Gilberto Luis Valente Costa

hp^2FEM - Uma Arquitetura de *Software* p Não-Uniforme para o Método de Elementos Finitos de Alta Ordem

119/2012

CAMPINAS
2012

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA

Gilberto Luis Valente da Costa

hp^2FEM - Uma Arquitetura de *Software* p Não-Uniforme para o Método de Elementos Finitos de Alta Ordem

Orientador: Prof. Dr. Marco Lúcio Bittencourt

Dissertação de Mestrado apresentada à Faculdade de Engenharia Mecânica da Universidade Estadual de Campinas, para a obtenção do título de Mestre em Engenharia Mecânica, na Área de Mecânica dos Sólidos e Projeto Mecânico.

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO DEFENDIDA PELO ALUNO Gilberto Luis Valente da Costa, E ORIENTADO PELO PROF. DR. Marco Lúcio Bittencourt.

.....
ASSINATURA DO ORIENTADOR

CAMPINAS
2012

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

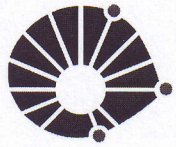
V234a

Valente da Costa, Gilberto Luis, 1983-
hp2FEM - uma arquitetura de software
p não-uniforme para o método de elementos finitos de alta
ordem \ Gilberto Luis Valente
da Costa. –Campinas, SP: [s.n.], 2012.

Orientador: Marco Lúcio Bittencourt.
Dissertação de Mestrado - Universidade Estadual de
Campinas, Faculdade de Engenharia Mecânica.

1. Arquitetura de software. 2. Método de elementos
finitos. 3. Framework (Programa de computador). I.
Bittencourt, Marco Lúcio, 1972-. II. Universidade
Estadual de Campinas. Faculdade de Engenharia
Mecânica. III. hp2FEM - uma arquitetura de software
p não-uniforme para o método de elementos finitos de alta.

Título em Inglês:	hp2FEM - a p non-uniform software architecture to the high order finite element method.
Palavras-chave em Inglês:	Software Architecture, Finite Element Method, Framework.
Área de concentração:	Mecânica dos Sólidos e Projeto Mecânico
Titulação:	Mestre em Engenharia Mecânica
Banca Examinadora:	Carlos Alberto Cimini Júnior, Edson Borin
Data da defesa:	27-08-2012
Programa de Pós Graduação:	Engenharia Mecânica



UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA
COMISSÃO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA
DEPARTAMENTO DE PROJETO MECÂNICO

DISSERTAÇÃO DE MESTRADO ACADÊMICO

hp^2FEM - Uma Arquitetura de *Software* p Não-Uniforme para o Método de Elementos Finitos de Alta Ordem

Autor: Gilberto Luis Valente da Costa
Orientador: Marco Lúcio Bittencourt

A Banca Examinadora composta pelos membros abaixo aprovou esta Dissertação:

Prof. Dr. Marco Lúcio Bittencourt, Presidente
DPM/FEM/UNICAMP

Prof. Dr. Carlos Alberto Cimini Júnior
DPM/FEM/UNICAMP

Prof. Dr. Edson Borin
IC/UNICAMP

Campinas, 27 de Agosto de 2012.

Dedicatória

Aos meus queridos avós, Pedro Catarino e Luiza Marçal, os quais sinto muito orgulho e agradeço por todo carinho e o apoio de sempre.

Agradecimentos

À Deus, minha fonte de força, fé e determinação. E como em todos os dias, agradeço pelo dom da vida.

Ao meu orientador, Prof. Dr. Marco Lúcio Bittencourt, por essa oportunidade e confiança na minha capacidade, pelo seu apoio e incentivo ao longo desse período.

Aos membros das bancas de qualificação e defesa, Prof. Dr. Carlos Alberto Cimini Júnior, Prof. Dr. Luiz Otávio Saraiva Ferreira e Prof. Dr. Edson Borin, pelos questionamentos e preciosas contribuições à este trabalho.

A toda minha família, aos meus avós Pedro Catarino Valente e Luiza Marçal Valente, pela formação que me deram, pela motivação e carinho, pelo apoio em minhas decisões. Aos meus tios(as), primo(as) que sempre me deram apoio durante esse tempo.

A uma pessoa, a qual confesso que sua ausência física ainda me faz passar por momentos difíceis. Em vários momentos desses 3 anos, sempre tive vontade de ligar e poder ouvir-la mais uma vez. As vezes demoramos a entender o quanto o amor de mãe é grande e verdadeiro. Obrigado Mãe! Por tudo que você fez por mim, pelo seu grande amor. Você colaborou e muito na minha formação. Sou eternamente grato!

Aos amigos e companheiros de laboratório: M. Sc. Fabiano Bargos, M. Sc. Jaime Izuka, M. Sc. Caio Rodrigues, M. Sc. Felipe Furlan, Dr. Carlos Mingoto, M. Sc. Jaime Delgado e aos engenheiros Jorge Suzuki, Oscar Rojas e Paola Gonzalez pela amizade, colaboração e apoio durante todo esse período.

Aos amigos em geral da Faculdade Engenharia Mecânica em especial em especial aos Cientistas da Computação Silas Alves e Wendell Diniz, e aos engenheiros Allan Dias (o tio) e Luan Franchini.

Aos amigos de convivência e república José Guilherme, Fabrício Luís, Everton Viana, Wander Fernandes, Filipe Ramos e Atílio Luís, pelo apoio e amizade de sempre.

Aos amigos colombianos, Camilo Gordillo, Ruben Dario, Jenny Lombo, Manuel Arcila, Miguel Angel, Camilo Ariza, Liz Katherine, Andrés Puerto, Maria Fernanda e tanto outros, pois não caberiam nesse livro. Agradeço pela amizade, apoio e por grandes momentos de convivência.

A todos amigos e familiares de quem eu possa não ter lembrado de colocar o nome aqui, mas que de alguma forma direta ou indireta colaboraram para que eu pudesse alcançar esta conquista.

A CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, pelo indispensável apoio financeiro.

O gênio consiste em um por cento de inspiração e noventa e nove por cento de transpiração.

Thomas Alva Edison

Resumo

VALENTE DA COSTA, Gilberto Luis. hp2FEM - uma arquitetura de software p não-uniforme para o método de elementos finitos de alta ordem. 2012. 122p. Dissertação (Mestrado). Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Campinas.

Este trabalho tem como objetivo principal a implementação de uma arquitetura de *software* para o Método de Elementos Finitos de Alta Ordem (MEF-AO), baseando-se no paradigma de programação orientada a objeto (POO) e no uso de técnicas de otimização de código fonte. O *software* foi escrito em linguagem C++ e desenvolvido sobre um *framework* com ferramentas que auxiliaram no desenvolvimento. A modelagem do sistema foi realizada de forma a facilitar e promover o reuso e manutenção do código. Buscou-se, também, a flexibilidade e generalização do MEF-AO ao permitir a variação nos procedimentos da construção das equações e o uso de malhas p não-uniforme. Neste caso, cada elemento pode ser interpolado com uma ordem polinomial diferente, além de permitir o uso de um algoritmo local de solução. Tal característica pode diminuir o número de operações e de armazenamento, pois o número de funções de forma é aumentado apenas onde é necessário o uso de mais pontos para interpolação da malha de solução. No final, o *software* é avaliado aplicando o problema de projeção para malha de quadrados e hexaedros.

Palavras-chave: Arquitetura de *Software*, Método de Elementos Finitos, *Framework* (Programa de computador).

Abstract

VALENTE DA COSTA, Gilberto Luis. hp2FEM - a p non-uniform software architecture to the high order finite element method. 2012. 122p. Dissertação (Mestrado). Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Campinas.

The main objective of this work is the implementation of a software architecture for the High-Order Finite Element Method (HO-FEM), based on the Object Oriented Paradigm (OOP) and on source-code optimization techniques. The software was written in C++ programming language and developed over a framework which provided tools that assisted the implementation. The system was modeled so to promote code reuse and maintainability. Furthermore, the system modeling also provided flexibility and generalization for the HO-FEM by allowing modifications on the procedures for equation assembling and the use of p-non-uniform meshes. In this case, each element can be interpolated with different polynomial order, and allows the application of an algorithm for local solution. Such features can reduce the number of operations for memory allocation, since the number of shape functions is increased only where a higher density of points is needed by the solution mesh. Finally, the software is assessed by applying the projection problem for meshes of squares and hexahedros.

Keywords: Software Architecture, Finite Element Method, Framework.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e Siglas

Matrizes e Vetores

$\{F\}$,$\{f\}$	-	Vetor de forças aplicadas
	-	Termos independentes
$\{F_e\}$	-	Vetor local de forças aplicadas
$F_{e,i}$	-	Coeficientes do vetor local de forças aplicadas
f_e^{1D}	-	Vetor de carga unidimensional associado à matriz de massa
f_i^{1D}	-	i-ésimo elemento do vetor de carga unidimensional associado à matriz de massa
$\{f_{1D}^m\}$	-	Vetor de carga associado à matriz de massa unidimensional
$[K]$	-	Matriz de rigidez global
$[K_e]$	-	Matriz de rigidez local
$[M]$	-	Matriz de massa global
$[M_{i,j}]$	-	Coeficientes da matriz de massa
$[M_{1D}]$, $\left[M_e^{1D}\right]$	-	Matriz de massa unidimensional
$[M_e]$	-	Matriz de massa local
$[M_e]^{-1}$	-	Inversa da matriz de massa local
$M_{e,ij}$	-	Coeficientes da matriz de massa local
$M_{e,ij}^{1D}$	-	Coeficientes da matriz de massa unidimensional
M_{ij}^{2D}	-	Coeficientes da matriz de massa bidimensional
$[N_{Sol,Pos}]$	-	Matriz de funções de interpolação da malha de solução nos pontos de colocação da malha de pós-processamento
$\{q\}$, $\{u_e\}$, $\{a_e\}$	-	Vetor de deslocamentos locais
$\{R\}$	-	Resíduo
$\{w\}$	-	Função ponderadora
$\{u\}$,$\{a\}$	-	Vetor de deslocamentos

$\left\{u_e^{Pos}\right\}$	-	Vetor de solução local da malha de pós-processamento
$\left\{u_e^{Sol}\right\}$	-	Vetor de solução local da malha de solução
$\{u_e^1\}, \{u_e^2\}, \{u_e^3\}$	-	Vetores de deslocamentos para os elementos 1, 2 e 3

Letras Latinas

b	-	Intensidade do termo independente
b_j	-	Coeficientes de aproximação
$d,'$	-	Derivada de primeira ordem
$''$	-	Derivada de segunda ordem
e	-	Erro na aproximação polinomial
$\|J\|$	-	Determinante do Jacobiano
L^p	-	Norma p
L_j	-	Comprimento de um elemento j
L_1, L_2	-	Comprimento dos elementos 1 e 2
R	-	Resíduo
w	-	Função ponderadora
$u(.)$	-	Função polinomial qualquer
$u_{ap}(.)$	-	Solução aproximada
u_i	-	Deslocamentos nodais para o nó i
u_{ij}	-	Deslocamentos nodais para o nó i no elemento j
u_{12}, u_{22}	-	Deslocamentos nodais do nó 2 nos elementos 1 e 2
$u_{11}, u_{12}, u_{22}, u_{23}, u_{33}, u_{34}$	-	Deslocamentos nodais calculados nos elementos 1, 2 e 3

Letras Gregas

Ω	-	Domínio
ξ_i	-	Coordenada local em [-1,1]
ξ, ξ_1, ξ_2	-	Coordenadas locais
ϕ_i,ϕ_j	-	Funções de interpolação
$\{\phi_i\}$	-	Base formada por um conjunto de funções ϕ_i
$\phi_a, \phi_b, \phi_p, \phi_q$	-	Funções de interpolação unidimensionais
ϕ_1, ϕ_2, ϕ_3	-	Funções de interpolação do polinômio de Lagrange

Siglas

ACDP	-	Ambiente Computacional para Desenvolvimento de Programas
DOFs	-	*Degrees of Freedom* (Graus de Liberdade)
EDP	-	Equações Diferenciais Parciais
GLC	-	Gramática Livre de Contexto
hp^2**FEM**	-	Programa do Método de Elementos Finitos de Alta performance - 2^a versão (*High Performace Finite Element Method Software*)
MEF	-	Método de Elementos Finitos
MEF-AO	-	Método de Elementos Finitos de Alta Ordem
POO	-	Programação Orientada a Objeto
	-	Paradigma de Orientação a Objetos
UML	-	*Unified Modeling Language*
XMI	-	*XML Metadata Interchange*
XML	-	*eXtended Markup Language*

Outras Notações

p-uniforme	-	Distribuição polinomial uniforme na malha de elementos finitos.
p-não-uniforme	-	Distribuição polinomial não uniforme na malha de elementos finitos.
$< , >$	-	Produto interno entre dois vetores
$\{\}^T$	-	Transposto de um vetor
1D	-	Elemento unidimensional
2D	-	Elemento bidimensional
3D	-	Elemento tridimensional

SUMÁRIO

1 Introdução

A simulação computacional é uma poderosa ferramenta utilizada em vários setores da indústria e em importantes centros de pesquisas relacionados às diversas áreas do conhecimento, como química, física, biologia molecular, meteorologia, e principalmente pelas engenharias. Uma simulação faz uso de um modelo numérico do problema em questão, cujos parâmetros podem ser modificados para fornecer informações relevantes a respeito da dinâmica do problema. Desta forma, a simulação permite reduzir amplamente os custos, tempo e recursos através da economia de equipamentos e a eliminação de longas e onerosas etapas de testes no processo de desenvolvimento e pesquisa.

De maneira geral, o desenvolvimento de uma simulação envolve dois passos fortemente relacionados: a modelagem matemática e a implementação algorítmica. O primeiro passo diz respeito à representação do problema em questão em linguagem matemática, de forma que todos os parâmetros relevantes e suas dependências e interações sejam descritas em equações bem definidas. O segundo passo remete ao problema de se representar o modelo matemático obtido de forma algorítmica. Logo, o modelo matemático é convertido em um algoritmo – escrito em uma linguagem de programação – discreto que será executado por um computador. Finalmente, o resultado destes dois passos será um *software* simulador capaz de produzir dados a respeito do problema.

Entretanto, há uma série de questões a serem estudadas durante o desenvolvimento da modelagem matemática e de sua implementação algorítmica para garantir que o *software* de simulação forneça resultados precisos e rápidos. A precisão do *software* está relacionada com a qualidade do modelo matemático, ou seja, é diretamente proporcional à proximidade do modelo matemático ao modelo real. Já o tempo de resposta do *software* está relacionado à velocidade de cálculo das unidades de processamento, à quantidade de unidades de processamento e à otimização do algoritmo em termos estruturais e arquiteturais, o que corresponde à forma em que será organizado os dados no *software*.

Não é difícil concluir que tanto a indústria quanto a academia buscam por simulações que sejam ao mesmo tempo precisas e rápidas. Dados pouco precisos podem levar a soluções erradas, e a longa espera para a obtenção de dados encarece o processo de simulação a ponto de torná-lo economicamente inviável. Consequentemente, os *softwares* de simulação devem tratar um grande volume de dados no menor tempo possível. O desenvolvimento de *softwares* com esta caracterís-

tica é contemplado por uma área de pesquisa chamada *High Performace Computing (HPC)*, como alguns autores referenciam, também chamada de Computação de Alto Desempenho (CAD).

Dentre as várias ferramentas disponíveis para a modelagem matemática, encontra-se o Método dos Elementos Finitos (MEF). O MEF é um método numérico utilizado para a obtenção de um modelo matemático aproximado de problemas de valor de contorno (HUGHES, 2000), fazendo uso das equações diferencias que representam o problema e as condições de contorno. Este método é aplicado à problemas onde não é possível encontrar uma solução analítica, devido às irregularidades geométricas e às não-linearidades.

Além disso, este método possui vantagens como permitir que as propriedades dos materiais sejam diferentes em elementos adjacentes. Entre outras, o MEF permite que as fronteiras irregulares possam ser aproximadas utilizando-se elementos com lados retos ou determinadas com exatidão usando elementos com fronteiras curvas. No entanto, a principal desvantagem da utilização do MEF é relacionada à necessidade de programas de computador e infraestrutura computacional. Além disso, o método necessita de uma grande quantidade de memória para a solução de problemas grandes e complexos.

Assim, no que diz respeito às simulações utilizando o MEF, é desejável que o método seja implementado de forma que garanta o alto desempenho da aplicação. É neste cenário que se encaixa este projeto de pesquisa, cuja proposta é desenvolver um conjunto de componentes de *software* de alto desempenho para uso em aplicações gerais do MEF. Neste sentido, este projeto propõe um conjunto de bibliotecas abertas, genéricas e portáveis que facilitem o desenvolvimento de simulações de alto desempenho. Para isso, devem fornecer ao usuário uma interface de *software* simples para algoritmos otimizados do MEF.

As bibliotecas propostas foram construídas de tal modo que possam ser utilizadas em diferentes aplicações. Por esta razão, foi dada grande ênfase na Engenharia de Software com o fim de conceber bibliotecas que fossem reutilizáveis, de fácil manutenção, flexíveis e seguras. Naturalmente, o estudo da modelagem de *software* esteve presente em todo o desenvolvimento do projeto.

Este documento resume o desenvolvimento deste projeto de pesquisa, salientando os objetivos da pesquisa, os métodos utilizados, o processo de desenvolvimento e os resultados já obtidos.

1.1 Motivação

A maior exigência de processamento computacional é devida tanto ao crescimento dos diversos setores da indústria quanto à informatização dos processos industriais, o que tem aumentando em grande escala o volume de dados a serem processados em computadores. Alguns exemplos, são as atividades relacionadas à física quântica, petróleo, previsão de tempo e bolsa de valores. Grandes empresas ligadas à estas áreas vêm utilizando supercomputadores com múltiplos núcleos de processamento e grande capacidade de memória.

Embora o aumento da capacidade de *hardware* tem um grande impacto na execução de algoritmos de alta complexidade, existem algumas limitações, como o custo para o uso de supercomputadores, podendo-se chegar a 200.000,00 dólares por hora. Além disso, estes computadores consumem uma grande quantidade de eletricidade, alcançando um consumo de 12,6 MW (MEUER *e outros*, 2012).

Nesse sentido, é importante a busca ou desenvolvimento de ferramentas de *software* para otimizar as operações a serem realizadas em uma CPU, tanto em computadores pessoais quanto em supercomputadores. Por esta razão, algumas vezes é necessário para o programador descer alguns níveis de abstração e entender o comportamento do *software* em um *hardware* específico, ou seja, o programador deve codificar levando em conta as características intrínsecas do *hardware* para o qual está codificando.

Uma vez que as ferramentas relativas à aplicação do MEF se encontram intimamente ligadas ao mundo da computação e considerando a importância das aplicações em diversos campos (AZEVEDO, 2003), o MEF é importante para o estudo feito nesse trabalho. Além de ser um método que demanda alto desempenho computacional.

Uma das formas de otimização da simulação computacional através do MEF acontece na modelagem numérica. Considerando-se o Método de Elementos Finitos de Alta Ordem (MEF-AO), essas otimizações tornam-se ainda mais necessárias devido ao alto índice de consumo de memória e processamento do algoritmo. Isso pode ser notado quando geramos funções de interpolação para apenas um elemento hexaédrico, com grau polinomial 10, o qual requer cerca de 14 MB de memória para armazenamento de funções de interpolação calculadas nos pontos de integração.

Além das considerações de alto desempenho do código, é fundamental usar a modelagem de elementos finitos que seja fácil na sua manipulação e entendimento. Ainda mais quando se trata da aplicação de alta ordem em um problema, que pode requerer um número maior de variáveis de entrada, aumentando a complexidade de manipulação dos dados. Nesse sentido, a adoção do paradigma da orientação por objeto tem como objetivo dar essa flexibilidade de manipulação das funções do algoritmo.

1.2 Trabalhos Correlatos

Os assuntos discutidos nesse trabalho estão ligados ao processo de desenvolvimento de *software* para o MEF-AO e as estratégias para solução do algoritmo. Considerando que muitos pesquisadores em vários países vêm trabalhando com tema de computação de alto desempenho direcionado aos métodos numéricos e algoritmos matemáticos, apresenta-se a seguir alguns trabalhos que tratam o MEF-AO dentro desse ambiente de CAD. No entanto, tanto em trabalhos direcionados à CAD, como outros que descrevem a modelagem, descrevem-se algumas características das arquiteturas de *software* para o MEF.

Um dos principais objetivos de alguns trabalhos que utilizam métodos numéricos, assim como o MEF, está em utilizar estratégias de implementação de maneira que o código gerado seja flexível e genérico, sem deixar de atender os requisitos de qualidade de um *software*. Assim, VOS (2011) demonstra como pode ser feito o encapsulamento do método espectral/hp em um *software* orientado a objeto, utilizando C++, que provê um ambiente necessário de algoritmos e estruturas de dados para emular os conceitos fundamentais do método.

Entre outros objetivos, VOS (2011) buscou a implementação de um código que seja computacionalmente eficiente, já que o método espectral/hp utiliza alta ordem polinomial. Este trabalho aplica o método a um problema elíptico com uma precisão definida. Considerando isso, VOS (2011) conclui que para problemas suaves, deve-se fixar a malha e variar a ordem polinomial de acordo com a precisão desejada. Já para problemas não-suaves, observou-se que o tempo de execução é minimizado fixando a ordem polinomial e refinando a malha.

Alguns trabalhos relacionados ao MEF não tem como preocupação principal a estruturação do *software* que implementa a modelagem ou como seu comportamento vai atender certos requisi-

tos direcionados à generalização do MEF. Nesse caso, focalizam-se nas funcionalidades principais do método direcionando à busca por alto desempenho com o uso de processamento paralelo. Essa opção vem permitindo a evolução em termos de aplicações mais complexas, com grande redução no tempo de processamento das simulações comparadas com as que utilizam processamento sequencial. O processamento paralelo pode ser aplicado tanto em *clusters* (com memória distribuída) como em computadores multi-núcleos (com memória compartilhada).

Os trabalhos de KAWABATA *e outros* (2009) e FU (2008) aplicam o método de elementos finitos em ambientes paralelos com o uso de clusters. Para KAWABATA *e outros* (2009) o objetivo foi identificar os gargalos na execução do MEF. Procurou-se saber os pontos onde ocorrem maior consumo de tempo e memória para aplicar a paralelização.

Em FU (2008), aplicou-se um algoritmo paralelo de decomposição de domínio na malha de elementos finitos para análise estrutural. O algoritmo aplicado por FU (2008) busca dividir um domínio, nesse caso a malha de elementos finitos, em vários sub domínios igualmente balanceados, onde cada sub-domínio é executado localmente em um processador de um ambiente de *clusters* aglomerados.

