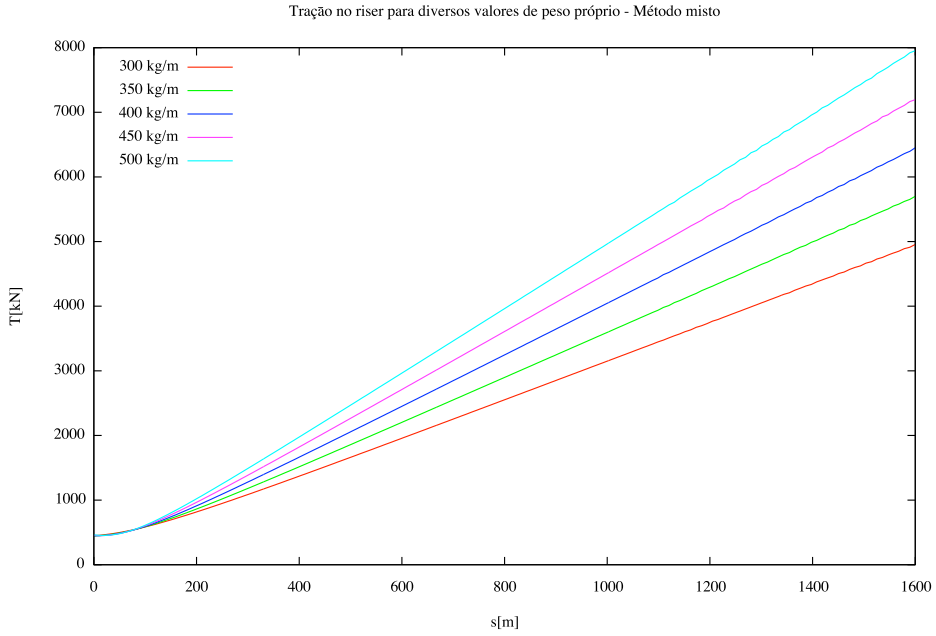
horizontal e na parte superior direção vertical ajuda a entender esse comportamento.

A variação de peso também contribui de modo relevante às distribuições de tração resultantes mudando não somente as magnitudes em questão mas também as relativas taxas de crescimento.

As variações de comprimento do cabo se demonstraram menos relevantes na distribuição de tração resultante porém essas variações se traduziram em mudanças



(a) Tração no riser para diversos valores de peso imerso com método clássico.



(b) Tração no riser para diversos valores de peso imerso com método misto.

Figura 16: Tração no riser para diversos valores de peso imerso.

substanciais no perfil resultante. A variação de comprimento, como dito, é uma etapa importante não somente na determinação da sensibilidade do modelo à esse parâmetro mas também na imposição iterativa da condição de contorno de tipo contato que ocorre no TDP.

4.2 Resultados do problema axissimétrico

Nesta seção serão descritos os resultados da parcela axissimétrica do problema, por isso uma descrição local foi feita de modo que somente uma porção pequena do comprimento da tubulação foi levada em consideração, porção colocada em condições de trabalho limítrofes, i.é. como se fosse a parte mais profunda da tubulação, submetida a pressões internas e externas equivalentes às experimentadas na parte inicial do riser (perto do TDP). As condições de contorno utilizadas para isolar os fenômenos axissimétricos dos restantes estudados na etapa global foram a de engasgamento na extremidade inferior e livre expansão axial na extremidade superior. Essa escolha foi feita para evitar que expansões/contrações axiais associadas às condições de contorno não condicionem o estado de esforço final[2]. Os códigos completos se encontram no apêndice A na página 1.

4.2.1 Resultados do problema de difusão de temperatura

Como dito anteriormente, a difusão de temperatura pode ser modelada como um problema *1D* simples pois, com a hipótese de isolamento perfeito, o único "carregamento" que é dependente de z é a temperatura externa. Essa temperatura apresenta gradientes na direção z não superiores a $0.1\frac{^\circ C}{m}$ como pode ser observado na figura 8. Comparando esses valores com os gradientes experimentados na direção radial (que são da ordem de $500\frac{^\circ C}{m}$) pode-se concluir a baixa relevância desse efeito, que portanto pode ser desprezado.

As informações usadas na simulação se encontram na tabela 4:

Tabela 4: Dados de difusão térmica

Raio interno (R_i) - $[m]$:	0.12
Raio externo (R_e) - $[m]$:	0.3
Espessura tubo interno (t_i) - $[m]$:	0.02
Espessura tubo externo (t_e) - $[m]$:	0.018
Comprimento tubulação (L) - $[m]$:	5
Coeficiente de condução térmica (k) tubos interno e externo - $\left[\frac{W}{m.K}\right]$:	50
Coeficiente de condução térmica (k) isolante - $\left[\frac{W}{m.K}\right]$:	0.16
Temperatura interna $[^\circ C]$:	95
Temperatura externa 1D $[^\circ C]$:	10
Temperatura externa 2D e $z \in [0, L]$ em metros $[^\circ C]$:	10+0.1z

A simulação *1D* fornece a distribuição de temperatura radial vista na figura 17 na página 83, simulação conduzida com uma malha de 200 elementos e aproximação de segunda ordem.

[2] Se por exemplo as duas extremidades fossem consideradas engastadas, eventuais expansões/contrações seriam impedidas por essas condições podendo gerar contributos de tensão axial σ_z exagerados e não relacionados aos esforços axissimétricos mas sim às próprias condições de contorno.

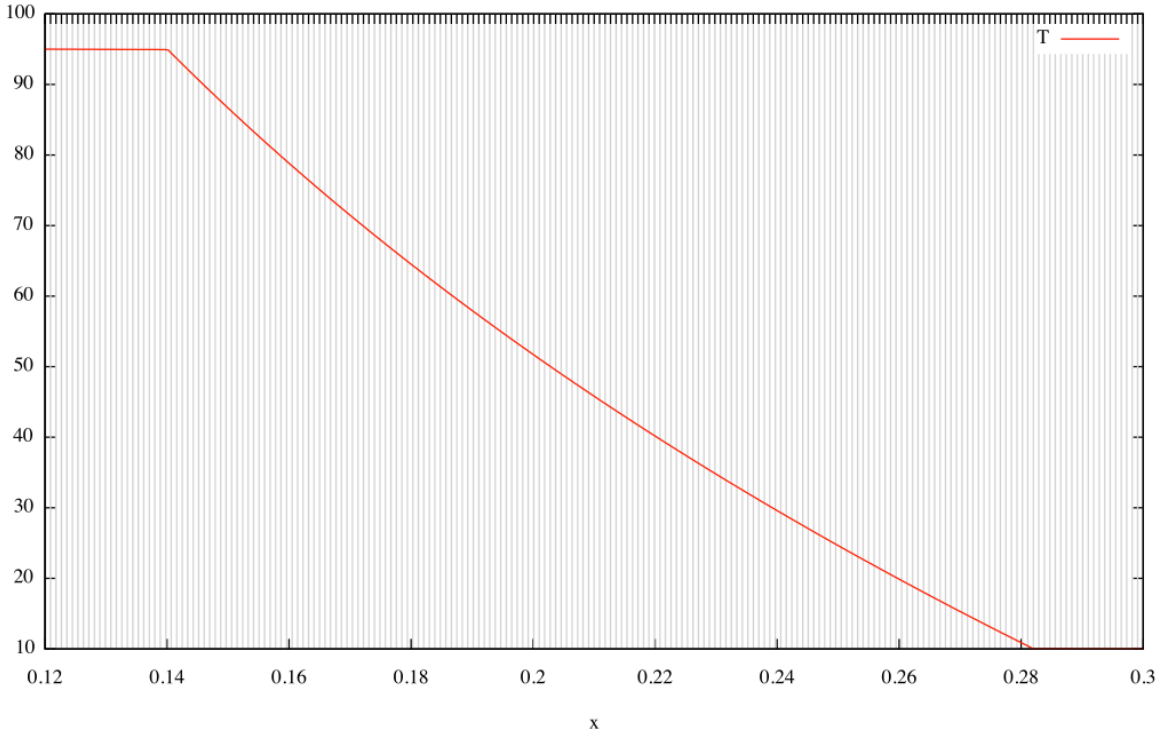


Figura 17: Distribuição de temperatura radial: malha de 200 elementos e aproximação de segunda ordem.

A evolução logarítmica concorda com o previsto pela solução analítica, veja (INCROPERA et al., 2012). Por escrúpulo é exposto o resultado da simulação em *2D* na figura 18 que confirma a validade do modelo *1D* dada a pequena variação da temperatura externa na direção axial.

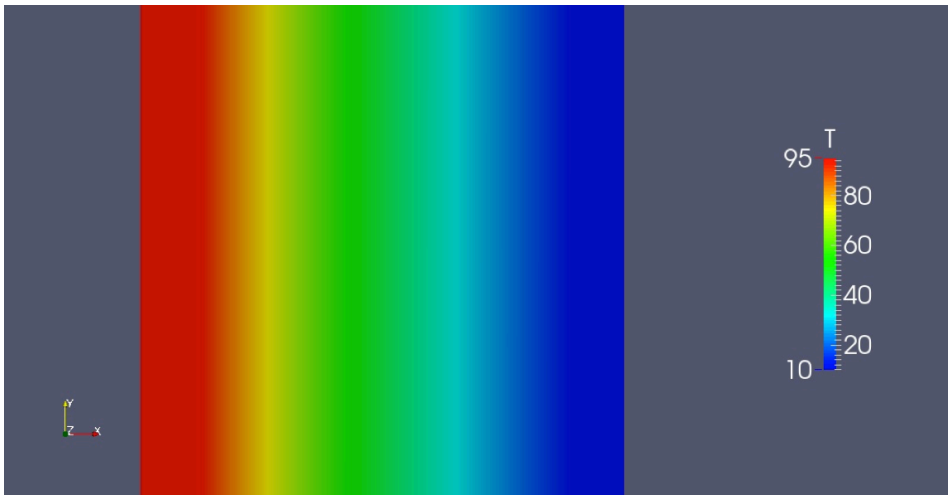


Figura 18: Distribuição de temperatura 2D: malha de 150x500 elementos e aproximação de segunda ordem.

Note como o perfil em direção radial segue o mesmo esquema da distribuição em *1D*.

4.2.2 Análise estrutural em ausência do efeito térmico

Na análise estrutural em ausência do efeito térmico, a formulação variacional é dada pela equação 3.38 na página 63 sem os termos a direita correspondentes à variação de temperatura e aos carregamentos de volume provenientes do peso imerso. A tabela 5 apresenta os principais parâmetros para o cálculo utilizados, esses são frutos de uma pesquisa dos valores realmente encontrados nas condições de trabalho da estrutura. A pressão externa é dada pela pressão hidrostática da água partindo de uma profundidade de 1500 metros e chegando a 1495 metros. A pressão interna é aproximadamente a pressão necessária, nos 5 primeiros metros, para a propulsão a $2.5\frac{m}{s}$ do óleo cru partindo de uma profundidade de 1500 metros e chegando à superfície da água num eventual cenário de inicio de operação[3]. O gradiente de pressão considera o efeito hidrostático e de perda de carga no fluxo.

A extremidade inferior da tubulação foi considerada engastada e a extremidade superior livre para translações axiais, dessa maneira é possível isolar os efeitos que decorrem exclusivamente dos fenômenos axissimétricos.

Tabela 5: Dados para o cálculo estrutural

Módulo de Young (E) tubos interno e externo - $[GPa]$:	200
Módulo de Young (E) material isolante - $[GPa]$:	5
Coeficiente de Poisson (ν) tubos interno e externo - adimensional:	0.3
Coeficiente de Poisson (ν) material isolante - adimensional:	0.4
Pressão interna (p_i) $[MPa]$ e $z \in [0, L]$ em metros:	52 - 0.0104z
Pressão externa (p_e) $[MPa]$ e $z \in [0, L]$ em metros:	15 - 0.01z
Densidade (ρ_m) tubos interno e externo - $\left[\frac{kg}{m^3}\right]$:	8000
Densidade (ρ_m) material isolante - $\left[\frac{kg}{m^3}\right]$:	1300
Densidade (ρ_f) água - $\left[\frac{kg}{m^3}\right]$:	1000
Aceleração da gravidade (g) $\left[\frac{m}{s^2}\right]$:	10

Na fase numérica pode-se notar que a malha *2D* é muito mais sensível a refinamento na direção radial que em direção axial. Essa observação é coerente com as expectativas vistos os grandes gradientes de esforços e de propriedades dos materiais experimentados na direção radial em relação a direção axial.

As imagens reproduzem a solução do problema para uma malha uniformemente distribuída pelo domínio que possui características conforme apresentado na tabela 6[4].

[3] Para maiores informações veja (JAN et al., 2010)

[4] Essas características se referem somente à parte implícita do problema, i.é. o problema nas deflexões que deriva da formulação variacional através do método de Galerkin. As informações referentes à parte explícita (i.é. cálculo das tensões) são omitidas já que essa etapa é muito menos onerosa.

Tabela 6: Dados da malha para a simulação sem efeitos térmicos

Número de nós:	641601
Número de elementos:	160000
Ordem de aproximação:	2
Tipo de elemento:	quadrilátero
Número de graus de liberdade (dimensão do sistema linear):	1283202
Número de graus de liberdade na fronteira de Dirichlet:	1602

As deflexões na condição de trabalho imposta são côngruas com a hipótese de pequenas deformações. A componente radial do gradiente de deslocamento radial ($\frac{\partial u_r}{\partial r}$) é superior à componente axial do gradiente de deslocamento axial ($\frac{\partial u_z}{\partial z}$) por todo o domínio e ambas são muito superiores à componente axial do gradiente de deslocamento radial ($\frac{\partial u_r}{\partial z}$) e à componente radial do gradiente de deslocamento axial ($\frac{\partial u_z}{\partial r}$). Além disso, a deformação axial do cabo é praticamente constante radialmente e axialmente, com exceção à extremidade engastada onde se verificam variações axiais de deformação. Como mencionado, os efeitos do engastamento são confinados às proximidades da extremidade sendo que não geram grandes gradientes em relação àqueles já presentes em zonas distantes. Veja a figura 19 na página 86 e a figura 20 na página 87.

As unidades de medida, se omitidas, são expostas em SI.

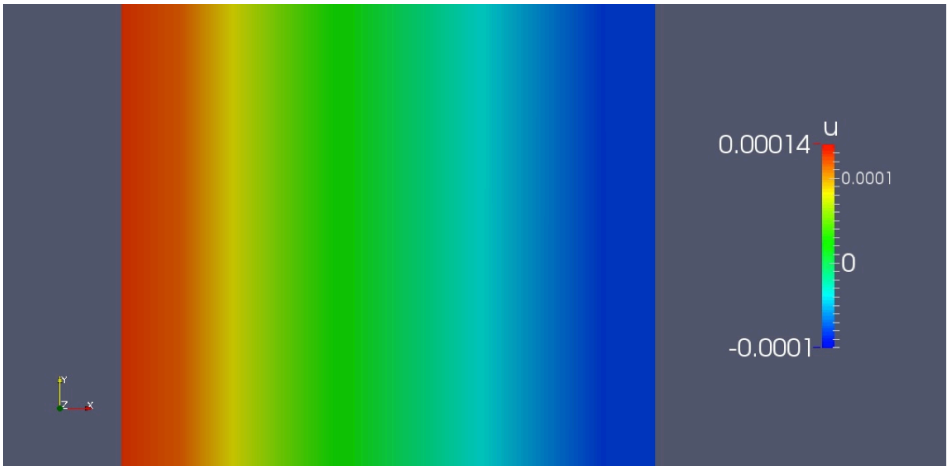
Mais interessante que o campo de deflexões para o estudo estrutural é o estado de tensões, que, dada a hipótese de axissimetria, é descrito completamente pelas componentes $[\sigma_r, \sigma_z, \tau_{rz}, \sigma_\theta]$. Essas podem ser obtidas facilmente através da seguinte equação constitutiva:

$$\begin{bmatrix} \sigma_r \\ \sigma_z \\ \tau_{rz} \\ \sigma_\theta \end{bmatrix} = \begin{bmatrix} \lambda+2\mu & \lambda & 0 & \lambda \\ \lambda & \lambda+2\mu & 0 & \lambda \\ 0 & 0 & \mu & 0 \\ \lambda & \lambda & 0 & \lambda+2\mu \end{bmatrix} \begin{bmatrix} \epsilon_r \\ \epsilon_z \\ \gamma_{rz} \\ \epsilon_\theta \end{bmatrix} \tag{4.1}$$

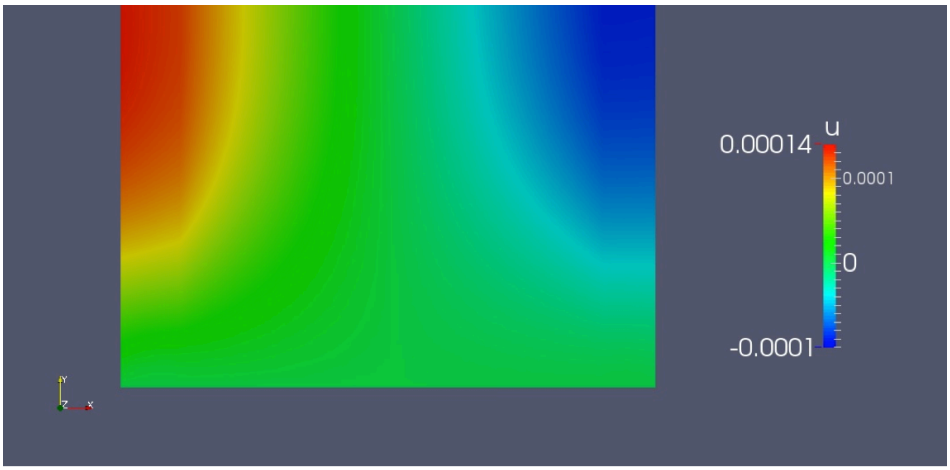
Sendo o vetor das deformações relacionado às deflexões conforme a equação 3.37 na página 63.

Da figura 21 na página 87 pode-se observar, como esperado, que a tensão σ_r é de compressão em todo o domínio chegando a uma amplitude máxima ($\approx$ -52 MPa) nas vizinhanças do raio interno. Essas tensões se abaixam, em módulo, consideravelmente ao longo da espessura do tubo interno e aumentam somente em correspondência ao tubo externo. Nessa figura fica evidente o efeito de proteção do material isolante por parte dos tubos metálicos do efeito das pressões.

A componente τ_{rz} apresenta magnitude significativamente inferior às outras tensões e não será discutida dada sua incidência marginal no estado de tensões final.



(a) Deflexão radial u_r em ausência do efeito térmico.



(b) Deflexão radial u_r em ausência do efeito térmico na extremidade.

Figura 19: Deflexão radial u_r em ausência do efeito térmico.

A tensão σ_z possui magnitude inferior a σ_r em todo o domínio e do perfil observado na figura 22 na página 88 e de uma análise detalhada no campo de deformações se nota que os contributos dominantes para essa tensão provém das deformações tangencial ϵ_θ e e axial ϵ_z.

Analogamente a σ_z, σ_θ sofre influência dominante do carregamento que deriva das pressões. Sendo a pressão interna superior, as amplitudes maiores se constatam nas vizinhanças do raio interno. Quanto às deformações, do campo de deflexões (figura 19 na página 86) se conclui que os contributos de ϵ_θ para os esforços se concentram nas proximidades dos raios interno e externo, com a parte interna que apresenta contributos positivos e externa negativos. Essa deformação apresenta valores almeno uma ordem de grandeza superiores à σ_r na região do raio interno e nessa zona se verificam as maiores amplitudes de tensão tangencial. Veja a figura 23 na página 88.

É possível resumir a relação entre as tensões e deformações afirmando que a deformação ϵ_r é a principal responsável pelo comportamento da componente σ_r enquanto

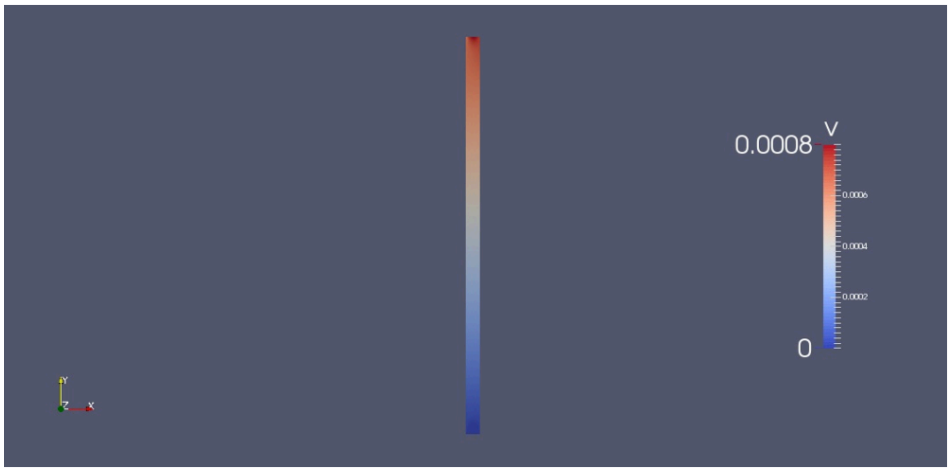


Figura 20: Deflexão axial u_z em ausência do efeito térmico.

Figura 21: Tensão radial σ_r em ausência do efeito térmico.

para σ_θ e σ_z as deformações ϵ_θ e ϵ_z dominam. Das equações constitutivas se nota que para cada uma das tensões mencionadas, a constante de proporcionalidade em relação ao correspondente contributo de deformação dominante, é superior às restantes explicando em parte a afirmação anterior. Contudo esse fato informa o analista que os valores das deformações ϵ_r, ϵ_θ e ϵ_z são equiparáveis porém superiores aos valores de τ_{rz}.

Por último, toma-se a *tensão de Von Mises* como parâmetro do estado de tensão local, que para o caso se escreve:

$$\sigma_{vm} = \sqrt{\frac{1}{2}[(\sigma_r - \sigma_\theta)^2 + (\sigma_r - \sigma_z)^2 + (\sigma_z - \sigma_\theta)^2 + 6\tau_{rz}^2]} \tag{4.2}$$

Essa tensão mede o estado de tensão distorsivo e é usada como critério de resistência para materiais dúcteis: o estado é considerado seguro se $\sigma_{vm} \leq \sigma_y$, sendo σ_y a tensão de escoamento (*yielding tension*).

A figura 24 na página 89 ilustra o estado de esforço usando a tensão de Von Mises e, como esperado, a zona crítica é a vizinhança do raio interno onde pode-se verificar

Figura 22: Tensão axial σ_z em ausência do efeito térmico.

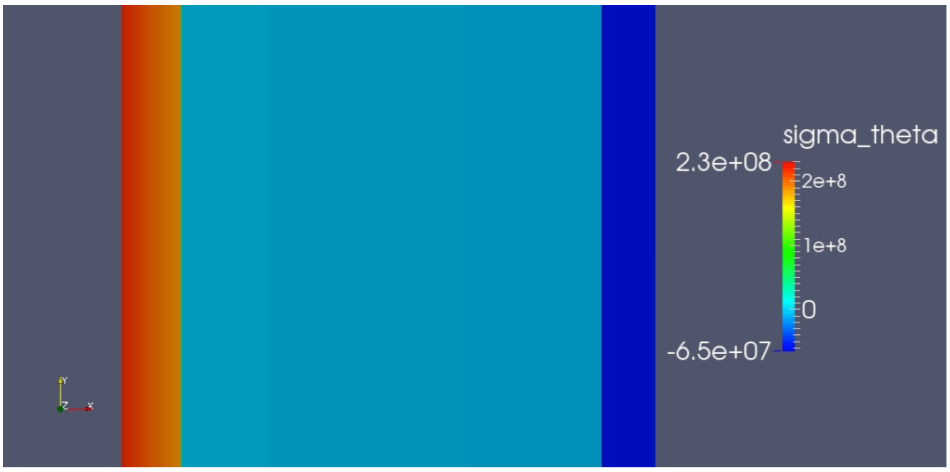


Figura 23: Tensão tangencial σ_θ em ausência do efeito térmico.

valores da ordem de $\sigma_{vm} \approx 240$ MPa.

4.2.3 Análise estrutural em presença do efeito térmico

O efeito térmico entra como forçante no sistema da elasticidade de acordo com a equação 3.38 na página 63. Assim deve-se resolver o problema de difusão de temperatura apresentado na seção 4.2.1 para posteriormente resolver o sistema da elasticidade e finalmente, com o campo de deflexão conhecido, proceder ao cálculo das deformações e das tensões, sendo que a relação constitutiva sofre uma alteração proveniente das deformações anelásticas. A nova relação entre deformações e tensões é dada conforme a equação 4.3.

$$\begin{bmatrix} \sigma_r \\ \sigma_z \\ \tau_{rz} \\ \sigma_\theta \end{bmatrix} = \begin{bmatrix} \lambda+2\mu & \lambda & 0 & \lambda \\ \lambda & \lambda+2\mu & 0 & \lambda \\ 0 & 0 & \mu & 0 \\ \lambda & \lambda & 0 & \lambda+2\mu \end{bmatrix} \begin{bmatrix} \epsilon_r - \alpha\Delta T \\ \epsilon_z - \alpha\Delta T \\ \gamma_{rz} \\ \epsilon_\theta - \alpha\Delta T \end{bmatrix} \tag{4.3}$$

Para esse problema o único dado adicional em relação àqueles já apresentados nas

Figura 24: Tensão equivalente de Von Mises σ_{vm} em ausência do efeito térmico.

tabelas 4 e 5 é o coeficiente de expansão térmica α que é exposto na tabela 7.

Tabela 7: Coeficiente de expansão térmica

Coeficiente de expansão térmica (α_m) tubos interno e externo - $\left[\frac{1}{K}\right]$:	13×10^{-6}
Coeficiente de expansão térmica (α_i) isolante - $\left[\frac{1}{K}\right]$:	50×10^{-6}

Em relação aos resultados obtidos na simulação em ausência de efeitos térmicos, esse caso apresenta muitas diferenças, confirmando a relevância dos efeitos térmicos: na figura 25, por exemplo, se observa que a deflexão radial experimenta valores até duas vezes superiores em relação ao problema sem efeitos térmicos (figura 19 na pagina 86). Quanto às tensões, a presença do efeito térmico é suficiente a mudar consideravelmente o

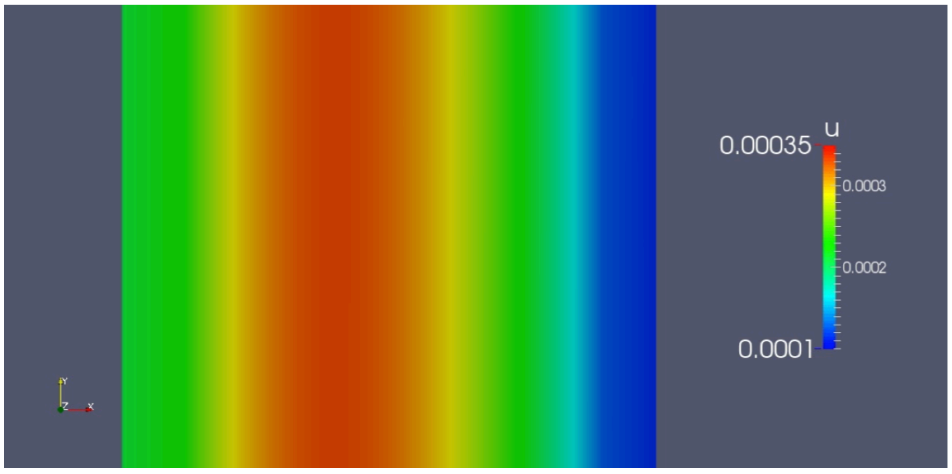
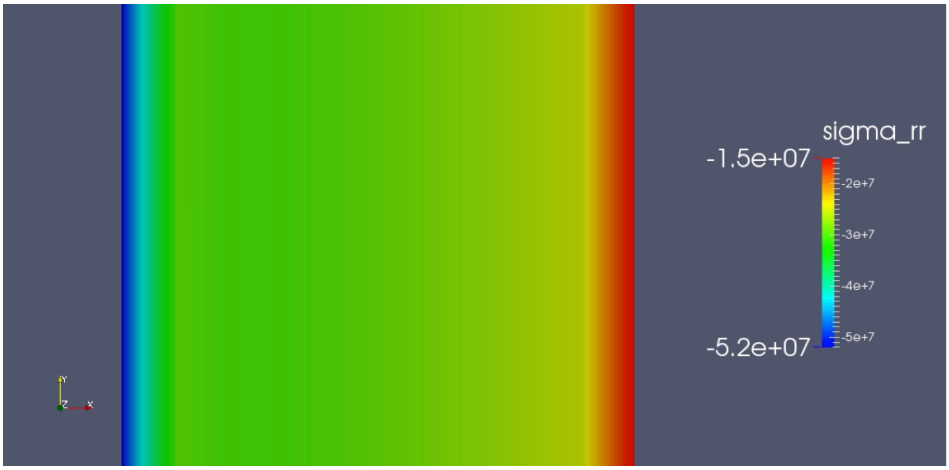


Figura 25: Deflexão radial u_r em presença do efeito térmico

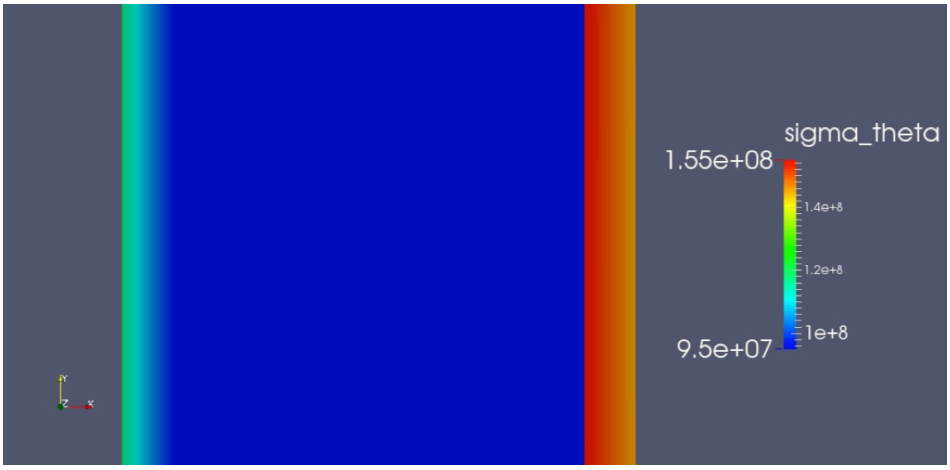
comportamento da tensão radial σ_r principalmente para a região do isolante onde o maior coeficiente de expansão térmica gera contributos de deformação anelástica consideráveis.

Ainda para a tensão radial σ_r, o tubo interno apresenta um comportamento similar ao caso não térmico onde a parcela dominante é dada pela deformação radial ϵ_r, que apesar de ser menor do que a deformação tangencial ϵ_θ, não é amortecida pela contributo anelástico quanto ϵ_θ. Além disso, pode-se observar que na porção correspondente ao tubo externo, a tensão radial é maior em presença do efeito térmico já que a parcela anelástica entra de maneira construtiva nas tensões de compressão.

Para a tensão tangencial σ_θ, o tubo interno apresenta magnitudes menores principalmente pelas deformações anelásticas que se opõem à deformação tangencial que, apesar dessa oposição, continua sendo dominante em σ_θ. Para o tubo esterno, os efeito anelásticos intervêm em maneira construtiva com ϵ_θ que, diversamente do caso atérmico, é ainda positivo para essa região. Essa interação resulta numa importante inversão de comportamento entre os casos de presença e ausência dos efeitos térmicos pois não somente muda a zona mais solicitada da estrutura mas também o tipo de solicitação (com os efeitos térmicos o tubo externo passa a ser solicitado em tração na direção tangencial). Veja a figura 26 na página 90.



(a) Tensão radial σ_r em presença do efeito térmico.



(b) Tensão tangencial σ_θ em presença do efeito térmico.

Figura 26: Tensão radial σ_r e tangencial σ_θ em presença do efeito térmico.

A tensão σ_z também verifica um relevante aumento de valores máximos quando é incluso o efeito térmico: os fenômenos anelásticos são suficientes a inverter o tipo de esforço solicitante na parte interna da tubulação e a aumentar consideravelmente as amplitudes do estado de trações na parte externa. A deformação anelástica contrasta (e vence) todas as outras na zona interna enquanto que para a parte externa se associa de maneira construtiva a ϵ_z e a ϵ_θ. Nesse caso, σ_z apresenta valores da ordem de 175 MPa (tração) para a parte externa e 80 MPa (compressão) para a parte interna. Nesse âmbito veja a figura 27 na página 91.

Mais uma vez os esforços distorsivos τ_{rz} são desprezíveis em relação aos restantes.

Figura 27: Tensão axial σ_z em presença do efeito térmico.

Finalmente, analisando a tensão equivalente de *Von Mises* (definida na equação 4.2 na página 87) pode-se notar a importância do efeito térmico já que essa assume valores ligeiramente inferiores para o tubo interno porém muito superiores para o tubo externo. A zona de valores máximos passa a ser no tubo externo na vizinhança entre o isolante e tubo (valores da ordem de 200 MPa). Nesse contexto veja a figura 28 na página 91.

Figura 28: Tensão equivalente de Von Mises σ_{vm} em presença do efeito térmico.

5 Discussão

A abordagem inédita utilizada no presente estudo permite uma abrangência de grande parte dos fenômenos aos quais é submetida a estrutura em questão. Na literatura, muitas vezes uma grande parcela dos fenômenos não é tratada e as investigações correm o risco de resultar errôneas ou incompletas, ainda mais quando a interação dos efeitos, como neste caso, é complexa e repleta de não-linearidades.

Os modelos considerados podem ser de utilizo para uma grande gama de configurações e situações de carregamento variando os diversos parâmetros definidos como profundidade, pressões, geometria, propriedades dos materiais, condições externas e etc. O trabalho foi conduzido em modo a cobrir em maneira horizontal as potencialidades dos códigos e dos modelos evidenciando os principais pontos críticos para a geometria em questão.

As hipóteses iniciais foram validadas a posteriori, algumas delas podem ser elencadas: pequenas deformações para problema local, prestações numéricas dos métodos clássico e misto, pequenas curvaturas no problema global dentre outras.

O método cooperativo desenvolvido responde a uma grande dificuldade que deriva das não-linearidades do problema e pode ser estendido a inúmeras outras situações, inclusive fora do âmbito do cálculo estrutural. A sua grande potencialidade está no fato de conseguir combinar as vantagens de dois métodos consagrados como a melhor eficiência e estabilidade do método clássico e a maior precisão do método misto. Problemas análogos que apresentem grande dependência de escala das não-linearidades, necessidade de maior precisão no cálculo das trações (ou de outra variável explícita para problemas que não sejam do cálculo estrutural) e/ou dificuldades na inicialização de um método para solução de sistemas não lineares podem potencialmente fruir da combinação de métodos utilizada no texto.

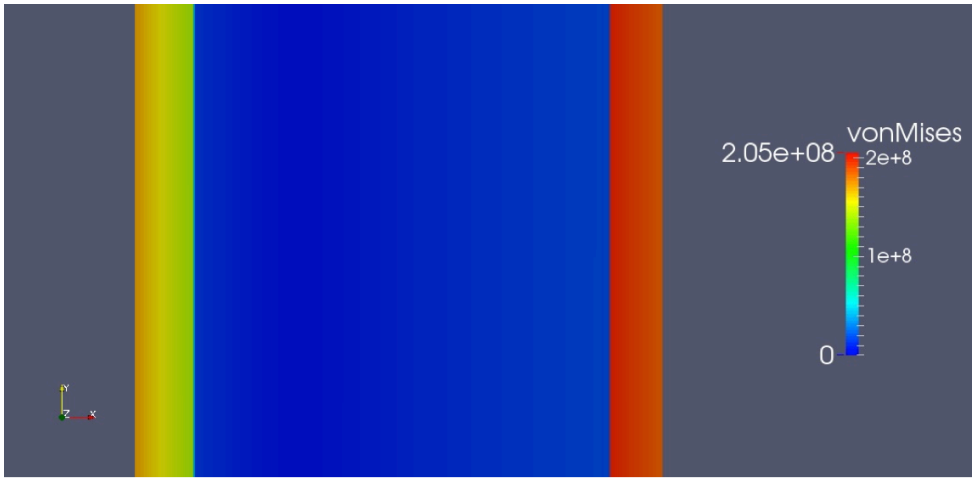
Por fim, apesar de não ter tratado fenômenos dinâmicos por questões de espaço e de objetivos, o *design* dos códigos foi concebido para permitir, com mínimos esforços a extensão da análise a uma situação tempo-dependente. Cabe ressaltar que a escolha operativa de não tratar os fenômenos dinâmicos deriva da menor relevância desses em relação aos esforços estáticos tratados quanto à analise de resistência. Estudos futuros, preocupados com o fenômeno da fadiga podem partir do presente trabalho evitando muitos dos problemas intrínsecos já tratados. Num cenário dinâmico, apesar da introdução de novas não-linearidades de carregamento de corrente[1], provavelmente o problema apresentaria

[1] Os carregamentos de corrente teriam de ser tratados com a velocidade relativa entre a corrente externa e a velocidade local da estrutura.

maior estabilidade numérica pela introdução da matriz de massa. Porém novos problemas como de escalas de tempo seriam introduzidos abrindo novas discussões.

Voltando ao caso estático, a sobreposição dos fenômenos globais e locais foi feita considerando que toda a tração que deriva dos carregamentos de corrente e do peso próprio imerso da estrutura é suportada somente pela porção metálica da seção transversal e que as tensões equivalentes são uniformemente distribuídas entre tubo interno e externo. As tensões na parte inferior do cabo sob tais hipóteses são por volta de 20 MPa. A sobreposição do fenômeno global pode ser vista na figura 29 na página 94 para os casos de presença e ausência de efeitos anelásticos.

(a) Tensão equivalente de Von Mises σ_{vm} em presença do efeito térmico.



(b) Tensão equivalente de Von Mises σ_{vm} em ausência do efeito térmico.

Figura 29: Sobreposição dos efeitos globais.

Esse último resultado traz consigo algumas discussões relevantes: note como, em relação aos estados sem os fenômenos globais nas figuras 28 na página 91 e 24 na página 89, os efeitos globais amplificam as diferenças entre os casos de presença e ausência de efeitos anelásticos pois no caso do tubo interno, diminuem o estado de esforço para o caso com efeito térmico e aumentam para o caso sem efeito térmico. Já no tubo esterno, para

ambos os casos aumentam os estados de tensões porém com uma grande diferença de magnitude dos valores.

É interessante observar como o efeito térmico "transfere" carregamento para o tubo externo. Apesar de não ter sido discutido o efeito da flexão, sabe-se que nas proximidades do TDP sua participação se torna relevante e esse fenômeno torna ainda mais delicada a mudança do ponto crítico ao tubo externo pois a flexão implica invariavelmente em valores de tensão equivalentes muito superiores nas partes mais distantes do baricentro.

Em futuros estudos, além dos efeitos dinâmicos mencionados seria de grande interesse a inclusão dos efeitos de flexão no modelo para delimitar o quando nos SCR (*steel catenary risers*) a resistência à flexão é efetivamente desprezível. Algumas boas referências para essa etapa se encontram em (PESCE; MARTINS; CHAKRABARTI, 2005) e em (ARANHA; MARTINS; PESCE, 1997).

6 Conclusão

No presente estudo, grande parte dos fenômenos de carregamento aos quais é submetido um *riser* em regime de operação foram tratados do ponto de vista da resistência estrutural. Uma abordagem não convencional foi usada no desacoplamento do estudo onde sob oportunas hipóteses foi possível decompor o problema em uma parte global e em uma local. Desse modo foi possível concentrar as não-linearidades em uma fase computacionalmente menos onerosa sem contudo perder precisão na análise.

Do ponto de vista numérico, na etapa global foi testada a eficiência de dois métodos consagrados (*mixed FEM* e *classic FEM*) e foram validadas as hipóteses sobre as vantagens de cada um. Ainda mais interessante foi a descoberta de como esses métodos podem funcionar em maneira ótima se usados em maneira cooperativa em procedimentos de refinamento de malha: dessa maneira, malhas que não exprimem convergência para ambos os métodos se usados em modo independente, podem ser atingidas quando abordadas com o método combinado. Desse modo foi possível resolver o problema da grande dependência de escala das não-linearidades.

Apesar da maior complexidade de implementação da parte global do problema, os fenômenos da parte local axissimétrica se demonstraram mais delicados e relevantes dados seus maiores contributos ao estado final de tensões. Foi observada a complexidade da interação entre os fenômenos e como a intuição pode falhar para situações assim complexas. Como exemplos pode-se citar como o efeito do aumento de comprimento da estrutura pode em última análise aliviar as trações resultantes apesar do amento do peso próprio total da estrutura ou como um peso próprio maior pode, através de maiores amplitudes de trações, ajudar a compensar efeitos anelásticos que derivam da distribuição de temperatura.

De qualquer maneira, a conclusão mais importante do presente estudo é sobre a importância relativa dos efeitos anelásticos em relação aos restantes fenômenos: apesar de serem comumente desprezados na literatura, foi possível verificar o quanto tais fenômenos incidem na resposta final podendo levar a estado de tração tensões que seriam de compressão e por fim podendo mudar completamente a zona mais solicitada e a distribuição final do estado de tensões.

Os efeitos estáticos considerados são suficientes a limitar uso de grande parte dos materiais industrialmente disponíveis pois pode-se chegar no presente caso a tensões equivalentes de Von Mises da ordem de 200 MPa para o aço, valor muito superior ao escoamento de muitos materiais dessa classe. Com isso o leitor pode compreender como é delicada a projetação eficiente desse tipo de estrutura dado que os próprios custos de material podem resultar proibitivos pelas grandes especificações de resistência.

Anexos

ANEXO A – Elasticidade linear estática

A.1 Equações do equilibrio

Existem duas maneiras para se obter as equações do equilíbrio intrínseco: por equilíbrio de um elemento diferencial ou pelo axioma de Euler associado à introdução do tensor de Cauchy. No presente texto será utilizado o segundo modo.

Axioma A.1. ***(Euler)*** *Dado um corpo Ω, cada sua parte $P \subset \Omega$ pode ser separada do seu complementar mediante a aparição de uma força superficial de contato.*

Postulado A.1. ***(Cauchy)*** *Se $P \subset \Omega$ é uma parte de um corpo Ω em equilíbrio, na superfície de separação entre P e seu complementar age uma força de densidade superficial $\vec{t}$ dependente somente do ponto $\vec{x}$ e da normal à superfície $\vec{n}$ em $\vec{x}$, i.e.*

$$t(\vec{x}, \vec{n}) \tag{A.1}$$

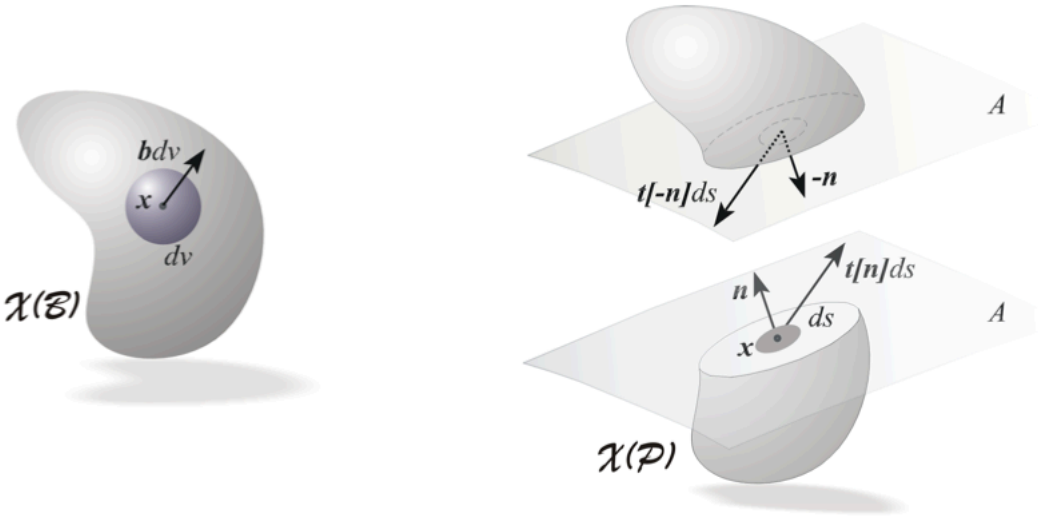


Figura 30: Axioma de Euler/Postulado de Cauchy

Fonte: Poeta60 (2014, http://it.wikipedia.org/wiki/Continuo_di_Cauchy)

Em um corpo equilibrado Ω, onde agem as forças $\vec{b}$ e $\vec{s}$ respectivamente volumétrica e superficial si há que $\forall P \subset \Omega$ as quantidades:

$$R(P) = \int_P \vec{b}(\vec{x}) \mathrm{d}V(\vec{x}) + \int_{\partial P} \vec{t}(\vec{x}, \vec{n}) \mathrm{d}A(\vec{x}) \tag{A.2a}$$

$$M(P,\vec{x_0}) = \int_P (\vec{x}-\vec{x}_0)\wedge\vec{b}(\vec{x})\mathrm{d}V(\vec{x}) + \int_{\partial P}(\vec{x}-\vec{x}_0)\wedge\vec{t}(\vec{x},\vec{n})\mathrm{d}A(\vec{x}) \tag{A.2b}$$

Respectivamente *resultante* e *momento resultante* se anulam. Note que:

$$\vec{t}(\vec{x},\vec{n}) = \vec{s}(\vec{x}) \quad em \quad \partial\Omega \tag{A.3}$$

Claramente as mesmas equações valem para o complementar de P (P^c). Portanto tem-se:

$$R(P) = M(P,\vec{x}_0) = 0 \quad \forall P\subset\Omega, \quad \forall\vec{x}_0\in\mathbb{R}^3 \tag{A.4}$$

Introduzindo o *tensor tensão de Cauchy*, assumindo dependência contínua e linear da densidade de força superficial $\vec{t}(\vec{x},\vec{n})$ em relação à normal $\vec{n}$ da superfície que divide o corpo, de modo que:

$$\vec{t}(\vec{x},\vec{n}) = \mathbf{T}(\vec{x}).\vec{n} \tag{A.5}$$

Assim podemos deduzir as equações de equilíbrio intrínseco usando os princípios de conservação dos momentos linear e angular e com o auxílio da fórmula de Gauss:

Teorema A.1. ***(Equações de Equilíbrio Intrínseco)*** *Seja Ω um corpo em equilíbrio sob ação das forças $\vec{b}:\Omega\to\mathbb{R}^3$ volumétrica e $\vec{s}:\partial\Omega\to\mathbb{R}^3$ superficial então, sendo $\boldsymbol{T}(\vec{x})$ o* tensor tensão de Cauchy *as equações de equilíbrio intrínseco são:*

$$\begin{cases} div(\mathbf{T}(\vec{x})) + \vec{b}(\vec{x}) = 0 & se\ \vec{x}\in\Omega \\ \mathbf{T}(\vec{x}).\vec{n} = \vec{s}(\vec{x}) & se\ \vec{x}\in\partial\Omega \\ \mathbf{T}(\vec{x}) = \mathbf{T}^t(\vec{x}) & \forall\vec{x}\in\Omega \end{cases} \tag{A.6}$$

Dem.: *Uma das partições possíveis é $P=\Omega$ e nesse caso a equação (A.2a) se reduz a*

$$\int_\Omega \vec{b}(\vec{x})\mathrm{d}V(\vec{x}) + \int_{\partial\Omega}\mathbf{T}(\vec{x}).\vec{n}\mathrm{d}A(\vec{x}) = 0$$

Usando o teorema de Gauss no segundo adendo se obtém a:

$$\int_\Omega \vec{b}(\vec{x})\mathrm{d}V(\vec{x}) + \int_\Omega div(\mathbf{T}(\vec{x}))\mathrm{d}A(\vec{x}) = 0$$

Como o mesmo raciocínio pode ser usado para um volume de controle qualquer, não necessariamente todo o domínio, a equação pode ser escrita na sua forma local:

$$div(\mathbf{T}(\vec{x})) + \vec{b}(\vec{x}) = 0 \quad \forall\vec{x}\in\Omega$$

Para a simetria se usa a notação de Einstein do cálculo tensorial. Seja portanto um genérico volume de controle $V\subset\Omega$ e a sua correspondente fronteira $\partial\Omega$. Denotando

ainda a componente normal à superfície do tensor tensão como $\vec{T}^{(n)}$ *e o braço do momento como* $\vec{r} = x_j\vec{e}_j$*, a equação (A.2b) fornece:*

$$M(P,\vec{x_0}) = \int_V \vec{r}\wedge\vec{b}(\vec{x})\mathrm{d}V(\vec{x}) + \int_{\partial V}\vec{r}\wedge\vec{t}(\vec{x},\vec{n})\mathrm{d}A(\vec{x}) = 0$$

Que em notação tensorial se escreve:

$$\int_V \epsilon_{ijk}{}^{1}x_j b_k \mathrm{d}V(\vec{x}) + \int_{\partial V}\epsilon_{ijk}x_j T_k^{(n)}\mathrm{d}A(\vec{x}) = 0$$

Notando que $T_k^{(n)} = \sigma_{mk}n_m$ *e usando o* teorema de Gauss*:*

$$\begin{aligned}
0 &= \int_V \epsilon_{ijk}x_j b_k \mathrm{d}V(\vec{x}) + \int_{\partial V}\epsilon_{ijk}x_j\sigma_{mk}n_m\mathrm{d}A(\vec{x}) \\
&= \int_V \epsilon_{ijk}x_j b_k \mathrm{d}V(\vec{x}) + \int_V (\epsilon_{ijk}x_j\sigma_{mk})_{,m}\mathrm{d}V(\vec{x}) \\
&= \int_V \epsilon_{ijk}x_j b_k \mathrm{d}V(\vec{x}) + \int_V (\epsilon_{ijk}x_{j,m}\sigma_{mk} + \epsilon_{ijk}x_j\sigma_{mk,m})\mathrm{d}V(\vec{x}) \\
&= \int_V \epsilon_{ijk}x_j(\sigma_{mk,m} + b_k)\mathrm{d}V(\vec{x}) + \int_V \epsilon_{ijk}x_{j,m}\sigma_{mk}\mathrm{d}V(\vec{x}),
\end{aligned}$$

A primeira integral se anula porque seu integrando corresponde à equação de equilíbrio já desenvolvida, portanto resta só o segundo termo que se reduz notando que $x_{j,m} = \delta_{j,m}$*. Assim com a arbitrariedade do volume de controle tem-se que:*

$$\epsilon_{ijk}\sigma_{jk} = 0 \quad \forall\vec{x}\in\Omega$$

Essa última corresponde em notação tensorial à condição de simetria. ♢

Em coordenadas cilíndricas as equações de equilíbrio (A.6) são:

$$\begin{cases}
\frac{\partial\sigma_r}{\partial r} + \frac{1}{r}(\frac{\partial\tau_{\theta r}}{\partial\theta} + (\sigma_r - \sigma_\theta)) + \frac{\partial\tau_{rz}}{\partial z} + b_r = 0 \\
\frac{\partial\tau_{\theta r}}{\partial r} + \frac{1}{r}(\frac{\partial\sigma_\theta}{\partial\theta} + 2\tau_{\theta r}) + \frac{\partial\tau_{\theta z}}{\partial z} + b_\theta = 0 \\
\frac{\partial\tau_{rz}}{\partial r} + \frac{1}{r}(\frac{\partial\tau_{\theta z}}{\partial\theta} + \tau_{rz}) + \frac{\partial\sigma_z}{\partial z} + b_z = 0
\end{cases} \tag{A.7}$$

A.2 Cinemática e Congruência em Pequenas Deformações

A.2.1 Cinemática do meio e equações de campo

Deformar um corpo significa alterar a posição relativa entre dois pontos (ou partículas). A posição e a evolução de um ponto podem ser descritas introduzindo o *vetor deslocamento* $\vec{u}(\vec{x},t)$ e o *vetor velocidade* $\vec{v}(\vec{x},t)$.

Para a descrição da cinemática se introduzem dois tipos de coordenadas:

[1] Símbolo de Levi-Civita

$$\epsilon_{ijk} = \begin{cases} +1 & \text{se } (i,j,k) = (1,2,3) \text{ ou } (2,3,1) \text{ ou } (3,1,2) \\ -1 & \text{se } (i,j,k) = (3,2,1) \text{ ou } (2,1,3) \text{ ou } (1,3,2) \\ 0 & \text{se } i = j \text{ ou } i = k \text{ ou } j = k \end{cases}$$

Coordenada espacial ou Euleriana $\vec{x} = (x_1, x_2, x_3)$ que representa a posição de um ponto em relação ao sistema de referência;

Coordenada material ou Lagrangeana $\vec{A} = (A_1, A_2, A_3)$. Sendo o ponto P o ponto que ocupa a posição $\vec{x}$ no instante t a correspondente coordenada material $\vec{A} = \vec{A}(\vec{x}, t)$ distingue a posição que P ocupava na configuração indeformada (que se convém no instante inicial $t = 0$).

Das coordenadas introduzidas derivam dois modos de descrever os vetores *deslocamento* e *velocidade*:

Coordenada espacial ou Euleriana

$$\vec{u}(\vec{x}, t) = \vec{x} - \vec{A}(\vec{x}, t), \quad \vec{v}(\vec{x}, t) = \frac{D\vec{u}(\vec{x}, t)}{Dt}^{2} \tag{A.8}$$

Coordenada material ou Lagrangeana

$$\vec{U}(\vec{A}, t) = \vec{x}(\vec{A}, t) - \vec{A}, \quad \vec{V}(\vec{x}, t) = \frac{D\vec{U}(\vec{A}, t)}{Dt} \tag{A.9}$$

Neste texto se adotará a descrição Lagrangeana. Note que para essa vale a seguinte:

$$\frac{\partial x_i}{\partial A_j} = \delta_{ij} + \frac{\partial U_i}{\partial A_j}, \quad dx_i = \sum_{j=1}^{3} \left(\delta_{ij} + \frac{\partial U_i}{\partial A_j} \right) \tag{A.10}$$

A.2.2 O tensor deformação

Para poder obter uma medida de deformação considera-se dois vetores passantes pelo ponto P de coordenada genérica no estado indeformado $\vec{A}$ com a seguinte orientação genérica:

$$\vec{ds}_0 = dA_1\vec{e}_1 + dA_2\vec{e}_2 + dA_3\vec{e}_3 \tag{A.11}$$

$$\vec{ds}_0^{\,*} = dA_1^*\vec{e}_1 + dA_2^*\vec{e}_2 + dA_3^*\vec{e}_3 \tag{A.12}$$

No estado deformado o ponto P ocupa a posição $\vec{x}(\vec{A}, t)$ e os vetores genéricos assumem a orientação, dada pelo campo de deflexão, seguinte:

$$\vec{ds} = dx_1\vec{e}_1 + dx_2\vec{e}_2 + dx_3\vec{e}_3 \tag{A.13}$$

$$\vec{ds}^{\,*} = dx_1^*\vec{e}_1 + dx_2^*\vec{e}_2 + dx_3^*\vec{e}_3 \tag{A.14}$$

Uma boa medida de deformação é dada pela quantidade:

$$\vec{ds}.\vec{ds}^{\,*} - \vec{ds}_0.\vec{ds}_0^{\,*}{}^{3} \tag{A.15}$$

---

[2] Derivada material: $\frac{D\vec{u}}{Dt} = \vec{u}_t + (\vec{u}.\nabla)\vec{u}$

[3] Note que: $\vec{ds}.\vec{ds}^{\,*} - \vec{ds}_0.\vec{ds}_0^{\,*} = |\vec{ds}||\vec{ds}^{\,*}|\cos\theta - |\vec{ds}_0||\vec{ds}_0^{\,*}|\cos\theta_0 = \sum_{i=1}^{3}(dx_i dx_i^* + dA_i dA_i^*)$

Portanto através da (A.10) se obtém que:

$$\begin{aligned} dx_i dx_i^* - dA_i dA_i^* &= \sum_{j,k=1}^{3} \left(\delta_{ij} + \frac{\partial U_i}{\partial A_j}\right)\left(\delta_{ik} + \frac{\partial U_i}{\partial A_k}\right) dA_j dA_k^* - dA_i dA_i^* \\ &= \sum_{j,k=1}^{3} \left(\frac{\partial U_i}{\partial A_k}\delta_{ij} + \frac{\partial U_i}{\partial A_j}\delta_{ik} + \frac{\partial U_i}{\partial A_k}\frac{\partial U_i}{\partial A_j}\right), \end{aligned}$$

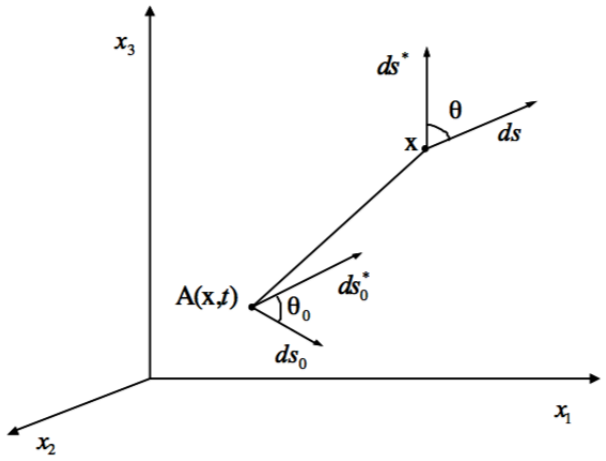


Figura 31: Deformação

Fonte: (SALSA, 2014)

Finalmente pode-se introduzir o *tensor deformação* $\mathbf{E} = \epsilon_{ij}$ com:

$$\epsilon_{ij} = \frac{1}{2}\left\{\frac{\partial U_i}{\partial A_j} + \frac{\partial U_j}{\partial A_i} + \sum_{k=1}^{3} \frac{\partial U_k}{\partial A_i}\frac{\partial U_k}{\partial A_j}\right\} \quad i,j = 1,2,3 \tag{A.16}$$

E então pode-se escrever:

$$\vec{ds}.\vec{ds}^{\,*} - \vec{ds}_0.\vec{ds}_0^{\,*} = 2\sum_{i,j=1}^{3} \epsilon_{ij} dA_i dA_j^* \tag{A.17}$$

O tensor deformação introduzido resume uma série de medidas de deformação nas suas componentes, em seguida observe algumas dessas medidas:

- Tomando $\vec{ds}_0 = \vec{ds}_0^{\,*} = ds_0\vec{e}_1$, tem-se que $\vec{ds} = \vec{ds}^{\,*}$ ($|\vec{ds}| = ds$) e substituindo na (A.17) tem-se:

$$ds^2 - ds_0^2 = 2\epsilon_{11} ds_0^2 \tag{A.18}$$

Usando o parâmetro $\gamma_1 = \frac{ds - ds_0}{ds_0}$ de deformação longitudinal conclui-se que:

$$\gamma_1 = \sqrt{1 + 2\epsilon_{11}} - 1 \tag{A.19}$$

Assim, com a hipótese de pequenas deformações, usando expansão de Taylor em torno de $\epsilon_{11} = 0$, obtém-se $\gamma_1 \approx \epsilon_{11}$, i.é., *a componente* ϵ_{ii} *representa a variação relativa de comprimento na direção do i-ésimo eixo.*

- Analogamente tomando $\vec{ds}_0 = ds_0\vec{e}_1$ e $\vec{ds}_0^{\,*} = ds_0^*\vec{e}_2$ a (A.17) fornece:

$$dsds^* \cos\theta = 2\epsilon_{12} ds_0 ds_0^* \tag{A.20}$$

Com o auxílio da (A.18) se tem ainda que $ds = (1+2\epsilon_{11})ds_0$ e que $ds^* = (1+2\epsilon_{22})ds_0^*$. Portanto introduzindo o ângulo $\gamma_{1,2} = \frac{\pi}{2} - \theta$ que é a variação de ângulo que ocorre na deformação entre os dois vetores inicialmente perpendiculares, tal que $\sin\gamma_{12} = \cos\theta$, tem-se que:

$$\sin\gamma_{12} = \frac{2\epsilon_{12}}{\sqrt{1+2\epsilon_{11}}\sqrt{1+2\epsilon_{22}}} \tag{A.21}$$

Novamente, fazendo uma expansão de Taylor em torno a $\epsilon_{12} = 0$ resta que $\gamma_{12} \approx 2\epsilon_{12}$, i.é., ϵ_{ij}, $i \neq j$ *representa a metade da variação do ângulo entre dois vetores que são paralelos aos eixos i-ésimo e j-ésimo no instante inicial (shear strains)* .