Os resultados obtidos por KAWABATA *e outros* (2009) foram satisfatórios, pois utilizando um toróide com 201.600 graus de liberdade, obteve-se melhor desempenho na montagem da matriz de rigidez, seguida da integração dos elementos e por último a solução do sistema. FU (2008) executou testes com 3 malhas diferentes de MEF sobre um modelo de tensão plano bidimensional de barragem, onde são alterados duas variáveis em seus testes: o grau de cada malha e o número de processadores diferentes. Com isso, o autor concluiu que com o aumento de variáveis se alcança um melhor desempenho computacional. Porém, observa-se que com o aumento do número de processadores, o desempenho melhora até um determinado limite, pois um número alto de processadores acarreta na queda de desempenho, devido à sobrecarga de comunicação e sua sincronização.

ANZT *e outros* (2010) tentam unir os conceitos de programação orientada a objeto com computação de alto desempenho. O trabalho discute sobre a implementação de um *software* para elementos finitos, chamado HiFlow, um projeto desenvolvido por uma equipe de 17 pessoas do Instituto de Tecnologia Karlsruhe na Alemanha. O HiFlow traz uma visão sobre simulação numérica, otimização numérica e computação de alto desempenho. Esse *software* é direcionado para aplicação em problemas de dinâmica dos fluidos. Contudo, incorporou-se módulos para tratar problemas de meteorologia, pesquisas sobre energia e bioinformática.

O pacote de *software* do HiFlow é dividido em 3 módulos chamados Mesh, DoF/FEM e álgebra linear. O primeiro sustenta operações como entrada e saída de arquivos de dados, iterações de entidades da malha e refinamento das células. O DOF/FEM trata os elementos e os graus de liberdade para armazenar e construir as equações do MEF. Por último, o módulo álgebra linear contém o módulo LAtoolbox utilizado para solução de sistemas lineares. Esse pacote foi desenvolvido em outro projeto e depois acoplado ao HiFlow.

Em sua conclusão, ANZT *e outros* (2010) descrevem que alcançam os principais objetivos do *software* baseado em orientação a objeto e o uso de dois níveis de comunicação de processamento paralelo: um com uso de CPUs (*Central Processing Unit*) para estruturas do MEF e o uso de GPUs (*Graphics Processing Unit*) para bibliotecas otimizadas de álgebra linear. Para ANZT *e outros* (2010), o usuário do *software* HiFlow está livre de qualquer conhecimento detalhado de *hardware*, tendo somente que se familiarizar com as interfaces e configurar os módulos disponíveis.

Existem outros programas comerciais ou de domínio público que implementam o MEF. Contudo, o objetivo desse capítulo não é descrevê-los de forma exaustiva, mas exibir um breve histórico de alguns *softwares* existentes, apontando algumas características em cada um. O ANSYS (STOLARSKI *e outros*, 2007) é um dos *softwares* mais populares de simulação para o MEF, utilizado em diversas universidades e em mais de 60 países. O ANSYS é privado, mas possui variações do programa não comerciais para academia. O *software* possibilita a solução numérica de diversos problemas físicos, tais como transferência de calor, fluídos, análise estrutural estática e dinâmica e também problemas de eletromagnetismo e acústica.

O Multiphysics COMSOL (COMSOL, 2012) é outro *software* privado e tem como objetivo simplificar as etapas da modelagem para solução com o MEF, tais como definição da geometria, malha, aspetos físicos do problema e visualização de resultados. O deal.II (BANGERTH *e outros*, 2007) é uma biblioteca em C++ de código aberto focado em soluções de Equações Diferenciais Parciais (EDP) baseado no MEF.

O DUNE (BASTIAN *e outros*, 2006) também é uma ferramenta para solução numérica de EDPs. No entanto, o DUNE não é restrito à aplicações do MEF, mas também implementa os métodos de volumes finitos e diferença finitas. De forma geral, o *software* baseia-se principalmente em interfaces abstratas, técnicas de programação genéricas e reuso de pacotes existentes.

O NASTRAN foi originalmente desenvolvido nos anos 1960 para a agência espacial ameri-

cana (NASA). É um pacote poderoso e grande com diversos módulos integrados para várias aplicações e distribuídos em muitas companhias. Entretanto, é um sistema escrito sobre metodologias de software e linguagem antigas no desenvolvimento de *software* (MCCORMICK, 1970).

O Nektar++ (NEC, 2012) é uma coleção de bibliotecas para o MEF e assim como o deal.II seu código é aberto. O Nektar++ foi originalmente desenvolvido para aplicações de problema de dinâmicas do fluídos. Atualmente, aplica-se algoritmos de elementos espectrais para diversos problemas de engenharia. O Nektar++ baseia-se nas soluções por aproximação global espectral/hp, módulo adicionado por VOS (2011).

1.3 Objetivos

O objetivo geral desse projeto é modelar e desenvolver uma arquitetura de *software* para o método de elementos finitos de alta ordem em linguagem C++, de maneira que permita flexibilidade e generalização, assim como a adição de novos componentes no programa quando necessário. Para isso, deve-se considerar o paradigma de orientação a objetos e alguns conceitos de eficiência em C++, visando diminuir o tempo de execução dos algoritmos. O *software* será implementado sobre a versão hp do MEF, com uso de adaptatividade polinomial para malhas com elementos de ordem uniforme e não-uniforme. Nesse sentido, os objetivos específicos são:

- Desenvolver um ambiente de suporte que permita modelagem, empregando a *Unified Modeling Language* (UML), através dos conceitos de orientação a objeto e flexibilidade na compilação do código C++ com uso de *scripts makefile* (STALLMAN *e outros*, 2010), testes de unidades de *software* e a análise de vazamento de memória.
- Modelar a arquitetura para o MEF-AO utilizando UML. Essa arquitetura inicial contém as classes com seus atributos e a assinatura dos métodos implementados, além de indicarem o relacionamento das classes e suas interfaces.
- Desenvolver o *software* baseando-se na modelagem UML aplicando os conceitos de orientação objeto e técnicas de otimização de código em C++.
- Implementar de um analisador, que converta dados de uma expressão (equação) em um valor numérico resultante. Esse analisador simbólico-numérico será incorporado ao programa para

uso em determinados casos, por exemplo, quando as forças em um elemento são dadas por funções simbólicas.

- Avaliar o problema de projeção como aplicação para diversas ordens, considerando refinamento polinomial para malhas com distribuições p-uniforme e não-uniforme. Serão aplicadas também outras estratégias de solução a serem abordadas tais como produto tensorial entre matrizes unidimensionais e a interpolação entre malhas de solução e pós-processamento.

1.4 Contribuições

Esse trabalho trata do desenvolvimento de uma arquitetura de *software* para o Método de Elementos Finitos de Alta Ordem (MEF-AO). O *software* hp^2FEM escrito em linguagem C++ é de domínio público e emprega estratégias de solução locais para otimizar o uso de recursos computacionais, resolvendo o problema proposto com a mesma aproximação de abordagens tradicionais. Atrelado a isso, desenvolve-se uma arquitetura aplicando conceitos de orientação a objeto, permitindo flexibilidade na alteração ou adição de novos módulos (como novas aplicações do MEF-AO ou métodos de solução de sistema lineares) sem comprometer a estrutura do *software*. A modelagem e implementação dessa arquitetura está baseada em uma versão experimental em Matlab, desenvolvida pelo grupo do Laboratório de Simulação Computacional da Faculdade de Engenharia Mecânica da UNICAMP, que por sua vez foi baseada em uma versão anterior em C++ (SILVA, 1997).

Aplicando o problema de projeção, discutido no Capítulo 4, desenvolve-se a tensorização de matrizes de massa e vetor de carregamento de elementos 1D na montagem de matrizes e vetores para elementos 2D e 3D (MIANO, 2009).

Na atual versão em C++, notam-se as características de flexibilidade e generalização do MEF-AO através de novas estruturas de dados para armazenamento de funções de interpolação e dados da malha de elementos finitos, assim como na reformulação de outras estruturas originais do *software* em Matlab. Além disso, nesse projeto implementa-se um método de solução local para o MEF-AO utilizando distribuição com grau não-uniforme das funções polinomiais em um mesmo grupo de elementos finitos, possibilitando o aumento do grau polinomial em regiões específicas.

O programa hp^2FEM considera diferentes tipos de malhas, permitindo o uso de diferentes

graus polinomiais nessas malhas. Assim, têm-se malhas para entrada, solução, mapeamento e pós-processamento. Em adição, desenvolve-se um método para interpolar a solução da malha de solução nos pontos de colocação da malha de pós-processamento, permitindo pós processar os elementos com grau polinomial diferente.

Por fim, para dar suporte à implementação do *software*, construiu-se um *framework* para modelagem, análise de erros, verificação de vazamento de memória. Além disso, desenvolveu-se um analisador simbólico-numérico, para conversão de funções simbólicas utilizadas para o MEF-AO em valores numéricos.

1.5 Organização do Texto

O texto foi organizado de forma que no Capítulo 2 serão detalhados os aspectos conceituais sobre o MEF-AO. Além disso, será exemplificado as estratégias de solução para o MEF-AO da arquitetura proposta. O capítulo descreve a estratégia local de solução e exemplifica os tipos de malhas utilizadas. Também são abordadas outras estratégias para solução como o produto tensorial entre matrizes unidimensionais, proposta de solução p não-uniforme e o algoritmo de interpolação entre malha de pós-processamento e solução.

O Capítulo 3 apresenta a arquitetura de software proposta. Inicialmente são detalhados os conceitos de engenharia de *software* e do paradigma de orientação a objeto, apresentando também especificações do *framework* utilizado. Em seguida, descrevem-se as características principais do programa através de diagramas de classes UML, dividindo o *software* em seis níveis. É demonstrado também como são definidas as configurações de entrada. Ao final, exibe-se a sequência de execução do MEF-AO por meio de diagramas de atividades UML.

O Capítulo 4 descreve o problema de projeção, o qual foi aplicado para validar o funcionamento da arquitetura do MEF-AO. Exibem-se os testes da adaptatividade p de elementos 2D e 3D usando as estratégias de solução propostas no Capítulo 2. As conclusões do trabalho estão no Capítulo 5, assim como suas perspectivas futuras. Além disso, exibe-se no Anexo A, os arquivos de entradas do hp^2FEM e no Apêndice A, os arquivos dos códigos do analisador simbólico-numérico implementado para prover suporte ao programa.

2 Método de Elementos Finitos de Alta Ordem

Nesse capítulo, apresentam-se conceitos sobre os métodos de elementos finitos e por resíduos ponderados. Exibe-se um exemplo de elasticidade para demonstrar a formulação do método e em seguida consideram-se conceitos de alta ordem e os passos para a construção de um algoritmo computacional para o MEF. Também discutem-se as estratégias a serem utilizadas na arquitetura de software proposta descrita no capítulo 3. Essas estratégias envolvem a forma de interpolação da malha de elementos finitos e a montagem das equações.

2.1 Método de Elementos Finitos

O MEF tem sido utilizado em várias áreas da ciência e engenharia, por tratar-se de um método matemático usado para solução de problemas de valor de contorno, baseado na discretização do domínio e que utiliza funções na aproximação da solução. Dessa forma, tendo a disposição de recursos computacionais, pode-se obter grande grau de precisão (KNOW, 1997). O MEF foi popularizado devido a uma de suas características, em que dado um domínio qualquer, o mesmo é dividido em um grupo aleatório de subdomínios com dimensões finitas. Com isso, se torna fácil a análise de cada elemento.

Contudo, desde o final da década de 60, o MEF teve diferentes desenvolvimentos com o objetivo na redução de erros de aproximação. Primeiramente, buscou-se aplicar adaptatividade nas malhas geradas, em que o objetivo era reduzir o tamanho do elemento, ou seja, refinar a malha, originando a versão h do método. Posteriormente, o tratamento foi focado no aumento do grau do polinômio das funções de interpolação sobre uma malha fixa, denominada versão p. Foi considerado posteriormente a combinação das duas versões, nomeando-se como a versão hp do MEF.

2.1.1 Método Direto

Considere um corpo contínuo e o carregamento externo aplicado. O objetivo é calcular os deslocamentos e as deformações oriundos das forças externas, além das tensões com as relações constitutivas. Efetua-se a discretização do corpo, obtendo-se subdomínios, ou seja, uma malha de

elementos finitos com nós e elementos, como pode ser visto na Figura 2.1.

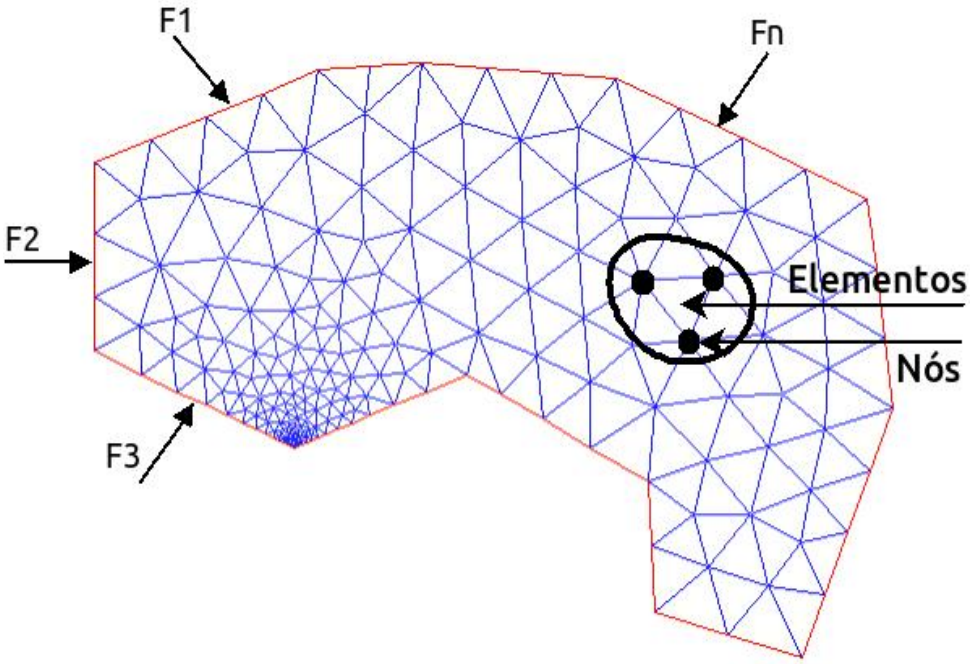


Figura 2.1: Malha de elementos finitos.

Cada grupo ou subdomínio pode ter seus elementos em uma, duas ou três dimensões. A aproximação para cada elemento é feita de forma independente. Considerando uma análise estática, cada elemento finito representa uma parte da rigidez do corpo.

Através da formulação adequada, incluindo as propriedades geométricas, do material e do corpo, obtêm-se uma matriz de rigidez para cada elemento. Assim sendo, a matriz de rigidez global $[K]$ é calculada com a superposição das matrizes de cada sub domínio $[K_e]$ e se realiza o mesmo o processo para se obter o vetor externo de carregamento $\{F\}$ sobre o corpo. Ao final, obtém-se o seguinte sistema de equações:

$$[K]\{u\} = \{F\}, \tag{2.1}$$

sendo $\{u\}$ o vetor de deslocamentos a serem calculados. Após a solução da equação (2.1), obtendo-se $\{u\}$, determinam-se os valores secundários: as deformações e tensões no corpo. Contudo, para a aplicação do MEF em geometrias mais complexas, faz-se necessário refinar a malha ao longo do contorno, aumentando o número de pontos nas regiões que possuem o comportamento mais curvo (BITTENCOURT, 2010).

2.1.2 Método de Resíduos Ponderados

Outra abordagem na formulação do MEF é o método de resíduos ponderados, o qual baseia-se na aproximação de uma solução satisfazendo as condições de contorno de um dado problema.

O primeiro passo é definir a função aproximada, em seguida essa função é substituída na equação diferencial do modelo, obtendo o resíduo **R**. Esse resíduo deve ser ponderado de alguma forma, a fim de aproximar-se a zero, se considerado em todo o domínio do problema (DANTAS, 2006). Isso é feito adotando-se uma segunda função, conhecida como função ponderadora **w**. A equação (2.2) representa o produto interno entre um resíduo **R** e uma função ponderadora **w** de uma equação diferencial do problema, resultando em uma equação integral no domínio Ω.

$$< \mathbf{R}, \mathbf{w} >= \int_{\Omega} \mathbf{R}\mathbf{w} \mathrm{d}\Omega = 0. \tag{2.2}$$

Aplicando integração por partes obtém-se a forma integral ou fraca do problema considerado. Usando a aproximação pelo MEF na forma fraca obtém-se um sistema de equações similar à equação 2.1 (KARNIADAKIS E S., 2005).

2.1.3 MEF de Alta Ordem

O MEF-AO é caracterizado pelo aumento do grau polinomial da funções de interpolação. Esse projeto atua nessa característica do MEF, com o propósito de desenvolver uma arquitetura de *software* que seja fácil de gerenciar, visto que a alta ordem agrega novas caraccterísticas ao MEF. Uma dessas características é a forma de distribuição dos pontos de colocação a serem interpolados nos elementos da malha. Logo, quando se caracteriza a aplicação do MEF-AO como p-não-uniforme, tem-se que a interpolação da malha pode ocorrer com diferentes pontos em cada elemento. Por outro lado, quando é definido uma malha como p-uniforme, todos elementos tem o mesmo número de funções de interpolação.

2.2 Esquema Computacional

De maneira geral, em grande parte da literatura, como em DANTAS (2006), o algoritmo básico para o MEF pode ser descrito através dos seguintes passos:

→ Passo 1 - Estudar e compreender o problema físico a ser resolvido.

→ Passo 2 - Etapa de pré-processamento. Discretização de um domínio Ω, gerando a malha em N subdomínios, determinando a forma do elemento e o tipo (1D, 2D ou 3D).

→ Passo 3 - Determinação do modelo de aproximação, gerando as funções de interpolação elementares para uma determinada grandeza (deslocamentos, pressão).

→ Passo 4 - Definir as equações de cada elemento. Por exemplo, dado um problema físico, obtêm-se a equação $[M_e]\{q\} = \{F_e\}$, onde $[M_e]$ é a matriz de massa, $\{q\}$ os coeficientes a serem calculados e $\{F_e\}$ o termo independente do elemento e.

→ Passo 5 - Montagem do sistema linear. Construção da equação geral com contribuição de todos N elementos do modelo, com o lado esquerdo $[M]$ e o lado direito $\{F\}$.

→ Passo 6 - Aplicação das condições de contorno homogêneas e não-homogêneas essenciais e naturais do MEF.

→ Passo 7 - Solução do sistema linear para a obtenção das incógnitas primárias.

→ Passo 8 - Pós-processamento e análise de resultados.

2.3 Abordagens de Solução para a Arquitetura hp^2FEM

O esquema computacional descrito anteriormente ilustra uma forma geral para obter a solução para um determinado tipo de problema usando o MEF. Nesta seção, discutem-se as principais estratégias de solução propostas nesse trabalho, considerando o MEF-AO. O uso das abordagens propostas incorpora aspectos positivos ao MEF, tais como melhor desempenho computacional, além de facilitar a análise de dados durante o pós-processamento dos resultados. Essas estratégias foram implementadas no *software* hp^2FEM.

2.3.1 Solução Local

A solução global, como visto na Seção 2.1.1, define uma forma de resolver a modelagem pelo MEF. Nesse caso, o sistema linear de equações dado pela expressão (2.3), será construído após o cálculo dos operadores da equação: a matriz de coeficientes $[M]$ e o vetor de termos independentes $\{F\}$. Contudo, o vetor de solução é calculado de forma global, após a montagem do sistema por meio da superposição das grandezas elementares $[M_e]$ e $\{F_e\}$, como pode ser visto na Figura 2.2.

$$[M]\{u\} = \{F\}. \tag{2.3}$$

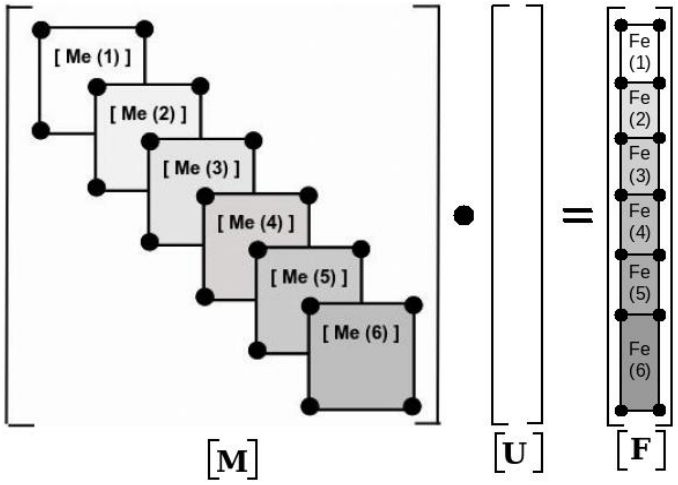


Figura 2.2: Solução global.

No método local, a solução é obtida em cada elemento. O vetor de solução global $\{u\}$ é calculado através da ponderação das solução locais com a medida do elemento. Assim, as matrizes e os vetores de carregamento não são sobrepostos, resolvendo o sistema linear localmente, como pode ser visto na Figura 2.3.

Um exemplo de aplicação do método de solução local, pode ser dado a seguir. Dado sistema de equações similar a (2.3), quer-se obter os coeficientes $\{u_e\}$ no elemento e. Essa solução poderá ser obtida invertendo a matriz de massa, ou seja,

$$\{u_e\} = [M_e]^{-1}\{F_e\}. \tag{2.4}$$

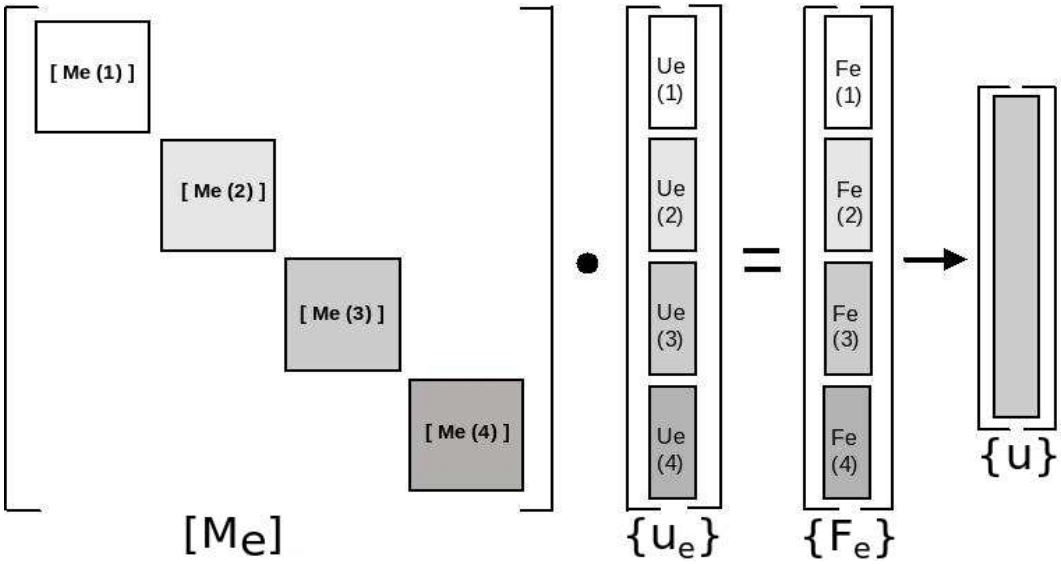


Figura 2.3: Solução local.