- Por último, considera-se o Jacobiano da transformação $\vec{A} \to \vec{x}(\vec{A}, t)$ que representa a porcentagem de variação de volume após a deformação. Da equação (A.10) tem-se que:

$$\frac{\partial(x_1, x_2, x_3)}{\partial(A_1, A_2, A_3)} = det\left(\delta_{ij} + \frac{\partial U_i}{\partial A_j}\right) \tag{A.22}$$

Desprezando os termos de ordem superior (pequenas deformações) tem-se que:

$$\frac{\partial(x_1, x_2, x_3)}{\partial(A_1, A_2, A_3)} = det\left(\delta_{ij} + \frac{\partial U_i}{\partial A_j}\right) \approx 1 + \sum_{i=1}^{3} \epsilon_{ii} \tag{A.23}$$

Se conclui que *o traço do tensor deformação representa a dilatação cúbica relativa de volume após a deformação.*

A.2.3 Linearização

Para pequenas deformações o tensor deformação se simplifica pois as componentes de ordem superior são desprezíveis, operando uma mudança de notação e observando que para problemas estáticos em pequenas deformações as descrições Euleriana e Lagrangeana coincidem se tem:

$$\epsilon_{ij}(\vec{u}) = \frac{1}{2}\left\{\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right\}, \quad i, j = 1, 2, 3 \tag{A.24}$$

O *coeficiente de dilatação cúbica* assume uma forma simplificada:

$$\Theta = \sum_{i=3} 3\epsilon_{ii}(\vec{u}) = div(\vec{u}) \tag{A.25}$$

Nesse caso as condições de contorno do problema tais como carregamentos ou deflexões impostas podem ser impostas em relação ao domínio de referência (indeformado) assim como as equações de equilíbrio desenvolvidas.

A.3 Relações constitutivas e equação de Navier

A.3.1 Relações constitutivas e Lei de Hooke

Na região elástica do comportamento de um material, existe uma relação linear entre deformação e tensão, ou seja *o tensor tensão é função linear do tensor deformação*. Isto quer dizer que a lei que rege o comportamento do material é do tipo:

$$T_{ij} = C_{ijkh}\epsilon_{kh} \tag{A.26}$$

Com C_{ijkh} as componentes de um tensor de quarta ordem $\mathbb{C}$. A homogeneidade do material, se presente, garante que as componentes C_{ijkh} sejam 81 constantes (constantes elásticas). Com a simetria de ambos os tensores as constantes se reduzem a 21 e finalmente com a isotropia do material[4] se reduzem a duas de modo que (*Lei de Hooke*):

$$T_{ij} = 2\mu\epsilon_{ij} + \lambda\delta_{ij}\epsilon_{kk} \quad ou \quad \mathbf{T} = 2\mu\mathbf{E} + \lambda Tr(\mathbf{E})\mathbf{I} \tag{A.27}$$

Onde as μ e λ são chamadas *constantes de Lamé*. A equação (A.27) é facilmente inversível observando que $\sum_{i=1}^{3} T_{ii} = (2\mu + 3\lambda)Tr(\mathbf{E})$ pode-se obter:

$$\epsilon_{ij} = \frac{T_{ij}}{2\mu} - \frac{\lambda\delta_{ij}}{2\mu(2\mu + 3\lambda)}T_{kk} \tag{A.28}$$

Na engenharia são mais conhecidas as constantes *módulo de Young* E e *coeficiente de Poisson* ν que são dadas pelas seguintes transformações:

$$E = \frac{\mu(2\mu + 3\lambda)}{\mu + \lambda} \tag{A.29}$$

$$\nu = \frac{\lambda}{2(\lambda + \mu)}, \quad \left(0 < \nu < \frac{1}{2}\right) \tag{A.30}$$

E a (A.28) pode ser rescrita como:

$$\epsilon_{ij} = \frac{1 + \nu}{E}T_{ij} - \frac{\nu}{E}\delta_{ij}T_{kk} \tag{A.31}$$

[4] Isotropia corresponde à invariância em relação a rotações de qualquer entidade. Seja uma matriz de rotação $\mathbf{Q}$ genérica; a invariância por rotação de $\mathbb{C}$ corresponde à:

$$\mathbf{Q}\mathbb{C}[\mathbf{E}]\mathbf{Q}^t = \mathbb{C}[\mathbf{Q}\mathbf{E}\mathbf{Q}^t] \quad \forall\mathbf{E} \quad \text{simétrico}$$

A.3.2 Equação de Navier

Partindo da equação (A.6), substituindo a relação (A.27) e observando que:

$$\begin{aligned}(\nabla.\mathbf{T})_i &= \sum_{j=1}^{3} \frac{\partial T_{ij}}{\partial x_j} \\ &= \mu \sum_{j=1}^{3} \frac{\partial}{\partial x_j}\left\{\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right\} + \lambda \frac{\partial}{\partial x_i} div(\vec{u}) \\ &= \mu \Delta u_i + (\mu + \lambda)\frac{\partial}{\partial x_i} div(\vec{u}),\end{aligned}$$

Pode-se obter a *equação de Navier*:

$$\mu \Delta \vec{u} + (\mu + \lambda)\nabla div(\vec{u}) + \vec{b} = 0 \quad \forall \vec{x} \in \Omega \tag{A.32}$$

Usando a identidade:

$$rot(rot(\vec{F})) = \nabla div(\vec{F}) - \Delta \vec{F}$$

A equação de Navier pode ser rescrita como:

$$(2\mu + \lambda)\nabla div(\vec{u}) - \mu rot(rot(\vec{u})) + \vec{b} = 0 \quad \forall \vec{x} \in \Omega \tag{A.33}$$

E finalmente pondo em evidência as constantes de Young e Poison:

$$\frac{E}{2(1+\nu)}\left(\Delta \vec{u} + \frac{1}{1-2\nu}\nabla div(\vec{u})\right) + \vec{b} = 0 \quad \forall \vec{x} \in \Omega \tag{A.34}$$

ANEXO B – Complementos de análise funcional

Neste complemento de texto são considerados alguns elementos de análise funcional extensivamente usados no corpo do texto. Para o leitor que não tem familiaridade com a análise funcional a leitura sequencial deste capítulo pode ser exaustiva pois é muito densa de conceitos que à primeira vista parecem muito abstratos, no entanto essa abstração intrínseca da análise funcional é a responsável pelo seu sucesso pois permite uma visão panorâmica de uma grande classe de problemas.

Um exemplo de aplicação dos conceitos enumerados está na própria monografia e possivelmente a melhor maneira de fruir deste complemento é uma leitura paralela com o corpo principal.

Para o leitor interessado, textos mais completos podem ser encontrados em (SALSA, 2010), (YOSIDA, 1974), (ADAMS, 1975).

B.1 Espaço Normado, de Banach e de Hilbert

Definição B.1. *Um* espaço normado $\mathbf{X}$ *é um espaço linear em* $\mathbb{R}$ *t.q. existe uma aplicação chamada* norma $||.|| : \mathbf{X} \to [0, \infty)$ *que verifica as três seguintes propriedades:*

i $||x|| = 0 \iff x = 0$ *(anulamento);*

ii $||\lambda x|| = |\lambda|||x|| \quad \forall \lambda \in \mathbb{R},\ \forall x \in \mathbf{X}$ *(homogeneidade);*

iii $||x + y|| \leq ||x|| + ||y|| \quad \forall x, y \in \mathbf{X}$ *(inequação triangular);*

Observação Uma norma define naturalmente uma noção de distância em um espaço normado: $d(x, y) = ||x - y|| = ||y - x||$.

Note que essa série de conceitos servirá para atribuir a um conjunto de funções uma estrutura ao quanto similar à estrutura familiar dos espaços $\mathbb{R}^n$, onde noções de norma, ângulo e distância são intuitivas.

Exemplo Em $\mathbb{R}^n$, seja $p \in [0, \infty)$ então normas possíveis são $||x|| = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$.

Exemplo Em $C([a, b])$ (espaço linear[1] de funções contínuas no intervalo $[a, b]$) uma norma possível é $||f|| = \max_{t \in [a,b]} |f(t)|$.

[1] $\forall \alpha, \beta \in \mathbb{R} \quad (\alpha f + \beta g)(t) = \alpha f(t) + \beta g(t) \quad \forall t \in [a, b],\ \forall f, g \in C([a, b])$

Observação Uma sequência de elementos de um espaço normado $\{x_i\}_{i\in\mathbb{N}}$ é dita convergente se $\exists x \in \mathbf{X} : \lim_{i\to\infty} ||x_i - x||_X = 0$.

Definição B.2. *Um* espaço de Banach $\mathbf{X}$ *é um espaço normado t.q. toda sequência de Cauchy*[2] *é convergente.*

Exemplo Em $C^k([a,b])$ espaço linear de funções contínuas e deriváveis k vezes com continuidade no intervalo $[a,b]$, a norma $||f|| = \sum_{j=0}^{k} \max_{t\in[a,b]} |f^{(j)}(t)|$ o faz um espaço de Banach.

Finalmente pode-se introduzir o conceito de *espaço de Hilbert* que traz consigo o conceito de *ângulo* entre elementos. Mas antes note a definição seguinte:

Definição B.3. *Seja um espaço linear* $\mathbf{H}$*. Um produto interno ou escalar é uma aplicação bilinear, simétrica e definida positiva de* $\mathbf{H}\times\mathbf{H}$ *em* $\mathbb{R}$ *i.é. uma função*

$$(.,.) : \mathbf{H}\times\mathbf{H} \to \mathbb{R}$$

que possui as seguintes propriedades:

i $(x,x) = 0 \iff x = 0$ *(anulamento);*

ii $(\alpha x + y, z) = \alpha(x,z) + (y,z) \quad \forall x,y,z \in \mathbf{H},\ \forall\alpha\in\mathbb{R}$ *(linearidade);*

iii $(x,y) = (y,x) \quad \forall x,y \in \mathbb{H}$ *(simetria);*

iv $\forall x \in \mathbb{H}\ (x,x) \geq 0$ *(positividade);*

Das propriedades ii e iii se conclui a bilinearidade.

Definição B.4. *Seja um espaço linear equipado de um produto interno. Esse é dito de Hilbert se a norma induzida por tal produto interno (*$||x|| = \sqrt{(x,x)}$*) faz do espaço um espaço de Banach ou completo.*

Proposição B.1. *Vice-versa, um espaço de Banach* $\mathbf{H}$ *é de Hilbert se a sua norma satisfaz a regra do paralelogramo:*

$$||x+y||^2 + ||x-y||^2 = 2||x||^2 + 2||y||^2 \tag{B.1}$$

Nesse caso o produto interno induzido pela norma é:

$$(x,y) = \frac{1}{2}\Big[||x+y||^2 - ||x||^2 - ||y||^2\Big] \tag{B.2}$$

[2] $\{x_i\}_{i\in\mathbb{N}}$ é de Cauchy se $\lim_{n,m\to\infty} ||x_n - x_m||_X = 0$

Exemplo Um exemplo de espaço de Hilbert é o espaço funcional de *funções a quadrado somável*, essas funções são denotadas pelo símbolo $L^2(\Omega)$ onde Ω é o domínio de referência:

$$L^2(\Omega) = \left\{ f : \Omega \to \mathbb{R} : \int_\Omega (f(\vec{x}))^2 \mathrm{d}\Omega < +\infty \right\} \tag{B.3}$$

Esse espaço equipado com o produto escalar $(f, g) = \int_\Omega f(\vec{x})g(\vec{x})\mathrm{d}\Omega$ e consequentemente com norma induzida $||f||_{L^2_\Omega} = \sqrt{\int_\Omega (f(\vec{x}))^2 \mathrm{d}\Omega}$ é de Banach.

Exemplo Analogamente ao espaço funcional anterior pode-se introduzir o espaço genérico $L^p(\Omega)$ com $p \in [1, \infty]$. Esses espaços são de Banach com a norma $||f||_{L^p_\Omega} = (\int_\Omega (f(\vec{x}))^p \mathrm{d}\Omega)^{\frac{1}{p}}$ para o caso $p \in [1, \infty)$ e $||f||_{L^\infty_\Omega} = esssup_{\vec{x}\in\Omega}|f(\vec{x})|$[3], no entanto essas normas não induzem um produto escalar e portanto esses espaços não são de Hilbert.

B.2 Funcionais e formas bilineares

Definição B.5. *Dado um espaço funcional* $\mathbf{V}$*, um funcional é uma aplicação sobre* $\mathbf{V}$ *com imagem em* $\mathbb{R}$*, i.é.:*

$$F : \mathbf{V} \to \mathbb{R}$$

Um funcional é dito linear se:

$$F(\lambda u + \gamma v) = \lambda F(u) + \gamma F(v) \quad \forall u, v \in V,\ \forall \lambda, \gamma \in \mathbb{R} \tag{B.4}$$

Um funcional é dito limitado se:

$$\exists C \geq 0 : \quad |F(u)| \leq C||u|| \quad \forall u \in \mathbf{V} \tag{B.5}$$

Uma notação comumente usada é $F(u)$ ou $\langle F, u \rangle$, note que, para operações de produto escalar, de funcionais e de norma pode ser necessária a especificação do espaço no qual se efetua a operação com o objetivo de evitar ambiguidades, por exemplo pode ser necessário usar a notação $(u, v)_H$, $\langle F, v \rangle_H$ e $||u||_H$.

Observação Um funcional limitado e linear que age sobre um espaço de Banach é também contínuo já que: $||x_n - x|| \to 0 \Longrightarrow |F(x_n) - F(x)| = |F(x_n - x)| \leq C||x_n - x|| \to 0$

Pode-se definir o *espaço dual* (denotado com $\mathbf{V}'$) de um espaço de Banach $\mathbf{V}$ composto de funcionais lineares e limitados sobre $\mathbf{V}$ i.é:

$$\mathbf{V}' = \{ F : \mathbf{V} \to \mathbb{R} : F \text{ é linear e limitado } \} \tag{B.6}$$

[3] *esssup* denota o estremo superior essencial de uma função, i.é., o mínimo valor M para o qual a medida da união de pontos $\vec{x}$ t.q. $|f(\vec{x})| > M$ seja nula. Ver (RUDIN, 1986) para teoria de medidas e análise real.

Tal espaço é de Banach se dotado da seguinte norma:

$$||F||_{V'} = \sup_{v \in V \setminus \{0\}} \frac{|F(v)|}{||v||_V} \tag{B.7}$$

O seguinte é um resultado muito importante da análise funcional:

Teorema B.1. ***representação de Riez*** *Seja* $\mathbf{H}$ *um espaço de Hilbert equipado de produto escalar* $(.,.)_H$*. Para todo funcional linear e limitado* F *de* $\mathbf{H}'$ *existe um único elemento* $x_F \in \mathbf{H}$ *tal que seja válida a:*

$$F(y) = (x_F, y)_H \quad \forall y \in \mathbf{H} \tag{B.8}$$

Reciprocamente, todo elemento $x \in \mathbf{H}$ *identifica um único funcional linear* F_x *de* $\mathbf{H}'$ *tal que:*

$$F_x(y) = (x, y)_H \quad \forall y \in \mathbf{H} \tag{B.9}$$

Em ambos os casos essa identificação é uma isometria, i.é.:

$$||F_x||_{H'} = ||x||_H \ e \ ||F||_{H'} = ||x_F||_H \tag{B.10}$$

Do teorema (B.1) se deduz que existe uma transformação bijetiva e isométrica entre um espaço de Hilbert e seu dual, essa aplicação é chamada *mapa de Riez* e a denotaremos com $R_H : \mathbf{H} \to \mathbf{H}'$. Note que:

$$R_H(x) = F_x \tag{B.11a}$$

$$R_H^{-1}(F) = x_F \tag{B.11b}$$

Exemplo Um exemplo de funcional linear e contínuo para um espaço do tipo $L^p(\Omega)$ é $F(f) = \int_\Omega f(\vec{x}).g(\vec{x})\mathrm{d}\Omega$ com $g \in L^q(\Omega)$ e $\frac{1}{p} + \frac{1}{q} = 1$. Esse resultado deriva (quanto limitação do funcional) de uma propriedade dos espaços L^p muito importante dita *desigualdade de Hölder*:

Proposição B.2. ***Desigualdade de Hölder*** *Sejam* $p, q \in [1, \infty]$ *dois expoentes conjugados, i.é., que satisfazem* $\frac{1}{p} + \frac{1}{q} = 1$*. Sejam ainda as funções* $f \in L^p(\Omega)$ *e* $g \in L^q(\Omega)$ *então vale a seguinte desigualdade:*

$$\left| \int_\Omega f.g\mathrm{d}\Omega \right| \leq \left(\int_\Omega f^p \mathrm{d}\Omega \right)^{\frac{1}{p}} \left(\int_\Omega g^q \mathrm{d}\Omega \right)^{\frac{1}{q}} \tag{B.12}$$

Ou em modo sucinto:

$$||f.g||_{L^1} \leq ||f||_{L^p} ||g||_{L^q} \tag{B.13}$$

Continuando o exemplo anterior, não só F é um exemplo de funcional como se pode demonstrar que todos os funcionais de $L^p(\Omega)$ possuem essa forma para $p \in [1, \infty)$, i.é. pode-se *identificar*[4] o espaço $(L^p(\Omega))'$ (dual de $L^p(\Omega)$) com o espaço $L^q(\Omega)$. O abuso de linguagem seguinte é usual: $L^q(\Omega)$ é o dual de $L^p(\Omega)$.

Ainda no âmbito do exemplo anterior note que no caso especial $L^2(\Omega)$, seu dual é ele mesmo e daí um exemplo do teorema (B.1).

A esse ponto é possível proceder ao conceito de forma:

Definição B.6. *Dado um espaço funcional normado* $\mathbf{V}$*, uma forma é uma aplicação* $a(.,.)$ *que associa a cada par de elementos de* $\mathbf{V}$*, um número real, i.é:*

$$a : \mathbf{V} \times \mathbf{V} \to \mathbb{R} \tag{B.14}$$

Uma forma é *bilinear* se vale:

$$\begin{aligned} a(\lambda u + \gamma w, v) &= \lambda a(u, v) + \gamma a(w, v) \quad \forall \lambda, \gamma \in \mathbb{R},\ \forall u, v, w \in \mathbf{V}, \\ a(u, \gamma w + \lambda v) &= \gamma a(u, w) + \lambda a(u, v) \quad \forall \lambda, \gamma \in \mathbb{R},\ \forall u, v, w \in \mathbf{V}; \end{aligned} \tag{B.15}$$

Uma forma é *contínua* se $\exists M > 0$ t.q. seja válida:

$$|a(u, v)| \leq M||u||_V||v||_V \quad \forall u, v \in \mathbf{V}; \tag{B.16}$$

Uma forma é *simétrica* se vale:

$$a(u, v) = a(v, u) \quad \forall u, v \in \mathbf{V}; \tag{B.17}$$

Uma forma é *positiva* se vale:

$$a(u, u) > 0 \quad \forall u \neq 0 \in \mathbf{V}; \tag{B.18}$$

Uma forma é *coerciva* se $\exists \alpha > 0$ t.q. seja válida:

$$a(u, u) \geq \alpha||u||_V^2 \quad \forall u \in \mathbf{V}; \tag{B.19}$$

Para finalizar a seção é introduzido o conceito de *tripla Hilbertiana* mas antes note a seguinte definição:

Definição B.7. *Sejam* $\mathbf{H}, \mathbf{V}$ *dois espaços de Hilbert. É dito que* $\mathbf{V}$ *é contido e imerso com continuidade em* $\mathbf{H}$ *(notação* $\mathbf{V} \hookrightarrow \mathbf{H}$*) se existe uma constante* $M > 0$ *t.q.* $||u||_H \leq M||u||_V\ \forall u \in \mathbf{V}$*.*

É dito que $\mathbf{V}$ *é denso em* $\mathbf{H}$ *se* $\forall u \in \mathbf{H}\ \exists$ *uma sequência* $\{u_n\}_{n\in\mathbb{N}}$ $u_n \in \mathbf{V}\ \forall n \in \mathbb{N}$ *t.q.* $||u_n - u||_H \to 0$*. Em palavras* $\mathbf{V}$ *é denso em* $\mathbf{H}$ *se para todo elemento* u *de* $\mathbf{H}$ *existe uma sucessão de* $\mathbf{V}$ *que aproxima* u *em norma* $\mathbf{H}$ *arbitrariamente bem.*

[4] Identificar pois se demonstra facilmente que além da correspondência ser bijetiva se tem que $||F_g||_{(L^p)'} = ||g||_{L^q}$ mas o funcional é dado pela associação de g com a integral no domínio Ω.

Observação O conceito de densidade pode ser entendido através dos números racionais $\mathbb{Q}$ e números reais $\mathbb{R}$: para todo $\epsilon > 0$ e para todo número real r existe um número racional a_n arbitrariamente próximo de r, ou seja $|a_n - r| < \epsilon$. O conceito de densidade pode ser estendido para espaços funcionais (que contrariamente aos espaços $\mathbb{R}^n$, são infinito dimensionais).

Definição B.8. *Sejam* $\mathbf{H}, \mathbf{V}$ *dois espaços de Hilbert com* $\mathbf{V} \hookrightarrow \mathbf{H}$ *com densidade, então* $\mathbf{H}' \hookrightarrow \mathbf{V}'$ *e a tripla Hilbertiana é dada pela*

$$\mathbf{V} \overset{denso}{\subset} \mathbf{H} \overset{Riez}{\equiv} \mathbf{H}' \overset{denso}{\subset} \mathbf{V}' \tag{B.20}$$

B.3 Diferenciação em espaços lineares

Nesta seção são expostos brevemente os conceitos de derivação em espaços funcionais lineares, para uma descrição mais detalhada veja (KOLMOGOROV; FOMIN, 1999).

Definição B.9 (Derivada forte ou de Fréchet)**.** *Sejam dois espaços normados* X *e* Y *e* F *uma aplicação de* X *em* Y*, definida em um conjunto aberto* $E \subset X$*. Essa aplicação é dita diferenciável em* $x \in E$ *se existe um operador linear limitado* $L_x : X \to Y$ *para o qual seja válida a seguinte:*

$$\forall\, \epsilon > 0, \exists\, \delta > 0 \;:\quad ||F(x+h) - F(x) - L_x(h)||_Y \leq \epsilon ||h||_X \quad \forall h \in X :\; ||h||_X < \delta \tag{B.21}$$

A $L_x(h)$ *(que é um elemento de* Y*) é dita diferencial forte ou de Fréchet de* F *avaliada em* $x \in E$ *na direção* h*. O operador* L_x *é dito derivada forte de* F *e é indicado com* $F'(x)$*.*

Outro conceito importante é o seguinte:

Definição B.10 (Derivada fraca ou de Gâteaux)**.** *Sejam dois espaços normados* X *e* Y *e* F *uma aplicação de* X *em* Y*. O diferencial fraco ou de Gâteaux de* F *em* x *é o limite (não sempre existente):*

$$DF(x;h) = \lim_{t \to 0} \frac{F(x+th) - F(x)}{t} \quad \forall h \in X \tag{B.22}$$

Com $t \in \mathbb{R}$ *e a igualdade entendida como* $\left|\left| DF(x;h) - \lim_{t\to 0} \frac{F(x+th)-F(x)}{t} \right|\right|_Y = 0$*. Se o operador* $DF(x;h)$ *é linear, pode ser escrito* $DF(x,h) = F'(x)[h]$ *e nesse caso* $F'(x)$ *é a derivada fraca ou de Gâteaux.*

B.4 Distribuições

As distribuições são uma extensão do conceito de função. A teoria foi criada com o propósito de manipular singularidades que se encontram em diversas aplicações: a *delda de Dirac* é um exemplo de distribuição como será visto adiante.

Antes de poder definir o conceito de distribuição deve-se ter em mente alguns conceitos que vêem apresentados a seguir:

Seja Ω um conjunto aberto de $\mathbb{R}^n$ e $f : \Omega \to \mathbb{R}$.

Definição B.11. *O suporte de uma função f é o menor subconjunto fechado do conjunto de pontos onde a função assume valor não nulo, i.é:*

$$suppf = \overline{\{\vec{x} : f(\vec{x} \neq 0)\}}^{5} \tag{B.23}$$

Definição B.12. *Uma função $f : \Omega \to \mathbb{R}$ é dita a suporte compacto em Ω se existe um conjunto compacto*[6] *$K \subset \Omega$ t.q. $suppf \subset K$.*

Pode-se agora dar a seguinte definição:

Definição B.13. *$\mathcal{D}(\Omega)$ é o espaço das funções infinitamente deriváveis e com suporte compacto em Ω, i.é:*

$$\mathcal{D}(\Omega) = \{f \in C^{\infty}(\Omega) : \exists K \subset \Omega,\ compacto : suppf \subset K\} \tag{B.24}$$

Para facilidade de leitura é introduzida a notação *multi-indices* conforme segue: seja $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ uma sequência de números naturais não negativos, então seja $f : \Omega \to \mathbb{R}$, com $\Omega \subset \mathbb{R}^n$ então usa-se a seguinte notação:

$$D^{\alpha} f(\vec{x}) = \frac{\partial^{|\alpha|} f(\vec{x})}{\partial^{\alpha_1} x_1 \partial^{\alpha_2} x_2 \ldots \partial^{\alpha_n} x_n}$$

Com $|\alpha| = \sum_{i=1}^{n} \alpha_i$.

O espaço $\mathcal{D}(\Omega)$ é também conhecido como espaço de funções teste. Neste espaço não é possível introduzir uma norma que o faça um espaço de Banach no entanto é possível, mesmo na ausência de norma introduzir uma convergência adequada:

Definição B.14. *Dada uma sequência $\{\phi_k\}_{k\in\mathbb{N}}$ de funções de $\mathcal{D}(\Omega)$, então essa é convergente a uma função ϕ de $\mathcal{D}(\Omega)$ ($\phi_k \longrightarrow \phi$ em $\mathcal{D}(\Omega)$) se:*

i O suporte das funções ϕ_k são todos contidos em um dado compacto $K \subset \Omega$;

ii $D^{\alpha}\phi_k \longrightarrow D^{\alpha}\phi$ uniformemente [7] *$\forall \alpha \in \mathbb{N}^n$;*

Com esses conceitos é possível dar a definição de distribuição:

---

5 A notação $\overline{A}$ significa o menor conjunto fechado que contém A, por exemplo $\overline{(a,b)} = [a,b]$.

6 $\Omega \subset \mathbb{R}^n$ é dito compacto se é fechado e limitado

7 $\phi_k \longrightarrow \phi$ uniformemente se $|\phi_k(\vec{x}) - \phi(\vec{x})| \overset{k\to\infty}{\longrightarrow} 0 \quad \forall \vec{x} \in \Omega$

Definição B.15. *O espaço das distribuições é o conjunto de funcionais lineares e contínuos*[8] *sobre* $\mathcal{D}(\Omega)$*. Em outras palavras, o espaço das distribuições é o dual de* $\mathcal{D}(\Omega)$*, denotado com* $\mathcal{D}^{'}(\Omega)$*.*

Pede-se ao leitor que não se desmotive com a quantidade de conceitos matemáticos introduzidos: a análise funcional parece sempre à primeira vista um instrumento sem fins práticos mas a realidade é que grande parte das aplicações modernas científicas necessitam dessa ferramenta para uma compreensão completa.

Exemplo Como dito anteriormente, a delta de Dirac é uma distribuição e a sua ação como funcional sobre uma função teste é a seguinte: seja $a \in \Omega$ então a correspondente delta δ_a age da seguinte maneira: $\langle \delta_a, \phi \rangle = \phi(a) \quad \forall \phi \in \mathcal{D}(\Omega)$.

A este ponto nos falta definir uma noção de convergência para $\mathcal{D}^{'}(\Omega)$:

Definição B.16. *Uma sequência de distribuições* $\{T_k\}_{k\in\mathbb{N}}$ *converge em* $\mathcal{D}^{'}(\Omega)$ *a uma distribuição* T *de* $\mathcal{D}^{'}(\Omega)$ *se*

$$\lim_{k\to\infty} \langle T_k, \phi \rangle = \langle T, \phi \rangle \quad \forall \phi \in \mathcal{D}(\Omega) \tag{B.25}$$

Exemplo Note que para toda função $f \in L^2(\Omega)$ é possível associar uma distribuição $T_f \in \mathcal{D}^{'}(\Omega)$ t.q. a sua ação em $\mathcal{D}(\Omega)$ seja:

$$\langle T_f, \phi \rangle = \int_\Omega f\phi \mathrm{d}\Omega \quad \forall \phi \in \mathcal{D}(\Omega)$$

Será visto que o contrário não é verdadeiro, i.é, existem distribuições que não possuem uma correspondente função em $L^2(\Omega)$ (δ delta de Dirac é uma delas[9]) no entanto vale o seguinte lema:

Lema B.1. *O espaço* $\mathcal{D}(\Omega)$ *é denso em* $L^2(\Omega)$*.*

E graças a esse lema é possível demonstrar que:

$$L^2(\Omega) \subset \mathcal{D}^{'}(\Omega) \tag{B.26}$$

A teoria das distribuição permite definir um novo tipo de derivada chamado *derivada distribucional* que generaliza o conceito de derivada:

[8] Note que o conceito de continuidade em $\mathcal{D}(\Omega)$ não coincide com o de limitação (pois $\mathcal{D}(\Omega)$ não é espaço de Banach): Um funcional contínuo T sobre $\mathcal{D}(\Omega)$ satisfaz $\forall\{\phi_k\}$ convergente se tem:

$$\phi_k \longrightarrow \phi \text{ em } \mathcal{D}(\Omega) \Longrightarrow \lim_{k\to\infty} \langle T, \phi_k \rangle = \langle T, \phi \rangle$$

[9] $\int_\Omega \delta^2 \mathrm{d}\Omega = \infty$

Seja portanto $T \in \mathcal{D}'(\Omega)$ com $\Omega \subset \mathbb{R}^n$. A derivada distribucional de T é a distribuição T^* que indicaremos com $T^* = \nabla T \in \mathcal{D}'(\Omega; \mathbb{R}^n)$ que satisfaz o teorema da divergência de Gauss:

$$\langle \nabla T, \phi \rangle = -\langle T, div\phi \rangle \quad \forall \phi \in \mathcal{D}(\Omega; \mathbb{R}^n)^{10} \tag{B.27}$$

Em outras palavras:

$$\langle \frac{\partial T}{\partial x_i}, \phi \rangle = -\langle T, \frac{\partial \phi}{\partial x_i} \rangle \quad \forall \phi \in \mathcal{D}(\Omega),\ i = 1, 2, \ldots, n \tag{B.28}$$

Do mesmo modo são definidas as derivadas de ordem superior:

Definição B.17. *Para todo multi-indice $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ é sempre definida, i.é existe sempre, a derivada distribucional de $T \in \mathcal{D}(\Omega)$ conforme a seguinte:*

$$\langle D^\alpha T, \phi \rangle = (-1)^{|\alpha|} \langle T, D^\alpha \phi \rangle \quad \forall \phi \in \mathcal{D}(\Omega) \tag{B.29}$$

Exemplo Note que a *função de Heaviside* $\chi_{(0,\infty]} = \begin{cases} 0 & \text{se } x \leq 0 \\ 1 & \text{se } x > 0 \end{cases}$ não é derivável em modo clássico pois apresenta uma singularidade. No entanto essa função pode ser derivada distribucionalmente:

$$\langle \frac{d\chi}{dx}, \phi \rangle = -\langle \chi, \frac{d\phi}{dx} \rangle = -\int_{-\infty}^{+\infty} \chi \frac{d\phi}{dx} dx = -\int_0^{+\infty} \frac{d\phi}{dx} dx = -\lim_{x \to \infty} \phi(x) + \phi(0) =$$
$$\overset{\phi \text{ tem suporte compacto}}{=} \phi(0) = \langle \delta_0, \phi \rangle \quad \forall \phi \in \mathcal{D}(\mathbb{R})$$

Note as seguintes propriedades válidas no âmbito da derivada distribucional que não são válidas para derivadas clássicas.

i Toda distribuição é infinitamente derivável distribucionalmente;

ii Se $T_n \longrightarrow T$ em $\mathcal{D}'(\Omega),\ n \to \infty \implies D^\alpha T_n \longrightarrow D^\alpha T$ em $\mathcal{D}'(\Omega),\ n \to \infty,\ \forall \alpha \in \mathbb{N}$, ou seja a derivação distribucional é uma operação contínua em $\mathcal{D}'(\Omega)$;

Observação Caso uma função f possua derivada clássica, a derivada distribucional da distribuição T_f que identifica f em sua ação, corresponde à distribuição que identifica f'. Em outras palavras usando um abuso de linguagem: a derivada distribucional coincide com a clássica caso essa segunda exista.

[10] $\mathcal{D}(\Omega; \mathbb{R}^n) = \{\phi = [\phi_1, \phi_2, \ldots, \phi_n] : \phi_i \in \mathcal{D}(\Omega)\}$

B.5 Espaços de Sobolev

Como visto anteriormente, as funções de $L^2(\Omega)$ são também distribuições, mas isso não significa que as suas derivadas distribucionais sejam elementos de $L^2(\Omega)$. A própria função $\chi_{[a,b]}$ está em $L^2([a,b])$ mas a sua derivada distribucional $\delta_a - \delta_b$ não pertence a tal espaço. Os espaços de sobolev são definidos nesse contexto:

Definição B.18. *Seja $\Omega \subset \mathbb{R}^n$ aberto e k um número natural positivo. Espaço de Sobolev de ordem k sobre Ω ($H^k(\Omega)$) é definido como o espaço formado de funções de $L^2(\Omega)$ que possuem todas derivadas distribucionais até a ordem k ainda elementos de $L^2(\Omega)$, i.é:*

$$H^k(\Omega) = \{f \in L^2(\Omega) : \ D^\alpha f \in L^2(\Omega), \ \forall \alpha \in \mathbb{N}^n : \ |\alpha| \leq k\} \tag{B.30}$$

Note que vale a seguinte imersão $H^{k+1}(\Omega) \subset H^k(\Omega)$ $k \geq 0$. Identificando o espaço $L^2(\Omega)$ com $H^0(\Omega)$.

Os espaços de Sobolev são di Hilbert se dotados do seguinte produto escalar:

$$(f,g)_{H^k(\Omega)} = \sum_{|\alpha| \leq k} \int_\Omega (D^\alpha f)(D^\alpha g) \mathrm{d}\Omega \tag{B.31}$$

Que por sua vez induz a seguinte norma:

$$||f||_{H^k(\Omega)} = \sqrt{(f,f)_{H^k(\Omega)}} = \sqrt{\sum_{|\alpha| \leq k} \int_\Omega (D^\alpha f)^2 \mathrm{d}\Omega} \tag{B.32}$$

Pode-se ainda definir as seguintes *seminormas*[11]:

$$|f|_{H^k(\Omega)} = \sqrt{\sum_{|\alpha| = k} \int_\Omega (D^\alpha f)^2 \mathrm{d}\Omega} \tag{B.33}$$

Dessa maneira pode-se simplificar a equação (B.32):

$$||f||_{H^k(\Omega)} = \sqrt{\sum_{m=0}^{k} |f|^2_{H^m(\Omega)}} \tag{B.34}$$

Exemplo Note que nem todas as funções presentes em espaços de Sobolev são contínuas: seja $\Omega = \{(x,y) \in \mathbb{R}^2 : x^2 + y^2 \leq 1\}$ ou seja a área delimitada pela circunferência de raio 1, então a função $f(x,y) = \left| ln \frac{1}{\sqrt{x^2+y^2}} \right|^k$ pertence à $H^1(\Omega)$ (verifique que as integrais do seu quadrado em Ω e do quadrado do seu gradiente em Ω são finitas) no entanto essa função possui uma singularidade na origem e portanto não é contínua em Ω.

Apesar disso é possível derivar uma relação entre espaços de funções contínuas e de funções em espaços de Sobolev:

[11] Possui todas as propriedades de norma menos o anulamento.

Teorema B.2. ***Imersão de Sobolev*** *Seja Ω um subconjunto aberto de $\mathbb{R}^n$. Então se a fronteira de Ω ($\partial\Omega$) é "suficientemente regular"*[12]*, então:*

$$H^k(\Omega) \subset C^m(\overline{\Omega}) \quad se\ k > m + \frac{n}{2} \tag{B.35}$$

Note que para uma dimensão ($n = 1$) as funções de $H^1([a, b])$ pertencem também a $C^0(\overline{\Omega})$ e portanto são contínuas. Já em duas dimensões, é necessário que f esteja em $H^2(\Omega)$ para garantir a continuidade.

Uma última observação para o caso das funções de $H^1(\Omega)$ que se anulam na fronteira do domínio ($H^1_0(\Omega)$) ou em parte do mesmo ($H^1_{\Gamma_D} = \{f \in H^1(\Omega) :\ f(\vec{x}) = 0 \forall \vec{x} \in \Gamma_D\}$) vale o seguinte resultado:

Proposição B.3. *Para as funções de $H^1_{\Gamma_D}$ a seminorma de $H^1(\Omega)$ é equivalente*[13] *à sua norma pois vale a seguinte desigualdade de Poincaré:*

$$||v||_{L^2(\Omega)} \leq C_\Omega |v|_{H^1(\Omega)} \quad \forall v \in H^1_{\Gamma_D} \tag{B.36}$$

A desigualdade de Poincaré vale para outros casos, um caso importante para o texto é o espaço das funções de $H^1(\Omega)$ a média nula ($H^1_{\int} = \{v \in H^1(\Omega) :\ \int_\Omega v \mathrm{d}\Omega = 0\}$).

[12] De modo heurístico, a regularidade de uma fronteira é medida pela derivabilidade do gráfico gerado por essa com um sistema cartesiano colocado em pontos da própria fronteira e pela propriedade do domínio de estar sempre de um só lado da fronteira

[13] Duas normas são equivalentes em um espaço funcional X se $\exists c > 0,\ C > 0$ constantes: $c||x||_X \leq ||x||_Y \leq C||x||_X \quad \forall x \in X$.

Referências

ADAMS, R. A. *Sobolev Spaces*. New York: Academic Press, 1975. Citado na página 109.

ARANHA, J. A. P.; MARTINS, C. A.; PESCE, C. P. Analytical approximation for the dynamic bending moment at the touchdown point of a catenary riser. *International Journal of Offshore and Polar Engineering*, p. 293–300, 1997. Citado na página 95.

BATH, K.-J. *Finite Element Procedures*. [S.l.]: Prentice-Hall, 1996. Citado na página 30.

BERGMAN, J. *Temperature of ocean water*. Webpage. Disponível em: <http://www-.windows2universe.org/earth/Water/temp.html>. Acesso em: 11 ago. 2014. Citado na página 60.

BREDERO Shaw. 2014. Webpage. Disponível em: <http://www.brederoshaw.com-/solutions/images/illustration_pip.jpg>. Acesso em: 04 ago. 2014. Citado na página 22.

CORIGLIANO, A.; TALIERCIO, A. *MECCANICA COMPUTAZIONALE*. [S.l.]: Esculapio, 2005. Citado na página 30.

CPLUSPLUS.COM. 2014. Webpage. Disponível em: <http://www.cplusplus.com-/reference/>. Acesso em: 8 ago. 2014. Citado na página 66.

DEVELOPERS libMesh. *libmesh*. 2014. Webpage. Disponível em: <http://libmesh.github-.io>. Acesso em: 20 out. 2014. Citado na página 67.

GELFAND, I. M.; FOMIN, S. V. *Calculus of Variations*. [S.l.]: Dover Publications, 2000. Citado na página 43.

OFFSHORE TECHNOLOGY CONFERENCE, 2002, Houston. *Optimized Design of Pipe-in-Pipe Systems*. Houston: DeepSea Engineering Management Ltd, 2002. Citado na página 23.

INCROPERA et al. *Fundamentals of Heat and Mass Transfer*. [S.l.]: Wiley, 2012. Citado na página 83.

IRVINE, H. M. *Cable Structures*. [S.l.]: MIT press, 1981. Citado na página 37.

JAN, K. et al. *Ultra high-pressure risers for deepwater drilling*. 2010. Disponível em: <http://www.offshore-mag.com/articles/print/volume-70/issue-3/drilling-_completion/ultra-high-pressure-risers-for-deepwater-drilling.html>. Citado na página 84.

KOLMOGOROV, A. N.; FOMIN, S. *Elements of the Theory of Functions and Functional Analysis*. [S.l.]: Dover Publications, 1999. Citado na página 114.

KOLMOGOROV, A. N.; FOMIN, S. V. *Introductory Real Analysis*. [S.l.]: Prentice-Hall, 1970. Citado na página 27.

KYRIAKIDES, S. Buckle propagation in pipe-in-pipe systems. part i. experiments. *Pergamon*, International Journal of Solids and Structures, n. 39, p. 351–366, 2002. Citado na página 23.

KYRIAKIDES, S.; VOGLER, T. J. Buckle propagation in pipe-in-pipe systems. part ii. analysis. *Pergamon*, International Journal of Solids and Structures, n. 39, p. 367–392, 2002. Citado na página 23.

LANGPOP.COM. *Normalized Comparison*. 2014. Webpage. Disponível em: <http://langpop.com>. Acesso em: 08 ago. 2014. Citado na página 67.

MONTANO, A.; RESTELLI, M.; SACCO, R. Numerical simulation of tethered buoy dynamics using mixed finite elements. *ELSEVIER*, Computer Methods in Applied Mechanics Engineering, n. 196, p. 4117–40129, 2007. Citado na página 50.

OPEN MPI - Message Passing Interface. 2014. Webpage. Disponível em: <http://www-.open-mpi.org/>. Acesso em: 18 set. 2014. Citado na página 66.

PESCE, C. P.; MARTINS, C. A.; CHAKRABARTI, S. *Numerical Modelling in Fluid-Structures Interactions*: Numerical computational of riser dynamics. [S.l.]: WIT PRESS, 2005. p. 253-309 p. Citado 4 vezes nas páginas 21, 41, 74 e 95.

PETSC - Portable Extensible Toolkitfor for Scientific Computation. 2014. Webpage. Disponível em: <www.mcs.anl.gov/petsc/>. Acesso em: 20 set. 2014. Citado na página 66.

POETA60. *Continuo di Cauchy*. 2014. Webpage. Disponível em: <http://it.wikipedia-.org/wiki/Continuo_di_Cauchy>. Acesso em: 28 jul. 2014. Citado na página 101.

PRATA, S. *C++ Primer Plus*. 6. ed. [S.l.]: Addison-Wesley, 2012. Citado na página 66.

QUARTERONI, A. *Modellistica numerica per problemi differenziali*. 4. ed. [S.l.]: Springer, 2008. Citado na página 35.

RUDIN, W. *Real and Complex Analysis*. [S.l.]: McGraw-Hill, 1986. Citado na página 111.

RUDIN, W. *FUNCTIONAL ANALYSIS*. [S.l.]: McGraw-Hill, Inc., 1991. Citado na página 27.

SACCO, R. *AN INTRODUCTION TO MIXED AND HYBRID FINITE ELEMENT METHODS IN COMPUTATIONAL FLUID-MECHANICS*. [S.l.], 2007. Citado na página 51.

SALSA, S. *Equazioni a derivate parziali - Metodi, modelli e applicazioni*. 2. ed. [S.l.]: Springer, 2010. Citado 2 vezes nas páginas 27 e 109.

SALSA, S. *Elasticità lineare*. [S.l.], 2014. Citado na página 105.

SANTOS, H. A. F. A.; ALMEIDA, C. I. On a pure complementary energy principle and a force-based finite element formulation for non-linear elastic cables. *ELSEVIER*, International Journal of Non-Linear Mechanics, n. 46, p. 395–406, 2011. Citado na página 37.

VIEIRA, P. *A practical introduction to finite element programming using libMesh*. [S.l.], 2009. Disponível em: <http://ptmat.fc.ul.pt/~pvieira/libmesh/>. Citado na página 67.

YOSIDA, K. *Functional Analysis*. [S.l.]: Springer-Verlag, 1974. Citado na página 109.

Apêndices

APÊNDICE A – Códigos

Os códigos são organizados em duas partes com três códigos para cada uma.

Na primeira parte são expostos os códigos do problema local: o primeiro é o código do problema da difusão de temperatura, o segundo o problema estrutural em ausência dos efeitos térmicos e o terceiro a junção dos dois problemas. Note que no terceiro código a parte correspondente ao problema de difusão é exatamente igual ao primeiro código. Para o problema estrutural é presente uma parcela de carregamento a mais na solução do campo de deflexões e uma parcela de deformação anelástica que entra no cálculo das tensões. Ambos esses novos fatores são computados a partir da distribuição de temperatura que provém da solução do problema de difusão.

Na segunda parte são expostos os códigos referentes à etapa global: no primeiro é presente a implementação do *header* das classes desenvolvidas com os protótipos de todas as funções e métodos principais, no segundo são definidos os métodos e funções e os algoritmos principais e no terceiro é presente um possível exemplo de um *main* que utilize as classes desenvolvidas.

Como *INPUTS* dinâmicos o problema local possui somente parâmetros numéricos como a dimensão do problema, o tipo resolutor dos sistemas lineares, a ordem de aproximação (primiera ou segunda) e o número de processores a serem utilizados no caso de paralelização. Esses parâmetros são configurados diretamente da linha de comando no lançamento do executável.

A parte global, além dos parâmetros numéricos já mencionados possui outros referentes ao método de Newton como tolerâncias e número máximo de iterações a ser ajustados diretamente da linha de comando. Além disso, o executável também precisa de um file chamado “data_input.txt” onde é feita a leitura dos parâmetros físicos do problema (comprimento do cabo, velocidade de corrente, etc).

A.1 Código do problema de difusão de temperatura

```
//
//  tempdiff.cpp
//
//
//  Created by rodrigo broggi on 16/08/14.
```

```
//
//

// C++ include files that we need
#include <iostream>
#include <algorithm>
#include <math.h>
#include <set>

// Basic include file needed for the mesh functionality.
#include "libmesh/libmesh.h"
#include "libmesh/mesh.h"
#include "libmesh/mesh_generation.h"
#include "libmesh/exodusII_io.h"
#include "libmesh/gnuplot_io.h"
#include "libmesh/linear_implicit_system.h"
#include "libmesh/equation_systems.h"

// Define the Finite Element object, quadrature rule and dof map
    indexing handling.
#include "libmesh/fe.h"
#include "libmesh/quadrature_gauss.h"
#include "libmesh/dof_map.h"

// Define useful datatypes for finite element
// matrix and vector components.
#include "libmesh/sparse_matrix.h"
#include "libmesh/numeric_vector.h"
#include "libmesh/dense_matrix.h"
#include "libmesh/dense_vector.h"

// Define the PerfLog, a performance logging utility.
// It is useful for timing events in a code and giving
// you an idea where bottlenecks lie.
#include "libmesh/perf_log.h"

// The definition of a geometric element
#include "libmesh/elem.h"
```

```
// To impose Dirichlet boundary conditions
#include "libmesh/dirichlet_boundaries.h"
#include "libmesh/analytic_function.h"
#include "libmesh/string_to_enum.h"
#include "libmesh/getpot.h"

// Bring in everything from the libMesh namespace
using namespace libMesh;
using namespace std;


//error function
void error (string & str){
    cerr<<str<<endl;
    exit(1);

}

// Function prototype.  This is the function that will assemble
// the linear system for our problem.  Note that the
// function will take the EquationSystems object and the
// name of the system we are assembling as input.  From the
// EquationSystems object we have acess to the Mesh and
// other objects we might need.
void assemble_tempdiff(EquationSystems & es,const string &
   system_name);

// Exact function prototype for temperature.
Real T_dirichlet (const Real x,const Real y);

// Exact value of conductivity constant (function of material)
Real k_cond (const Real x);

// Define a wrapper for exact_solution that will be needed below
void exact_solution_wrapper (DenseVector<Number> & output, const
    Point & p, const Real);
```

```
// Begin the main program.
int main (int argc, char** argv)
{
    // Initialize libMesh and any dependent libaries, like in
       example 2.
    LibMeshInit init (argc, argv);

    // Declare a performance log for the main program
    // PerfLog perf_main("Main Program");

    // Create a GetPot object to parse the command line
    GetPot command_line (argc, argv);

    // Check for proper calling arguments.
    if (argc < 3)
    {
        if (init.comm().rank() == 0)
            cerr << "Usage:\n"
            <<"\t " << argv[0] << " -d 1(2)" << " -n 15"
            << endl;

        // This handy function will print the file name, line
           number,
        // and then abort.  Currrently the library does not use
           C++
        // exception handling.
        libmesh_error();
    }

    // Brief message to the user regarding the program name
    // and command line arguments.
    else
    {
        cout << "Running " << argv[0];

        for (int i=1; i<argc; i++)
            cout << " " << argv[i];

        cout << endl << endl;
```

```
    }


    // Read problem dimension from command line.  Use int
    // instead of unsigned since the GetPot overload is
       ambiguous
    // otherwise.
    int dim = 2;
    if ( command_line.search(1, "-d") )
         dim = command_line.next(dim);

    // Skip higher-dimensional examples on a lower-dimensional
       libMesh build
    libmesh_example_assert(dim <= LIBMESH_DIM, "2D/3D␣support");

    // Create a mesh with user-defined dimension.
    // Read number of elements from command line
    int ps = 15;
    if ( command_line.search(1, "-n") )
         ps = command_line.next(ps);

    // Read FE order from command line
    string order = "SECOND";
    if ( command_line.search(2, "-Order", "-o") )
         order = command_line.next(order);

    // Read FE Family from command line
    string family = "LAGRANGE";
    if ( command_line.search(2, "-FEFamily", "-f") )
         family = command_line.next(family);

    // Cannot use discontinuous basis.
    if ((family == "MONOMIAL") || (family == "XYZ"))
    {
         if (init.comm().rank() == 0)
              cerr << "ex4␣currently␣requires␣a␣C^0␣(or␣higher)␣FE
                 ␣basis." << endl;
         libmesh_error();
    }
```

```
// Create a mesh, with dimension to be overridden later,
   distributed
// across the default MPI communicator.
Mesh mesh(init.comm());

// Use the MeshTools::Generation mesh generator to create a
   uniform
// grid on the square [-1,1]^D.  We instruct the mesh
   generator
// to build a mesh of 8x8 \p Quad9 elements in 2D, or \p
   Hex27
// elements in 3D.

//Problem domain
const Real R_i = 0.12;
const Real R_e = 0.30;
const Real ti = 0.020;
const Real te = 0.018;
const Real L = 10;



Real halfwidth = dim > 1 ? 1. : 0.;
Real halfheight = dim > 2 ? 1. : 0.;

if ((family == "LAGRANGE") && (order == "FIRST"))
{
    // No reason to use high-order geometric elements if we
       are
    // solving with low-order finite elements.
    MeshTools::Generation::build_cube (mesh,
                                       ps,
                                       (dim>1) ? ps : 0,
                                       (dim>2) ? ps : 0,
                                       R_i, R_e,
                                       0, L,
                                       -halfheight,
                                           halfheight,
```

```
                                                (dim==1)     ? EDGE2 :
                                                ((dim == 2) ? QUAD4 :
                                                    HEX8));
  }

  else
  {
      MeshTools::Generation::build_cube (mesh,
                                                ps,
                                                (dim>1) ? ps : 0,
                                                (dim>2) ? ps : 0,
                                                R_i, R_e,
                                                0, L,
                                                -halfheight,
                                                    halfheight,
                                                (dim==1)     ? EDGE3 :
                                                ((dim == 2) ? QUAD9 :
                                                    HEX27));
  }


  // Print information about the mesh to the screen.
  mesh.print_info();


  // Create an equation systems object.
  EquationSystems equation_systems (mesh);

  // Declare the system and its variables.
  // Create a system named "Diffusion"
  LinearImplicitSystem& system =
  equation_systems.add_system<LinearImplicitSystem> ("
     Diffusion");


  // Add the variable "T" to "Diffusion".  "T"
  // will be approximated using second-order approximation.
  unsigned int T_var = system.add_variable("T",
```

```
                                                    Utility::
                                                      string_to_enum<
                                                      Order>    (order)
                                                      ,
                                                    Utility::
                                                      string_to_enum<
                                                      FEFamily>(family
                                                      ));

  // Give the system a pointer to the matrix assembly
  // function.
  system.attach_assemble_function (assemble_tempdiff);

  // Construct a Dirichlet boundary condition object

  // Indicate which boundary IDs we impose the BC on
  // We either build a line, a square or a cube, and
  // here we indicate the boundaries IDs in each case
  set<boundary_id_type> boundary_ids;

  // the dim==1 mesh has two boundaries with IDs 0 and 1

  if(dim==1){
       boundary_ids.insert(0);
       boundary_ids.insert(1);
  }

  // the dim==2 mesh has four boundaries with IDs 0, 1, 2 and
     3
  if(dim>=2)
  {
       boundary_ids.insert(1);
       boundary_ids.insert(3);
  }

  // Program made for 1D or 2D dimensions
  if(dim>2)
  {
       string s1("Code supports just 1D and 2D problems!");
```

```
        error(s1);
    }

    // Create a vector storing the variable numbers which the BC
        applies to
    vector<unsigned int> variables(1);
    variables[0] = T_var;

    // Create an AnalyticFunction object that we use to project
       the BC
    // This function just calls the function exact_solution via
       exact_solution_wrapper
    AnalyticFunction<> exact_solution_object(
       exact_solution_wrapper);

    DirichletBoundary dirichlet_bc(boundary_ids,
                                   variables,
                                   &exact_solution_object);

    // We must add the Dirichlet boundary condition _before_
    // we call equation_systems.init()
    system.get_dof_map().add_dirichlet_boundary(dirichlet_bc);


    // Initialize the data structures for the equation system.
    equation_systems.init();

    // Print information about the system to the screen.
    equation_systems.print_info();
    mesh.print_info();

    // Solve the system "Diffusion"
    system.solve();

    // After solving the system write the solution
    // to a GMV-formatted plot file.
    if(dim == 1)
    {
```

```
        GnuPlotIO plot(mesh,"Distribuzione di temperatura
           radiale, 1D",GnuPlotIO::GRID_ON);
        plot.write_equation_systems("gnuplot_script",
           equation_systems);
    }
#ifdef LIBMESH_HAVE_EXODUS_API
    else
    {
        ExodusII_IO (mesh).write_equation_systems ((dim == 3) ?
                                                   "out_3.e" : "
                                                      out_2.e",
                                                      equation_systems
                                                      );
    }
#endif // #ifdef LIBMESH_HAVE_EXODUS_API

    // All done.
    return 0;
}



// We now define the matrix assembly function for the
// Diffusion system.  We need to first compute element
// matrices and right-hand sides, and then take into
// account the boundary conditions.
void assemble_tempdiff(EquationSystems& es,
                       const string& system_name)
{
    // It is a good idea to make sure we are assembling
    // the proper system.
    libmesh_assert_equal_to (system_name, "Diffusion");


    // Declare a performance log.  Give it a descriptive
    // string to identify what part of the code we are
    // logging, since there may be many PerfLogs in an
    // application.
    PerfLog perf_log ("Matrix Assembly");
```

```
  // Get a constant reference to the mesh object.
  const MeshBase& mesh = es.get_mesh();

  // The dimension that we are running
  const unsigned int dim = mesh.mesh_dimension();

  // Get a reference to the LinearImplicitSystem we are
     solving
  LinearImplicitSystem& system = es.get_system<
     LinearImplicitSystem>("Diffusion");

  // A reference to the \p DofMap object for this system.  The
      \p DofMap
  // object handles the index translation from node and
     element numbers
  // to degree of freedom numbers.  We will talk more about
     the \p DofMap
  // in future examples.
  const DofMap& dof_map = system.get_dof_map();

  // Get a constant reference to the Finite Element type
  // for the first (and only) variable in the system.
  FEType fe_type = dof_map.variable_type(0);

  // Build a Finite Element object of the specified type.
     Since the
  // \p FEBase::build() member dynamically creates memory we
     will
  // store the object as an \p AutoPtr<FEBase>.  This can be
     thought
  // of as a pointer that will clean up after itself.
  AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));

  // A 5th order Gauss quadrature rule for numerical
     integration.
  QGauss qrule (dim, FIFTH);
```

```
    // Tell the finite element object to use our quadrature rule
       .
    fe->attach_quadrature_rule (&qrule);

    // Declare a special finite element object for
    // boundary integration.
    AutoPtr<FEBase> fe_face (FEBase::build(dim, fe_type));

    // Boundary integration requires one quadraure rule,
    // with dimensionality one less than the dimensionality
    // of the element.
    QGauss qface(dim-1, FIFTH);

    // Tell the finte element object to use our
    // quadrature rule.
    fe_face->attach_quadrature_rule (&qface);

    // Here we define some references to cell-specific data that
    // will be used to assemble the linear system.
    // We begin with the element Jacobian * quadrature weight at
        each
    // integration point.
    const vector<Real>& JxW = fe->get_JxW();

    // The physical XY locations of the quadrature points on the
        element.
    // These might be useful for evaluating spatially varying
       material
    // properties at the quadrature points.
    const vector<Point>& q_point = fe->get_xyz();

    // The element shape functions evaluated at the quadrature
       points.
    const vector<vector<Real> >& phi = fe->get_phi();

    // The element shape function gradients evaluated at the
       quadrature
    // points.
    const vector<vector<RealGradient> >& dphi = fe->get_dphi();
```

```
// Define data structures to contain the element matrix
// and right-hand-side vector contribution.  Following
// basic finite element terminology we will denote these
// "Ke" and "Fe". More detail is in example 3.
DenseMatrix<Number> Ke;
DenseVector<Number> Fe;

// This vector will hold the degree of freedom indices for
// the element.  These define where in the global system
// the element degrees of freedom get mapped.
vector<dof_id_type> dof_indices;

// Now we will loop over all the elements in the mesh.
// We will compute the element matrix and right-hand-side
// contribution.
MeshBase::const_element_iterator       el     = mesh.
   active_local_elements_begin();
const MeshBase::const_element_iterator end_el = mesh.
   active_local_elements_end();

for ( ; el != end_el; ++el)
{
    // Start logging the shape function initialization.
    // This is done through a simple function call with
    // the name of the event to log.
    perf_log.push("elem init");

    // Store a pointer to the element we are currently
    // working on.  This allows for nicer syntax later.
    const Elem* elem = *el;

    // Get the degree of freedom indices for the
    // current element.  These define where in the global
    // matrix and right-hand-side this element will
    // contribute to.
    dof_map.dof_indices (elem, dof_indices);

    // Compute the element-specific data for the current
```

```
// element.  This involves computing the location of the
// quadrature points (q_point) and the shape functions
// (phi, dphi) for the current element.
fe->reinit (elem);

// Zero the element matrix and right-hand side before
// summing them.  We use the resize member here because
// the number of degrees of freedom might have changed
   from
// the last element.  Note that this will be the case if
    the
// element type is different (i.e. the last element was
   a
// triangle, now we are on a quadrilateral).
Ke.resize (dof_indices.size(),
           dof_indices.size());

Fe.resize (dof_indices.size());

// Stop logging the shape function initialization.
// If you forget to stop logging an event the PerfLog
// object will probably catch the error and abort.
perf_log.pop("elem init");

// Now we will build the element matrix.  This involves
// a double loop to integrate the test funcions (i)
   against
// the trial functions (j).
//
// We have split the numeric integration into two loops
// so that we can log the matrix and right-hand-side
// computation seperately.
//
// Now start logging the element matrix computation
perf_log.push ("Ke");

for (unsigned int qp=0; qp<qrule.n_points(); qp++){

    const Real x = q_point[qp](0);
```

```
            for (unsigned int i=0; i<phi.size(); i++)
                for (unsigned int j=0; j<phi.size(); j++)
                    Ke(i,j) += JxW[qp]*(x*k_cond(x)*dphi[i][qp]*
                        dphi[j][qp]);
        }

        // Stop logging the matrix computation
        perf_log.pop ("Ke");

        // Now we build the element right-hand-side contribution
            .
        // This involves a single loop in which we integrate the
        // "forcing function" in the PDE against the test
            functions.
        //
        // Start logging the right-hand-side computation
        perf_log.push ("Fe");


        // Null right hand side
        for (unsigned int i=0; i<phi.size(); i++)
            Fe(i) += 0;


        // Stop logging the right-hand-side computation
        perf_log.pop ("Fe");

        // If this assembly program were to be used on an
            adaptive mesh,
        // we would have to apply any hanging node constraint
            equations
        // Also, note that here we call
            heterogenously_constrain_element_matrix_and_vector
        // to impose a inhomogeneous Dirichlet boundary
            conditions.
        dof_map.
            heterogenously_constrain_element_matrix_and_vector (
```

```
            Ke, Fe, dof_indices);

        // The element matrix and right-hand-side are now built
        // for this element.  Add them to the global matrix and
        // right-hand-side vector.  The \p SparseMatrix::
           add_matrix()
        // and \p NumericVector::add_vector() members do this
           for us.
        // Start logging the insertion of the local (element)
        // matrix and vector into the global matrix and vector
        perf_log.push ("matrix insertion");

        system.matrix->add_matrix (Ke, dof_indices);
        system.rhs->add_vector    (Fe, dof_indices);

        // Start logging the insertion of the local (element)
        // matrix and vector into the global matrix and vector
        perf_log.pop ("matrix insertion");
    }

    // That's it.  We don't need to do anything else to the
    // PerfLog.  When it goes out of scope (at this function
       return)
    // it will print its log to the screen.
}

// Exact function prototype for temperature in the boundary.
Real T_dirichlet (const Real x,const Real y = 0.){

    const Real R_i = 0.12;
    const Real R_e = 0.30;
    const Real L = 10;

    //linear change on external temperature with gradient equal
       to the max
    //verified in ocean conditions
    if (x == R_i)
        return 95;
```

```
    else if (x == R_e)
        return 10 + 0.1*y;

    else{
        string s1("Function T_dirichlet called for x out of
            domain bonds!");
        error(s1);

    }



}

//function that define material conductivity
Real k_cond (const Real x){
    //steel inner and outer conductivity constant
    const Real k_inner_steel = 50;
    const Real k_outer_steel = 50;

    //insulation conductivity constant
    const Real k_insulation = 0.16;

    //overall internal and external radius
    const Real R_i = 0.12;
    const Real R_e = 0.30;

    //internal and external steel pipe thickness
    const Real ti = 0.020;
    const Real te = 0.018;

    //insulation thickness
    const Real t_ins = R_e - R_i - ti - te;

    if (x >= R_i && x <= R_i+ti )
        return k_inner_steel;

    else if (x > R_i + ti && x < R_e-te )
        return k_insulation;
```

```
    else if (x >= R_e - te && x <= R_e )
        return k_outer_steel;

    else{
        string s1("Function k_cond called for x out of domain
            bonds!");
        error(s1);

    }



}

//wrapper of exact solution used to impose Dirichlet conditions
void exact_solution_wrapper (DenseVector<Number> & output, const
    Point & p, const Real){

    const Real R_i = 0.12;
    const Real R_e = 0.30;


    output(0) = T_dirichlet(p(0), (LIBMESH_DIM>1)?p(1):0);


}
```