Considerando uma malha unidimensional discretizada em 3 elementos (ver Figura 2.4), interpolam-se os elementos em um intervalo de $[-1,1]$, sendo os coeficientes da matriz de massa e do vetor de carga determinados através das equações (2.5) e (2.6), respectivamente,

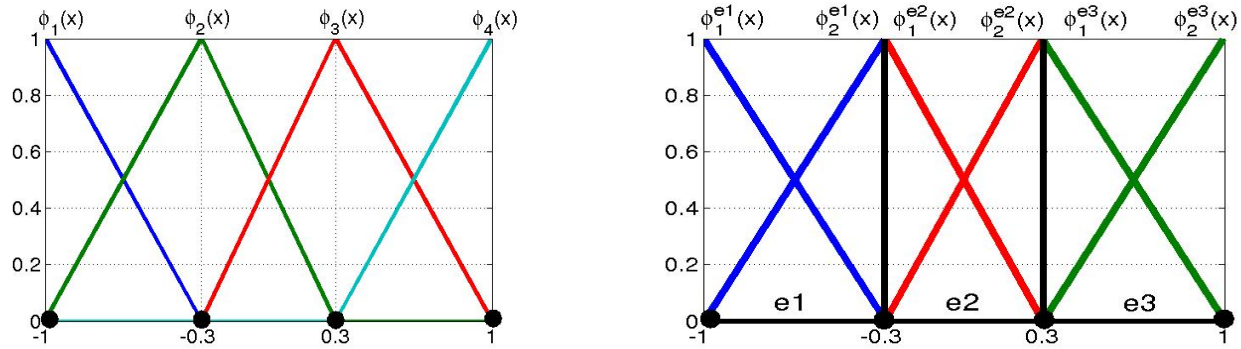
$$M_{e,ij} = \int_{-1}^{1} \phi_i(\xi_1)\phi_j(\xi_1)\,|J|\,d\xi_1, \tag{2.5}$$

$$F_{e,i} = \int_{-1}^{1} b(\xi_1)\phi_i(\xi_1)\,|J|\,d\xi_1, \tag{2.6}$$

onde, ϕ_i e ϕ_j são as funções de interpolação nas coordenadas locais ξ_1, $|J|$ o determinante do Jacobiano e $b(\xi_1)$ é a intensidade do termo independente.

O vetor de solução local $\{u_e\}$ é obtido em cada elemento através da equação (2.4). Logo, obtêm-se respectivamente os vetores soluções: $\{u_e^1\} = \begin{Bmatrix} u_{11} \\ u_{12} \end{Bmatrix}$, $\{u_e^2\} = \begin{Bmatrix} u_{22} \\ u_{23} \end{Bmatrix}$ e $\{u_e^3\} = \begin{Bmatrix} u_{33} \\ u_{34} \end{Bmatrix}$, onde, u_{11}, u_{12}, u_{22}, u_{23}, u_{33} e u_{34} são as soluções nos graus de liberdade da malha unidimensional nos elementos 1, 2 e 3 representados na Figura 2.4.

As Figuras 2.4(a) e 2.4(b) ilustram o processo de superposição das funções de forma no domínio global e no domínio de cada elemento da malha. A Figura 2.4(a) exibe o comportamento



(a) Interpolação no intervalo de -1 a 1 em todo o domínio.

(b) Interpolação linear em elementos locais.

Figura 2.4: Interpolação por polinômio de Lagrange.

das funções de interpolação globais $\phi_1(x)$, $\phi_2(x)$, $\phi_3(x)$ e $\phi_4(x)$ em toda malha, o qual deve-se considerar que a construção da solução global é feita após superposição das funções elementares. A Figura 2.4(b) ilustra o cálculo da solução nos elementos de forma independente, ou seja, as soluções são determinadas localmente.

Contudo é necessário calcular o vetor global de solução $\{u\}$, que é determinado a partir da ponderação das soluções locais com a medida do elemento. De forma geral, o vetor $\{u\}$ pode ser obtido através da seguinte equação (FURLAN, 2011).

$$u_i = \frac{\sum_{j=1}^{n} u_{ij} L_j}{\sum_{j=1}^{n} L_j}, \tag{2.7}$$

sendo n o número de elementos que compartilham o mesmo grau de liberdade. O índice i indica o grau de liberdade no sistema global e o índice j indica o número do elemento. Sendo assim, u_i representa a solução do vetor global no grau de liberdade i, u_{ij} representa a solução local para i calculada de acordo com o elemento j e L_j o comprimento do elemento. O cálculo para o coeficiente no ponto 2 com a contribuição dos coeficientes locais dos elementos 1 e 2 é por exemplo,

$$u_2 = \frac{u_{21} L_1 + u_{22} L_2}{L_1 + L_2} \tag{2.8}$$

2.3.2 Produto Tensorial de Matrizes de Massa

As funções de interpolação para elementos 2D e 3D são calculadas através do produto tensorial das funções de forma dos elementos 1D (BARGOS, 2009). Os termos da matriz de massa e do vetor de carregamento para elementos 2D ou 3D podem ser obtidos a partir dessas funções. Esse procedimento é denominado padrão e indicado por "STANDARD".

Contudo, uma outra forma de se calcular as matrizes de massa pode ser determinado a partir dos termos das matrizes de elementos unidimensionais (MIANO, 2009). Na arquitetura do hp^2FEM, identifica-se esse procedimento como "D1_MATRICES". Como será visto no Capítulo 3, "STANDARD" e "D1_MATRICES" são palavras chaves definidas nos arquivos de entrada.

O sistema de equações para o problema de projeção em um elemento unidimensional no sistema local de coordenadas ξ_1 é dado por (MIANO, 2009)

$$\left[M_e^{1D}\right]\{a_e\} = \left\{f_e^{1D}\right\}. \tag{2.9}$$

Os coeficientes da matriz de massa e do termo independente são calculados como

$$M_{e,ij}^{1D} = \int_{-1}^{1} \phi_i \phi_j(\xi_1) d\xi_1, \tag{2.10}$$

$$f_i^{1D} = \int_{-1}^{1} b(\xi_1)\phi_i(\xi_1) d\xi_1, \tag{2.11}$$

Para elementos bidimensionais, a aproximação de um problema de projeção, definido no quadrado local $\Omega = [\xi_1,\xi_2] \in [-1,1] \times [-1,1]$, pode ser tratado obtendo-se os coeficientes das matrizes de massa através das seguintes equações (KARNIADAKIS E S., 2005)

$$M_{ij}^{2D} = \int_{-1}^{1}\int_{-1}^{1} \phi_i(\xi_1,\xi_2)\phi_j(\xi_1,\xi_2) d\xi_1 d\xi_2, \tag{2.12}$$

Contudo, sabe-se que as funções de interpolação para elementos bidimensionais podem ser

construídas através do produto tensorial das funções de base unidimensionais

$$\phi_i\left(\xi_1,\xi_2\right) = \phi_a\left(\xi_1\right)\phi_b\left(\xi_2\right), \tag{2.13}$$

$$\phi_j\left(\xi_1,\xi_2\right) = \phi_p\left(\xi_1\right)\phi_q\left(\xi_2\right). \tag{2.14}$$

Substituindo as expressões (2.13) e (2.14) em (2.12), pode-se isolar os termos nas direções ξ_1 e ξ_2 em duas integrais. A partir daí, determina-se que os coeficientes da matriz de massa para quadrados serão dados pelo produto dos coeficientes da matriz de massa unidimensional nas direções ξ_1 e ξ_2, obtendo o resultado:

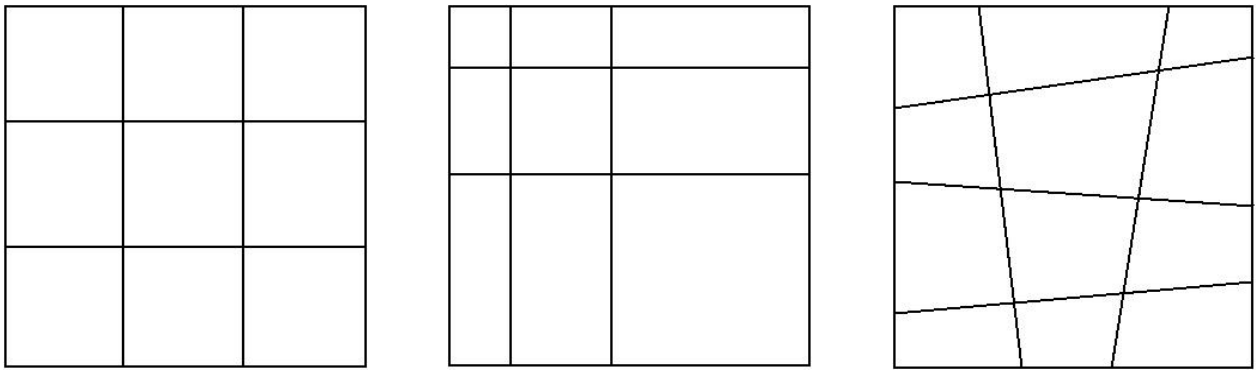
$$\begin{aligned} M_{ij}^{2D} &= \int_{-1}^{1}\int_{-1}^{1}\left[\phi_a(\xi_1)\phi_b(\xi_2)\right]\left[\phi_p(\xi_1)\phi_q(\xi_2)\right]d\xi_1 d\xi_2 \\ &= \left(\int_{1}^{-1}\phi_a(\xi_1)\phi_p(\xi_1)d\xi_1\right)\left(\int_{1}^{-1}\phi_b(\xi_2)\phi_q(\xi_2)d\xi_2\right) \\ &= M_{ap}^{1D}M_{bq}^{1D}. \end{aligned} \tag{2.15}$$

2.3.3 Interpolação entre malha de pós-processamento e malha de solução

O *software* hp^2FEM aborda a possibilidade de utilizar 3 tipos de malha, classificadas como uniforme, não-uniforme e não-estruturada. Como pode ser visto através da Figura 2.5, estes três tipos de malha podem ser utilizadas para representar o mesmo domínio, porém a forma dos elementos é diferente em cada uma delas. Para a malha uniforme, todos os elementos tem a mesma forma regular nas duas direções. Na malha não-uniforme, os elementos são regulares mas com proporções diferentes. E por fim, na malha não estruturada, os elementos possuem formas variadas.

A classificação mostrada pela Figura 2.5 descreve as malhas de acordo com a forma dos elementos. Contudo no *software* hp^2FEM, há outros tipos de abordagem de malha, que estão relacionadas ao grau polinomial e ao tipo de tarefa, a qual está destinada. Baseado nisso, o *software* é construído com 4 tipos de malhas: entrada, solução, mapeamento e pós-processamento.

A primeira malha, Entrada, é definida com ordem de interpolação igual a 1. A malha de solução é gerada, a partir da malha de entrada, em um dos módulos hp^2FEM, o qual gera novas coordenadas sobre a malha de entrada através da interpolação da funções de base nos pontos de colocação. A malha de mapeamento é utilizada para o cálculo da medida do elemento e de trans-

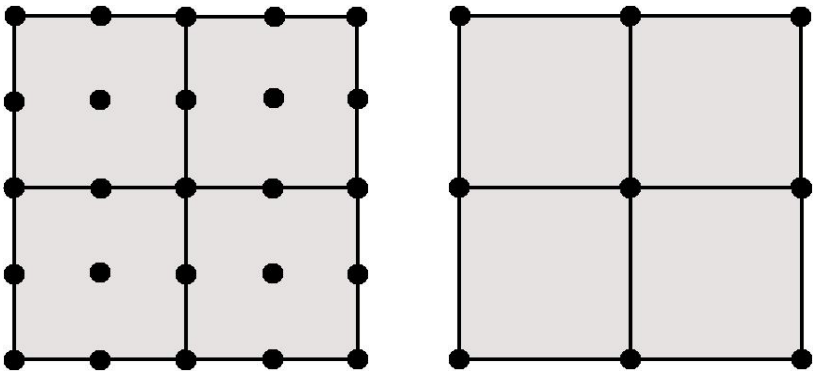


(a) Malha uniforme. (b) Malha não-uniforme. (c) Malha não-estruturada.

Figura 2.5: Classificação dos tipos de malha.

formação de coordenadas global-local. Por último, a malha de pós-processamento possui uma única ordem e é gerada para análise de resultados.

Considerando o MEF-AO, podem ocorrer situações nas quais é gerada grande quantidade de dados, devido ao alto grau polinomial na malha de solução. Dependendo da análise do pós-processamento que se deseja obter, esta pode ser dificultada pelo volume de dados apresentados. Por exemplo, se o objetivo for apenas analisar graficamente o comportamento da malha de elementos, pode não ser necessário utilizar na malha de pós-processamento o mesmo grau da malha de solução. A Figura 2.6 mostra um caso em que apesar da malha de solução ser interpolada com grau 2, a malha de pós-processamento exibe resultados com grau 1.



(a) Malha de solução. (b) Malha de pós-processamento.

Figura 2.6: Malhas de solução e pós-processamento.

Contudo, os resultados obtidos na malha de pós-processamento devem ser condizentes

com a solução para garantir que não haverá degradação do resultado. Para obter essa relação, implementou-se um algoritmo de interpolação entre a malha de solução e pós-processamento. Esse processo é exemplificado na Figura 2.7, considerando elementos unidimensionais. A Figura 2.7 exibe a interpolação em apenas um elemento 1D entre as malhas de pós-processamento e solução.

Figura 2.7: Elemento unidimensional - Interpolação da malha de solução com a malha de pós-processamento.

Considerando coordenadas locais em um intervalo $[-1,1]$, a malha de solução obtida no elemento da Figura 2.7 gera as seguintes funções baseadas em polinômio de Lagrange:

$$\begin{aligned} \phi_1(\xi_1) &= -\tfrac{1}{2}\xi_1\left(1-\xi_1\right), \\ \phi_2(\xi_1) &= \left(1-\xi_1^2\right), \\ \phi_3(\xi_1) &= \tfrac{1}{2}\xi_1\left(1+\xi_1\right). \end{aligned} \tag{2.16}$$

A interpolação entre as malhas é realizada utilizando as funções de base da malha de solução nos pontos de colocação. Logo, o resultado final dessa interpolação para obter o vetor solução da malha de pós-processamento em cada elemento é dado matricialmente por:

$$\left\{u_e^{Pos}\right\} = \left[N_{Sol,Pos}\right]\left\{u_e^{Sol}\right\}, \tag{2.17}$$

sendo $\left\{u_e^{Pos}\right\}$ o vetor de solução da malha de pós-processamento, $[N_{Sol,Pos}]$ a matriz de interpolação entre as malhas, onde as linhas representam os pontos de colocação do elemento da malha de pós-processamento e as colunas as funções da malha de solução e $\left\{u_e^{Sol}\right\}$ a solução já obtida na malha de solução. Para exemplificar com mais detalhes, dado os pontos da malha de pós-processamento -1 e 1 na Figura 2.7 e os valores da funções de base ϕ_1 a ϕ_3, a matriz $[N_{Sol,Pos}]$ resultaria nos seguintes valores:

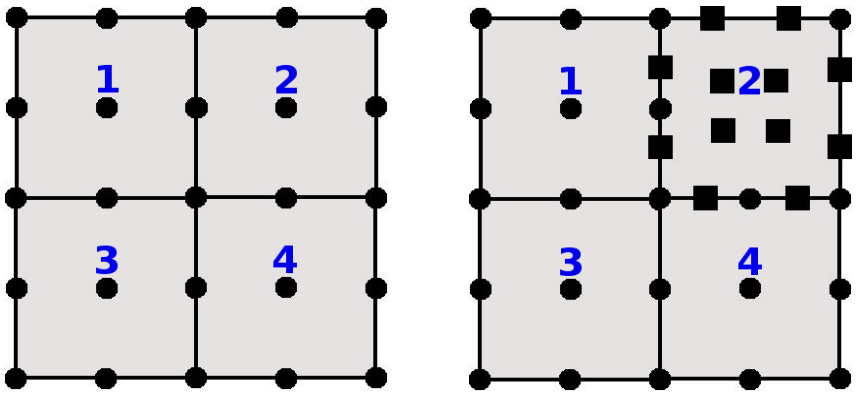
$$N_{Sol,Pos} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.18}$$

e o vetor resultado da malha de pós-processamento será de dimensão igual a 2×1 com o grau 1. Supondo que o resultado do vetor solução $\left\{u_e^{Sol}\right\}$ seja $\{0{,}5; 0{,}1; 0{,}3\}^T$, o resultado para a malha de pós-processamento será $\{0{,}5; 0{,}3\}^T$. Observa-se que mesmo interpolando a malha de pós-

processamento com grau menor do que da malha de solução, os dados permanecem coerentes, descartando apenas a solução para os nós internos da malha de solução do elemento 1D na Figura 2.7.

2.3.4 Solução p-Uniforme e p-Não-Uniforme

Uma outra abordagem para o MEF-AO é dada pela forma de interpolar ou gerar pontos na malha de elementos finitos. A solução obtida pelo método p-uniforme gera o mesmo grau polinomial para toda a malha. Entretanto, considerando a solução local descrita na Seção 2.3.1, a malha pode ser interpolada com graus diferentes em cada elemento, o qual denominamos no *software* hp^2FEM como p-não-uniforme. Uma ilustração entre a diferença de interpolação p-uniforme e p-não-uniforme é representada pela Figura 2.8 para uma malha com 4 elementos quadrados. Na solução para p-uniforme, a malha na Figura 2.8(a) é interpolada com grau polinomial 2 para todos elementos. Para a solução na Figura 2.8(b), a interpolação nos elementos emprega as ordens [2, 3, 2, 2] (da esquerda para direita, de cima para baixo).



(a) Malha p-uniforme. (b) Malha p-não-uniforme.

Figura 2.8: Malhas p-uniforme e p-não-uniforme.

A solução p-não-uniforme pode colaborar no ganho de desempenho para problemas mais complexos, onde a interpolação com grau maiores pode ser aplicada apenas em locais ou elementos da malha, onde seja mais necessário. Contudo, nesse trabalho, essa abordagem procura apenas verificar o funcionamento adequado, ou seja, se o erro calculado para p-uniforme e p-não-uniforme são similares.

As classificações de malha e estratégias anteriores apresentadas para essa arquitetura de *software* podem ser combinadas para obter diferentes tipos análises a respeito da avaliação de custo computacional e convergência do erro. Um exemplo é a combinação de malhas uniformes e não-uniformes com p-uniforme e p-não-uniforme, em que pode ser avaliado a convergência do erro entre soluções p-uniforme e p-não-uniforme. Estratégias como essas serão aplicadas para o problema de projeção descrito na Seção 3.6.2.

No algoritmo implementado para o problema de projeção, a combinação de malha uniforme com p-uniforme permite o cálculo da matriz de massa (lado esquerdo da equação, como visto em 2.3) e a matriz jacobiana apenas para o primeiro elemento da malha. Isso ocorre, pois os termos da matriz de massa local e o jacobiano da transformação são constantes, sendo necessário que sejam calculados somente em uma única vez durante o processo de solução da malha.

Para a combinação de malha não-uniforme com p-uniforme, apenas os termos do jacobiano são calculados para cada elemento, uma vez que, nesse caso, a matriz de massa local ainda é constante ao longo dos elementos da malha. Para o caso geral, onde tem-se malhas p-não-uniforme e não estruturadas, a matriz de massa e o jacobiano são construídos para cada elemento.

3 Arquitetura de *Software* Proposta

Nesse capítulo, abordam-se conceitos relacionados as técnicas de estruturação e engenharia de *software*, assim como o paradigma de orientação por objeto utilizados no desenvolvimento do *software* hp^2FEM. Além disso, apresenta-se o conjunto de ferramentas utilizadas para dar suporte ao desenvolvimento.

Em seguida, apresenta-se a arquitetura proposta demonstrando com as visões arquiteturais com as principais decisões tomadas durante o desenvolvimento, os principais componentes e os seus relacionamentos. Em resumo, a representação da arquitetura é dada através das visões lógica e de processos. Assim, na Seção 3.4 é visto a organização geral do código usando diagramas de classes e por camadas. Alguns detalhes da configuração das entradas de dados são descritos na Seção 3.5. E por último, na Seção 3.6, apresenta-se o fluxo dos principais pacotes do código, demonstrando o funcionamento do hp^2FEM.

3.1 Conceitos Gerais de Arquitetura e Engenharia de *Software*

A arquitetura de um *software* é um conceito de fácil entendimento e intuitivo, porém é difícil de definição. Alguns autores como CLEMENTS *e outros* (2010) e GARLAN e SHAW (1994) definem a arquitetura de *software* como uma estrutura que descreve o sistema como um todo e que colabora na tomada de decisões de alto nível, desempenhando o papel de ligação entre as fases de requisitos e processos.

Outros conceitos são comuns à engenharia de *software*. Dentre esses, têm-se a modularização e decomposição, com os quais sistemas de *software* são particionados em componentes modulares menores, cada um tratando de característica particular do sistema (OLIVEIRA, 2007). Por outro lado, deve-se levar em consideração na arquitetura de um *software* a descrição de como os seus componentes devem interagir, estando de acordo com os principais requisitos: desempenho, escalabilidade, interoperabilidade, portabilidade e confiabilidade.