A.2 Código do problema estrutural sem efeitos térmicos

```
// C++ include files that we need
#include <iostream>
#include <algorithm>
#include <math.h>
#include <fstream>
#include <iomanip>
```

```
// libMesh includes
#include "libmesh/libmesh.h"
#include "libmesh/mesh.h"
#include "libmesh/mesh_generation.h"
#include "libmesh/exodusII_io.h"
#include "libmesh/gnuplot_io.h"
#include "libmesh/linear_implicit_system.h"
#include "libmesh/equation_systems.h"
#include "libmesh/fe.h"
#include "libmesh/quadrature_gauss.h"
#include "libmesh/dof_map.h"
#include "libmesh/sparse_matrix.h"
#include "libmesh/numeric_vector.h"
#include "libmesh/dense_matrix.h"
#include "libmesh/dense_submatrix.h"
#include "libmesh/dense_vector.h"
#include "libmesh/dense_subvector.h"
#include "libmesh/perf_log.h"
#include "libmesh/elem.h"
#include "libmesh/boundary_info.h"
#include "libmesh/zero_function.h"
#include "libmesh/dirichlet_boundaries.h"
#include "libmesh/string_to_enum.h"
#include "libmesh/getpot.h"

//Handling errors
void error (std::string & str){
    std::cerr<<str<<std::endl;
    exit(1);

    }

// Bring in everything from the libMesh namespace
using namespace libMesh;

//Problem geometric parameters:
inline Real R_i(){ return 0.12; }
inline Real R_e(){ return 0.30; }
```

```
inline Real t_i(){ return 0.02; }
inline Real t_e(){ return 0.018; }
inline Real Length(){ return 10.0; }


//Problem materials parameters:
Real mu (const Real x);

Real lambda (const Real x);

Real rho (const Real x);


//Problem Newmann b.c:
Real N_section (const Real x);

Real PressurexR (const Real x,const Real y);

//Golbal problem input
Real N_global (const Real x);

// Matrix and right-hand side assemble
void assemble_elasticity(EquationSystems& es,
                         const std::string& system_name);

void compute_stresses(EquationSystems& es);

// Begin the main program.
int main (int argc, char** argv)
{
  // Initialize libMesh and any dependent libaries
  LibMeshInit init (argc, argv);

  GetPot command_line (argc, argv);

  // Initialize the cantilever mesh
  const unsigned int dim = 2;

  // Skip this 2D example if libMesh was compiled as 1D-only.
```

```
//libmesh_example_assert(dim <= LIBMESH_DIM, "2D support");

  int psr = 50;
  if ( command_line.search(1, "-nx") )
       psr = command_line.next(psr);

  int psl = 50;
  if ( command_line.search(1, "-ny") )
       psl = command_line.next(psl);

  // Read FE order from command line
  std::string order = "SECOND";
  if ( command_line.search(2, "-Order", "-o") )
  order = command_line.next(order);
  std::cout<<"order: "<<order<<std::endl;


// Create a 2D mesh distributed across the default MPI
   communicator.
Mesh mesh(init.comm(), dim);

if ((order == "FIRST")){
  MeshTools::Generation::build_square (mesh,
                                       psr, psl,
                                       R_i(), R_e(),
                                       0., Length(),
                                       QUAD4);
  }
 else{
  MeshTools::Generation::build_square (mesh,
                                       psr, psl,
                                       R_i(), R_e(),
                                       0., Length(),
                                       QUAD9);
 }

// Print information about the mesh to the screen.
mesh.print_info();
```

```
// Create an equation systems object.
EquationSystems equation_systems (mesh);

// Declare the system and its variables.
// Create a system named "Elasticity"
LinearImplicitSystem& system =
  equation_systems.add_system<LinearImplicitSystem> ("
     Elasticity");


// Add two displacement variables, u and v, to the system
unsigned int u_var = system.add_variable("u", Utility::
   string_to_enum<Order> (order), LAGRANGE);
unsigned int v_var = system.add_variable("v", Utility::
   string_to_enum<Order>   (order), LAGRANGE);


system.attach_assemble_function (assemble_elasticity);

// Construct a Dirichlet boundary condition object
// We impose a "clamped" boundary condition on the
// "lowear" and "upper" boundaries, i.e. bc_id = 0,2
std::set<boundary_id_type> boundary_ids;
boundary_ids.insert(0);
//boundary_ids.insert(2);

// Create a vector storing the variable numbers which the BC
   applies to
std::vector<unsigned int> variables(2);
variables[0] = u_var; variables[1] = v_var;

// Create a ZeroFunction to initialize dirichlet_bc
ZeroFunction<> zf;

DirichletBoundary dirichlet_bc(boundary_ids,
                               variables,
                               &zf);
```

```
  // We must add the Dirichlet boundary condition _before_
  // we call equation_systems.init()
  system.get_dof_map().add_dirichlet_boundary(dirichlet_bc);


  // Also, initialize an ExplicitSystem to store stresses
  ExplicitSystem& stress_system =
    equation_systems.add_system<ExplicitSystem> ("StressSystem")
      ;

  stress_system.add_variable("sigma_rr", CONSTANT, MONOMIAL);
  stress_system.add_variable("sigma_zz", CONSTANT, MONOMIAL);
  stress_system.add_variable("sigma_rz", CONSTANT, MONOMIAL);
  stress_system.add_variable("sigma_theta", CONSTANT, MONOMIAL);
  stress_system.add_variable("vonMises", CONSTANT, MONOMIAL);

  // Initialize the data structures for the equation system.
  equation_systems.init();

  // Print information about the system to the screen.
  equation_systems.print_info();

  // Solve the system
  system.solve();


  // Post-process the solution to compute the stresses
  compute_stresses(equation_systems);

  // Plot the solution
#ifdef LIBMESH_HAVE_EXODUS_API
  // Use single precision in this case (reduces the size of the
     exodus file)
  ExodusII_IO exo_io(mesh, /*single_precision=*/true);

  // First plot the displacement field using a nodal plot
  std::set<std::string> system_names;
  system_names.insert("Elasticity");
```

```
  exo_io.write_equation_systems("displacement_and_stress.exo",
      equation_systems,&system_names);

  // then append element-based discontinuous plots of the
      stresses
  exo_io.write_element_data(equation_systems);
#endif // #ifdef LIBMESH_HAVE_EXODUS_API

  // All done.
  return 0;
}


Real mu (const Real x){
    //steel inner and outer constants
    const Real E_pipe_mod = 2e11;
    const Real poisson_pipe_mod = 0.3;

    //mu pipe
    const Real mu_p = E_pipe_mod/(2*(1+poisson_pipe_mod));

    //insulation constants
    const Real E_ins_mod = 5e9;
    const Real poisson_ins_mod = 0.4;


    const Real mu_i = E_ins_mod/(2*(1+poisson_ins_mod));



    if (x >= R_i() && x <= R_i()+t_i() )
        return mu_p;

    else if (x > R_i() + t_i() && x < R_e()-t_e() )
        return mu_i;

    else if (x >= R_e() - t_e() && x <= R_e() )
        return mu_p;
```

```
    else{
        std::string s1("Function mu called for x out of domain
            bonds!");
        error(s1);

    }




}

Real lambda (const Real x){
    //steel inner and outer constants
    const Real E_pipe_mod = 2e11;
    const Real poisson_pipe_mod = 0.3;

    //mu pipe
    const Real lambda_p = (E_pipe_mod*poisson_pipe_mod)/((1+
        poisson_pipe_mod)*(1-2*poisson_pipe_mod));

    //insulation constants
    const Real E_ins_mod = 5e9;
    const Real poisson_ins_mod = 0.4;


    const Real lambda_i = (E_ins_mod*poisson_ins_mod)/((1+
        poisson_ins_mod)*(1-2*poisson_ins_mod));


    if (x >= R_i() && x <= R_i()+t_i() )
        return lambda_p;

    else if (x > R_i() + t_i() && x < R_e()-t_e() )
        return lambda_i;

    else if (x >= R_e() - t_e() && x <= R_e() )
        return lambda_p;
```

```
    else{
        std::string s1("Function lambda called for x out of
            domain bonds!");
        error(s1);

    }




}

Real rho (const Real x){
    //steel inner and outer constants
    const Real rho_pipe = 8000;


    const Real rho_ins = 1300;


    if (x >= R_i() && x <= R_i()+t_i() )
        return rho_pipe;

    else if (x > R_i() + t_i() && x < R_e()-t_e() )
        return rho_ins;

    else if (x >= R_e() - t_e() && x <= R_e() )
        return rho_pipe;

    else{
        std::string s1("Function rho called for x out of domain
            bonds!");
        error(s1);

    }
```

```
}


Real PressurexR (const Real x,const Real y = 0.){


    //linear change on external temperature with gradient equal
       to the max
    //verified in ocean conditions
    if (x == R_i())
        return -R_i()*(104e6 - 10500*y); //104e6 -

    else if (x == R_e())
        return R_e()*(1.5e7 - 10000*y);

    else{
        std::cerr<<"x = "<<x<<"y = "<<y<<std::endl;
        std::string s1("Function PressurexR called for x out of
           domain bonds!");
        error(s1);

    }



}
void assemble_elasticity(EquationSystems& es,
                         const std::string& system_name)
{
    //making sure we will assemble the right system
    libmesh_assert_equal_to (system_name, "Elasticity");

    //getting the mesh reference
    const MeshBase& mesh = es.get_mesh();

    //getting dimension of the system
    const unsigned int dim = mesh.mesh_dimension();
```

```
LinearImplicitSystem& system = es.get_system<
    LinearImplicitSystem>("Elasticity");

//getting the variable numbers
const unsigned int u_var = system.variable_number ("u");
const unsigned int v_var = system.variable_number ("v");

//getting dof map and the fe type (both variables have the
    same type otherwise need to be changed)
const DofMap& dof_map = system.get_dof_map();
FEType fe_type = dof_map.variable_type(0);

//having a "dynamic" pointer of the prescribed type, getting
    default quadrature order
AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));
QGauss qrule (dim, fe_type.default_quadrature_order());
fe->attach_quadrature_rule (&qrule);

//same for boundary elements
AutoPtr<FEBase> fe_face (FEBase::build(dim, fe_type));
QGauss qface(dim-1, fe_type.default_quadrature_order());
fe_face->attach_quadrature_rule (&qface);


//jacobian and quadrature weights for numeric integration (
    taken in the quadrature points)
const std::vector<Real>& JxW = fe->get_JxW();
const std::vector<std::vector<Real> >& phi = fe->get_phi();
const std::vector<std::vector<RealGradient> >& dphi = fe->
    get_dphi();

const std::vector<Point>& q_point = fe->get_xyz();



DenseMatrix<Number> Ke;
DenseVector<Number> Fe;

//Submatrix for handling
```

```
DenseSubMatrix<Number>
Kuu(Ke), Kuv(Ke),
Kvu(Ke), Kvv(Ke);

DenseSubVector<Number>
Fu(Fe),
Fv(Fe);

std::vector<dof_id_type> dof_indices;
std::vector<dof_id_type> dof_indices_u;
std::vector<dof_id_type> dof_indices_v;

MeshBase::const_element_iterator        el     = mesh.
   active_local_elements_begin();
const MeshBase::const_element_iterator end_el = mesh.
   active_local_elements_end();


for ( ; el != end_el; ++el)
{
  const Elem* elem = *el;

  dof_map.dof_indices (elem, dof_indices);
  dof_map.dof_indices (elem, dof_indices_u, u_var);
  dof_map.dof_indices (elem, dof_indices_v, v_var);

  const unsigned int n_dofs   = dof_indices.size();
  const unsigned int n_u_dofs = dof_indices_u.size();
  const unsigned int n_v_dofs = dof_indices_v.size();



  fe->reinit (elem);

  Ke.resize (n_dofs, n_dofs);
  Fe.resize (n_dofs);

  Kuu.reposition (u_var*n_u_dofs, u_var*n_u_dofs, n_u_dofs,
     n_u_dofs);
```

```
Kuv.reposition (u_var*n_u_dofs, v_var*n_u_dofs, n_u_dofs,
    n_v_dofs);

Kvu.reposition (v_var*n_u_dofs, u_var*n_v_dofs, n_v_dofs,
    n_u_dofs);
Kvv.reposition (v_var*n_u_dofs, v_var*n_u_dofs, n_v_dofs,
    n_v_dofs);

Fu.reposition (u_var*n_u_dofs, n_u_dofs);
Fv.reposition (v_var*n_u_dofs, n_v_dofs);

for (unsigned int qp=0; qp<qrule.n_points(); qp++)
{

    const Real x = q_point[qp](0);
    const Real y = q_point[qp](1);

    for (unsigned int i=0; i<n_u_dofs; i++){
      for (unsigned int j=0; j<n_u_dofs; j++)
      {


        //See the weak formulation

        Kuu(i,j) += JxW[qp]*2*mu(x)*x*dphi[j][qp](0)*dphi[
            i][qp](0);

        Kuu(i,j) += (JxW[qp]*2*mu(x)*phi[j][qp]*phi[i][qp
            ])/x;

        Kuu(i,j) += JxW[qp]*2*mu(x)*x*dphi[j][qp](1)*dphi[
            i][qp](1);

        Kuu(i,j) += JxW[qp]*lambda(x)*x*dphi[j][qp](0)*
            dphi[i][qp](0);

        Kuu(i,j) += JxW[qp]*lambda(x)*dphi[j][qp](0)*phi[i
            ][qp];
```

```
        Kuu(i,j) += JxW[qp]*lambda(x)*phi[j][qp]*dphi[i][
            qp](0);

        Kuu(i,j) += (JxW[qp]*lambda(x)*phi[j][qp]*phi[i][
            qp])/x;


      }
    }
    for (unsigned int i=0; i<n_u_dofs; i++)
      for (unsigned int j=0; j<n_v_dofs; j++)
      {

        Kuv(i,j) += JxW[qp]*2*mu(x)*x*dphi[j][qp](0)*dphi[
            i][qp](1);

        Kuv(i,j) += JxW[qp]*lambda(x)*x*dphi[j][qp](1)*
            dphi[i][qp](0);

        Kuv(i,j) +=  JxW[qp]*lambda(x)*dphi[j][qp](1)*phi[
            i][qp];

      }

    for (unsigned int i=0; i<n_v_dofs; i++)
      for (unsigned int j=0; j<n_u_dofs; j++)
      {

          Kvu(i,j) += JxW[qp]*2*mu(x)*x*dphi[j][qp](1)*
              dphi[i][qp](0);

          Kvu(i,j) += JxW[qp]*lambda(x)*x*dphi[j][qp](0)*
              dphi[i][qp](1);

          Kvu(i,j) +=  JxW[qp]*lambda(x)*phi[j][qp]*dphi[i
              ][qp](1);
      }

    for (unsigned int i=0; i<n_v_dofs; i++)
```

```
      for (unsigned int j=0; j<n_v_dofs; j++)
      {

        Kvv(i,j) += JxW[qp]*(2*mu(x)+lambda(x))*x*dphi[j][
            qp](1)*dphi[i][qp](1);

        Kvv(i,j) += JxW[qp]*2*mu(x)*x*dphi[j][qp](0)*dphi[
            i][qp](0);
      }

    //body load - uncomment in case with body load
    //for (unsigned int i=0; i<n_v_dofs; i++)
      //Fv(i) += JxW[qp]*10*x*(1000-rho(x))*phi[i][qp];


  }

  for (unsigned int side=0; side<elem->n_sides(); side++)
    if (elem->neighbor(side) == NULL)
      {
        const std::vector<std::vector<Real> >&  phi_face =
            fe_face->get_phi();
        const std::vector<Real>& JxW_face = fe_face->
            get_JxW();
        const std::vector<Point>& q_point_face = fe_face->
            get_xyz();

        fe_face->reinit(elem, side);

        // Apply pressure on lateral sides (Newmann)
        if( !mesh.boundary_info->has_boundary_id (elem,
            side, 0) && !mesh.boundary_info->
            has_boundary_id (elem, side, 2) )
        {
          for (unsigned int qp=0; qp<qface.n_points(); qp
              ++)
          {
            const Real x = q_point_face[qp](0);
            const Real y = q_point_face[qp](1);
```

```
            for (unsigned int i=0; i<n_u_dofs; i++)
            {
                const Real R_i = 0.12;
                const Real R_e = 0.30;

                if (mesh.boundary_info->has_boundary_id (
                   elem, side, 1) || mesh.boundary_info->
                   has_boundary_id (elem, side, 3)){
                    Fu(i) += -JxW_face[qp]*PressurexR(x,y)
                       *phi_face[i][qp];

                }

            }
        }
    }

    // Apply pressure on upper side (Newmann)
      if( mesh.boundary_info->has_boundary_id (elem,
         side, 2) )
      {
          //loop over quadrature face ponts
          for (unsigned int qp=0; qp<qface.n_points();
              qp++)
          {

              //Getting current quadrature point
                 coordinates:
              const Real x = q_point_face[qp](0);
              const Real y = q_point_face[qp](1);

              //loop over the radial dofs since the
                 pressure acts just upon this
                 direction
              for (unsigned int i=0; i<n_u_dofs; i++)
              {
```

```
                                //See the weak formulation to
                                    understand the term
                                Fv(i) += JxW_face[qp]*N_section(x)*
                                    phi_face[i][qp];

                            }
                        }
                    }

                }


        dof_map.constrain_element_matrix_and_vector (Ke, Fe,
            dof_indices);

        system.matrix->add_matrix (Ke, dof_indices);
        system.rhs->add_vector     (Fe, dof_indices);


    }



}


void compute_stresses(EquationSystems& es)
{
  const MeshBase& mesh = es.get_mesh();

  const unsigned int dim = mesh.mesh_dimension();

  LinearImplicitSystem& system = es.get_system<
      LinearImplicitSystem>("Elasticity");

  unsigned int displacement_vars[2];
  displacement_vars[0] = system.variable_number ("u");
  displacement_vars[1] = system.variable_number ("v");
  const unsigned int u_var = system.variable_number ("u");
```

```
const DofMap& dof_map = system.get_dof_map();
FEType fe_type = dof_map.variable_type(u_var);
AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));
QGauss qrule (dim, fe_type.default_quadrature_order());
fe->attach_quadrature_rule (&qrule);

const std::vector<Real>& JxW = fe->get_JxW();
const std::vector<std::vector<Real> >& phi = fe->get_phi();
const std::vector<std::vector<RealGradient> >& dphi = fe->
   get_dphi();
const std::vector<Point>& q_point = fe->get_xyz();

// Also, get a reference to the ExplicitSystem
ExplicitSystem& stress_system = es.get_system<ExplicitSystem>(
   "StressSystem");
const DofMap& stress_dof_map = stress_system.get_dof_map();
unsigned int sigma_vars[4];
sigma_vars[0] = stress_system.variable_number ("sigma_rr");
sigma_vars[1] = stress_system.variable_number ("sigma_zz");
sigma_vars[2] = stress_system.variable_number ("sigma_rz");
sigma_vars[3] = stress_system.variable_number ("sigma_theta");
unsigned int vonMises_var = stress_system.variable_number ("
   vonMises");

// Storage for the stress dof indices on each element
std::vector< std::vector<dof_id_type> > dof_indices_var(system
   .n_vars());
std::vector<dof_id_type> stress_dof_indices_var;

// To store the stress tensor on each element
DenseVector<Number> elem_sigma;

MeshBase::const_element_iterator el = mesh.
   active_local_elements_begin();
const MeshBase::const_element_iterator end_el = mesh.
   active_local_elements_end();

std::fstream fs;
```

```
fs.open ("strains.txt", std::fstream::in | std::fstream::out |
    std::fstream::app);

for ( ; el != end_el; ++el)
  {
    const Elem* elem = *el;

    for(unsigned int var=0; var<2; var++)
      {
        dof_map.dof_indices (elem, dof_indices_var[var],
            displacement_vars[var]);
      }

    fe->reinit (elem);

    elem_sigma.resize(4);

    for (unsigned int qp=0; qp<qrule.n_points(); qp++)
      {

            const Real x = q_point[qp](0);
            const Real y = q_point[qp](1);
            const unsigned int n_x_dofs = dof_indices_var[0].
               size();
            const unsigned int n_y_dofs = dof_indices_var[1].
               size();


             // Get the gradient at this quadrature point
             Gradient displacement_gradient_x;
             Gradient displacement_gradient_y;
             for(unsigned int l=0; l<n_x_dofs; l++)
             {
                 displacement_gradient_x.add_scaled(dphi[l][qp
                    ], system.current_solution(dof_indices_var
                    [0][l]));
             }

             for(unsigned int l=0; l<n_y_dofs; l++)
```

```
{
    displacement_gradient_y.add_scaled(dphi[l][qp
        ], system.current_solution(dof_indices_var
        [1][l]));
}

Real epsilon_theta = 0;

//std::cout<<std::endl;
//std::cout<<"node values and phi values on
    quadrature: "<<std::endl;

for(unsigned int l=0; l<n_x_dofs; l++)
{
    //std::cout<<"("<<system.current_solution(
        dof_indices_var[0][l])<<","<<phi[l][qp]<<")
        "<<"\t";
    epsilon_theta += phi[l][qp]*system.
        current_solution(dof_indices_var[0][l])/x;
}


//Uncomment in case you want strain information
//fs<<"o = ("<<x<<","<<y<<")"<<std::endl;
//fs<<"d(ur)/dr = "<<displacement_gradient_x(0)
    <<"\t"<<"d(ur)/dz = "<<displacement_gradient_x
    (1)<<std::endl;
//fs<<"d(uz)/dr = "<<displacement_gradient_y(0)
    <<"\t"<<"d(uz)/dz = "<<displacement_gradient_y
    (1)<<std::endl;
//fs<<"epsilon_theta  = "<<epsilon_theta<<std::
    endl<<std::endl;

Real global_traction = 0.0;

//Comment or not if you want to account for the
    global effect
global_traction = N_global(x);
```

```
            //The integration is done to have more precise
               constants
            elem_sigma(0) += JxW[qp]*((lambda(x)+ 2*mu(x))*(
               displacement_gradient_x(0)) + lambda(x)*(
               displacement_gradient_y(1)+epsilon_theta));
            elem_sigma(1) += JxW[qp]*((lambda(x)+ 2*mu(x))*(
               displacement_gradient_y(1)) + lambda(x)*(
               displacement_gradient_x(0)+epsilon_theta) +
               global_traction);
            elem_sigma(2) += JxW[qp]*(mu(x)*(
               displacement_gradient_x(1)+
               displacement_gradient_y(0)));
            elem_sigma(3) += JxW[qp]*((lambda(x)+ 2*mu(x))*(
               epsilon_theta) + lambda(x)*(
               displacement_gradient_y(1)+
               displacement_gradient_x(0)));



    }

  // Get the average stresses by dividing by the element
     volume
  elem_sigma.scale(1./elem->volume());


  // load elem_sigma data into stress_system
  for(unsigned int i=0; i<4; i++)
  {
        stress_dof_map.dof_indices (elem,
           stress_dof_indices_var, sigma_vars[i]);

        // We are using CONSTANT MONOMIAL basis functions,
           hence we only need to get
        // one dof index per variable
        dof_id_type dof_index = stress_dof_indices_var[0];

        if( (stress_system.solution->first_local_index() <=
           dof_index) &&
```

```
            (dof_index < stress_system.solution->
               last_local_index()) )
          {
            stress_system.solution->set(dof_index,
               elem_sigma(i));
          }

      }


      // Also, the von Mises stress
      Number vonMises_value = std::sqrt( 0.5*( pow(elem_sigma(0)
          - elem_sigma(1),2.) +
                                               pow(elem_sigma(1)
                                                   - elem_sigma
                                                  (4),2.) +
                                               pow(elem_sigma(4)
                                                   - elem_sigma
                                                  (0),2.) +
                                               6.*(pow(
                                                  elem_sigma(2)
                                                  ,2.))
                                               ) );



      stress_dof_map.dof_indices (elem, stress_dof_indices_var,
         vonMises_var);
      dof_id_type dof_index = stress_dof_indices_var[0];
      if( (stress_system.solution->first_local_index() <=
         dof_index) &&
          (dof_index < stress_system.solution->last_local_index
             ()) )
        {
          stress_system.solution->set(dof_index, vonMises_value)
             ;
        }
```

```
    }

  fs.close();
  // Should call close and update when we set vector entries
     directly
  stress_system.solution->close();
  stress_system.update();
}


Real N_section (const Real x){
    //steel inner and outer traction constant
    const Real N_inner_pipe = 0.e6;
    const Real N_outer_pipe = 0.e6;

    //insulation conductivity constant
    const Real N_insulation = 0.0;



    if (x >= R_i() && x <= R_i()+t_i() )
        return x*N_inner_pipe;

    else if (x > R_i() + t_i() && x < R_e()-t_e() )
        return x*N_insulation;

    else if (x >= R_e() - t_e() && x <= R_e() )
        return x*N_outer_pipe;

    else{
        std::string s1("Function k_cond called for x out of
           domain bonds!");
        error(s1);

    }

}

Real N_global (const Real x){
```

```
    //steel inner and outer traction constant
    const Real N_inner_pipe = 20.e6;
    const Real N_outer_pipe = 20.e6;

    //insulation conductivity constant
    const Real N_insulation = 0.0;



    if (x >= R_i() && x <= R_i()+t_i() )
        return N_inner_pipe;

    else if (x > R_i() + t_i() && x < R_e()-t_e() )
        return N_insulation;

    else if (x >= R_e() - t_e() && x <= R_e() )
        return N_outer_pipe;

    else{
        std::string s1("Function k_cond called for x out of
           domain bonds!");
        error(s1);

    }

}
```

A.3 Código do problema estrutural com efeitos térmicos

A.4 Código do problema da análise global

A.4.1 *Header* - Protótipo das classes e das funções

```
//
//  Cable_Equation.h
//
//
//  Created by rodrigo broggi on 16/10/14.
//
```

```
//

#ifndef _Cable_Equation_h
#define _Cable_Equation_h

//STD library
#include <iostream>
#include <algorithm>
#include <sstream>
#include <math.h>
#include <string>
#include <set>
#include <vector>

//Boost library
#include<boost/scoped_ptr.hpp>

//Libmesh library
#include "libmesh/libmesh.h"
#include "libmesh/mesh.h"
#include "libmesh/mesh_generation.h"
#include "libmesh/exodusII_io.h"
#include "libmesh/gnuplot_io.h"
#include "libmesh/equation_systems.h"
#include "libmesh/fe.h"
#include "libmesh/quadrature_gauss.h"
#include "libmesh/dof_map.h"
#include "libmesh/sparse_matrix.h"
#include "libmesh/numeric_vector.h"
#include "libmesh/dense_matrix.h"
#include "libmesh/dense_vector.h"
#include "libmesh/linear_implicit_system.h"
#include "libmesh/perf_log.h"
#include "libmesh/boundary_info.h"
#include "libmesh/utility.h"

// To impose Dirichlet boundary conditions
#include "libmesh/dirichlet_boundaries.h"
#include "libmesh/analytic_function.h"
```

```
#include "libmesh/zero_function.h"
#include "libmesh/string_to_enum.h"
#include "libmesh/getpot.h"

#include "libmesh/dense_submatrix.h"
#include "libmesh/dense_subvector.h"

#include "libmesh/elem.h"


                        /*General auxiliary functions used
                           in both methods*/


inline libMesh::Real denominator(const libMesh::Real x, const
   libMesh::Real y,const libMesh::Real exp){ return pow((pow(x
   ,2.)+pow(y,2.)),exp); };

inline libMesh::Real sgn(const libMesh::Real x){ return (x > 0)
   ? 1. : ( x == 0 ? 0. : -1.0 ); };

// Local horizontal force (depends on the local cable
   positioning and on its derivative)
libMesh::Real fx(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl);

// Local vertical force (depends on the local cable positioning
   and on its derivative)
libMesh::Real fy(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl);

// wrapper of exact solution used to impose Dirichlet conditions
void exact_solution_wrapper(libMesh::DenseVector<libMesh::Number
   > & output, const libMesh::Point & p, const libMesh::Real);


                        /*Auxiliary functions for the weak
                           formulation of Classic method*/
```

```
libMesh::Real LA1(const libMesh::Real xl, const libMesh::Real yl
    );

libMesh::Real LA2(const libMesh::Real xl, const libMesh::Real yl
    );

libMesh::Real LB1(const libMesh::Real x, const libMesh::Real y,
    const libMesh::Real xl, const libMesh::Real yl);

libMesh::Real LB2(const libMesh::Real x, const libMesh::Real y,
    const libMesh::Real xl, const libMesh::Real yl);

libMesh::Real A11(const libMesh::Real xl,const libMesh::Real yl)
    ;

libMesh::Real A12(const libMesh::Real xl,const libMesh::Real yl)
    ;

libMesh::Real A21(const libMesh::Real xl,const libMesh::Real yl)
    ;

libMesh::Real A22(const libMesh::Real xl,const libMesh::Real yl)
    ;

libMesh::Real B11(const libMesh::Real x,const libMesh::Real  y,
    const libMesh::Real xl, const libMesh::Real yl);

libMesh::Real B12(const libMesh::Real x,const libMesh::Real  y,
    const libMesh::Real xl, const libMesh::Real yl);

libMesh::Real B21(const libMesh::Real x,const libMesh::Real  y,
    const libMesh::Real xl, const libMesh::Real yl);

libMesh::Real B22(const libMesh::Real x,const libMesh::Real  y,
    const libMesh::Real xl, const libMesh::Real yl);

libMesh::Real C11(const libMesh::Real x,const libMesh::Real  y,
    const libMesh::Real xl, const libMesh::Real yl);
```

```
libMesh::Real C12(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl);

libMesh::Real C21(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl);

libMesh::Real C22(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl);

void assemble_cable_linear_CLASSIC(libMesh::EquationSystems & es
   , const std::string & system_name);


                        /*Auxiliary functions for the weak
                          formulation of Classic method*/


libMesh::Real LA1(const libMesh::Real xl, const libMesh::Real yl
   , const libMesh::Real t);

libMesh::Real LA2(const libMesh::Real xl, const libMesh::Real yl
   , const libMesh::Real t);

libMesh::Real LB1(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real LB2(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real LB3(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);


libMesh::Real A11(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t);
```

```
libMesh::Real A12(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t);

libMesh::Real A13(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t);

libMesh::Real A21(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t);

libMesh::Real A22(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t);

libMesh::Real A23(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t);

libMesh::Real B11(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real B12(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real B13(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real B21(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real B22(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real B23(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);
```

```
libMesh::Real C11(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real C12(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real C13(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real C21(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real C22(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

libMesh::Real C23(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t);

void assemble_cable_linear_MIXED(libMesh::EquationSystems & es,
   const std::string & system_name);



//This class contains the problem's data, including current
   profile and initial approximation solution for the Newton's
   method.
class Cable_Problem_Data {

  //Variables input Data
  libMesh::Real cable_length;
  libMesh::Real cable_diameter;
  libMesh::Real cable_axial_stiffness;
  libMesh::Real cable_submerged_weight;
```

```
  libMesh::Real ocean_depth;
  libMesh::Real current_modulus;

  //Function input Data:
  //This is a pointer to a function that contains the current
     profile function ( V(y) = current_modulus*(*current_profile
     (y)) )
  libMesh::Real (*current_profile)(const libMesh::Real y);

  //Function input Data
  void (*initial_solution)(libMesh::DenseVector<libMesh::Number>
      & xy, const libMesh::Point & s, const libMesh::Real);

public:

  Cable_Problem_Data(const libMesh::Real length, const libMesh::
     Real diameter,
                     const libMesh::Real axial_stiffness, const
                        libMesh::Real s_weight,
                     const libMesh::Real o_depth, const libMesh
                        ::Real c_modulus,
                     libMesh::Real (*function1)(const libMesh::
                        Real),
                     void (*function2)(libMesh::DenseVector<
                        libMesh::Number> &, const libMesh::Point
                         &, const libMesh::Real));

  Cable_Problem_Data(std::istream & INPUT, libMesh::Real (*
     function1)(const libMesh::Real),
                     void (*function2)(libMesh::DenseVector<
                        libMesh::Number> &, const libMesh::Point
                         &, const libMesh::Real));

  //Providing data functions
  inline libMesh::Real L() const { return cable_length; };
  inline libMesh::Real EA() const { return cable_axial_stiffness
     ; };
  inline libMesh::Real q() const { return cable_submerged_weight
     ; };
```

```
  inline libMesh::Real Cd() const { return ( (0.5)*0.47*
     cable_diameter*1000.*(pow(current_modulus,2.)) ); };
  inline libMesh::Real O_Depth() const { return ocean_depth; };

  //Current profile
  inline libMesh::Real f(const libMesh::Real y) const { return (
      (*current_profile)(y) ); };
  //Current profile derivative
  inline libMesh::Real fl(const libMesh::Real y) const { return
     ( ((*current_profile)(y+0.0001) - (*current_profile)(y
     -0.0001))/(0.0002) ); };

  void initial_solution_wrapper(libMesh::DenseVector<libMesh::
     Number> & xy, const libMesh::Point & s,
                                const libMesh::Real t = 0) const
                                    { return (*initial_solution)
                                   (xy,s,t); };

  void print() const;

  friend class Cable_Equation;

  friend class Cable_Equation_CLASSIC;

  friend class Cable_Equation_MIXED;

};




class Cable_Equation {

protected:
  static Cable_Problem_Data data;
  GetPot command_line;


public:
```

```
  //This constructor initializes internal data (physical for
     data structure and numerical for GetPot object)
  Cable_Equation(const Cable_Problem_Data & Data, const GetPot &
      Command_Line);

  //This function solve the problem with the data provided and
     gives output in gnuplot format
  virtual void solve_cable_problem_complete() = 0;

  //Prints physical data
  void print_data() const { data.print(); };



                    /*Auxiliary funtions worth for both
                       Classic and Mixed methods*/


  // Local horizontal force (depends on the local cable
     positioning and on its derivative)
  friend libMesh::Real fx(const libMesh::Real x, const libMesh::
     Real y, const libMesh::Real xl, const libMesh::Real yl);

  // Local vertical force (depends on the local cable
     positioning and on its derivative)
  friend libMesh::Real fy(const libMesh::Real x, const libMesh::
     Real y, const libMesh::Real xl, const libMesh::Real yl);

  // wrapper of exact solution used to impose Dirichlet
     conditions
  friend void exact_solution_wrapper(libMesh::DenseVector<
     libMesh::Number> & output, const libMesh::Point & p, const
     libMesh::Real);



};
```

```
class Cable_Equation_CLASSIC : public Cable_Equation {

  libMesh::Mesh mesh;

  libMesh::EquationSystems equation_systems;

                                /*Weak formulation coefficients -
                                   Auxiliary functions to the
                                   assembling function*/

  friend libMesh::Real LA1(const libMesh::Real xl, const libMesh
     ::Real yl);

  friend libMesh::Real LA2(const libMesh::Real xl, const libMesh
     ::Real yl);

  friend libMesh::Real LB1(const libMesh::Real x, const libMesh
     ::Real y,const libMesh::Real xl, const libMesh::Real yl);

  friend libMesh::Real LB2(const libMesh::Real x, const libMesh
     ::Real y,const libMesh::Real xl, const libMesh::Real yl);


  friend libMesh::Real A11(const libMesh::Real xl,const libMesh
     ::Real yl);

  friend libMesh::Real A12(const libMesh::Real xl,const libMesh
     ::Real yl);

  friend libMesh::Real A21(const libMesh::Real xl,const libMesh
     ::Real yl);

  friend libMesh::Real A22(const libMesh::Real xl,const libMesh
     ::Real yl);
```

```
  friend libMesh::Real B11(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl);

  friend libMesh::Real B12(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl);

  friend libMesh::Real B21(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl);

  friend libMesh::Real B22(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl);

  friend libMesh::Real C11(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl);

  friend libMesh::Real C12(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl);

  friend libMesh::Real C21(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl);

  friend libMesh::Real C22(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl);

  //This is the most important function: it assemble the actual
     Newton linear system iteration
  friend void assemble_cable_linear_CLASSIC (libMesh::
     EquationSystems & es, const std::string & system_name);

  //Post-processing function - calculates the traction from the
     solver implicit problem
  void compute_traction();

public:

  //Constructor
  Cable_Equation_CLASSIC(const Cable_Problem_Data & Data, const
     GetPot & Command_Line, const libMesh::LibMeshInit & INIT);
```

```
  //Function that handles data and put together all problem
     parts, solve by Newton's method the problem and prints (
     GNUPLOT format) the OUTPUT
  void solve_cable_problem_complete();

};

class Cable_Equation_MIXED: public Cable_Equation {

  libMesh::Mesh mesh;

  libMesh::EquationSystems equation_systems;

                              /*Weak formulation coefficients -
                                 Auxiliary functions to the
                                 assembling function*/

  friend libMesh::Real LA1(const libMesh::Real xl, const libMesh
     ::Real yl, const libMesh::Real t);

  friend libMesh::Real LA2(const libMesh::Real xl, const libMesh
     ::Real yl, const libMesh::Real t);

  friend libMesh::Real LB1(const libMesh::Real x, const libMesh
     ::Real y,const libMesh::Real xl, const libMesh::Real yl,
     const libMesh::Real t);

  friend libMesh::Real LB2(const libMesh::Real x, const libMesh
     ::Real y,const libMesh::Real xl, const libMesh::Real yl,
     const libMesh::Real t);

  friend libMesh::Real LB3(const libMesh::Real x, const libMesh
     ::Real y,const libMesh::Real xl, const libMesh::Real yl,
     const libMesh::Real t);


  friend libMesh::Real A11(const libMesh::Real xl,const libMesh
     ::Real yl, const libMesh::Real t);
```

```
friend libMesh::Real A12(const libMesh::Real xl,const libMesh
    ::Real yl, const libMesh::Real t);

friend libMesh::Real A13(const libMesh::Real xl,const libMesh
    ::Real yl, const libMesh::Real t);

friend libMesh::Real A21(const libMesh::Real xl,const libMesh
    ::Real yl, const libMesh::Real t);

friend libMesh::Real A22(const libMesh::Real xl,const libMesh
    ::Real yl, const libMesh::Real t);

friend libMesh::Real A23(const libMesh::Real xl,const libMesh
    ::Real yl, const libMesh::Real t);

friend libMesh::Real B11(const libMesh::Real x,const libMesh::
    Real y, const libMesh::Real xl, const libMesh::Real yl,
    const libMesh::Real t);

friend libMesh::Real B12(const libMesh::Real x,const libMesh::
    Real y, const libMesh::Real xl, const libMesh::Real yl,
    const libMesh::Real t);

friend libMesh::Real B13(const libMesh::Real x,const libMesh::
    Real y, const libMesh::Real xl, const libMesh::Real yl,
    const libMesh::Real t);

friend libMesh::Real B21(const libMesh::Real x,const libMesh::
    Real y, const libMesh::Real xl, const libMesh::Real yl,
    const libMesh::Real t);

friend libMesh::Real B22(const libMesh::Real x,const libMesh::
    Real y, const libMesh::Real xl, const libMesh::Real yl,
    const libMesh::Real t);

friend libMesh::Real B23(const libMesh::Real x,const libMesh::
    Real y, const libMesh::Real xl, const libMesh::Real yl,
    const libMesh::Real t);
```

```
  friend libMesh::Real C11(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl,
     const libMesh::Real t);

  friend libMesh::Real C12(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl,
     const libMesh::Real t);

  friend libMesh::Real C13(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl,
     const libMesh::Real t);

  friend libMesh::Real C21(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl,
     const libMesh::Real t);

  friend libMesh::Real C22(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl,
     const libMesh::Real t);

  friend libMesh::Real C23(const libMesh::Real x,const libMesh::
     Real  y, const libMesh::Real xl, const libMesh::Real yl,
     const libMesh::Real t);

  //This is the most important function: it assemble the actual
     Newton linear system iteration
  friend void assemble_cable_linear_MIXED (libMesh::
     EquationSystems & es, const std::string & system_name);


public:

  //Constructor
  Cable_Equation_MIXED(const Cable_Problem_Data & Data, const
     GetPot & Command_Line, const libMesh::LibMeshInit & INIT);

  //Function that handles data and put together all problem
     parts, solve by Newton's method the problem and prints (
     GNUPLOT format) the OUTPUT
```

```
  void solve_cable_problem_complete();

};


#endif
```