O gerenciamento de um *software* pode ser separado por visões, devido à complexidade em descrever a arquitetura (CLEMENTS *e outros*, 2010). As visões em uma documentação de *software* define cada parte do sistema e consiste em reduzir a quantidade de informações a serem

tratadas pelo arquiteto em um dado momento. CLEMENTS *e outros* (2010) propõem cinco tipos de visões padrão: dados, módulo, implantação, execução e implementação. Contudo, pode-se determinar visões adicionais para manifestar outras questões de um determinado sistema como a visão da interface do usuário e a de segurança. Abaixo são apresentadas algumas visões, as quais são muito utilizadas nas documentações de *softwares* genéricos:

- Visão de dados - Utilizada quando o sistema possui uma base de dados, onde a sua estrutura necessita ser modelada.
- Visão de módulo - Geralmente representada por diagramas UML (*Unified Modeling Language*) que descrevem as unidades do sistema, representando a planta para a construção do *software*.
- Visão de implantação - Descreve a estrutura de *hardware* do sistema. É muito utilizada em sistema distribuídos.
- Visão de execução - Mostra o fluxo de execução do sistema facilitando a compreensão do seu funcionamento, podendo ser representada por diagramas UML de sequência ou atividades. É útil para examinar o desempenho e outras propriedades do sistema em tempo de execução.
- Visão de implementação - Pode ser representada por diagramas de componentes e descreve como os arquivos da arquitetura de *software* são organizados em diretórios tanto para o ambiente de produção como para o de desenvolvimento.
- Visão de caso de uso - Mostra um conjunto de cenários e/ou os casos de usos que exibem funcionalidades centrais e significativas do sistema.

Algumas dessas visões foram utilizadas nesse trabalho. A forma com que se buscou organizar a estrutura do *software* demonstra o uso da visão de implementação. A visão de execução foi utilizada de forma indireta e colaborou na correção de alguns acoplamentos dos módulos. Por último, destaca-se a visão de módulo, utilizada durante todo o processo de desenvolvimento, tanto no planejamento inicial como na reformulação de alguns componentes.

3.1.1 Técnicas de Estruturação de *Software*

Durante a estruturação de um *software*, principalmente em sistemas mais complexos, devem ser considerados conceitos como: modularização, procedimento de particionamento em que cada parte do projeto de *software* seja bem delimitada; visão hierárquica, proporcionando uma visualização do sistema com uma divisão em diferentes níveis de abstração; baixo acoplamento entre os elementos, que determina uma maior independência entre as partes; e a alta coesão, que agrupa as partes inter-relacionadas, reduzindo a troca de mensagens entre as partes do sistema (RUBIRA E BRITO, 2007).

Além disso, um projeto de arquitetura de *software* pode ser determinado por meio de dois enfoques principais: *top-down* e o *bottom-up*. No primeiro, o sistema é decomposto recursivamente "de cima para baixo", gerando módulos ou funções até que esses sejam reconhecidos como módulos facilmente implementados. O enfoque *top-down* descreve o estado de um sistema centralizado e compartilhado com as funções atuantes. Neste projeto, o enfoque desenvolvido foi o *bottom-up*, que está mais relacionado com as características do paradigma de programação de orientação a objetos, descrito mais adiante. Nesse caso, o sistema é visto como uma coleção de blocos, e o estado do sistema é decentralizado entre os objetos do sistema, sendo assim, cada objeto opera sobre seu próprio estado (VASCONCELOS *e outros*, 2006). Essa forma de implementação permite o desenvolvimento independente das partes do *software* ajudando a obter um maior foco na implementação do objeto a ser tratado. Colabora ainda com o trabalho em um grupo de desenvolvedores, no qual cada um pode atuar em diferentes partes do *software* sem a necessidade de possuir o programa por completo.

3.1.2 Paradigma de Programação Orientada a Objeto (POO)

No processo de desenvolvimento do *software* também incluímos a escolha de um paradigma de programação. No programa hp^2FEM, optou-se pelo modelo de orientação a objeto que representa melhor objetos do mundo real no domínio da aplicação e nas funcionalidades requeridas pelos mesmos, por meio de sua interação. Esse paradigma, inicialmente criado em 1967 por Dahl e Nygaard, e consolidado em 1976 com a linguagem Smaltalk, descreve conceitos de classes, objetos, tipos, encapsulamento de informações, polimorfismo, parametrização, métodos virtuais e herança (RUBIRA E BRITO, 2007). Esses conceitos definem como os sistemas serão estruturados. Alguns

desses, são descritos a seguir:

- A característica de herança em POO é representada por uma hierarquia de classes especializadas, onde a mais específica herda métodos e atributos de uma classe mais geral, respectivamente, denominadas como sub-classe e super-classe.
- O conceito de encapsulamento é o efeito de programar dados e métodos em um mesmo objeto, protegendo a estrutura interna do usuário do objeto. Com isso, uma das vantagens adquiridas é poder modificar um objeto internamente sem afetar outros módulos do sistema que utilizam-se do objeto modificado.
- O polimorfismo permite criar sistemas mais claros e flexíveis, podendo ser dado de duas formas, dinâmico e estático. O polimorfismo estático permite criar funções com o mesmo nome com argumentos diferentes. A definição de qual método será chamado é definido em tempo de compilação através dos argumentos que foram passados. O polimorfismo dinâmico está relacionado ao conceito de herança e permite a redefinição de um método da super-classe na sub-classe. A decisão de qual método será chamado é dado em tempo de execução (DEITEL E DEITEL, 2008).
- Os métodos ou funções virtuais geralmente são utilizados para resolver o problema de ambiguidade propiciado pelo polimorfismo dinâmico. Chamadas dos métodos de uma classe derivada de uma classe base podem ser ambíguas, visto que um objeto da classe derivada por ser referenciado tanto pela classe base como pela derivada. Com o uso da função virtual, declarada na classe base com a palavra-chave *virtual*, o compilador gera versões de uma função e força sempre a chamada das funções dos objetos referenciados. Aconselha-se o uso de funções virtuais quando a função sobrescrita for modificada futuramente.

A modelagem de um *software* pode ser feita por meio de UML. Essa linguagem surgiu em 1995 da união dos métodos de modelagem de *software* propostos por Booch, Jacobson e Raumbaugh (FERNANDES, 2011). Sua modelagem pode ser feita independentemente da linguagem de programação a ser utilizada. Durante a criação do modelo UML, podem ser arquitetados diversos tipos de diagramas, com diferentes finalidades para cada etapa do projeto.

Dentre esses tipos de diagramas, no hp^2FEM foi utilizado o diagrama de classe, representando os diversos módulos que constituem o *software*. Também conhecidos como diagramas estáticos, descrevem o sistema por meio das definições das classes com seus métodos e atributos e

o relacionamento entre elas. Nesse relacionamento, indica-se, por exemplo, as associações entre classes, bem como quais classes são tipos de atributos das outras classes. Contudo, não é exibido a troca de mensagens entre essas classes.

3.2 Infraestrutura de *Software* Construída (*Framework)*

Durante o projeto foram empregadas ferramentas que fossem capazes de gerenciar cada módulo do *software* hp^2FEM de maneira independente durante as fases de modelagem, implementação, tratamentos de erros, verificação de vazamento de memória, entre outras, e a criação de um conversor de expressões simbólicas para valores numéricos. Essas ferramentas são descritas a seguir.

3.2.1 Ferramenta de Modelagem e Documentação

Os diagramas e a documentação do programa hp^2FEM foram desenvolvidos a partir da ferramenta Metamill de multiplataforma e modelagem UML 2.3, desenvolvido pela companhia Metamill Software de Luxemburgo (HAVEN, 2011).

Esse *software* foi utilizado principalmente para criação de diagramas de classes ao longo desse projeto. Através de interface gráfica, pode-se inserir informações sobre as classes, como os atributos, métodos e parâmetros. Do mesmo modo, foi possível a adição do código implementado para uma determinada funcionalidade e posteriormente gerar os códigos direcionando a um diretório configurado.

O Metamill armazena todas as informações em um único arquivo baseado em XMI (*XML Metadata Interchange*) (OMG, 2011), sendo possível editá-lo separadamente. As importações e exportações dos diagramas podem ser realizadas nesse formato.

Embora a documentação das classes pudesse ser gerada utilizando-se o próprio Metamill, criou-se uma opção de documentação utilizando o Doxygen (HEESCH, 2012), um *software* que gera documentação em formatos html, latex, rtf e xml a partir do código fonte. Assim, as palavras-chaves utilizadas pelo Doxygen foram inseridas dentro das interfaces do Metamill.

De forma geral, os passos de modelagem, codificação e documentação desse projeto realizaram-se de forma concorrente aplicando-se a estratégia *bottom-up* discutida na Seção 3.1.1.

3.2.2 Monitoramento de Memória

A tarefa de gerenciamento de memória dinâmica em programas escritos na linguagem C++ deve sempre estar a cargo do programador. Dependendo da complexidade do sistema, erros no uso incorreto de memória dinâmica podem não ser percebidos pelo compilador. Considerando isso e o uso repetitivo de vetores extensos no projeto hp^2FEM, utilizou-se o *software* Valgrind (BROWN, 2011) para o monitoramento de memória.

Esse *software* possui algumas ferramentas como o Memcheck, que através de um mapa de *bits* aponta quais áreas de memórias estão ou não alocadas e quais estão alocadas e inicializadas. Com essa ferramenta, o Valgrind consegue detectar erros como: vazamento de memória, leitura ou escrita em regiões de memória já desalocadas ou não alocadas, uso de variáveis ou ponteiros não iniciados, leitura ou escrita em regiões de memória que ultrapassa o limite de memória alocada, o uso incorreto de funções que desalocam ou alocam o bloco de memória.

Atrelado ao Valgrind, pode-se utilizar o Vakyrie, uma ferramenta de interface gráfica baseado em Qt4 (SUMMERFIELD, 2010), que exibe o monitoramento de memória através do arquivo de saída em XML *eXtended Markup Language* (HEITLINGER, 2001) do Valgrind. Nesse monitoramento, como mostra a Figura 3.1, é possível visualizar uma árvore recursiva do fluxo de execução do programa até o momento em que se encontra o erro devido ao uso incorreto de memória.

O Código 3.1 apresenta um *script* programado em *Shell Script* para gerar os arquivos gráfico e em formato texto com informações sobre a memória utilizada no programa. Juntamente ao *script*, exibe-se a lista dos principais parâmetros do *software* Valgrind na geração desses arquivos.

- *-v* - Exibe vários aspectos sobre o programa.
- *–tool=<Nome da ferramenta a ser utilizada pelo Valgrind>*
    - *callgrind* - É uma ferramenta de análise que armazena o histórico de chamadas entre funções de um programa.

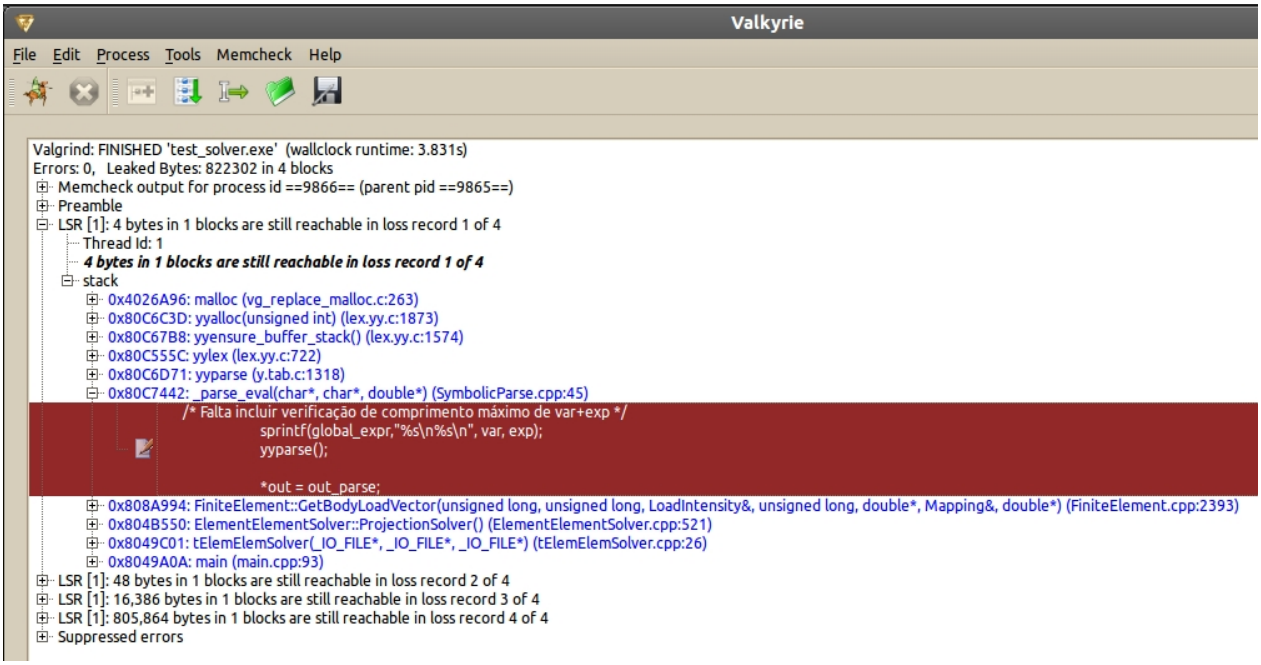


Figura 3.1: Monitoramento do uso de memória.

- *memcheck* - Ferramenta padrão do Valgrind que permite monitorar o uso de memória.

- *–track-origins=yes* - Ativa o rastreamento de variavéis não inicializadas.
- *–leak-check=full* - Exibe detalhes sobre cada vazamento de memória individualmente.
- *–show-reachable=yes* - Exibe todos os blocos.
- *–xml=yes* - Exibe as partes importantes em XML.
- *–xml="LogFile".xml* - Especifica o arquivo em que o Valgrind deve escrever sua saída XML.

Código 3.1: *Script* em Shell Script para verificar o uso de memória.

```
1 #! /usr/bin/bash
2
3 RUNFILE=./test_solver.exe
4 LOGFILE=./test_solver.xml
5 MEMCHECK=./memcheck.out
6
7 valgrind -v --tool=callgrind --tool=memcheck --track-origins=yes
8 --leak-check=full --show-reachable=yes --xml=yes --xml-file=$LOGFILE
9 $RUNFILE > $MEMCHECK
```

```
10
11 valkyrie -l $LOGFILE
```

3.2.3 Analisador Simbólico-Numérico

Alguns dados de entrada do hp^2FEM, tais como condições de contorno e carregamentos, podem utilizar expressões simbólicas ou funções que a princípio não conhecem os valores das suas variáveis. Devido à essas expressões, desenvolveu-se um analisador simbólico-numérico, que converte uma expressão, dado os valores das variáveis, em um valor numérico resultado de uma função de entrada. A Figura 3.2 representa esse fluxo de execução. Para isso, construiu-se a biblioteca do analisador através das ferramentas *lex* e *yacc* para sistemas Unix, que são geradores de analisadores léxicos e sintáticos.

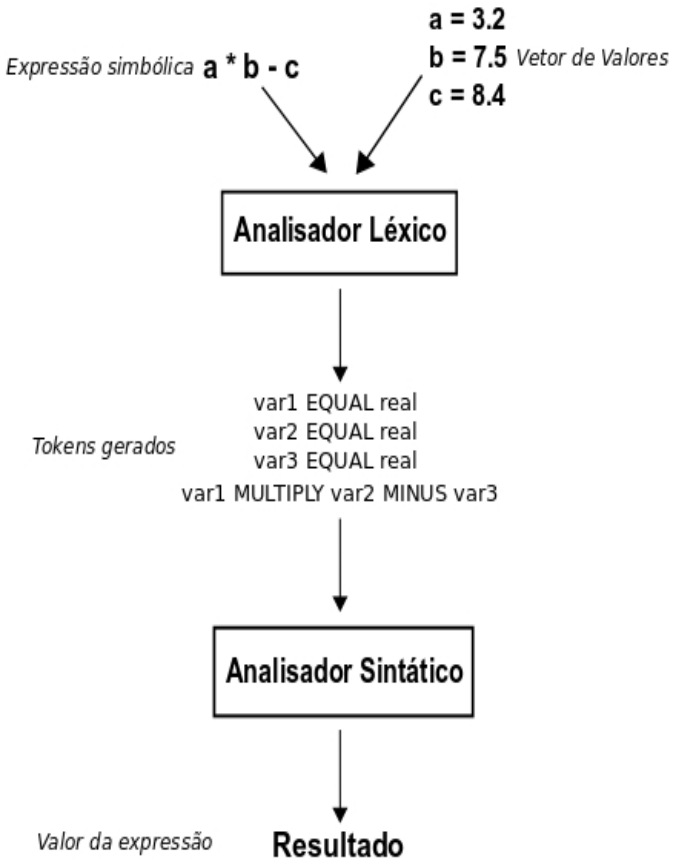


Figura 3.2: Analisador simbólico-numérico.

A descrição da análise léxica é realizada através de expressões regulares que são construídas em um arquivo com a extensão '.l'. A partir desse arquivo, a ferramenta lex produz um autômato finito (máquina de estados) que serve para reconhecer padrões de uma linguagem, como a que estará descrita no arquivo '.l'. Juntamente com rotinas em C ou C++, esse autômato é gravado em um arquivo 'lex.yy.c', que será compilado e ligado ao *software* (SANTOS, 2006).

O arquivo '.l' é dividido em 3 seções, como mostrado no Exemplo 3.1, sendo que a última pode ser excluída.

```
seção de declarações
\%\%
seção de regras
\%\%
seção de rotinas
```

Exemplo 3.1: Formato do arquivo '.l' para gerar o analisador léxico.

A seção de declarações é o local onde são colocadas as declarações de variáveis, constantes e/ou bibliotecas a serem incluídas no código. Essas definições devem estar entre os caracteres especiais '%{' e '%}'. Em seguida, na zona de regras, pode-se definir o formato dos *tokens* a partir de expressões regulares, com objetivo de simplificar o seu uso. O Exemplo 3.2 exemplifica o uso dessas substituições.

```
variable    [a-z][A-Z]
real        [0-9]+[eE][+-]?[0-9]+
MULTIPLY    ``*''
```

Exemplo 3.2: Seção de declarações do arquivo '.l'.

A seção de regras é o local em que se definem as expressões regulares, associando uma determinada ação semântica definida por uma linguagem, podendo ser C ou C++ (SANTOS, 2006). Essas expressões são definidas por um conjunto de regras que permite identificar padrões como uma sequência de caracteres pertencente a uma certa linguagem. O Exemplo 3.3 exibe a seção de regras utilizada no analisador simbólico-numérico.

O lado esquerdo é onde ocorre o reconhecimento da expressão regular, nesse caso a expressão foi definida anteriormente na seção de declaração (Exemplo 3.2). O lado direito define a ação

```
{real}      {
              yylval = atof(yytext);
              return NUMBER;
            }
{MULTIPLY} {return MULTIPLY; }
```

Exemplo 3.3: Seção de regras do arquivo '.l' empregada no analisador simbólico-numérico.

semântica a ser executada. Em um dos exemplos anteriores é identificado a expressão 'real', posteriormente são executadas as ações em que se converte o conjunto de *strings* reconhecido em um valor 'real', por conseguinte, esse valor é passado ao analisador sintático como *token* 'NUMBER'. Por fim, a seção de código do arquivo '.l' é facultativa e onde pode-se definir rotinas auxiliares em C que serão copiadas inteiramente para o arquivo `lex.yy.c`. O arquivo completo que foi utilizado nesse trabalho para gerar o analisador léxico pode ser visto na Seção A.1 do Apêndice A.

O próximo passo do analisador simbólico-numérico é o analisador sintático produzido através da ferramenta *yacc*, a qual gera uma rotina chamada *yyparse*. A análise sintática a ser gerada identifica um sequência de texto descrita por uma GLC - Gramática Livre de Contexto. Uma GLC pode ser definida por $G = \{V, \sum, P, S\}$, onde V são os símbolos não terminais ou variáveis; $\sum$ representa os símbolos terminais, ou seja um conjunto do alfabeto definido pela linguagem em G; P descreve as regras de substituições ou produções gramaticais; e por último S que define o símbolo inicial da gramática e deve ser não terminal.

Para uma breve exemplificação, considere uma gramática GLC, dada por $G = \{T,S,i,j,P,S\}$, onde P são as produções:
$S \rightarrow T$
$T \rightarrow jTj$
$T \rightarrow iTi$
$T \rightarrow i$
Logo, uma derivação que pode ser obtida com essa gramática seria: $S \rightarrow T \rightarrow jTj \rightarrow jiTij \rightarrow jiiij$.

Assim como nesse exemplo, a definição da gramática aqui considerada seguiu o mesmo parâmetro, porém para o uso de equações simbólicas. Seguindo esses passos, a gramática desenvolvida é colocada dentro de um arquivo '.y' e o analisador sintático é gerado pela ferramenta *yacc*. Esse arquivo tem características parecidas com o arquivo '.l', separado por três seções: declaração, gra-

mática e código entre os caracteres especiais '%%'.

Na seção declaração, deve-se redeclarar os *tokens* que possuem mais de um caractere com palavra-chave '%token'. Outros *tokens* devem ser redeclarados para tratamento de ambiguidades da gramática, em que uma mesma sequência de entrada é reconhecida por produções distintas do lado direito da regra. Essas redeclarações utilizam definições de associatividade e prioridade da ferramenta *yacc*. Abaixo segue um trecho do analisador, exemplificando a redeclaração de alguns *tokens*:

```
\%token   NEW_LINE
\%left    PLUS MINUS
\%left    MULTIPLY DIVIDE
\%right   POWER
```

Exemplo 3.4: Seção de declarações do arquivo '.y' para o analisador simbólico-numérico.

O efeito de prioridade dos *tokens* é definido pela ordem decrescente de declaração no arquivo '.y'. Portanto, as operações do Exemplo 3.4 de multiplicação e divisão serão priorizadas em relação as operações de soma e subtração em uma determinada função. O critério de associatividade é determinado pelas palavras-chaves '%left' e '%right'. No exemplo, o *token* 'POWER' representa potenciação, isto é, dado a expressão x^2 seria o mesmo que x elevado a 2. Nesse caso, a palavra-chave '%right' define que a associatividade da operação estará a direita. Por exemplo, se uma função for descrita pela *string* xˆyˆ2, o uso da associatividade define que a primeira operação a ser executada será yˆ2, o mesmo que escrever a função da seguinte maneira xˆ(yˆ2).

A seção de código do arquivo '.y' implementa as funções auxiliares em C, como por exemplo funções para o tratamento de erro. Por último, a seção de código define a gramática GLC como descrita anteriormente. No Exemplo 3.5, tem-se uma parte da gramática descrita no código do analisador. Seguindo a descrição anterior, define-se um conjunto de regras (produções), onde um símbolo não terminal é colocado à esquerda do separador ':' e conhecido como alvo. No lado direito, coloca-se uma sequência de símbolos (terminais e não terminais) concluído por ';'. Quando há várias sequências, essas devem ser separadas por '|'. O arquivo para gerar o analisador sintático pode ser visto na Seção A.2 do Apêndice A.

Atrelado aos arquivos '.l' e '.y' para os analisadores léxico e sintático, foi implementado um arquivo '.cpp' para interface com o código hp^2FEM, conforme pode ser visto na Seção A.3

```
Input:
       /* Empty */
       | Input Line
       ;
Line:
      NEW_LINE
      | Expression NEW_LINE
      | VARIABLE EQUAL Expression { int ind = $1; sym[ind] = $3; }
      ;
Expression:
            NUMBER { \$$ = $1; }
            | Expression PLUS Expression  { $$ = $1 + $3; }
            | Expression DIVIDE Expression  { $$ = $1 - $3; }
            ;
```

Exemplo 3.5: Seção de código do arquivo '.y' para o analisador simbólico-numérico.

do Apêndice A. Como demonstrado na Figura 3.2, essa interface recebe como entrada dois *arrays* de *strings*: um que armazena o valor das variáveis e outro a expressão ou equação simbólica a ser calculada, exemplos: "$x = 2\ \ y = 3\ \ z = 5{,}5$" (variáveis) e "$x * 5 + y^3 - sin(z)$" (equação). Entre as tarefas das rotinas auxiliares estão a conversão de variáveis em caixa alta para caixa baixa, ou seja, X para x; e a junção dos dois *arrays* de entrada em um único *array* de caracteres, considerando que o analisador simbólico-numérico somente poderá receber uma entrada.

A saída do analisador é armazenada por uma variável global do tipo *double* com o cálculo da equação simbólica. Para facilitar a solução das equações no analisador sintático, o armazenamento das variáveis foi realizado em um *array* $sym[]$ de valores numéricos, sendo que cada índice do *array* é representado pela ordem em que variável aparece no alfabeto português. Assim, dado a entrada das variáveis por "$x = 2\ \ y = 3\ \ z = 5{,}5$", os campos de índices 23, 24 e 25 do vetor de variáveis $sym[]$ estará armazenando os valores 2, 3 e $5{,}5$ respectivamente.

Em geral, a equação de entrada é quebrada pelo analisador sintático em partes como $expressao\ OPERADOR\ expressao$. Para cada produção gramatical como essa, executa-se a operação correspondente dentro de uma ação semântica definida em linguagem C. A operação é executada sobre os valores já armazenados no vetor de variáveis $sym[]$ ou constantes numéricas. Quando a última operação for executada, o valor resultante da equação será armazenado na variável global de saída, concluindo a execução do analisador simbólico-numérico.

3.3 Abordagem Adotada para a Arquitetura

Esta seção fornece uma visão geral da arquitetura do *software* hp^2FEM. O projeto, desenvolvido em linguagem C++, utiliza todas as ferramentas descritas na Seção 3.2. A modelagem e implementação foi baseada em uma versão anterior desenvolvida em Matlab pelo grupo do Laboratório de Simulação Computacional da Faculdade de Engenharia Mecânica da Universidade Estadual de Campinas. Contudo, houve uma versão anterior a essa, escrita em C++ (SILVA, 1997), porém com objetivos mais específicos. SILVA (1997) apresentou a otimização estrutural e análise de sensibilidade em elasticidade linear e não-linear em problemas bi e tridimensionais.

No entanto, o escopo desse projeto trata do desenvolvimento de uma arquitetura para o MEF-AO de maneira generalizada, modular, otimizada e de fácil usabilidade. A seguir, exibe-se a descrição da arquitetura através de diagramas de classes e atividades, além dos dados de entrada para o *software*. Para a validação do código, considera-se o procedimento local de projeção descrito na Seção 2.3.1.

3.3.1 Metas e Restrições Arquiteturais do hp^2FEM

O programa hp^2FEM tem como principal objetivo permitir a flexibilidade e a generalização na implementação do MEF-AO, facilitando o seu uso, manutenção e o acoplamento de novas bibliotecas. Juntamente a isso, busca-se a eficiência do uso de recursos computacionais, tanto na modelagem, como na implementação através de boas práticas de programação em C++.

Esse projeto foi desenvolvido através da implementação independente para cada módulo, começando a partir de pacotes mais básicos, até as aplicações a serem incluídas nas camadas superiores do sistema. Para cada pacote foi gerada uma biblioteca estática com uso de Makefiles (STALLMAN *e outros*, 2010) e posteriormente incluídas em pacotes de níveis mais elevados. Concluídos os módulos, construiu-se a árvore de camadas dos algoritmos principais como será visto adiante.

Além dos requisitos gerais dados na Seção 1.3, o hp^2FEM apresenta as seguintes restrições e metas de desenvolvimento mais específicas:

1. A funções de formas a serem construídas devem permitir o uso de bases modal e nodal. Para esse projeto foram implementados classes para os polinômios de Lagrange truncado, Lagrange padrão, Jacobi, Hermite e Lobato (KARNIADAKIS E S., 2005).

2. A implementação realizada suporta condições de contorno homogêneas e não homogêneas, podendo serem aplicadas em diferente entidades como nós, arestas e faces dos elementos e linhas, superfícies e volumes da geometria.

3. O hp^2FEM faz o uso de diferentes tipos de matrizes como as simétricas e esparsas. Essas classes foram construídas no *software* sem o uso de bibliotecas externas. As operações e alocação essas estruturas são feitas de forma unidimensional.

4. Armazenamento das relações entre a geometria e malha.

5. Cálculo de carregamento nodais equivalentes com intensidades aplicadas às arestas e faces dos elementos e linhas, superfícies e volumes da geometria.

6. Uso de diversos tipos de materiais através de generalização, sendo possível acrescentar materiais específicos quando necessário.

7. Construção de incidência e coordenadas nodais de alta ordem para os elementos, considerando o caso de vários grupos de elementos de formas diferentes.

8. Gerenciamento automático da disposição dos pontos de interpolação no elemento.

9. Adaptatividade p com uso de ordem polinomial não-uniforme, ou seja, variação polinomial em uma mesma malha do grupo de elementos.

10. Definição de mais de um tipo de grupo de elementos na malha de elementos finitos a ser gerado, permitindo formas diferentes para a mesma malha de elementos finitos.

11. Estratégias de solução através do produto tensorial de operadores de elementos unidimensionais para o cálculo das funções de interpolação e operadores de elementos 2D e 3D.

12. Conversão de funções simbólicas para valores numéricos através do analisador simbólico-numérico

13. Tratamento de malhas uniformes, não-uniformes e não-estruturadas.

14. Uso de diferentes tipos de malhas para cada etapa do programa. Nesse caso, o *software* possui objetos de malha para entrada, solução, mapeamento e pós-processamento, podendo utilizar diferentes ordens polinomiais em cada malha.

3.4 Visão Lógica

A visão lógica será demonstrada através da subdivisão do hp^2FEM em camadas e pacotes principais. Em cada pacote, exibem-se diagramas de classes e o relacionamento entre essas. Além disso, serão descritas as principais operações e atributos de classes mais significativas ao programa.

3.4.1 Visão Geral da Arquitetura

A Figura 3.3 apresenta uma visão geral da arquitetura organizada por camadas e pacotes e classes mais importantes. A seguir, apresenta-se uma descrição sucinta das camadas, desde a camada de solução até as camadas bases dos sistema. Já na Seção 3.4.2, apresenta-se de forma inversa, iniciando-se das classes bases, uma descrição detalhada das camadas por meio de diagrama de classes.

- **Camada Solver**
  Representada principalmente pela classe ***Solver***. Este nível, trata as aplicações do MEF-AO. Como visto na Figura 3.3, os métodos de solução são divididos em duas classes principais, ***ElementElementSolver*** e ***GlobalSolver***.

- **Camada Model**
  Descreve as principais características do modelo de elementos finitos como a numeração e os tipos de graus de liberdade (classes ***Equations*** e ***DOFs***), as coordenadas de cada nó (classe ***Nodes***), as condições de contorno (classe ***BoundaryConditions***), o conjunto de cargas aplicadas (classe ***LoadSets***), as relações entre malha e geometria (classe ***GeometryMesh***), a topologia da malha (representando pela classe ***MeshTopology***). Dentro deste componente é importante ressaltar a classe ***HighOrder*** que gera e gerencia os graus de liberdade de alta ordem polinomial.

- **Camada Group**
  Este nível é representado por um grupo de elementos finitos que possuem características idênticas como a dimensão, material, forma do elemento e funções de interpolação (classe ***Group***). As principais classes são descritas a seguir:

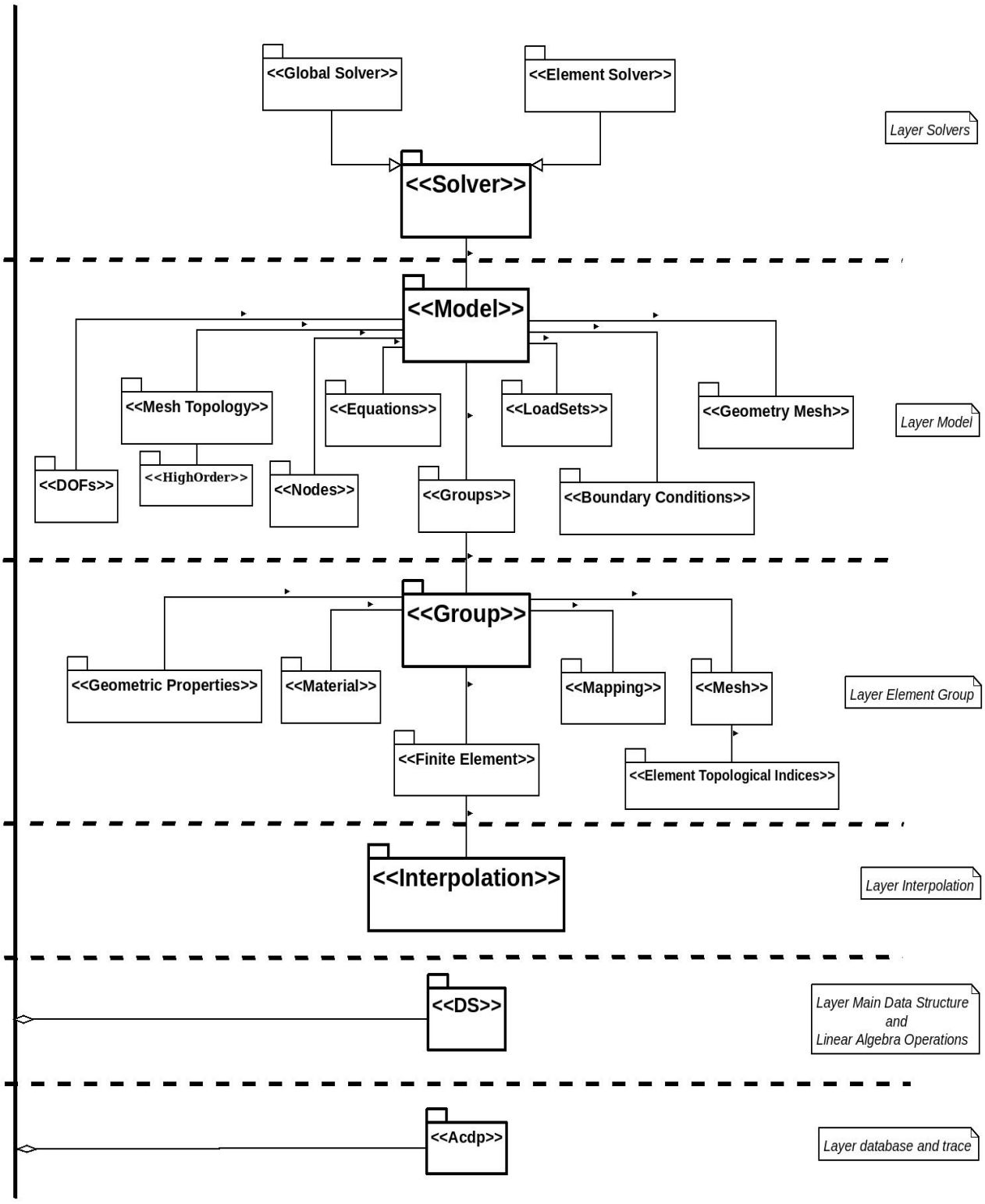


Figura 3.3: Visão geral da arquitetura hp^2FEM - Disposição dos componentes em camadas.

- Classe ***Material***: define as propriedades específicas de um determinado material, calcula os tensores e as componentes de tensão.
- Classe ***Mapping***: calcula a matriz de jacobiano e seu determinante em cada ponto de integração.
- Classe ***Mesh***: armazena dados da malha, tais como a incidência e coordenadas dos elementos e a numeração dos graus de liberdade dos nós do elemento.
- Classe ***ElementTopologicalIndices***: está agregada à classe ***Mesh*** e é responsável por construir e armazenar a numeração local das entidades topológicas dos elementos.
- Classe ***FiniteElement***: é responsável pelo cálculo dos operadores do elemento, tais como matrizes de massa e rigidez e vetores de carregamento.

◦ **Camada Interpolation**

Composta pelas classes ***CollocationPoints***, ***NumericalIntegration***, ***Polynomials1D*** e ***ShapeFunctions***. Essas classes calculam, respectivamente, os pontos de colocação, os pontos de integração, os polinômios e suas derivadas, os quais são utilizados no cálculo das funções de forma na classe ***ShapeFunctions***.

◦ **Camada Data Structure - DS**

Nesta camada estão as estruturas básicas do hp^2FEM responsáveis por armazenar informações numéricas sobre o método de elementos finitos em *arrays* e operações sobre esses valores como métodos para solução de sistema de equações.

◦ **Camada ACDP**

No modelo hp^2FEM, este módulo não está incluso no desenho UML da arquitetura proposta, ou seja, nesse nível as implementações foram realizadas de forma independente das classes do modelo. Esse ambiente é composto por funções auxiliares em linguagem C para dar suporte a todo código, como, rotinas de armazenamento e gerenciamento de arquivos, depuração e banco de dados.

3.4.2 Descrição e Relacionamento Estático dos Principais Pacotes

Camada ACDP

O pacote ACDP (Ambiente Computacional para Desenvolvimento de Programas), implementa procedimentos para gerenciamento de bancos de dados, armazenamento e leitura de arquivos, tratamento de erros e depuração. Essas operações utilizam estruturas de dados dinâmicas como pilha, lista e fila (GUIMARÃES E FEIJÓO, 1989).

Entretanto, as principais rotinas utilizadas nesse projeto são: ***geterror*** para indicação de erros específicos e ***FindKeyWord***, que localiza as palavras-chaves dos arquivos de entrada. Entre outras rotinas, encontram-se as auxiliares para ordenação de estrutura de dados construídas para o *software* hp^2FEM, usados principalmente na camada **Model** e no componente ***MeshTopology***. Além disso, empregam-se as funções de banco de dados para o armazenamento, em formato binário, de dados gerados pelo programa.

Para a leitura de dados é sempre utilizado a função ***getEnumIndex***, para conversão de caracteres, como determinadas palavras-chaves ou indicação da forma dos elementos finitos, para valores enumerados em C++. Para essas informações de entrada são construídos uma lista de estruturas de enumerados para armazenar dados de forma numérica, facilitando a manipulação dos dados em C++, como comparações entre palavras-chaves.

Camada DS - *Data Structure*

Nesta camada estão as estruturas de dados básicas do hp^2FEM. Essas estruturas contêm diferentes tipos de matrizes como matriz esparsa e simétrica, além de vetores, que realizam operações de produtos matriciais e escalares, como também operações para solução de sistemas de equações lineares. Nesse nível, encontram-se também classes *arrays*, que armazenam valores numéricos, e também objetos. Destacam-se as classes ***OneIndexTable*** e ***TwoIndexTable***, que são tabelas de armazenamento de dados para o reaproveitamento dos mesmos em mais de um módulo do *software*. Essas tabelas possuem vetores de índices e dados. O vetor de índices acessa e gerencia o vetor de dados. Essas tabelas são utilizadas principalmente para o gerenciamento dos valores das funções

de interpolação.

As estruturas de dados definidas nessas camadas são principalmente usadas na construção das equações de elementos finitos. Como pode ser visto na Figura 3.4, existem classes para matrizes padrão, simétrica e esparsa, além da classe para vetores. Existe uma relação de dependência entre essas classes devido à utilização de instâncias em operações de algumas classes em outras.

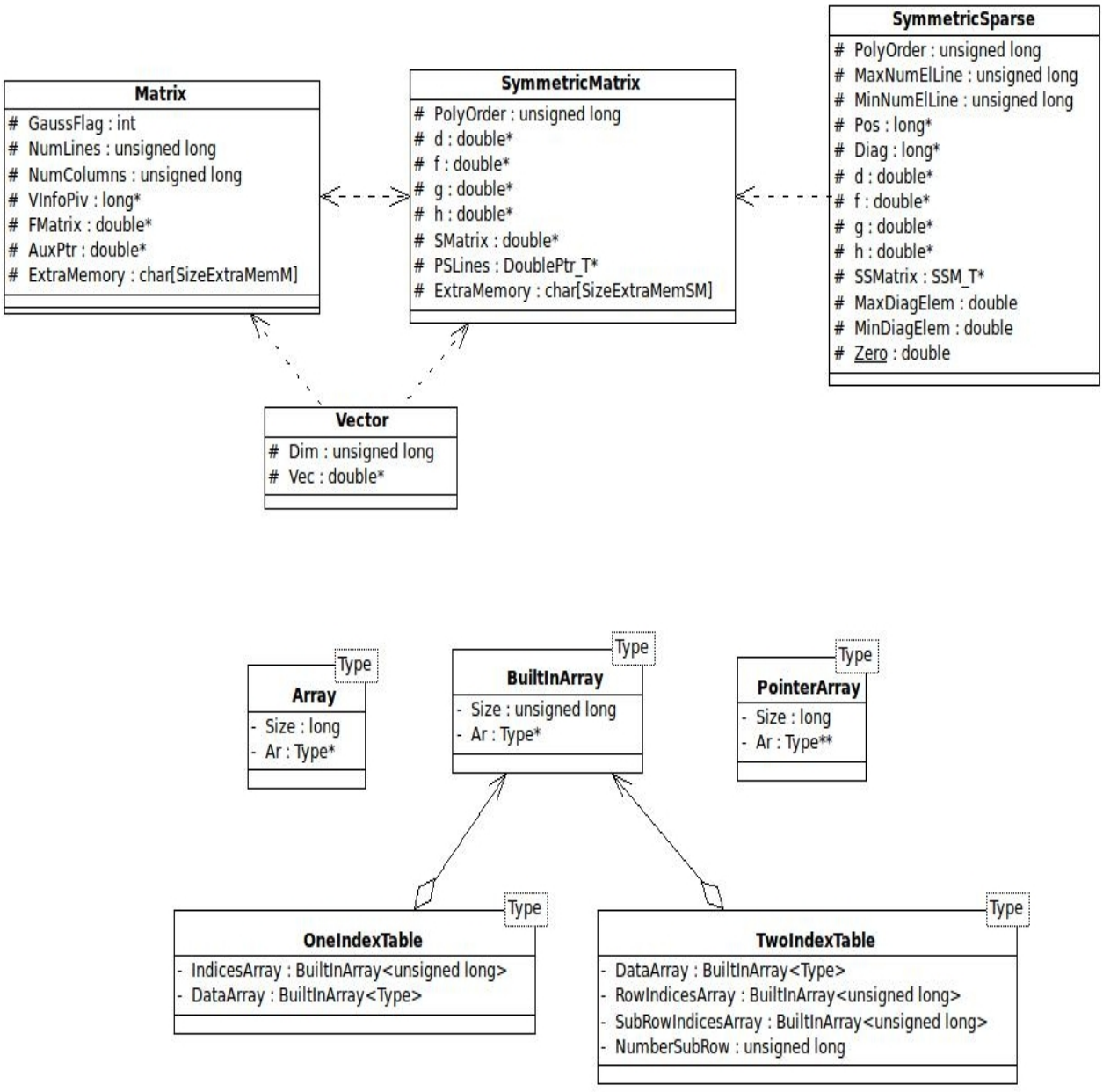


Figura 3.4: Diagrama de classe ***DS***.

Entre as rotinas principais e em comum nas classes de matrizes estão os procedimentos para: acesso aos elementos, produto escalar e matricial, inserção de submatriz em uma matriz e cálculo de normas. Destacamos nessas classes, principalmente para o MEF, as operações para soluções de equações lineares como o método de eliminação de *Gauss* com rotinas adjacentes para a triangularização e solução por meio da fatorização LU.

As classes ***SymmetricMatrix*** e ***SymmetricSparse*** adicionam outras rotinas para solução de sistemas de equações. Nessas classes, têm-se métodos iterativos de *Jacobi*, *Gauss-Seidel* e gradiente conjugado padrão e pré-condicionado para matrizes simétricas e positivas definidas. Em relação à alocação e armazenamento de dados, a classe de matriz simétrica aloca e atribui valores somente para a parte triangular superior. A classe de matriz esparsa aloca apenas as posições para coeficientes não nulos.

Na parte inferior da Figura 3.4 estão as classes de *arrays* e tabelas para armazenamento de dados do MEF, como as funções de interpolação, numeração de equações, incidência e coordenadas de elementos. As classes *arrays* permitem criar instâncias para armazenamento de uma sequência dados de um mesmo tipo.

A diferença entre essas classes está nos tipos de dados instanciados. A classe ***BuiltInArray*** foi modelada para ser instanciada utilizando-se tipos de variáveis básicos em C++, *float* ou *int*. No entanto, a ***Array*** é usada para armazenar instâncias de uma classe. À partir da ***PointerArray***, o objeto instanciado é uma sequência de ponteiros, os quais referenciam valores dos mesmos tipos que podem ser utilizados pela classe ***Array***.

Embora estruturas da própria linguagem estejam disponíveis, optou-se pela construção de objetos *arrays*, sem uso de biblioteca externas, para obter flexibilidade no uso de recursos de otimização do código futuramente. Essas classes utilizam *templates*, um artifício em C++ que permite passar como parâmetros os tipos a serem usados, definindo classes, estruturas e funções mais genéricas (VANDEVOORDE E JOSUTTIS, 2003). As rotinas em destaque dessas classes são as de acesso aos elementos, busca, cálculo de valores máximo e mínimo, leitura e escrita em banco de dados e arquivos. A arquitetura inicial dessas classes foi apresentada por BITTENCOURT (2000).

Igualmente ao caso anterior, as classes ***OneIndexTable*** e ***TwoIndexTable*** utilizam tipos parametrizados. De maneira geral, a ***OneIndexTable*** define uma estrutura para separar grupo de informações, o qual utiliza-se um vetor para armazenar todos as informações sobre um respectivo objeto

e um outro vetor de endereços que guarda os índices iniciais para cada grupo de informações.