A.4.2 Implementação das funções membro e auxiliares - *source code*

```
//
//  Cable_Equation.C
//
//
//  Created by rodrigo broggi on 18/10/14.
//
//

#include "Cable_Equation.h"


                                        /*Some friend
                                          functions
                                          implementations*/

                                    /*Auxiliary funtions worth
                                      for both Classic and
                                      Mixed methods*/

// Local horizontal force (depends on the local cable
   positioning and on its derivative)
libMesh::Real fx(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl) {

  return (Cable_Equation::data.Cd()*sgn(Cable_Equation::data.f(y
     )*yl)*(pow(Cable_Equation::data.f(y),2.))*pow(yl,3)/
     denominator(xl,yl,3./2.) );

};
```

```
// Local vertical force (depends on the local cable positioning
   and on its derivative)
libMesh::Real fy(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl) {

  return ( Cable_Equation::data.q() - Cable_Equation::data.Cd()*
     sgn(Cable_Equation::data.f(y)*yl)*(pow(Cable_Equation::data
     .f(y),2.))*xl*pow(yl,2)/denominator(xl,yl,3./2.) );

};

// wrapper of exact solution used to impose Dirichlet conditions
void exact_solution_wrapper(libMesh::DenseVector<libMesh::Number
   > & output, const libMesh::Point & p, const libMesh::Real) {

  output(1) = Cable_Equation::data.O_Depth();

};









                                                //Cable_Problem_Data
                                                    functions:



Cable_Problem_Data::Cable_Problem_Data(const libMesh::Real
   length, const libMesh::Real diameter,
                                       const libMesh::Real
                                          axial_stiffness, const
                                           libMesh::Real
                                          s_weight,
                                       const libMesh::Real
                                          o_depth, const libMesh
                                          ::Real c_modulus,
                                       libMesh::Real (*function1
                                          )(const libMesh::Real)
```

```
                                                    ,
                                                void (*function2)(libMesh
                                                    ::DenseVector<libMesh
                                                    ::Number> &, const
                                                    libMesh::Point &,
                                                    const libMesh::Real))
                                                    :
                                                cable_length(length),
                                                    cable_diameter(
                                                    diameter),
                                                    cable_axial_stiffness(
                                                    axial_stiffness),
                                                cable_submerged_weight(
                                                    s_weight), ocean_depth
                                                    (o_depth),
                                                    current_modulus(
                                                    c_modulus),
                                                current_profile(function1
                                                    ), initial_solution(
                                                    function2) {};

Cable_Problem_Data::Cable_Problem_Data(std::istream & INPUT,
    libMesh::Real (*function1)(const libMesh::Real),
                                                void (*function2)(libMesh
                                                    ::DenseVector<libMesh
                                                    ::Number> &, const
                                                    libMesh::Point &,
                                                    const libMesh::Real))
                                                    {

  INPUT >> cable_length;
  INPUT >> cable_diameter;
  INPUT >> cable_axial_stiffness;
  INPUT >> cable_submerged_weight;
  INPUT >> ocean_depth;
  INPUT >> current_modulus;

  current_profile = function1;
  initial_solution = function2;
```

```
};


void Cable_Problem_Data::print() const {


  std::cout<<"******************************PROBLEM
     INFORMATION******************************"<<std::endl<<
     std::endl;

  std::cout<<"Cable length: "<<cable_length<<std::endl;
  std::cout<<"Cable diameter: "<<cable_diameter<<std::endl;
  std::cout<<"Cable axial stiffness: "<<cable_axial_stiffness<<
     std::endl;
  std::cout<<"Cable submerged weight: "<<cable_submerged_weight
     <<std::endl;
  std::cout<<"Ocean depth: "<<ocean_depth<<std::endl;
  std::cout<<"Current modulus: "<<current_modulus<<std::endl;

  std::ofstream initial_solution_data("
     gnuplot_script_initial_xy_data");

  int n = 500;

  const libMesh::Real step = cable_length/n;

  libMesh::DenseVector<libMesh::Number> local(2);

  for (int i = 0; i < n; i++) {
      initial_solution_wrapper(local,step*i);
      initial_solution_data <<local(0)<<"\t"<<local(1)<<std::
         endl;
  }

  initial_solution_data.close();

  std::cout<<"******************************PROBLEM
     INFORMATION END******************************"<<std::endl
```

```
    <<std::endl;

};


//END Cable_Problem_Data functions


                                                    //Cable_Equation
                                                      functions:

Cable_Problem_Data Cable_Equation::data(0.,0.,0.,0.,0.,0.,NULL,
    NULL);


Cable_Equation::Cable_Equation(const Cable_Problem_Data & Data,
    const GetPot & Command_Line): command_line(Command_Line) {

  data.cable_length = Data.cable_length;
  data.cable_diameter = Data.cable_diameter;
  data.cable_axial_stiffness = Data.cable_axial_stiffness;
  data.cable_submerged_weight = Data.cable_submerged_weight;
  data.ocean_depth = Data.ocean_depth;
  data.current_modulus = Data.current_modulus;

  data.current_profile = Data.current_profile;
  data.initial_solution = Data.initial_solution;

};



//END Cable_Equation functions




                                                    //
                                                      Cable_Equation_CLASSIC
```

```
                                                    functions:



Cable_Equation_CLASSIC::Cable_Equation_CLASSIC(const
    Cable_Problem_Data & Data, const GetPot & Command_Line,
                                                    const libMesh::
                                                        LibMeshInit &
                                                        INIT) :
                                                        Cable_Equation
                                                        (Data,
                                                        Command_Line),
                                                         mesh(INIT.
                                                        comm()),
                                                    equation_systems(
                                                        mesh) { };



void Cable_Equation_CLASSIC::solve_cable_problem_complete() {

  // Create a mesh with user-defined dimension.
  // Read libMesh::Number of elements from command line
  int ps = 200;
  if ( command_line.search(1, "-n") )
       ps = command_line.next(ps);

  // Read FE order from command line
  std::string order = "SECOND";
  if ( command_line.search(2, "-Order", "-o") )
       order = command_line.next(order);

  // Read number non linear loops
  int nl_steps = 200;
  if ( command_line.search(1, "-nll") )
       nl_steps = command_line.next(nl_steps);

  // Read number linear steps
  int l_steps = 500;
```

```
if ( command_line.search(1, "-nl") )
     l_steps = command_line.next(l_steps);

libMesh::Real tol_nl = 1800;
//libMesh::Real tol_exp = -3.;
if ( command_line.search(1, "-tol") )
     tol_nl = command_line.next(tol_nl);

// Generate 1D mesh in the interval [0,L] with number of
   elements ps and order "order".
libMesh::MeshTools::Generation::build_line (mesh,
                                            ps,
                                            0., data.L(),
                                            (order == "FIRST") ?
                                               libMesh::EDGE2 : libMesh
                                               ::EDGE3);

// Printing mesh info to the screen
std::cout<<"*****************************-MESH INFORMATION
   -*********************************"<<std::endl<<std::endl
   ;
mesh.print_info();
std::cout<<"*****************************-MESH INFORMATION
   END-*****************************"<<std::endl<<std::endl;

//Refresh mesh into equation_system
equation_systems.reinit();

// Declare the system and its variables (and relative orders)
libMesh::LinearImplicitSystem & system = equation_systems.
   add_system<libMesh::LinearImplicitSystem> ("Catenary");

unsigned int x_var = system.add_variable ("x", libMesh::
   Utility::string_to_enum<libMesh::Order>   (order));
unsigned int y_var = system.add_variable ("y", libMesh::
   Utility::string_to_enum<libMesh::Order>   (order));

// Give the system a pointer to the assembly function
```

```
system.attach_assemble_function(assemble_cable_linear_CLASSIC)
    ;

// Construct two Dirichlet boundary conditions, one omogeneus
    and the other nonomogeneus object

// Indicate which boundary IDs we impose the BC on
std::set<libMesh::boundary_id_type> boundary_ids_omogeneus;

// the dim==1 mesh has two boundaries with IDs 0 and 1
boundary_ids_omogeneus.insert(0);

// Create a vector storing the variable numbers which the BC
    applies to
std::vector<unsigned int> variables_omogeneus(2);
variables_omogeneus[0] = x_var;
variables_omogeneus[1] = y_var;

// Create a ZeroFunction to initialize dirichlet_bc
libMesh::ZeroFunction<> zf;

// Create a DirichletBoundary object with position, variables
    and values
libMesh::DirichletBoundary dirichlet_bc_omogeneus(
    boundary_ids_omogeneus, variables_omogeneus, &zf);

// We must add the Dirichlet boundary condition _before_
// we call equation_systems.init()
system.get_dof_map().add_dirichlet_boundary(
    dirichlet_bc_omogeneus);

// Construct a Dirichlet non-homogeneus boundary condition
    object

// Indicate which boundary IDs we impose the BC on
std::set<libMesh::boundary_id_type> boundary_ids;

boundary_ids.insert(1);
```

```
// Create a vector storing the variable numbers which the BC
   applies to
std::vector<unsigned int> variables(1);
variables[0] = y_var;

// Create an AnalyticFunction object that we use to project
   the BC
// This function just calls the function exact_solution via
   exact_solution_wrapper
libMesh::AnalyticFunction<> exact_solution_object(
   exact_solution_wrapper);

libMesh::DirichletBoundary dirichlet_bc(boundary_ids,
   variables, &exact_solution_object);

// We must add the Dirichlet boundary condition _before_
// we call equation_systems.init()
system.get_dof_map().add_dirichlet_boundary(dirichlet_bc);


// Also, initialize an ExplicitSystem to store traction
libMesh::ExplicitSystem& stress_system = equation_systems.
   add_system<libMesh::ExplicitSystem> ("TractionSystem");

stress_system.add_variable("T", libMesh::CONSTANT, libMesh::
   MONOMIAL);

// Initialize data structures for equation_system object and
   print its information
equation_systems.init ();

std::cout<<"*****-EQUATION SYSTEM
   INFORMATION-*****"<<std::endl
   <<std::endl;
equation_systems.print_info();
std::cout<<"*****-EQUATION SYSTEM
   INFORMATION END-*****"<<std::endl
   <<std::endl;
```

```
//Create a performance-logging
libMesh::PerfLog perf_log("Systems Catenary");

//Get a reference to the Catenary system
libMesh::LinearImplicitSystem& catenary_static_system =
    equation_systems.get_system<libMesh::LinearImplicitSystem>(
    "Catenary");

// Number of steps and tolerance criterion for the nonlinear
    iterations
const unsigned int n_nonlinear_steps = nl_steps;
const libMesh::Real nonlinear_tolerance = tol_nl;

// It is convenient also to define a max linear solver
    iterations for the linear system when the convergence is
    not certain
equation_systems.parameters.set<unsigned int>("linear solver
    maximum iterations") = l_steps;

//Project an initial solution with an AnalyticFunction object:

// This function just calls the function initial_solution via
    initial_solution_wrapper
libMesh::AnalyticFunction<> initial_solution_object(*
    Cable_Equation::data.initial_solution);

//Using an initial exact solution it is proected in the
    aproximated functional space
system.project_solution(&initial_solution_object);


//Plotting the initial "solution"
libMesh::GnuPlotIO plot1(mesh,"Initial position CLASSIC",
    libMesh::GnuPlotIO::GRID_ON);
plot1.write_equation_systems("gnuplot_script_initial_CLASSIC",
    equation_systems);
```

```
// Get a copy of the nonlinear current iteration solution (to
   test whether to exit or not the loop)
libMesh::AutoPtr<libMesh::NumericVector<libMesh::Number> >
   last_nonlinear_soln (catenary_static_system.solution->clone
   ());

// Setting linear solve tolerance
const libMesh::Real initial_linear_solver_tol = 1.e-10;
equation_systems.parameters.set<libMesh::Real> ("linear solver
   tolerance") = initial_linear_solver_tol;

// Beginning nonlinear loop
for (unsigned int l=0; l<n_nonlinear_steps; ++l) {

  // Update last nonlinear solution
  last_nonlinear_soln->zero();
  last_nonlinear_soln->add(*catenary_static_system.solution);

  // Assemble and solve linear system
  perf_log.push("linear solve");
  equation_systems.get_system("Catenary").solve();
  perf_log.pop("linear solve");

  // Compute the difference between current and last nonlinear
      iterations
  last_nonlinear_soln->add (-1., *catenary_static_system.
     solution);

  last_nonlinear_soln->close();

  // Compute the L2 norm and the H1 of the solution difference
     :

  const libMesh::Real norm_delta = system.calculate_norm(*
     last_nonlinear_soln,libMesh::L2);
  const libMesh::Real normh1 = system.calculate_norm(*
     last_nonlinear_soln,libMesh::H1);
```

```
    // Get the number of iterations required to solve the linear
        system and its final residual
    const unsigned int n_linear_iterations =
       catenary_static_system.n_linear_iterations();

    const libMesh::Real final_linear_residual =
       catenary_static_system.final_linear_residual();

    // Print out the convergence info for both linear and
       nonlinear iterations
    std::cout << "----------Newton: "<<l<<"-th step
       ----------"<<std::endl;
    std::cout << "Linear solver converged at step: " <<
       n_linear_iterations <<std::endl
    << ", final residual: "<< final_linear_residual <<std::endl
    << "  Nonlinear convergence: ||u - u_old||_L2 = " <<
       norm_delta <<std::endl
    << "  Nonlinear convergence: ||u - u_old||_H1 = " << normh1
       << std::endl;
    std::cout << "
       ----------
       "<<std::endl;

    // Terminate solution iteration if the difference between
       last and current nonlinear solutions is sufficiently
       small and if the most recent linear system was solved to
       a sufficient tolerance
    if ((norm_delta < nonlinear_tolerance) && (
       catenary_static_system.final_linear_residual() <
       nonlinear_tolerance)) {

        std::cout << " Nonlinear solver converged at step "
        << l
        << std::endl;
        break;
    }

    // Decrease the linear system tolerance. To obtain the
       quadratic convergence with Newton method the linear
```

```
      system tolerance needs to decrease as we get closer to
      the solution.
    //equation_systems.parameters.set<libMesh::Real> ("linear
      solver tolerance") =
    //std::min(Utility::pow<2>(final_linear_residual),
      initial_linear_solver_tol);

  } // end nonlinear loop

  // Post-process the solution to compute the stresses
  compute_traction();

  libMesh::GnuPlotIO plot(mesh,"Stationary position classic",
     libMesh::GnuPlotIO::GRID_ON);
  plot.write_equation_systems("gnuplot_script_CLASSIC",
     equation_systems);




  return ;

};

                                        /*Weak formulation
                                          coefficients
                                          definition for
                                          CLASSIC method*/

libMesh::Real LA1(const libMesh::Real xl, const libMesh::Real yl
   ) {

  return ( Cable_Equation::data.EA()*(denominator(xl,yl,0.5)-1.)
     *xl/denominator(xl,yl,0.5) );

};

libMesh::Real LA2(const libMesh::Real xl, const libMesh::Real yl
   ) {
```

```
  return ( Cable_Equation::data.EA()*(denominator(xl,yl,0.5)-1.)
     *yl/denominator(xl,yl,0.5) );

};

libMesh::Real LB1(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl) {

  return ( denominator(xl,yl,0.5)*fx(x,y,xl,yl) );

};

libMesh::Real LB2(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl) {

  return ( denominator(xl,yl,0.5)*fy(x,y,xl,yl) );

};


libMesh::Real A11(const libMesh::Real xl,const libMesh::Real yl)
    {

  return  ( Cable_Equation::data.EA()*(denominator(xl,yl,3./2.)-
     pow(yl,2))/(denominator(xl,yl,3./2.)) );

};

libMesh::Real A12(const libMesh::Real xl,const libMesh::Real yl)
    {

  return  ( Cable_Equation::data.EA()*xl*yl/(denominator(xl,yl
     ,3./2.)) );

};

libMesh::Real A21(const libMesh::Real xl,const libMesh::Real yl)
    {
```

```
  return  ( Cable_Equation::data.EA()*xl*yl/(denominator(xl,yl
      ,3./2.)) );

};

libMesh::Real A22(const libMesh::Real xl,const libMesh::Real yl)
    {

  return  ( Cable_Equation::data.EA()*(denominator(xl,yl,3./2.)-
      pow(xl,2.))/(denominator(xl,yl,3./2.)) );

};

libMesh::Real B11(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl) {

  return  ( -2.*Cable_Equation::data.Cd()*sgn(Cable_Equation::
      data.f(y)*yl)*(pow(Cable_Equation::data.f(y),2.))*xl*pow(yl
      ,3.)/(denominator(xl,yl,2.)) );

};

libMesh::Real B12(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl) {

  return  ( Cable_Equation::data.Cd()*sgn(Cable_Equation::data.f
      (y)*yl)*(pow(Cable_Equation::data.f(y),2.))*
              (3.*pow(xl,2.)*pow(yl,2.) + pow(yl,4.))/(denominator
                 (xl,yl,2.)) );

};

libMesh::Real B21(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl) {

  return  ( (Cable_Equation::data.q()*xl/denominator(xl,yl,0.5))
      + Cable_Equation::data.Cd()*
```

```
                sgn(Cable_Equation::data.f(y)*yl)*(pow(
                    Cable_Equation::data.f(y),2.))*(pow(xl,2.)*pow(
                    yl,2.) - pow(yl,4.))/(denominator(xl,yl,2.)) );

};

libMesh::Real B22(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl) {

  return  ( (Cable_Equation::data.q()*yl/denominator(xl,yl,0.5))
      - 2*Cable_Equation::data.Cd()*
                sgn(Cable_Equation::data.f(y)*yl)*(pow(
                    Cable_Equation::data.f(y),2.))*(pow(xl,3.)*yl)/(
                    denominator(xl,yl,2.)) );

};

libMesh::Real C11(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl) { return  0;
   };

libMesh::Real C12(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl) {

  return  ( 2.*Cable_Equation::data.Cd()*sgn(Cable_Equation::
     data.f(y)*yl)*Cable_Equation::data.f(y)*
                Cable_Equation::data.fl(y)*(pow(yl,3.))/(
                   denominator(xl,yl,1.)) );

};

libMesh::Real C21(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl) { return  0;
   };

libMesh::Real C22(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl) {
```

```
  return  ( -2.*Cable_Equation::data.Cd()*sgn(Cable_Equation::
      data.f(y)*yl)*Cable_Equation::data.f(y)*Cable_Equation::
      data.fl(y)*xl*(pow(yl,2))/(denominator(xl,yl,1.)) );

};




                                                /*Assemble
                                                   function for
                                                   Classic
                                                   method*/



void assemble_cable_linear_CLASSIC(libMesh::EquationSystems& es,
    const std::string& system_name) {

  // Confirm if we are assembling the right system
  libmesh_assert_equal_to(system_name, "Catenary");

  // Getting reference to the mesh object
  const libMesh::MeshBase& mesh = es.get_mesh();

  // Get dimension of the problem
  const unsigned int dim = mesh.mesh_dimension();

  // Get reference to the catenary system and its variables
     number ids
  libMesh::LinearImplicitSystem & catenary_static_system = es.
     get_system<libMesh::LinearImplicitSystem> ("Catenary");

  const unsigned int x_var = catenary_static_system.
     variable_number ("x");
  const unsigned int y_var = catenary_static_system.
     variable_number ("y");

  // get finite element type for "x" (same as the "y" type) -
     both are displacement types
```

```
libMesh::FEType fe_disp_type = catenary_static_system.
    variable_type(x_var);

// Build a Finite Element object (pointer) of the specified
    type for the displacement variables
libMesh::AutoPtr<libMesh::FEBase> fe_disp  (libMesh::FEBase::
    build(dim, fe_disp_type ));

// Gauss quadrature rule for numerical integration apropriate
    for the displacement finite element type
libMesh::QGauss qrule (dim, fe_disp_type.
    default_quadrature_order());

// Setting the quadrature rule to the finite elemnt objects (
    fe_disp is a pointer that will refer to different elements)
fe_disp->attach_quadrature_rule (&qrule);

// Defining references to cell-specific data that will be used
     to assemble the linear system:

// Get the Jacobian * (quadratre weight) for each quadrature
    point point
const std::vector<libMesh::Real>& JxW = fe_disp->get_JxW();

// Get the element functions evaluated at the quadrature
    points (first indice is the quadrature point and second is
    the node to witch the shape function is related)
const std::vector<std::vector<libMesh::Real> >& phi = fe_disp
    ->get_phi();

// Get the element functions gradients evaluated at the
    quadrature points (first indice is the quadrature point and
     second is the node to witch the shape function is related)
     - , each element is a libMesh::RealGradient type (dphi[qp
    ][i](j) is the gradient of the shape function related to
    the ith node evaluated in the qpth quadrature point in the
    jth direction)
const std::vector<std::vector<libMesh::RealGradient> >& dphi =
     fe_disp->get_dphi();
```

```
// A reference to the DofMap object for this system (this
   object handles the index translation from node and element
   numbers to degree of freedom)
const libMesh::DofMap & dof_map = catenary_static_system.
   get_dof_map();

// Data structures to contain the element mathix and right-
   hand-side vector (rhs) contribution.
libMesh::DenseMatrix<libMesh::Number> Ke;
libMesh::DenseVector<libMesh::Number> Fe;


libMesh::DenseSubMatrix<libMesh::Number>
    Kxx(Ke), Kxy(Ke),
    Kyx(Ke), Kyy(Ke);

libMesh::DenseSubVector<libMesh::Number>
    Fx(Fe), Fy(Fe);

// This vector will hold the element dof indices (where in the
    global system the element dof get mapped)
std::vector<libMesh::dof_id_type> dof_indices;
std::vector<libMesh::dof_id_type> dof_indices_x;
std::vector<libMesh::dof_id_type> dof_indices_y;

// Get element iterator (the active is used for mesh-
   refinement situations)
libMesh::MeshBase::const_element_iterator       el     = mesh.
   active_local_elements_begin();
const libMesh::MeshBase::const_element_iterator end_el = mesh.
   active_local_elements_end();

for ( ; el != end_el; ++el) {

  // Store the element we are working at in a pointer (elem)
  const libMesh::Elem* elem = *el;
```

```
    // Get the global dof for the current element and the size
       of them (how many nodes in each element)
    dof_map.dof_indices (elem, dof_indices);
    dof_map.dof_indices (elem, dof_indices_x, x_var);
    dof_map.dof_indices (elem, dof_indices_y, y_var);

    const unsigned int n_dofs   = dof_indices.size();
    const unsigned int n_x_dofs = dof_indices_x.size();
    const unsigned int n_y_dofs = dof_indices_y.size();

    //Compute the cell-specific data mentioned earlier
    fe_disp->reinit  (elem);

    //Prevent cases in witch there exixts diferent elements
       types in mesh (triangles and quadrilateral)
    Ke.resize (n_dofs, n_dofs);
    Fe.resize (n_dofs);

    //The DenseSubMatrix.repostition () member takes the (
       row_offset, column_offset, row_size, column_size).
    //Similarly, the DenseSubVector.reposition () member takes
       the (row_offset, row_size)
    Kxx.reposition (x_var*n_x_dofs, x_var*n_x_dofs, n_x_dofs,
       n_x_dofs);
    Kxy.reposition (x_var*n_x_dofs, y_var*n_x_dofs, n_x_dofs,
       n_y_dofs);

    Kyx.reposition (y_var*n_x_dofs, x_var*n_y_dofs, n_y_dofs,
       n_x_dofs);
    Kyy.reposition (y_var*n_x_dofs, y_var*n_x_dofs, n_y_dofs,
       n_y_dofs);


    Fx.reposition (x_var*n_x_dofs, n_x_dofs);
    Fy.reposition (y_var*n_y_dofs, n_y_dofs);

    //Build element matrix and RHS using numerical integration (
       note that the previus step solution is required hear)
    for (unsigned int qp=0; qp<qrule.n_points(); qp++) {
```

```
// Values to hold previus solution and its gradient
libMesh::Number   x = 0.;
libMesh::Number   y = 0.;

libMesh::Gradient grad_x;
libMesh::Gradient grad_y;

//Compute previus Newton iterate solution and gradients on
    the current quadrature point
for (unsigned int l=0; l<n_x_dofs; l++) {

  x += phi[l][qp]*catenary_static_system.current_solution
     (dof_indices_x[l]);

  y += phi[l][qp]*catenary_static_system.current_solution
     (dof_indices_y[l]);

  grad_x.add_scaled (dphi[l][qp],catenary_static_system.
     current_solution (dof_indices_x[l]));

  grad_y.add_scaled (dphi[l][qp],catenary_static_system.
     current_solution (dof_indices_y[l]));

}

// Store computed values
const libMesh::Real  xl = grad_x(0);
const libMesh::Real  yl = grad_y(0);


for (unsigned int i=0; i<n_x_dofs; i++) {

  Fx(i) += JxW[qp]*(LA1(xl,yl)*dphi[i][qp](0) - LB1(x,y,xl
     ,yl)*phi[i][qp] - A11(xl,yl)*xl*dphi[i][qp](0) - A12(
     xl,yl)*yl*dphi[i][qp](0) +
                    B11(x,y,xl,yl)*xl*phi[i][qp] + B12(x,y
                       ,xl,yl)*yl*phi[i][qp] + C11(x,y,xl,
                       yl)*x*phi[i][qp] + C12(x,y,xl,yl)*y
```

```
                        *phi[i][qp]);

    Fy(i) += JxW[qp]*(LA2(xl,yl)*dphi[i][qp](0) - LB2(x,y,xl
        ,yl)*phi[i][qp] - A21(xl,yl)*xl*dphi[i][qp](0) -
                    A22(xl,yl)*yl*dphi[i][qp](0) + B21(x,y
                        ,xl,yl)*xl*phi[i][qp] + B22(x,y,xl,
                        yl)*yl*phi[i][qp] + C21(x,y,xl,yl)*
                        x*phi[i][qp] +
                    C22(x,y,xl,yl)*y*phi[i][qp]);



    for (unsigned int j=0; j<n_x_dofs; j++) {

      Kxx(i,j) += JxW[qp]*((-A11(xl,yl)*dphi[j][qp](0)*dphi[
          i][qp](0)) + (B11(x,y,xl,yl)*dphi[j][qp](0)*phi[i][
          qp]) +
                              (C11(x,y,xl,yl)*phi[j][qp]*phi[i
                                  ][qp]) );    // Newton term

      Kxy(i,j) += JxW[qp]*((-A12(xl,yl)*dphi[j][qp](0)*dphi[
          i][qp](0)) + (B12(x,y,xl,yl)*dphi[j][qp](0)*phi[i][
          qp]) +
                              (C12(x,y,xl,yl)*phi[j][qp]*phi[i
                                  ][qp]) );    // Newton term

      Kyx(i,j) += JxW[qp]*((-A21(xl,yl)*dphi[j][qp](0)*dphi[
          i][qp](0)) + (B21(x,y,xl,yl)*dphi[j][qp](0)*phi[i][
          qp]) +
                              (C21(x,y,xl,yl)*phi[j][qp]*phi[i
                                  ][qp]) );    // Newton term

      Kyy(i,j) += JxW[qp]*((-A22(xl,yl)*dphi[j][qp](0)*dphi[
          i][qp](0)) + (B22(x,y,xl,yl)*dphi[j][qp](0)*phi[i][
          qp]) +
                              (C22(x,y,xl,yl)*phi[j][qp]*phi[i
                                  ][qp]) );    // Newton term

    }
```

```
      }

    } // end of the quadrature point qp-loop

    dof_map.heterogenously_constrain_element_matrix_and_vector (
       Ke, Fe, dof_indices);

    catenary_static_system.matrix->add_matrix (Ke, dof_indices);
    catenary_static_system.rhs->add_vector    (Fe, dof_indices);

  } // end of element loop

  return;

};


                                        /*Post-processing
                                           function for classic
                                           method to calculate
                                           traction*/


void Cable_Equation_CLASSIC::compute_traction() {

  //Getting mesh reference
  const libMesh::MeshBase& mesh = equation_systems.get_mesh();

  //Getting dimension of problem
  const unsigned int dim = mesh.mesh_dimension();

  //Getting reference to the "Elasticity" system that should be
     already solved
  libMesh::LinearImplicitSystem& system = equation_systems.
     get_system<libMesh::LinearImplicitSystem>("Catenary");

  //Getting variables numbers
  unsigned int displacement_vars[2];
```

```
displacement_vars[0] = system.variable_number ("x");
displacement_vars[1] = system.variable_number ("y");
const unsigned int u_var = system.variable_number ("x");

//Getting dof map, fe type, creating fe Auto pointer and
   attach quadrature rule
const libMesh::DofMap& dof_map = system.get_dof_map();
libMesh::FEType fe_type = dof_map.variable_type(u_var);
libMesh::AutoPtr<libMesh::FEBase> fe (libMesh::FEBase::build(
   dim, fe_type));
libMesh::QGauss qrule (dim, fe_type.default_quadrature_order()
   );
fe->attach_quadrature_rule (&qrule);

//getting Jacobianxweight and shape function and its gradients
    references
const std::vector<libMesh::Real>& JxW = fe->get_JxW();
const std::vector<std::vector<libMesh::Real> >& phi = fe->
   get_phi();
const std::vector<std::vector<libMesh::RealGradient> >& dphi =
    fe->get_dphi();


// Also, get a reference to the ExplicitSystem
libMesh::ExplicitSystem& traction_system = equation_systems.
   get_system<libMesh::ExplicitSystem>("TractionSystem");
const libMesh::DofMap& traction_dof_map = traction_system.
   get_dof_map();
unsigned int T_var = traction_system.variable_number("T");

// Storage for the stress dof indices on each element
std::vector< std::vector<libMesh::dof_id_type> >
   dof_indices_var(system.n_vars());
std::vector<libMesh::dof_id_type> traction_dof_indices_var;

// To store the stress tensor on each element
libMesh::Real elem_sigma;

//element iterator to loop all over the mesh
```

```
libMesh::MeshBase::const_element_iterator el = mesh.
    active_local_elements_begin();
const libMesh::MeshBase::const_element_iterator end_el = mesh.
    active_local_elements_end();

//mesh elements loop
for ( ; el != end_el; ++el) {

  const libMesh::Elem* elem = *el;

  //getting dof indices map for the displacement variables
  for(unsigned int var=0; var<2; var++)
    dof_map.dof_indices (elem, dof_indices_var[var],
        displacement_vars[var]);


  //reinitilize element properties
  fe->reinit (elem);

  //set to zero storing element traction vector
  elem_sigma = 0;

  //loop over the quadrature points (it is performed an
     integration of the traction on each element and after
  //it, the integral is divided by the element area (volume or
      length))
  for (unsigned int qp=0; qp<qrule.n_points(); qp++) {

    //Getting quadrature point r-coordinate and dof number of
       the element on each variable
    const unsigned int n_x_dofs = dof_indices_var[0].size();
    const unsigned int n_y_dofs = dof_indices_var[1].size();


    // Get the gradients at this quadrature point:
    //(it is the sum of all shape function gradients weighted
       by the respective dof displacement solution)
    libMesh::Gradient displacement_gradient_x;
    libMesh::Gradient displacement_gradient_y;
```

```
      for(unsigned int l=0; l<n_x_dofs; l++)
        displacement_gradient_x.add_scaled(dphi[l][qp], system.
           current_solution(dof_indices_var[0][l]));


      for(unsigned int l=0; l<n_y_dofs; l++)
        displacement_gradient_y.add_scaled(dphi[l][qp], system.
           current_solution(dof_indices_var[1][l]));



      //The integration is done to have more precise stress
         constants on each element (adding quadrature point
         contribute):
      //to understand those terms see the elastic law and
         congruence for axisymmetrical problems in cylindrical
         coordinates
      elem_sigma += JxW[qp]*(data.EA()*(sqrt(
         displacement_gradient_x(0)*displacement_gradient_x(0) +
          displacement_gradient_y(0)*displacement_gradient_y(0))
         -1));



    }

    // Get the average stresses by dividing by the element
       volume
    elem_sigma = elem_sigma/(elem->volume()*1000);

    // load elem_sigma data into traction_system

    //getting dof indices map for the current variable
    traction_dof_map.dof_indices (elem, traction_dof_indices_var
       , T_var);

    // We are using CONSTANT MONOMIAL basis functions, hence we
       only need to get
    // one dof index per variable
```

```
    libMesh::dof_id_type dof_index = traction_dof_indices_var
        [0];

    //setting the solution
    if( (traction_system.solution->first_local_index() <=
        dof_index) && (dof_index < traction_system.solution->
        last_local_index()) )
      traction_system.solution->set(dof_index, elem_sigma);


  }

  // Should call close and update when we set vector entries
      directly
  traction_system.solution->close();
  traction_system.update();
}


//END Cable_Equation_CLASSIC functions:


/*HERE*/


                                                    //
                                                        Cable_Equation_mixed
                                                         functions:


Cable_Equation_MIXED::Cable_Equation_MIXED(const
    Cable_Problem_Data & Data, const GetPot & Command_Line,
                                                    const libMesh::
                                                        LibMeshInit & INIT
                                                        ) : Cable_Equation
                                                        (Data,Command_Line
                                                        ),
                                                    mesh(INIT.comm()),
                                                        equation_systems(
                                                        mesh) { };
```

```
void Cable_Equation_MIXED::solve_cable_problem_complete() {

  // Create a mesh with user-defined dimension.
  // Read number of elements from command line
  int ps = 200;
  if ( command_line.search(1, "-n") )
    ps = command_line.next(ps);

  // Read FE order from command line
  std::string order = "SECOND";
  if ( command_line.search(2, "-Order", "-o") )
    order = command_line.next(order);

  // Read FE order from command line
  std::string order_t = "CONSTANT";
  if ( command_line.search(2, "-Ordert", "-ot") )
    order_t = command_line.next(order_t);

  // Read number non linear loops
  int nl_steps = 200;
  if ( command_line.search(1, "-nll") )
    nl_steps = command_line.next(nl_steps);

     // Read number linear steps
  int l_steps = 500;
  if ( command_line.search(1, "-nl") )
    l_steps = command_line.next(l_steps);

  libMesh::Real tol_nl = Cable_Equation::data.L();
  if ( command_line.search(1, "-tol") )
    tol_nl = command_line.next(tol_nl);

  // Generate 1D mesh in the interval [0,L] with number of
     elements ps and order "order".
  libMesh::MeshTools::Generation::build_line (mesh,
                                              ps,
```

```
                                                    0., Cable_Equation
                                                        ::data.L(),
                                                    (order == "FIRST")
                                                        ? libMesh::
                                                      EDGE2 : libMesh
                                                        ::EDGE3);


// Printing mesh info to the screen
std::cout<<"*****************************-MESH INFORMATION
    -*********************************"<<std::endl<<std::endl
    ;
mesh.print_info();
std::cout<<"*****************************-MESH INFORMATION
    END-*****************************"<<std::endl<<std::endl;

//Refresh mesh into equation_system
equation_systems.reinit();

// Declare the system and its variables (and relative orders)
libMesh::LinearImplicitSystem & system = equation_systems.
    add_system<libMesh::LinearImplicitSystem> ("Catenary");

unsigned int x_var = system.add_variable ("x", libMesh::
    Utility::string_to_enum<libMesh::Order>    (order));
unsigned int y_var = system.add_variable ("y", libMesh::
    Utility::string_to_enum<libMesh::Order>    (order));
unsigned int t_var;

if (order_t == "FIRST")
  t_var = system.add_variable ("t", libMesh::Utility::
      string_to_enum<libMesh::Order>    (order_t));

else
  t_var = system.add_variable ("t", libMesh::CONSTANT, libMesh
      ::MONOMIAL);

// Give the system a pointer to the assembly function
system.attach_assemble_function (assemble_cable_linear_MIXED);
```

```
// Construct two Dirichlet boundary conditions, one omogeneus
   and the other nonomogeneus object

// Indicate which boundary IDs we impose the BC on
std::set<libMesh::boundary_id_type> boundary_ids_omogeneus;

// the dim==1 mesh has two boundaries with IDs 0 and 1
boundary_ids_omogeneus.insert(0);

// Create a vector storing the variable numbers which the BC
   applies to
std::vector<unsigned int> variables_omogeneus(2);
variables_omogeneus[0] = x_var;
variables_omogeneus[1] = y_var;

// Create a ZeroFunction to initialize dirichlet_bc
libMesh::ZeroFunction<> zf;

// Create a DirichletBoundary object with position, variables
   and values
libMesh::DirichletBoundary dirichlet_bc_omogeneus(
   boundary_ids_omogeneus, variables_omogeneus, &zf);

// We must add the Dirichlet boundary condition _before_
// we call equation_systems.init()
system.get_dof_map().add_dirichlet_boundary(
   dirichlet_bc_omogeneus);

// Construct a Dirichlet boundary condition object

// Indicate which boundary IDs we impose the BC on
std::set<libMesh::boundary_id_type> boundary_ids;

boundary_ids.insert(1);

// Create a vector storing the variable numbers which the BC
   applies to
```

```
std::vector<unsigned int> variables(1);
variables[0] = y_var;

// Create an AnalyticFunction object that we use to project
   the BC
// This function just calls the function exact_solution via
   exact_solution_wrapper
libMesh::AnalyticFunction<> exact_solution_object(
   exact_solution_wrapper);

libMesh::DirichletBoundary dirichlet_bc(boundary_ids,
   variables, &exact_solution_object);

// We must add the Dirichlet boundary condition _before_
// we call equation_systems.init()
system.get_dof_map().add_dirichlet_boundary(dirichlet_bc);


// Initialize data structures for equation_system object and
   print its information
equation_systems.init ();

equation_systems.print_info();

//Create a performance-logging
libMesh::PerfLog perf_log("Systems Catenary");

//Get a reference to the Catenary system
libMesh::LinearImplicitSystem& catenary_static_system =
   equation_systems.get_system<libMesh::LinearImplicitSystem>(
   "Catenary");

// Number of steps and tolerance criterion for the nonlinear
   iterations
const unsigned int n_nonlinear_steps = nl_steps;
const libMesh::Real nonlinear_tolerance = tol_nl;

// It is convenient also to define a max linear solver
   iterations for the linear system when the convergence is
```

```
    not certain
equation_systems.parameters.set<unsigned int>("linear solver
    maximum iterations") = l_steps;

//Project an initial solution with an AnalyticFunction object:

// This function just calls the function initial_solution via
    initial_solution_wrapper
libMesh::AnalyticFunction<> initial_solution_object(*
    Cable_Equation::data.initial_solution);

// An initial exact solution is projected in the aproximated
    functional space
system.project_solution(&initial_solution_object);

//Plotting the initial "solution"
libMesh::GnuPlotIO plot1(mesh,"Initial position",libMesh::
    GnuPlotIO::GRID_ON);
plot1.write_equation_systems("gnuplot_script_initial_MIXED",
    equation_systems);

// Get a copy of the nonlinear current iteration solution (to
    test whether to exit or not the loop)
libMesh::AutoPtr<libMesh::NumericVector<libMesh::Number> >
    last_nonlinear_soln (catenary_static_system.solution->clone
    ());

// Setting linear solve tolerance
const libMesh::Real initial_linear_solver_tol = 1.e-10;
equation_systems.parameters.set<libMesh::Real> ("linear solver
    tolerance") = initial_linear_solver_tol;

// Beginning nonlinear loop
for (unsigned int l=0; l<n_nonlinear_steps; ++l) {

  // Update last nonlinear solution
  last_nonlinear_soln->zero();
  last_nonlinear_soln->add(*catenary_static_system.solution);
```

```
// Assemble and solve linear system
perf_log.push("linear solve");
equation_systems.get_system("Catenary").solve();
perf_log.pop("linear solve");


// Compute the difference between current and last nonlinear
    iterations
last_nonlinear_soln->add (-1., *catenary_static_system.
   solution);

last_nonlinear_soln->close();

// Compute the L2 norm and the H1 of the solution difference
   :

const libMesh::Real norm_delta_x = system.calculate_norm(*
   last_nonlinear_soln,0,libMesh::L2);
const libMesh::Real norm_delta_y = system.calculate_norm(*
   last_nonlinear_soln,1,libMesh::L2);
const libMesh::Real norm_delta_t = system.calculate_norm(*
   last_nonlinear_soln,2,libMesh::L2);
const libMesh::Real norm_delta = sqrt(norm_delta_x*
   norm_delta_x + norm_delta_y*norm_delta_y);
const libMesh::Real norm_delta_all = system.calculate_norm(*
   last_nonlinear_soln,libMesh::L2);

// Get the number of iterations required to solve the linear
    system and its final residual
const unsigned int n_linear_iterations =
   catenary_static_system.n_linear_iterations();

const libMesh::Real final_linear_residual =
   catenary_static_system.final_linear_residual();

// Print out the convergence info for both linear and
   nonlinear iterations
```

```
    std::cout << "-------------Newton: "<<l<<"-th step
       ------------------------------------------ "<<std::endl;
    std::cout << "Linear solver converged at step: " <<
       n_linear_iterations <<std::endl
    << ", final residual: "<< final_linear_residual <<std::endl
    << "  Nonlinear convergence: ||u - u_old||_L2 = " <<
       norm_delta_all <<std::endl
    << "  Nonlinear convergence: ||xy - xy_old||_L2 = " <<
       norm_delta <<std::endl
    << "  Nonlinear convergence: ||t - t_old||_L2 = " <<
       norm_delta_t <<std::endl;
    std::cout << "
       ------------------------------------------------------------------------------
       "<<std::endl;

    // Terminate solution iteration if the difference between
       last and current nonlinear solutions is sufficiently
       small and if the most recent linear system was solved to
       a sufficient tolerance
    if ((norm_delta < nonlinear_tolerance) && (
       catenary_static_system.final_linear_residual() <
       nonlinear_tolerance)) {

      std::cout << " Nonlinear solver converged at step "
      << l
      << std::endl;
      break;

    }

    // Decrease the linear system tolerance. To obtain the
       quadratic convergence with Newton method the linear
       system tolerance needs to decrease as we get closer to
       the solution.
    //equation_systems.parameters.set<libMesh::Real> ("linear
       solver tolerance") =
    //std::min(Utility::pow<2>(final_linear_residual),
       initial_linear_solver_tol);
```

```
  } // end nonlinear loop


  libMesh::GnuPlotIO plot(mesh,"Stationary position mexed
     formulation",libMesh::GnuPlotIO::GRID_ON);
  plot.write_equation_systems("gnuplot_script_MIXED",
     equation_systems);


  return ;

};


                                        /*Weak formulation
                                           coefficients
                                           definition for MIXED
                                           method*/

libMesh::Real LA1(const libMesh::Real xl, const libMesh::Real yl
   , const libMesh::Real t) {

  return ( Cable_Equation::data.EA()*xl*((t*1000.)/(
     Cable_Equation::data.EA()+(t*1000.))) );

};

libMesh::Real LA2(const libMesh::Real xl, const libMesh::Real yl
   , const libMesh::Real t) {

  return ( Cable_Equation::data.EA()*yl*((t*1000.)/(
     Cable_Equation::data.EA()+(t*1000.))) );

};

libMesh::Real LB1(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {
```

```
  return ( (1.+((t*1000.)/Cable_Equation::data.EA()))*fx(x,y,xl,
     yl) );

};

libMesh::Real LB2(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {

  return ( (1.+((t*1000.)/Cable_Equation::data.EA()))*fy(x,y,xl,
     yl) );

};

libMesh::Real LB3(const libMesh::Real x, const libMesh::Real y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {

  return ( (0.5)*denominator(xl,yl,2.) - (0.5)*(pow((1.+((t
     *1000.)/Cable_Equation::data.EA())),2)) );

};


libMesh::Real A11(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t) {

  return  ( Cable_Equation::data.EA()*((t*1000.)/(Cable_Equation
     ::data.EA()+(t*1000.))) );

};

libMesh::Real A12(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t) {

  return  0;

};
```

```
libMesh::Real A13(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t) {

  return ( xl*pow((Cable_Equation::data.EA()/(Cable_Equation::
     data.EA()+(t*1000.))),2.) );

};

libMesh::Real A21(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t) {

  return 0;

};

libMesh::Real A22(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t) {

  return ( Cable_Equation::data.EA()*((t*1000.)/(Cable_Equation
     ::data.EA()+(t*1000.))) );

};

libMesh::Real A23(const libMesh::Real xl,const libMesh::Real yl,
    const libMesh::Real t) {

  return ( yl*pow((Cable_Equation::data.EA()/(Cable_Equation::
     data.EA()+(t*1000.))),2.) );

};

libMesh::Real B11(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {

  return ( -3.*(1.+((t*1000.)/Cable_Equation::data.EA()))*
     Cable_Equation::data.Cd()*
              sgn(Cable_Equation::data.f(y)*yl)*(pow(
                 Cable_Equation::data.f(y),2.))*xl*pow(yl,3.)/(
```

```
                denominator(xl,yl,5./2.)) );

  };

libMesh::Real B12(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {
  return  ( 3.*(1+((t*1000.)/Cable_Equation::data.EA()))*
     Cable_Equation::data.Cd()*
            sgn(Cable_Equation::data.f(y)*yl)*(pow(
               Cable_Equation::data.f(y),2.))*(pow(xl,2.)*pow(yl
               ,2.))/(denominator(xl,yl,5./2.)) );

};

libMesh::Real B13(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) { return 0.; };

libMesh::Real B21(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {

  return  ( (1.+((t*1000.)/Cable_Equation::data.EA()))*
     Cable_Equation::data.Cd()*
             sgn(Cable_Equation::data.f(y)*yl)*(pow(
                Cable_Equation::data.f(y),2.))*(2*pow(xl,2.)*pow
                (yl,2.) - pow(yl,4.))/(denominator(xl,yl,5./2))
                );

};

libMesh::Real B22(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {

  return  ( -(1.+((t*1000.)/Cable_Equation::data.EA()))*
     Cable_Equation::data.Cd()*
```

```
                sgn(Cable_Equation::data.f(y)*yl)*(pow(
                    Cable_Equation::data.f(y),2.))*(2*pow(xl,3.)*yl
                    - xl*pow(yl,3.))/(denominator(xl,yl,5./2)) );

};

libMesh::Real B23(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) { return 0.; };

libMesh::Real C11(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) { return  0; };

libMesh::Real C12(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {

  return  ( 2.*(1+((t*1000.)/Cable_Equation::data.EA()))*
     Cable_Equation::data.Cd()*
            sgn(Cable_Equation::data.f(y)*yl)*Cable_Equation::
               data.f(y)*Cable_Equation::data.fl(y)*(pow(yl,3.))
               /(denominator(xl,yl,3./2.)) );

};

libMesh::Real C13(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {

  return  ( fx(x,y,xl,yl)/Cable_Equation::data.EA() );

};

libMesh::Real C21(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t ){ return  0; };
```

```
libMesh::Real C22(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t) {

  return  ( -2.*(1.+((t*1000.)/Cable_Equation::data.EA()))*
     Cable_Equation::data.Cd()*
              sgn(Cable_Equation::data.f(y)*yl)*Cable_Equation::
                 data.f(y)*Cable_Equation::data.fl(y)*xl*(pow(yl
                 ,2.))/(denominator(xl,yl,3./2.)) );

};

libMesh::Real C23(const libMesh::Real x,const libMesh::Real  y,
   const libMesh::Real xl, const libMesh::Real yl, const libMesh
   ::Real t){ return  ( fy(x,y,xl,yl)/Cable_Equation::data.EA()
   ); };

void assemble_cable_linear_MIXED(libMesh::EquationSystems& es,
   const std::string& system_name) {

  // Confirm if we are assembling the right system
  libmesh_assert_equal_to (system_name, "Catenary");

  // Getting reference to the mesh object
  const libMesh::MeshBase& mesh = es.get_mesh();

  // Get dimension of the problem
  const unsigned int dim = mesh.mesh_dimension();

  // Get reference to the catenary system and its variables
     number ids
  libMesh::LinearImplicitSystem& catenary_static_system = es.
     get_system<libMesh::LinearImplicitSystem> ("Catenary");

  const unsigned int x_var = catenary_static_system.
     variable_number ("x");
  const unsigned int y_var = catenary_static_system.
     variable_number ("y");
```

```
const unsigned int t_var = catenary_static_system.
    variable_number ("t");

// get finite element type for "x" (same as the "y" type) -
    both are displacement types
libMesh::FEType fe_disp_type = catenary_static_system.
    variable_type(x_var);

// get finite element type for "T"
libMesh::FEType fe_trac_type = catenary_static_system.
    variable_type(t_var);

// Build a Finite Element object (pointer) of the specified
    type for the displacement variables
libMesh::AutoPtr<libMesh::FEBase> fe_disp  (libMesh::FEBase::
    build(dim, fe_disp_type ));

// Build a Finite Element object (pointer) of the specified
    type for the traction variable
libMesh::AutoPtr<libMesh::FEBase> fe_trac  (libMesh::FEBase::
    build(dim, fe_trac_type ));

// Gauss quadrature rule for numerical integration apropriate
    for the displacement finite element type
libMesh::QGauss qrule (dim, fe_disp_type.
    default_quadrature_order());

// Setting the quadrature rule to the finite elemnt objects (
    fe_disp is a pointer that will refer to different elements)
fe_disp->attach_quadrature_rule (&qrule);
fe_trac->attach_quadrature_rule (&qrule);

// Defining references to cell-specific data that will be used
     to assemble the linear system:

// Get the Jacobian * (quadratre weight) for each quadrature
    point point
const std::vector<libMesh::Real>& JxW = fe_disp->get_JxW();
```

```
// Get the element functions evaluated at the quadrature
   points (first indice is the node to witch the shape
   function is related and second is the quadrature point)
const std::vector<std::vector<libMesh::Real> >& phi = fe_disp
   ->get_phi();

// Get the element functions gradients evaluated at the
   quadrature points (first indice is the node to witch the
   shape function is related and second is the quadrature
   point) - , each element is a libMesh::RealGradient type (
   dphi[qp][i](j) is the gradient of the shape function
   related to the ith node evaluated in the qpth quadrature
   point in the jth direction)
const std::vector<std::vector<libMesh::RealGradient> >& dphi =
    fe_disp->get_dphi();

// Same for traction
const std::vector<std::vector<libMesh::Real> >& psi = fe_trac
   ->get_phi();

// A reference to the DofMap object for this system (this
   object handles the index translation from node and element
   numbers to degree of freedom)
const libMesh::DofMap & dof_map = catenary_static_system.
   get_dof_map();

// Data structures to contain the element mathix and right-
   hand-side vector (rhs) contribution.
libMesh::DenseMatrix<libMesh::Number> Ke;
libMesh::DenseVector<libMesh::Number> Fe;


libMesh::DenseSubMatrix<libMesh::Number>
    Kxx(Ke), Kxy(Ke), Kxt(Ke),
    Kyx(Ke), Kyy(Ke), Kyt(Ke),
    Ktx(Ke), Kty(Ke), Ktt(Ke);

libMesh::DenseSubVector<libMesh::Number>
  Fx(Fe), Fy(Fe), Ft(Fe);
```

```
// This vector will hold the element dof indices (where in the
    global system the element dof get mapped)
std::vector<libMesh::dof_id_type> dof_indices;
std::vector<libMesh::dof_id_type> dof_indices_x;
std::vector<libMesh::dof_id_type> dof_indices_y;
std::vector<libMesh::dof_id_type> dof_indices_t;

// Get element iterator (the active is used for mesh-
   refinement situations)
libMesh::MeshBase::const_element_iterator       el     = mesh.
   active_local_elements_begin();
const libMesh::MeshBase::const_element_iterator end_el = mesh.
   active_local_elements_end();

for ( ; el != end_el; ++el) {

  // Store the element we are working at in a pointer (elem)
  const libMesh::Elem* elem = *el;

  // Get the global dof for the current element and the size
     of them (how many nodes in each element)
  dof_map.dof_indices (elem, dof_indices);
  dof_map.dof_indices (elem, dof_indices_x, x_var);
  dof_map.dof_indices (elem, dof_indices_y, y_var);
  dof_map.dof_indices (elem, dof_indices_t, t_var);

  const unsigned int n_dofs   = dof_indices.size();
  const unsigned int n_x_dofs = dof_indices_x.size();
  const unsigned int n_y_dofs = dof_indices_y.size();
  const unsigned int n_t_dofs = dof_indices_t.size();

  //Compute the cell-specific data mentioned earlier
  fe_disp->reinit  (elem);
  fe_trac->reinit (elem);

  //Prevent cases in witch there exixts diferent elements
     types in mesh (triangles and quadrilateral)
  Ke.resize (n_dofs, n_dofs);
```

```
    Fe.resize (n_dofs);

    //The DenseSubMatrix.repostition () member takes the (
       row_offset, column_offset, row_size, column_size).
    //Similarly, the DenseSubVector.reposition () member takes
       the (row_offset, row_size)
    Kxx.reposition (x_var*n_x_dofs, x_var*n_x_dofs, n_x_dofs,
       n_x_dofs);
    Kxy.reposition (x_var*n_x_dofs, y_var*n_x_dofs, n_x_dofs,
       n_y_dofs);
    Kxt.reposition (x_var*n_x_dofs, t_var*n_x_dofs, n_x_dofs,
       n_t_dofs);

    Kyx.reposition (y_var*n_x_dofs, x_var*n_x_dofs, n_y_dofs,
       n_x_dofs);
    Kyy.reposition (y_var*n_x_dofs, y_var*n_x_dofs, n_y_dofs,
       n_y_dofs);
    Kyt.reposition (y_var*n_x_dofs, t_var*n_x_dofs, n_y_dofs,
       n_t_dofs);

    Ktx.reposition (t_var*n_x_dofs, x_var*n_x_dofs, n_t_dofs,
       n_x_dofs);
    Kty.reposition (t_var*n_x_dofs, y_var*n_x_dofs, n_t_dofs,
       n_y_dofs);
    Ktt.reposition (t_var*n_x_dofs, t_var*n_x_dofs, n_t_dofs,
       n_t_dofs);


    Fx.reposition (x_var*n_x_dofs, n_x_dofs);
    Fy.reposition (y_var*n_x_dofs, n_y_dofs);
    Ft.reposition (t_var*n_x_dofs, n_t_dofs);

    //Build element matrix and RHS using numerical integration (
       note that the previus step solution is required hear)
    for (unsigned int qp=0; qp<qrule.n_points(); qp++) {

      // Values to hold previus solution and its gradient
      libMesh::Number   x = 0.;
      libMesh::Number   y = 0.;
```

```
libMesh::Number   t = 0.;

libMesh::Gradient grad_x;
libMesh::Gradient grad_y;

//Compute previus Newton iterate solution and gradients on
    the current quadrature point
for (unsigned int l=0; l<n_x_dofs; l++) {

  x += phi[l][qp]*catenary_static_system.current_solution
     (dof_indices_x[l]);
  y += phi[l][qp]*catenary_static_system.current_solution
     (dof_indices_y[l]);
  grad_x.add_scaled (dphi[l][qp],catenary_static_system.
     current_solution (dof_indices_x[l]));
  grad_y.add_scaled (dphi[l][qp],catenary_static_system.
     current_solution (dof_indices_y[l]));

}

//Same for traction
for (unsigned int l=0; l<n_t_dofs; l++)
  t += psi[l][qp]*catenary_static_system.current_solution
     (dof_indices_t[l]);

// Store computed values
const libMesh::Real  xl = grad_x(0);
const libMesh::Real  yl = grad_y(0);

for (unsigned int i=0; i<n_x_dofs; i++) {

  Fx(i) += JxW[qp]*(LA1(xl,yl,t)*dphi[i][qp](0) - LB1(x,y,
     xl,yl,t)*phi[i][qp] - A11(xl,yl,t)*xl*dphi[i][qp](0)
     -
                    A12(xl,yl,t)*yl*dphi[i][qp](0) - A13(
                       xl,yl,t)*(t*1000.)*dphi[i][qp](0) +
                        B11(x,y,xl,yl,t)*xl*phi[i][qp] +
                    B12(x,y,xl,yl,t)*yl*phi[i][qp] + C12(x
                       ,y,xl,yl,t)*y*phi[i][qp] + C13(x,y,
```

```
                                xl,yl,t)*(t*1000.)*phi[i][qp]);

      Fy(i) += JxW[qp]*(LA2(xl,yl,t)*dphi[i][qp](0) - LB2(x,y,
          xl,yl,t)*phi[i][qp] - A21(xl,yl,t)*xl*dphi[i][qp](0)
          -
                              A22(xl,yl,t)*yl*dphi[i][qp](0) - A23(
                                  xl,yl,t)*(t*1000.)*dphi[i][qp](0) +
                                   B21(x,y,xl,yl,t)*xl*phi[i][qp] +
                              B22(x,y,xl,yl,t)*yl*phi[i][qp] + C21(x
                                  ,y,xl,yl,t)*x*phi[i][qp] + C22(x,y,
                                  xl,yl,t)*y*phi[i][qp] +
                              C23(x,y,xl,yl,t)*(t*1000.)*phi[i][qp])
                                  ;

      for (unsigned int j=0; j<n_x_dofs; j++) {

        Kxx(i,j) += JxW[qp]*((-A11(xl,yl,t)*dphi[j][qp](0)*
            dphi[i][qp](0)) + (B11(x,y,xl,yl,t)*dphi[j][qp](0)*
            phi[i][qp]) +
                                (C11(x,y,xl,yl,t)*phi[j][qp]*phi
                                    [i][qp]) );    // Newton term

        Kxy(i,j) += JxW[qp]*((-A12(xl,yl,t)*dphi[j][qp](0)*
            dphi[i][qp](0)) + (B12(x,y,xl,yl,t)*dphi[j][qp](0)*
            phi[i][qp]) +
                                (C12(x,y,xl,yl,t)*phi[j][qp]*phi
                                    [i][qp]) );    // Newton term

        Kyx(i,j) += JxW[qp]*((-A21(xl,yl,t)*dphi[j][qp](0)*
            dphi[i][qp](0)) + (B21(x,y,xl,yl,t)*dphi[j][qp](0)*
            phi[i][qp]) +
                                (C21(x,y,xl,yl,t)*phi[j][qp]*phi
                                    [i][qp]) );    // Newton term

        Kyy(i,j) += JxW[qp]*((-A22(xl,yl,t)*dphi[j][qp](0)*
            dphi[i][qp](0)) + (B22(x,y,xl,yl,t)*dphi[j][qp](0)*
            phi[i][qp]) +
                                (C22(x,y,xl,yl,t)*phi[j][qp]*phi
                                    [i][qp]) );    // Newton term
```

```
    }

    for (unsigned int j=0; j<n_t_dofs; j++) {

      Kxt(i,j) += JxW[qp]*1000*(-A13(xl,yl,t)*psi[j][qp]*
          dphi[i][qp](0) + C13(x,y,xl,yl,t)*psi[j][qp]*phi[i
          ][qp]);

      Kyt(i,j) += JxW[qp]*1000*(-A23(xl,yl,t)*psi[j][qp]*
          dphi[i][qp](0) + C23(x,y,xl,yl,t)*psi[j][qp]*phi[i
          ][qp]);

    }

  }

  for (unsigned int i=0; i<n_t_dofs; i++) {

    Ft(i) += JxW[qp]*1000*(-LB3(x,y,xl,yl,t)*psi[i][qp] + xl
        *xl*psi[i][qp] + yl*yl*psi[i][qp] -
                          (1.+((t*1000.)/Cable_Equation::
                              data.EA()))*((t*1000)/
                              Cable_Equation::data.EA())*psi[
                              i][qp]);

    for (unsigned int j=0; j<n_x_dofs; j++) {

      Ktx(i,j) += JxW[qp]*1000*(xl*dphi[j][qp](0)*psi[i][qp
          ]);

      Kty(i,j) += JxW[qp]*1000*(yl*dphi[j][qp](0)*psi[i][qp
          ]);

    }

    for (unsigned int j=0; j<n_t_dofs; j++)
      Ktt(i,j) += JxW[qp]*1000*1000*(-(1+((t*1000.)/
          Cable_Equation::data.EA()))*(1./Cable_Equation::
```

```
                data.EA())*psi[j][qp]*psi[i][qp]);

        }

    } // end of the quadrature point qp-loop

    dof_map.heterogenously_constrain_element_matrix_and_vector (
       Ke, Fe, dof_indices);

    catenary_static_system.matrix->add_matrix (Ke, dof_indices);
    catenary_static_system.rhs->add_vector    (Fe, dof_indices);

  } // end of element loop

  return;

}


//END Cable_Equation_MIXED functions
```