A ***TwoIndexTable*** tem o mesmo intuito, porém com adição de um vetor como atributo denominado ***SubRowIndicesArray***, como pode ser visto na Figura 3.4. A idéia é criar sub grupos de dados dentro do grupo de dados endereçado pelo *RowIndicesArray*. Essa estrutura pode ser exemplificada por meio da Figura 3.5, onde são criados dois grupos de dados, sendo cada grupo divido em 2 sub grupos. Os *arrays* de índices armazenam o endereço do vetor de dados iniciando em 0. Na última posição dos *arrays* de índices é armazenado o total de elementos para facilitar o cálculo do número de dados para último grupo ou sub grupo de elementos.

Essas duas últimas classes dessa camada foram implementadas totalmente nesse trabalho. O uso dessas tabelas pode ser exemplificado através do funcionamento do hp^2FEM. Os valores das funções de interpolação nos pontos de integração são armazenados no vetor de dados e são separados em sub grupos de funções, de acordo com a ordem de integração e em grupos por meio do grau polinomial. Sendo assim, é possível acessar todas funções para todas ordens do integrando ou acessar somente as funções dado a ordem polinomial e uma ordem específica de integração.

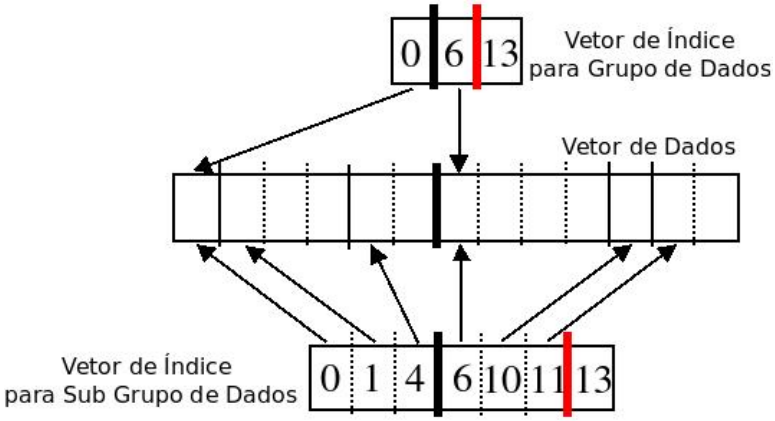


Figura 3.5: Representação estrutural da classe ***TwoIndexTable***.

Camada *Interpolation*

Por meio do pacote ***Polynomials1D*** são executadas as primeiras rotinas da camada ***Interpolation*** responsáveis pela geração das funções polinomiais e suas derivadas para elementos unidimensionais. Esse pacote gera funções para bases nodal e modal através do cálculo dos polinômios de Lagrange padrão e truncado, Jacobi, Hermite e Lobatto como pode ser visto pelo diagrama da

Figura 3.6. As coordenadas e o grau do polinômio são as principais entradas para a construção dessas funções.

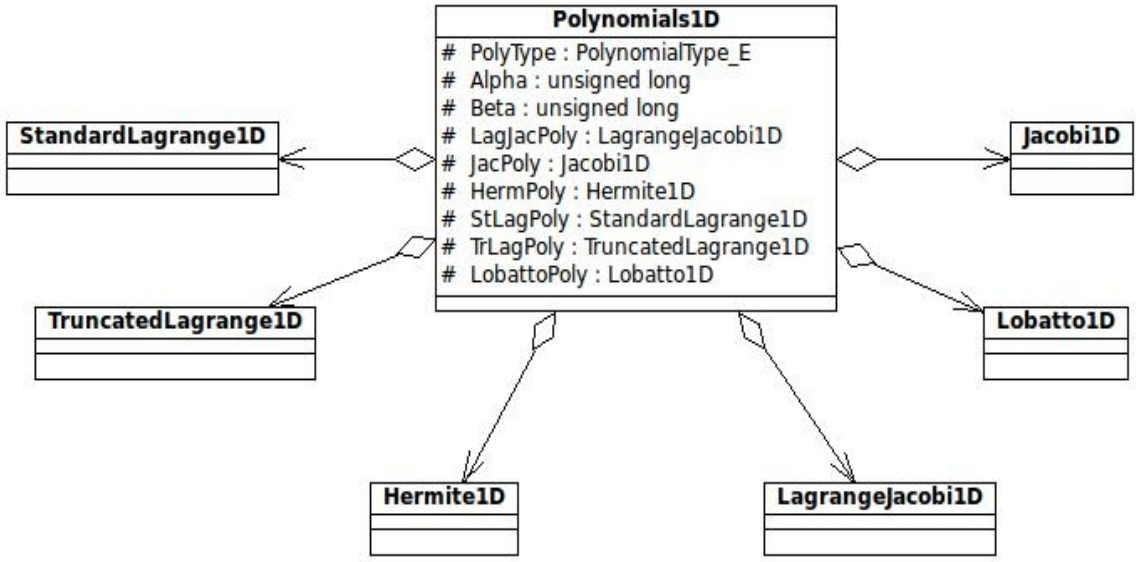
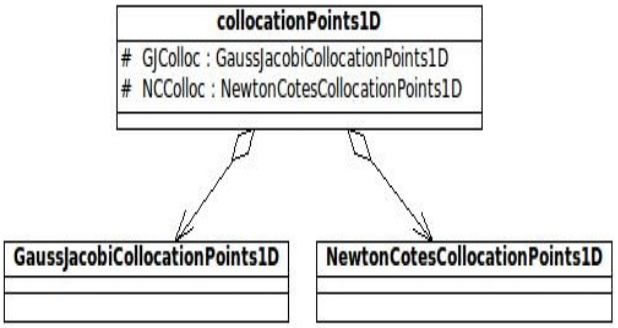
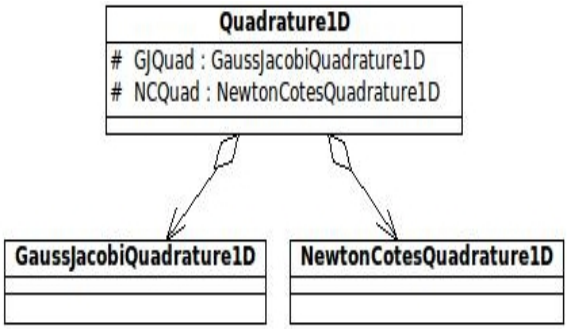


Figura 3.6: Diagrama de classe ***Polynomials1D***.

A Figura 3.7 descreve os diagramas das classes ***CollocationPoints1D*** e ***Quadrature1D***. Nessas classes são calculadas as coordenadas e ponderações para os pontos de colocação e integração unidimensionais. O número de coordenadas é determinado de acordo com o tipo de colocação ou regra de quadratura, como Gauss-Jacobi, Gauss-Radau-Jacobi, Gauss-Lobatto-Jacobi e Newton-Cotes (KARNIADAKIS E S., 2005), que é especificado no arquivo de entrada, assim como o grau do polinômio. A diferença entre essas classes é que para objetos ***Quadrature1D*** são também determinados os coeficientes de ponderação.



(a) ***CollocationPoints1D***.



(b) ***Quadrature1D***.

Figura 3.7: Diagramas de classes ***Quadrature1D*** e ***CollocationPoints1D***.

As classes anteriores são encapsuladas pelo conjunto de classes dos componentes ***CollocationPoints*** e ***NumericalIntegration*** que possuem o mesmo propósito. Contudo, essas classes também armazenam pontos para elementos 2D e 3D, estando modeladas estruturas para as seguintes formas de elementos: linha, quadrado, triângulo, tetraedro e hexaedro. Para os elementos não unidimensionais, as coordenadas são armazenadas de acordo com o vetor de índices de tensorização passados como entrada na ***CollocationPoints***, o qual foram criados na classe ***ShapeFunctions*** (Figura 3.9 descrita adiante). Para ***NumericalIntegration***, esses índices são calculados internamente para armazenamento das coordenadas e do produto tensorial entre os coeficientes de ponderação em elementos 2D e 3D.

Para armazenar as coordenadas, utilizam-se instâncias das classes de tabelas de dados da Seção 3.4.2 . Essa implementação permite flexibilidade ao MEF-AO, armazenando pontos para vários graus polinomiais. No caso das classes ***CollocationPoints*** e ***NumericalIntegration***, o uso da tabela ***OneIndexTable*** representa um grupo de dados como um conjunto de coordenadas para um certo grau polinomial. Juntamente a isso, aloca-se uma variável para auxiliar o acesso a essa tabela. Essas variáveis são declaradas no código como ***IPSets*** e ***CPSets***, que são *arrays*, cujos os índices indicam o grau do polinômio e os valores armazenados formam o índice do vetor de endereços da tabela ***OneIndexTable***, onde as coordenadas estão armazenadas.

Considere um exemplo do uso do vetor ***CPSets*** da classe ***LineCollocationPoints*** na Figura 3.8. Dadas as entradas de ordens polinomiais 2, 5 e 7, cria-se o vetor com dimensão da maior ordem polinomial. Nesse exemplo, percebe-se que o grau do polinômio são os índices do vetor ***CPSets*** e que os valores armazenados indicam os índices para acesso às linhas com informações da tabela ***OneIndexTable***. As posições inválidas do ***CPSets*** são preenchidas com -1. Assim, os pontos de colocação de grau 2 são armazenados na linha 0 da tabela; para o grau 5, os pontos estão na linha 1; e para grau 7, as coordenadas de colocação estão na linha 2 da tabela.

Figura 3.8: Vetores auxiliares para acesso às tabelas ***OneIndexTable*** e ***TwoIndexTable***.

Os diagramas das classes ***CollocationPoints*** e ***NumericalIntegration*** são similares ao do componente ***ShapeFunctions***, como mostrado na Figura 3.9. Além de encapsular as classes ante-

riores, esse pacote calcula as funções de interpolação e suas derivadas nos pontos de integração e colocação armazenados anteriormente. Para o armazenamento desses valores é definido para cada classe da ***ShapeFunctions*** (Figura 3.9), um conjunto de quatro atributos do tipo ***TwoIndexTable***. Para a construção das funções de forma para elementos 2D ou 3D, realiza-se produto tensorial entre as funções de interpolação dos elementos unidimensionais (BARGOS, 2009).

Camada *Group*

A Figura 3.10 apresenta as principais estruturas da camada ***FiniteElementGroup***. Entre essas está a classe ***Mapping*** que é responsável pelo cálculo das matrizes jacobianas e seus determinantes usados para o mapeamento da geometria dos elementos. Nesse componente, também computa-se a medida dos elementos, como comprimento, área ou volume. As entradas principais para determinar esses valores são: as coordenadas nodais de mapeamento, a dimensão e a forma do elemento. Utilizam-se os procedimentos ***STANDARD*** e ***D1_MATRICES***, nessa etapa do código, exemplificados no Capítulo 2.

A classe ***Mesh***, que também está agregada à classe ***FiniteElementGroup***, pode ser instanciada em 4 objetos com finalidades para armazenamento de malhas de elementos de entrada, solução, mapeamento e pós-processamento (ver Seção 2.3.3). Para cada uma dessas, pode-se utilizar um grau de interpolação diferente. Uma das informações armazenadas na ***Mesh*** é o número local de entidades topológicas de elementos para cada grau, usando uma instância da classe ***ElementTopologicalIndices***. A topologia contém dados como o número de arestas, faces e nós, que são determinadas a partir da forma do elemento e ordens polinomiais.

Outros valores armazenados na ***Mesh*** são as coordenadas, a numeração dos graus de liberdade, número de nós, ordem polinomial e incidência. Esses dados estão associados a cada elemento, o que permite configurar uma malha com ordem polinomial não-uniforme. Além disso, as coordenadas deformadas também são armazenadas depois do cálculo do vetor de solução da equação geral do MEF. O armazenamento da numeração dos graus de liberdade é realizado em dois métodos: um é para armazenar os graus de liberdade na malha de solução, utilizando os dados do objeto ***Equation*** da camada ***Model***, e outro para malha de pós-processamento, cuja a numeração global é criada através da incidência e do número de graus de liberdade para cada nó.

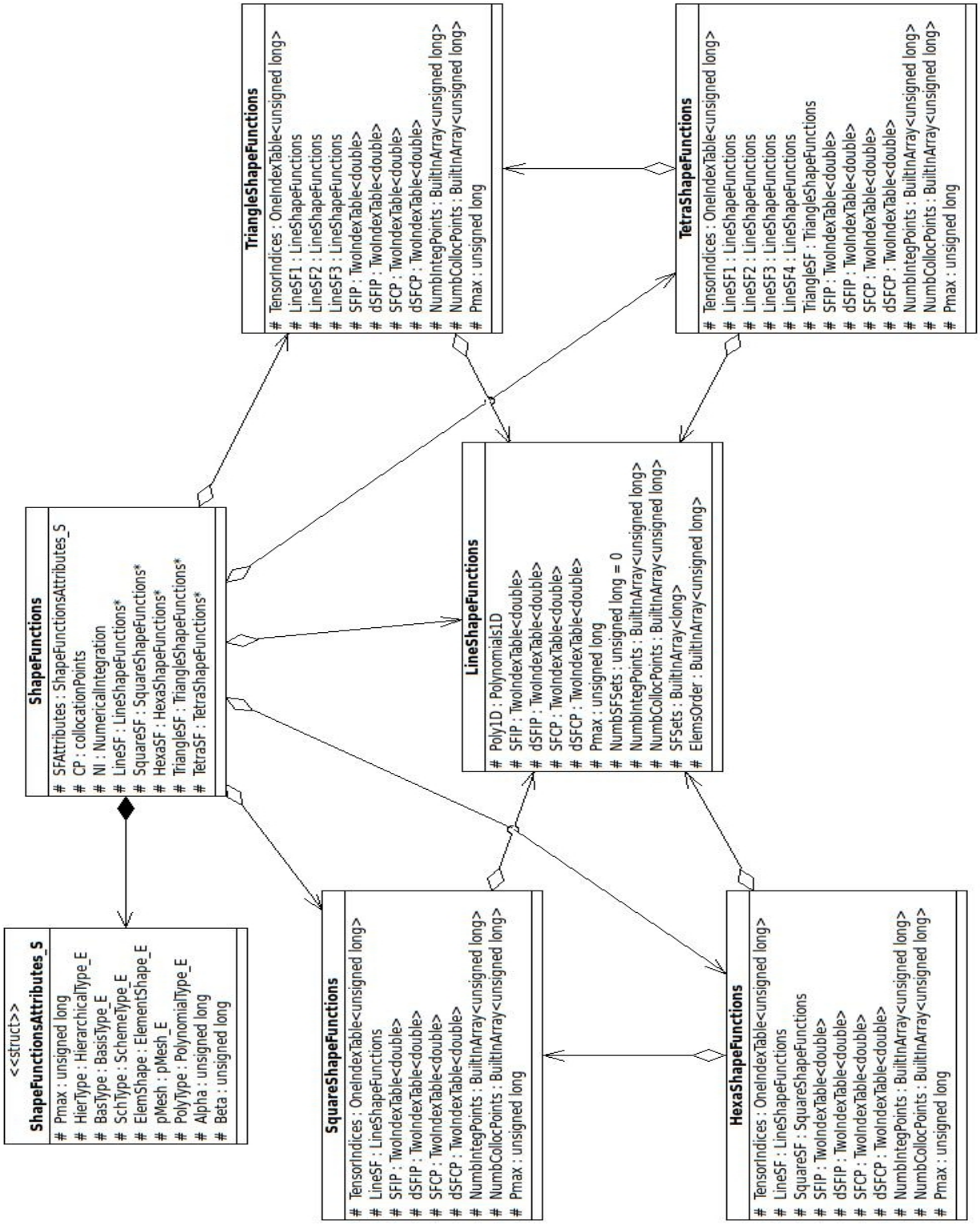


Figura 3.9: Diagrama de classe ***ShapeFunctions***.

Figura 3.10: Conjunto de classes da camada ***Group***.

O conjunto de classes ilustrado na Figura 3.11 é responsável por implementar uma hierarquia de materiais. Armazena-se também o tipo de aplicação a ser tratada por um conjunto de tipos enumerados em C++. Essas aplicações podem ser, por exemplo, *Poisson* e *Reynolds*, estado plano de tensão, entre outros. O uso de características de herança e polimorfismo dinâmico, discutidos Seção 3.1.2, permite flexibilidade na construção e redefinição de novos tipos de materiais.

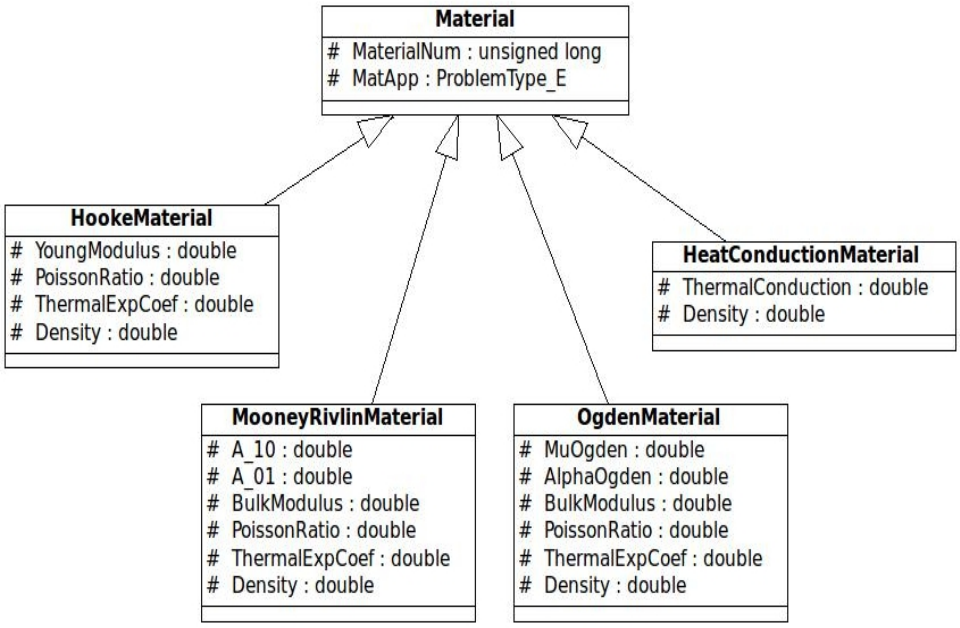


Figura 3.11: Diagrama de classe ***Material***.

A classe ***FiniteElement*** é outro objeto da classe ***FiniteElementGroup***. É responsável principalmente pelo cálculo dos operadores elementares, por exemplo, vetores de carregamento, matrizes de massa e rigidez, como visto na Seção 2.1.1. Como podemos ver através da estrutura ***FiniteElementAttributes_S*** do diagrama de classes na Figura 3.12, um objeto ***FiniteElement*** armazena propriedades gerais para solução de um elemento como a sua forma e dimensão, o número de graus de liberdade por nó, os tipos de aplicação, malha, procedimento de cálculo (D1_MATRICES ou STANDARD) e grau polinomial.

Além desses atributos, a classe ***FiniteElement*** associa, por composição, a classe ***GradientMeasure***, vista no diagrama. Essa classe opera de forma auxiliar para a construção de operadores que compõem as formulações definidas na ***FiniteElement***. Por exemplo, a construção da matriz de derivadas globais das funções de interpolação para o cálculo da matriz de rigidez.

Dentro da ***FiniteElement***, a construção dos vetores de carregamento e das matrizes de massa

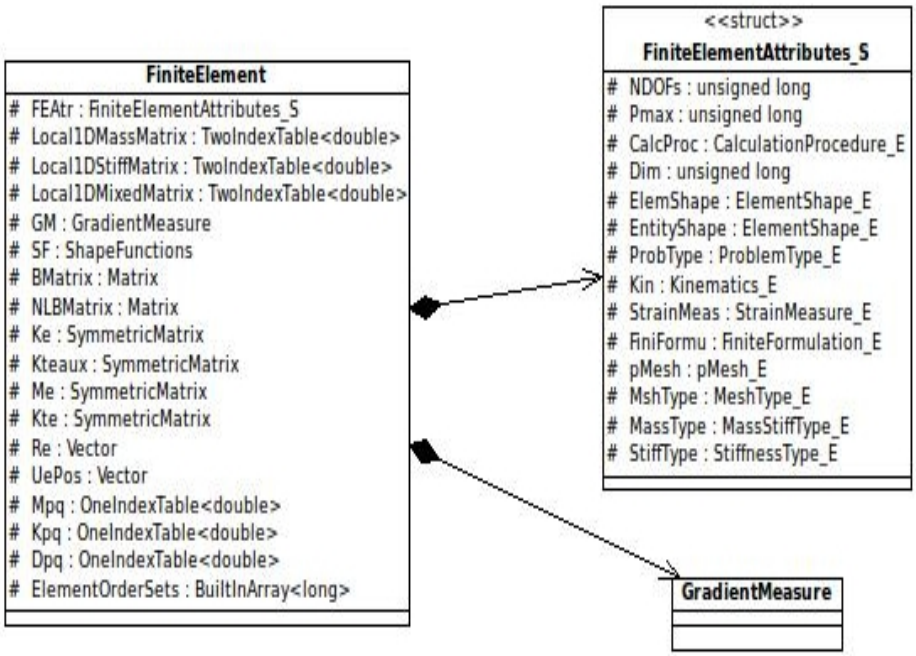


Figura 3.12: Diagrama de classe ***FiniteElement***.

e rigidez pode ser feita através do produto tensorial de operadores unidimensionais para elementos 2D e 3D (procedimento de cálculo D1_MATRICES, ver Seção 2.3.2).

Outro método a ser destacado é a interpolação da solução global para a malha de pós-processamento. A operação realiza-se através do cálculo das funções de forma nos pontos de colocação da malha de pós-processamento, obtendo um novo vetor solução através do produto entre a matriz obtida da interpolação e o vetor resultante da malha de solução, como exemplificado na Seção 2.3.3.

A classe ***GeometricProperties*** (Figura 3.10) descreve as propriedades geométricas de um determinado elemento, por exemplo, um elemento de barra que tem como propriedade a área da seção transversal. Os principais atributos são o número de propriedades e seus valores associados a todos ou diferentes elementos do grupo.

A classe ***FiniteElementGroup*** se destaca por agregar as classes descritas anteriormente e por operar como interface através da passagem de diversos dados para a formulação do MEF às outras classes. Grande parte desses dados são carregados internamente na classe ***FiniteElementGroup***. Além disso, a classe armazena o número do grupo de elementos, o número de elementos e os

elementos inicial e final.

A agregação da classe ***Mesh*** na ***FiniteElementGroup*** se dá com a definição de quatros tipos de objetos de malha de elementos finitos, correspondendo às malhas de entrada, solução, mapeamento e pós-processamento. A malha de mapeamento possui um atributo que indica o tipo de mapeamento a ser realizado, podendo ser isoparamétrico ou não-isoparamétrico. O primeiro (isoparamétrico) utiliza o mesmo grau para aproximação da solução na malha de mapeamento, o segundo (não-isoparamétrico) permite usar grau diferente da malha de solução no mapeamento, podendo ser de ordem polinomial uniforme ou não-uniforme.

Para a construção de alguns objetos do grupo de elementos é realizado operações auxiliares na ***FiniteElementGroup***. Uma dessas operações é a construção de um vetor de diferentes ordens polinomiais por meio da junção dos graus dos elementos para cada tipo de malha. Esse vetor é passado ao objeto ***ShapeFunctions*** da camada ***Interpolation*** para o cálculo da funções de forma.

Camada *Model*

Embora a classe ***FEGroups*** esteja apresentada na Figura 3.10, essa faz parte da camada ***Model*** e é responsável por armazenar todos os grupos de elementos. Possui como atributos, o número total de elementos globais, o número de grupos e as máximas ordens polinomiais para as malhas de mapeamento e solução.

Entre as operações que se destacam na ***FEGroups*** estão as que convertem o número de um elemento do sistema global para o local ou vice-versa, a leitura da incidência dos elementos na malha de entrada e a leitura do grau da malha de pós-processamento. Há também rotinas para retorno do número do grupo em que se encontra um determinado elemento e retorno do número total de graus de liberdade e elementos globais.

Seguindo a descrição das classes ou componentes que compõem a camada *Model*, apresenta-se, na Figura 3.13, a classe ***GeometryMesh***, que constrói e associa as propriedades entre malha e geometria. As informações associadas são utilizadas quando é necessário a troca de dados entre malha e geometria. Por exemplo, dado um carregamento sobre uma superfície da geometria, deseja-se obter todos os nós, arestas e faces de elementos que estão agregadas nessa superfície.

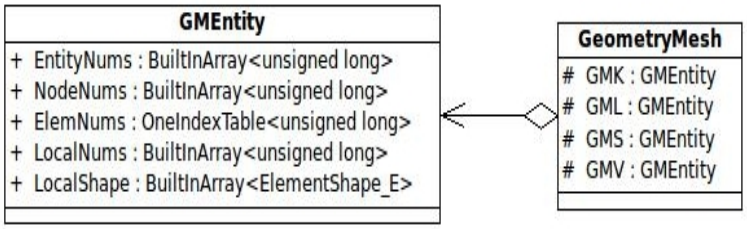


Figura 3.13: Diagrama de classe ***GeometryMesh***.

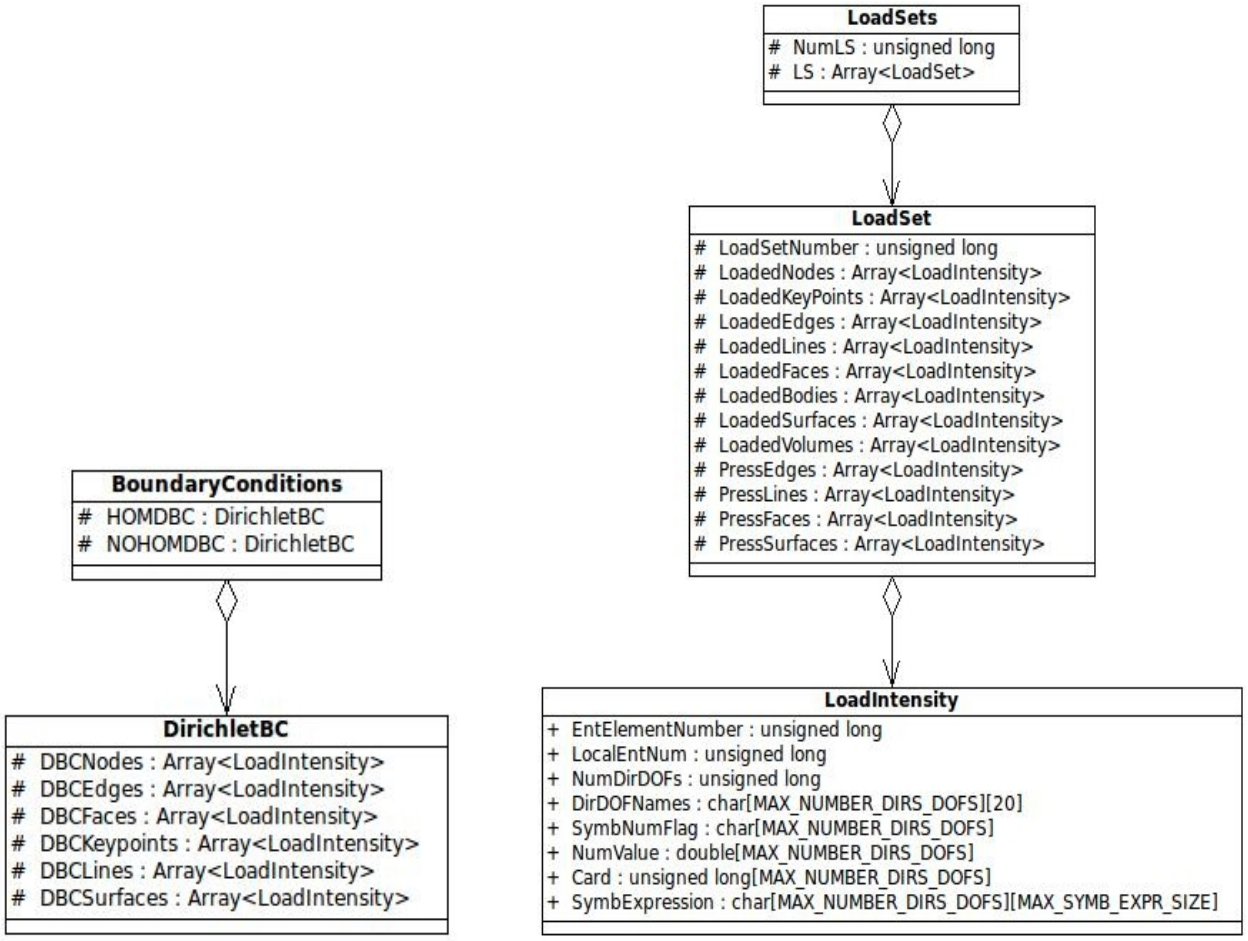
Na Figura 3.14, têm-se as classes que representam as condições de contorno. A classe ***BoundaryConditions*** agrega as condições de contorno homogêneas e não homogêneas (KARNIADAKIS E S., 2005) através de objetos da classe ***DirichletBC***. As condições de contorno foram modeladas para serem aplicadas às entidades da geometria ou malha, como nós, arestas, faces, linhas e superfícies.

Para cada entidade de uma instância da classe ***DirichletBC***, armazena-se o número do elemento, número local da entidade, o número de graus de liberdade, a identificação do grau de liberdade e sua cardinalidade em relação ao vetor de graus de liberdade e o valor da condição de contorno aplicada.

A classe ***LoadSets*** (Figura 3.14(b)) é responsável por armazenar um conjunto de carregamento de um modelo. Assim, como na ***BoundaryConditions***, armazena valores para entidades da malha e da geometria. Dependendo do tipo de entidade, o carregamento aplicado pode ser uma função simbólica, o que implica o uso do analisador Simbólico-Numérico (ver Seção 3.2.3).

Os objetos do conjunto de classes ***MeshTopology***, como visto na Figura 3.15, são utilizados para a construção da topologia da malha, através de tabelas que armazenam a relação entre as entidades da malha. Essas tabelas são descritas a seguir:

- Tabela **NodeNode**: Armazena para cada nó da malha os seus nós vizinhos.
- Tabela **ElementElement**: Armazena para cada elemento da malha os seus elementos vizinhos.
- Tabela **NodeEquation**: Para cada nó, armazena a numeração dos graus de liberdade vizinhos.
- Tabela **NodeElement**: Para cada nó, é armazenado os seus elementos vizinhos. Para esse tipo



(a) ***BoundaryConditions***. (b) ***LoadSets***.

Figura 3.14: Diagramas de classes ***BoundaryConditions*** e ***LoadSets***.

de tabela são instanciados dois objetos da classe ***MeshTopology***, um para malha de entrada e outra para a malha de solução.

Na tabela **NodeElement** as relações são construídas por meio da incidência dos elementos. Além desse parâmetro, a tabela **ElementElement** utiliza-se da topologia dos elementos, e semelhante à tabela **NodeNode**, usa a tabela **NodeElement**, que deve ser previamente definida.

A classe ***HighOrder***, mostrada na Figura 3.15 é responsável por gerar, compatibilizar e gerenciar a incidência de alta ordem dos elementos. Nessa classe estão armazenados o número de graus de liberdade e nós das entidades elementares (aresta, face, corpo). A geração de pontos ou nós é realizada nas malhas de pós-processamento, mapeamento e solução. A ***HighOrder*** utiliza a

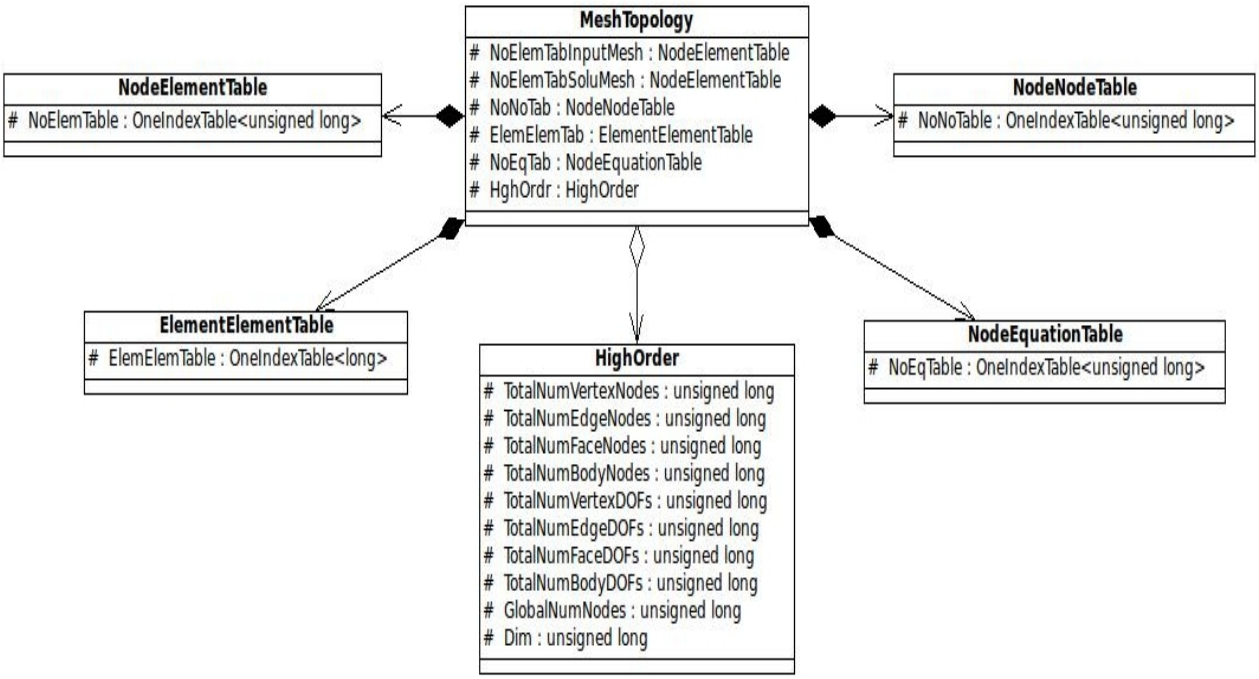


Figura 3.15: Diagrama de classe ***MeshTopology***.

topologia dos elementos, a forma do elemento e as funções de interpolação nos pontos de colocação para gerar novas incidências e coordenadas. Após isso, a topologia dos elementos é reconstruída por meio da redefinição dos nós nas entidades topológicas.

As classes ***Nodes***, ***DOFs*** e ***Equations*** (Figura 3.16) são estruturas auxiliares que armazenam as coordenadas nodais da malha de entrada, os tipos de graus de liberdade e a numeração das equações, respectivamente. A classe ***Nodes*** contém a dimensão e o número de nós dos elementos e implementa rotinas para acesso às coordenadas de um dado nó e construções de tabelas de relação entre elementos e coordenadas. A classe ***Equations*** armazena os graus de liberdade em uma tabela ***OneIndexTable***, associando a cada nó os tipos de graus de liberdade na seguinte sequência: graus de liberdade restritos para condições de contorno homogênea e não homogênea. Essa contagem de graus de liberdade efetua-se sobre cada entidade da malha como nós, arestas, faces e volume.

A classe ***Model*** (Figura 3.16) atua como ligação entre aplicações e a construção de dados do MEF-AO, agregando e gerenciando as classes descritas anteriormente. Exemplos importantes desse gerenciamento são as chamadas ao carregamento dos dados para as classes como a numeração dos graus de liberdade; chamadas para construções de incidências e coordenadas nas malhas de entrada, solução e pós-processamento, sendo o armazenamento nas 2 últimas ocorrendo após a construção

do objeto ***HighOrder***.

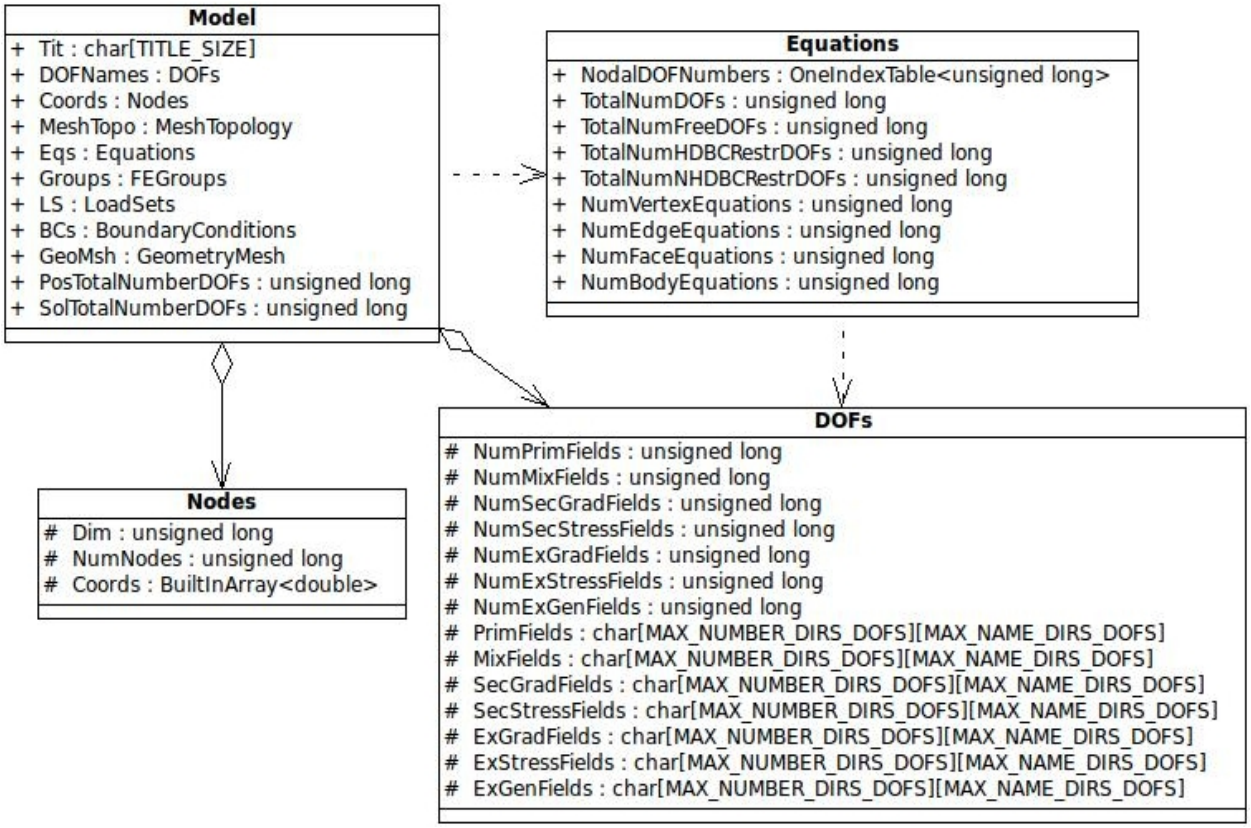


Figura 3.16: Diagrama de classe ***Model***.

Camada *Solver*

Na camada ***Solver***, utiliza-se o mecanismo de herança em POO. A classe ***Solver*** possui métodos gerais para aplicações usando o MEF.

Como mostrado na Figura 3.17, a classe ***Solver*** agrega a estrutura ***SolverParam_S*** utilizada para armazenar parâmetros da solução como o tipo de problema a ser tratado e seus valores relacionados. A estrutura ***TheoreticalSolution_S***, também agregada, armazena dados da solução teórica imposta que pode ser simbólica ou numérica. Sendo simbólica, deve ser representada por coordenadas espaciais X, Y e Z e a coordenada de tempo T.

Dentre os principais atributos da classe ***Solver*** estão o objeto da classe ***Model***; o vetor solução

da malha; o vetor de soma das medidas dos elementos em cada grau de liberdade; matriz para armazenamento dos tipos de erros calculados. Esses erros são calculados em duas rotinas tanto para o padrão de montagem das funções de interpolação *D1_MATRICES*, quanto para o *STANDARD*.

A classe ***GlobalSolver*** é derivada de ***Solver*** e responsável por resolver os problemas de maneira global. Sendo assim, as matrizes de massa ou rigidez, armazenadas como atributos nessa classe, são construídas a partir da superposição de matrizes de cada elemento (operação de montagem), calculando-se o vetor de solução do MEF após essa operação.

A outra classe derivada, ***ElementElementSolver***, trata a solução de maneira local como visto na Seção 2.3.1. Nesse caso, determina-se a solução em cada elemento e a montagem da solução global é ponderada de acordo com a medida do elemento. Na Seção 3.6, será mostrado como ocorre a execução elemento a elemento para o problema de projeção, considerando 3 tipos de caso: malhas uniforme e não-uniformes com distribuição polinomial uniforme, malhas não-estruturadas ou aplicação da distribuição polinomial não-uniforme (ver Seção 2.3).

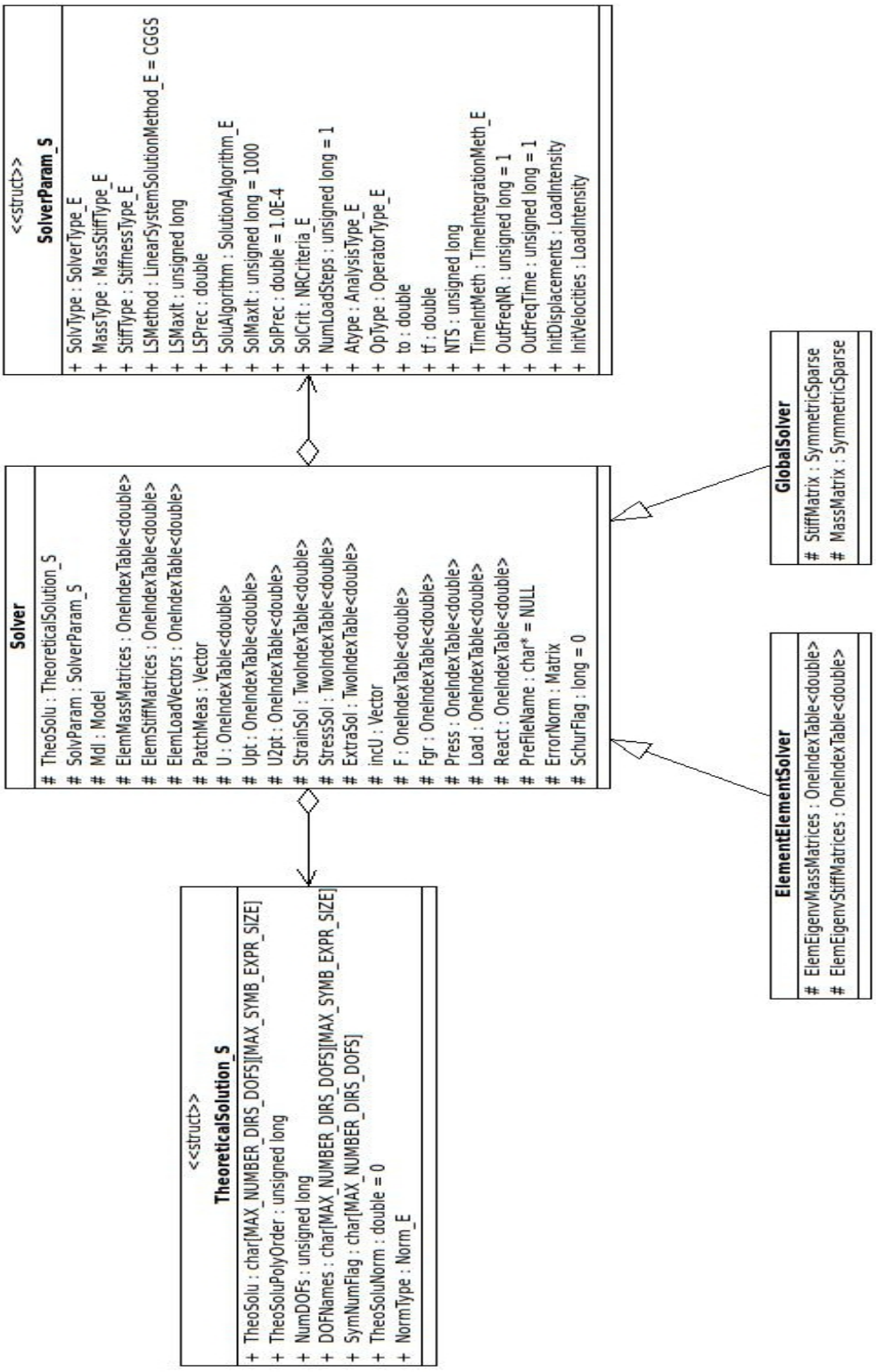


Figura 3.17: Diagrama de classe ***Solver***.

3.5 Entrada de dados

A entrada de dados do *software* hp^2FEM é dividido em dois arquivos com extensões .fem e .def no formato texto. Esses arquivos, descritos no Anexo A, contêm palavras-chaves, a qual cada uma é associada a determinadas informações para a especificação do modelo de elementos finitos.

O arquivo .fem armazena dados gerais da malha como número de elementos, a forma dos elementos para cada grupo, coordenadas e incidência. O arquivo .def armazena informações tanto em relação a toda a malha como para um grupo específico. As informações gerais desse arquivo são relativas ao tipo de análise e método de solução utilizado. Já os dados específicos são os métodos de integração, tipos de base polinomial, condições de contorno, forças aplicadas, grau polinomial, tipo de malha e tipo de distribuição polinomial.