A.4.3 *main*

```
//
//  Cable_all.C
//
//
//  Created by rodrigo broggi on 16/10/14.
//
//

#include "Cable_Equation.h"


//This functions are input data to the problem and are to be
   changed for each different physical problem:

//Example of constant profile solution
```

```
inline libMesh::Real current_profile(const libMesh::Real y) {
   return 1; }

//Example of parabolic initial solution (this example of initial
    exact parabolic solution is a bit complex so it also needs
   some auxiliary functions other then its own that are defined
   below as well):
void initial_solution(libMesh::DenseVector<libMesh::Number> &
   output, const libMesh::Point & p, const libMesh::Real);

inline libMesh::Real Length() { return 2000.; }

inline libMesh::Real Depth() { return 1500.; }

inline libMesh::Real fnonlin_h(const libMesh::Real h) { return (
   Length() - (h/2.)*sqrt(4.*(pow((Depth()/h),2.))+1) - (pow(h
   ,2.)/(4.*Depth()))*asinh(2.*Depth()/h)); }

inline libMesh::Real D_fnonlin_h(const libMesh::Real h) { return
    (-(1./2.)*sqrt(4.*(pow((Depth()/h),2.))+1.) + 2.*pow((Depth
   ()/h),2.)*(1./(sqrt(4.*(pow((Depth()/h),2))+1.))) - (h/(2.*
   Depth()))*asinh(2.*(Depth()/h)) + (1./2.)*(1./(4*(pow((Depth
   ()/h),2))+1))); }

inline libMesh::Real fnonlin_x(const libMesh::Real x, const
   libMesh::Real s, const libMesh::Real h) { return (s - (x/2.)*
   sqrt(4.*pow((Depth()*x/(h*h)),2.)+1.) - (pow(h,2.)/(4.*Depth
   ()))*asinh(2.*Depth()*x/(h*h))); }

inline libMesh::Real D_fnonlin_x(const libMesh::Real x, const
   libMesh::Real s, const libMesh::Real h) { return (-(1./2.)*
   sqrt(4.*(pow((Depth()*x/(h*h)),2.))+1.) - 2.*pow((Depth()*x/(
   h*h)),2.)*(1./(sqrt(4.*(pow((Depth()*x/(h*h)),2.))+1.))) -
   (1./2.)*(1./(4.*(pow((Depth()*x/(h*h)),2.))+1.)));}

libMesh::Real find_h();

//end auxiliary functions for initial solution
```

```
int main (int argc, char** argv)
{
    libMesh::LibMeshInit init (argc, argv);

    // Create a GetPot object to parse the command line (mostly
       numeric parameters)
    GetPot command_line (argc, argv);


    //Reading and initialising Problem physical Data
    std::ifstream data_input("data_input.txt");
    Cable_Problem_Data data(data_input,&current_profile,&
       initial_solution);
    data_input.close();

    //Print the physical data
    data.print();

    boost::scoped_ptr<Cable_Equation> cable_equation;

    std::string type = "Both";

    libMesh::Mesh mesh(init.comm());

    //Handle case in witch just one case matter
    if ( command_line.search(1, "-type") ) {
        type = command_line.next(type);

        if (type == "CLASSIC"){
            cable_equation.reset(new Cable_Equation_CLASSIC(data
               ,command_line,init));
            cable_equation->print_data();
            cable_equation->solve_cable_problem_complete();
        }
        else if (type == "MIXED"){
            cable_equation.reset(new Cable_Equation_MIXED(data,
               command_line,init));
```

```
            cable_equation->print_data();
            cable_equation->solve_cable_problem_complete();
        }

        else{
            std::cerr<<"The argument of -type should be either
               MIXED or CLASSIC. User passed: "<<type<<std::endl
               ;
            exit(1);
        }

    }

    else{

        cable_equation.reset(new Cable_Equation_CLASSIC(data,
           command_line,init));
        cable_equation->print_data();
        cable_equation->solve_cable_problem_complete();

        cable_equation.reset(new Cable_Equation_MIXED(data,
           command_line,init));
        cable_equation->solve_cable_problem_complete();


    }


    return 0;
}


libMesh::Real find_h() {

    //First thing is to find the x cordinate of the plataform (
       named h)
    libMesh::Real h_c;
    libMesh::Real h_old = Depth()*0.5;
```

```
    libMesh::Real error = 1;

    unsigned int i = 0;

    unsigned int MAXIT = 50;


    while (error > 0.001 &&  i < MAXIT) {

        h_c = h_old - (fnonlin_h(h_old)/D_fnonlin_h(h_old));

        error = abs(h_c - h_old);

        h_old = h_c;

        i++;
    }

    return h_c;
}

void initial_solution(libMesh::DenseVector<libMesh::Number> &
   output, const libMesh::Point & p, const libMesh::Real) {

    const libMesh::Real h = find_h();

    libMesh::Real x_c;

    libMesh::Real x_old = p(0);

    libMesh::Real error = 1;

    unsigned int i = 0;

    unsigned int MAXIT = 50;


    while (error > 0.001 &&  i < MAXIT) {
```

```
        x_c = x_old - (fnonlin_x(x_old,p(0),h)/D_fnonlin_x(x_old
            ,p(0),h));

        error = abs(x_c - x_old);

        x_old = x_c;

        i++;
    }

    //Parabola initialization
    output(0) = x_c;
    output(1) = (Depth()/pow(h,2))*(pow(x_c,2));
    ( (output.size() < 3) ? output(1) = (Depth()/pow(h,2))*(pow(
       x_c,2)) : output(2) = 4.e2 + (1.e3 - 4.e2)*p(0)/(1800.) )
       ;
}

void initial_solution_mixed(libMesh::DenseVector<libMesh::Number
   > & output, const libMesh::Point & p, const libMesh::Real) {

    const libMesh::Real h = find_h();

    libMesh::Real x_c;

    libMesh::Real x_old = p(0);

    libMesh::Real error = 1;

    unsigned int i = 0;

    unsigned int MAXIT = 50;


    while (error > 0.001 &&  i < MAXIT) {

        x_c = x_old - (fnonlin_x(x_old,p(0),h)/D_fnonlin_x(x_old
            ,p(0),h));
```

```
        error = abs(x_c - x_old);

        x_old = x_c;

        i++;
    }

    std::cout<<std::endl<<"Size seen: "<<output.size()<<std::
        endl;

    //Parabola initialization
    output(0) = x_c;
    output(1) = (Depth()/pow(h,2))*(pow(x_c,2));
    ( (output.size() < 3) ? output(1) = (Depth()/pow(h,2))*(pow(
        x_c,2)) : output(2) = 4.e2 + (1.e3 - 4.e2)*p(0)/(1800.) )
        ;
}
```