Os dados contidos nos arquivos são valores numéricos, palavras-chaves e identificadores. Para leitura de caracteres que não são palavras-chaves (os identificadores), converte-se para uma estrutura relativa do tipo enumerado em C++. Os atributos dentro dessa estrutura podem ser representados por um tipo inteiro em C++, o que permite comparações mais rápidas na implementação ao invés do uso de cadeias de caracteres.

Um exemplo dessa representação pode ser visto na Figura 3.18, o qual define uma estrutura do tipo enumerado contendo os tipos de polinômios a serem utilizados no *software* hp^2FEM. Nesse caso, após a leitura da palavra-chave "*INTERPOLATION*", lê-se o tipo de polinômio como uma *string* e em seguida converte-se o para tipo *PolynomialType_E*, assumindo um dos valores da estrutura.

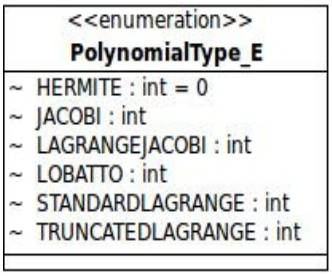


Figura 3.18: Tipo enumerado ***PolynomialType_E***.

3.6 Visualização de Processos

Os diagramas de atividades a seguir exibem o fluxo de execução do *software* hp^2FEM. Os diagramas foram construídos com base em UML e são compostos principalmente por atividades, ações, objetos do sistema e nós inicial, final e de decisões. As características dos diagramas podem ser dadas pelas formas geométricas ou das descrições(MELO, 2010). As ações são representadas por retângulos mais curvados e palavras-chaves no infinitivo, as atividades por palavras como substantivo, os objetos por palavras sublinhadas e as tomadas de decisão por losangos.

3.6.1 Fluxo Geral do *Software* hp^2FEM

O diagrama da Figura 3.19 representa o fluxo de execução do programa descrito de forma geral, considerando a aplicação de qualquer problema ou tipo de ***solver***. Algumas atividades desse diagrama podem ser decompostas em várias sub-atividades, umas sendo mais extensas e outras menores. Em resumo, esse diagrama representa a leitura e a construção de dados para a malha de grupos dos elementos.

Das classes descritas na Seção 3.4.2, algumas instâncias foram destacadas, representando-as como objetos no diagrama da Figura 3.19. Em um primeiro passo, cria-se o objeto ***Solver*** e carregam-se as informações associadas à solução, podendo ser global ou local. Outras informações são lidas nessa etapa de acordo com o tipo de problema a ser tratado. Na sequência, o objeto ***Model***, interno ao ***Solver***, realiza a leitura de outros dados comuns na malha. No próximo nível, encontra-se o objeto ***Group***, podendo ser alocado como único objeto ou um vetor de objetos, onde cada grupo de elementos guardam informações em comum (ver Seção 3.4.2).

Dentro do objeto ***Group*** está o objeto ***ShapeFunctions*** que assume a tarefa de construir as funções de interpolação. Essa é uma outra etapa que pode ser decomposta em várias sub-atividades da camada ***Interpolation***. Em seguida, realiza-se a leitura das incidências dos elementos, e caso haja mais grupos, executa-se o próximo objeto ***Group***. Após as chamadas das rotinas dos grupos de elementos, serão executadas as atividades de leitura das forças aplicadas e propriedades geométricas da malha.

Outra atividade extensa é a construção da topologia da malha. Destaca-se o objeto ***HighOr-***

der, que através das funções de interpolação, gera mais pontos nos elementos da malha para a solução. Seguindo o fluxo, efetua-se a atividade de numeração das equações, que em seguida serão armazenadas no objeto da malha de solução. Contudo, para a malha de pós-processamento, as equações são construídas novamente baseando na mesma implementação de geração de pontos da classe ***HighOrder***.

O armazenamento das equações também é realizado para cada grupo. Após todo esse processo, o solver local ou global será executado. A seguir veremos o fluxo de execução do problema de projeção tratado de forma local, ou seja, solução elemento por elemento. Esse problema será detalhado também na Seção 4.1.2.

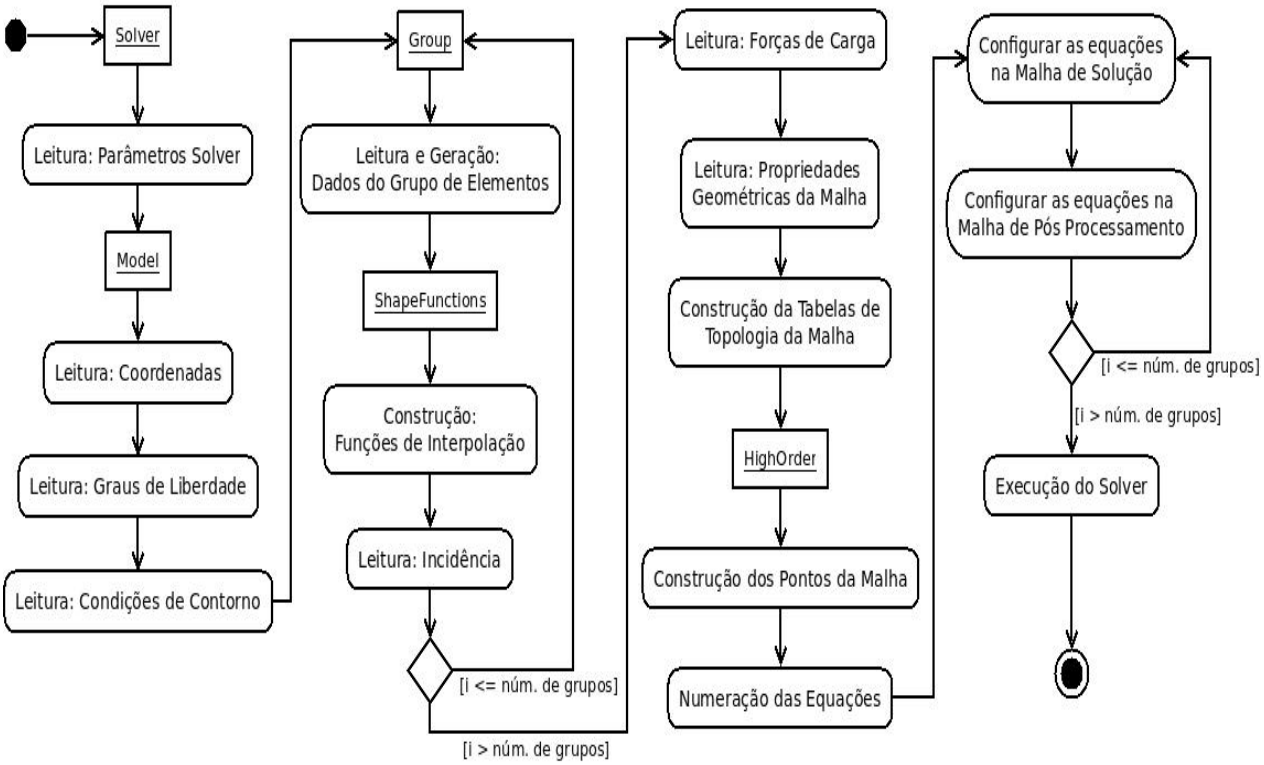


Figura 3.19: Diagrama de atividades global do hp^2FEM.

3.6.2 Fluxo do Solver para Problema de Projeção

Na Figura 3.20, exibe-se o diagrama de atividades do problema de projeção por meio de principais ações dessa rotina e atividades do hp^2FEM. O diagrama refere-se a 3 casos da implementação realizada no projeto: soluções para malha uniforme com distribuição polinomial uniforme, malha

não-estruturada e distribuição polinomial não-uniforme.

O estado inicial desse diagrama representa o estado final do diagrama de atividades anterior Figura 3.19, substituindo a atividade de "Execução do *Solver*". Após a chamada da rotina do problema de projeção, aloca-se o vetor de solução de acordo com o grau polinomial determinado da malha de pós-processamento. Do mesmo modo, aloca-se o vetor de medida do elemento que armazena em cada grau de liberdade a medida dos elementos que o compartilha.

No próximo passo, acessam-se as informações referente ao grupo como tipo de malha e distribuição polinomial, grau da malha de pós-processamento, forma do elemento e número de elementos do grupo. Os primeiros desses dados são usados para a tomada de decisão que vêm em sequência. Se a malha e a distribuição polinomial for uniforme, os próximos passos serão executados apenas para o primeiro elemento da malha do grupo. Caso seja uma malha não-estruturada ou distribuição polinomial não-uniforme as mesmas atividades serão executadas para todos elementos.

Posteriormente, realiza-se acesso ao grau da malha, coordenadas e a alocação da matriz de massa e vetor de carga a serem calculados. Antes do cálculo da matriz de massa, o algoritmo testa se o procedimento de cálculo é o caso D1_MATRICES, o qual as matrizes de massa serão calculadas apenas para elementos unidimensionais. No passo seguinte, a atividade de cálculo da matriz de massa possui internamente o mesmo nó de decisão (verificando-se o procedimento é D1_MATRICES ou STANDARD). Assim, caso os elementos sejam 2D ou 3D, determina-se o cálculo da matriz de massa através do produto tensorial entre as matrizes 1D ou pelo procedimento padrão (STANDARD), sendo esse utilizando-se das próprias funções de interpolação dos elementos 2D ou 3D.

Após isso, executa-se a triangularização da matriz de massa pelo método de Gauss. Em sequência há uma tomada de decisão em que se calcula o vetor de medida do elemento para o caso da malha ser uniforme e p-uniforme. Seguindo o fluxo, acessa-se as coordenadas na malha de mapeamento e a carga aplicada é calculada. Finalmente, determina-se a solução local através do método de Gauss para a solução de sistema lineares. Para o caso do problema de projeção, a solução obtida representa os coeficientes da combinação de funções aproximadas (ver Seção 4.1.2).

Seguindo o curso de execução, verifica-se o grau da malha de solução é igual ao da malha de pós-processamento. Caso afirmativo, a solução local é armazenada no vetor global. Caso contrário, interpolam-se os pontos de colocação da malha de pós-processamento usando as funções de

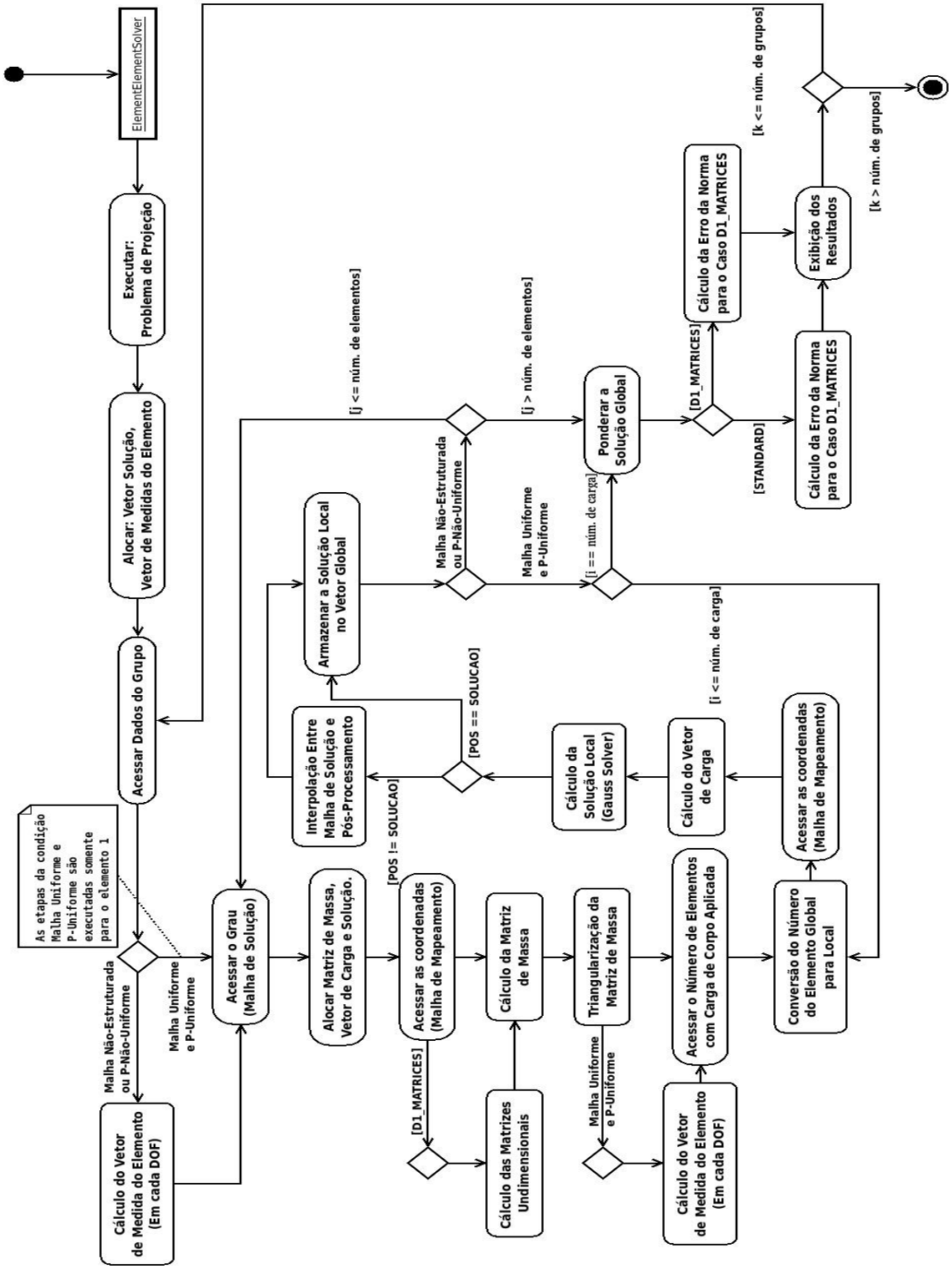


Figura 3.20: Diagrama de atividades do *Solver* elemento por elemento - Problema de projeção.

interpolação da solução. Nos próximos passos, para a malha não-estruturada ou p-não-uniforme, verifica-se há mais elementos, podendo voltar ao passo em que se acessa o grau do elemento, para calcular outra matriz de massa. Para a malha uniforme e p-uniforme, verifica-se há mais carga de corpo a ser aplicada, para retornar ou não ao passos que levarão ao cálculo de outro vetor de carga até a solução local.

Após ao cálculo de todas as soluções locais, a solução global é ponderada, utilizando-se do vetor da medida do elemento calculado nas etapas anteriores. Em seguida, os erros nas normas infinita e L_2 serão calculados para o caso STANDARD ou D1_MATRICES. Os resultados serão exibidos no próximo passo. E caso haja mais grupos, todos os passos relativos ao cálculo da matriz de massa, vetor carga e soluções locais serão refeitos, caso contrário, a execução é finalizada.

4 Casos de Validação e Testes

Nesse capítulo, consideram-se testes com o objetivo de validar o funcionamento da arquitetura do hp^2FEM e as estratégias de solução propostas na Seções 2.3 e 3.3.1 para o problema de projeção. Sendo assim, avalia-se o código por meio do comportamento do erro na norma L_2 da solução aproximada, considerando-se a variação em p para malhas de quadrados e hexaedros.

De acordo com estudos anteriores para soluções suaves ou contínuas, a adaptatividade p, o qual se fixa o número de elementos e varia-se a ordem polinomial, converge de forma exponencial (KARNIADAKIS E S., 2005).

Para cada caso das estratégias de solução, realizam-se diferentes análises, avaliando todos os casos em um *solver* local, ou seja, quando determina-se a solução elemento por elemento. Na comparação entre os procedimentos de cálculo "D1_MATRICES"e "STANDARD", verifica-se o custo computacional, avaliando o tempo de execução e o pico de maior memória consumida. Tal procedimento, foi avaliado para malhas uniforme com p-uniforme. Para o caso p-não-uniforme, compara-se o erro de cada variação p com os das malhas p-uniforme. Por conseguinte, o erro da melhor solução da adaptatividade p para p-uniforme é comparado com algumas combinações polinomiais da malha p-não-uniforme.

No caso de malhas não-estruturadas, também avalia-se a convergência com a variação p para o caso "STANDARD". No caso de interpolação entre as malhas de pós-processamento e solução, verifica-se a relação entre as soluções das malhas são coerentes. Sendo assim, a malha de pós-processamento deve manter a solução para determinados graus de liberdade quando o grau do polinômio é menor que o da malha de solução.

Devido aos testes de custo computacional, é importante considerar que os casos a seguir foram avaliados em uma máquina com a seguinte configuração: Processador Intel Core 2 Duo T5500, velocidade do processador 1.66 GHz, Barramento 667 MHz, cache L2 2MB e memória *RAM DDR2* de 2GB.

4.1 Aplicação

4.1.1 Problema de Projeção

O problema de projeção descreve a aproximação u_{ap} de uma função u por uma combinação linear de funções de base ϕ_i, como ilustrado esquematicamente na Figura 4.1. O conjunto das funções de interpolação definem um espaço de aproximação sobre o qual a função u é projetada.

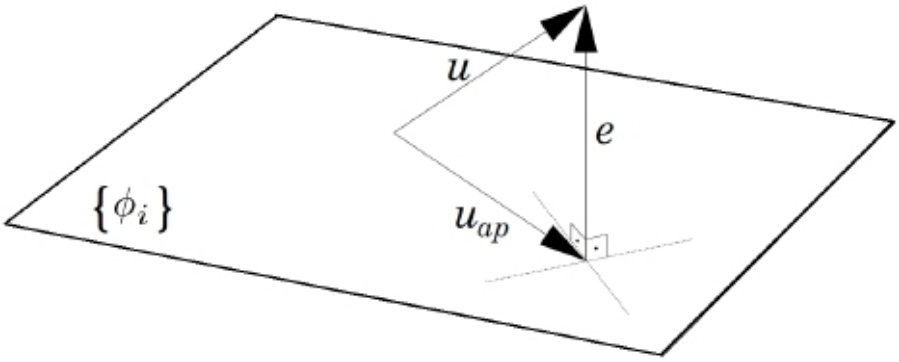


Figura 4.1: Problema de projeção.

Podemos aproximar a função u contínua por meio de uma combinação linear das funções de interpolação ϕ_i como

$$u \approx u_{ap} = \sum_{i=0}^{n} u_i \phi_i, \tag{4.1}$$

sendo u_i os coeficientes da aproximação. Para uma base nodal, u_i é o valor da função u calculada nas coordenadas nodais. No problema de projeção, as funções de interpolação $\{\phi_i\}$ determinam um espaço, representado pelo plano da Figura 4.1. A combinação linear de funções define a aproximação da função u_{ap}. O erro entre a função u e a função aproximada u_{ap} é dado por:

$$e = u - u_{ap}. \tag{4.2}$$

O erro e representa a "distância" entre a função u e o "plano" definido por $\{\phi_i\}$. De forma análoga ao caso que a menor distância entre um ponto e um plano é o vetor normal do plano que passa por esse ponto, o erro e é ortogonal ao "plano" $\{\phi_i\}$. Dessa maneira, tem-se a seguinte relação de

ortogonalidade entre a função e e cada função de interpolação ϕ_i

$$\int_{\Omega} e\phi_i d\Omega = 0. \tag{4.3}$$

Substituindo-se (4.2) em (4.3), tem-se

$$\int_{\Omega} (u - u_{ap})\, \phi_i d\Omega = 0. \tag{4.4}$$

Logo,

$$\int_{\Omega} u_{ap}\phi_i d\Omega = \int_{\Omega} u\phi_i d\Omega. \tag{4.5}$$

Substituindo-se a expressão (4.1) em (4.5) obtém-se

$$\sum_{i=0}^{n} u_i \int_{\Omega} \phi_i\phi_j d\Omega \;\; = \;\; \int_{\Omega} u\phi_j d\Omega \qquad\qquad j = 0,1,...,n. \tag{4.6}$$

Essa equação pode ser reescrita como

$$\sum_{i,j=0}^{n} M_{ij}u_j = b_j, \tag{4.7}$$

sendo M_{ij} os coeficientes da matriz de massa dados por

$$M_{ij} = \int_{\Omega} \phi_i\phi_j d\Omega \tag{4.8}$$

e b_j os elementos do vetor de carga de corpo

$$b_j = \int_{\Omega} u\phi_j d\Omega. \tag{4.9}$$

4.1.2 Erro na Norma L_2

A norma L_2 é um caso particular das normas canônicas que também são conhecidas como normas L_p (GRADSHTEYN *e outros*, 2000). No caso da norma L_2, o erro da aproximação da solução para problemas que possuem solução analítica como o de projeção é calculado da seguinte

forma:

$$\|e\|_{L_2} = \sqrt{\int_{\Omega} \|u - u_{ap}\|^2 \, d\Omega}. \tag{4.10}$$

4.2 Procedimentos de Cálculo D1_MATRICES e STANDARD

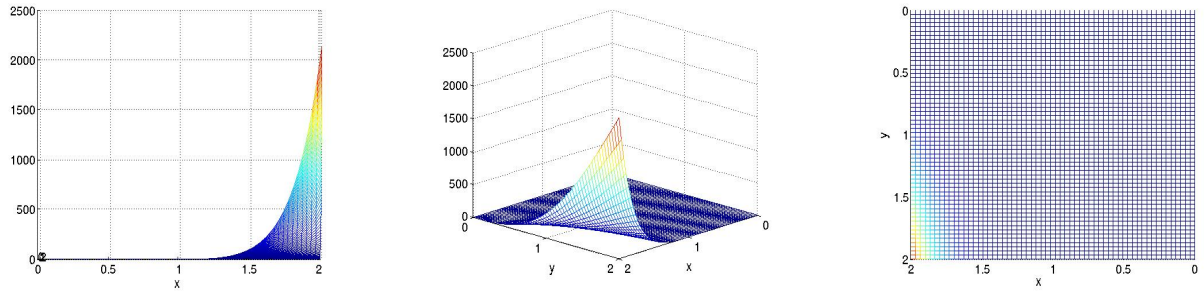
Como visto na Seção 2.3.2, os procedimentos de cálculo D1_MATRICES e STANDARD definem a abordagem de cálculo da matriz de massa (4.8) e do carregamento (4.9) para os elementos 2D ou 3D.

As funções a serem aproximadas em elementos 2D (Quadrado) e 3D (Hexaedro) são dadas nas equações (4.11) e (4.12), respectivamente, cujos os gráficos são ilustrados nas Figuras 4.2 e 4.3. Para a função (4.12), exibe-se os gráficos fixando a variável z em zero.

$$f(x,y) = \exp(\pi x) \sin\left(\frac{\pi y}{4}\right) (x-1)^2 y^2. \tag{4.11}$$

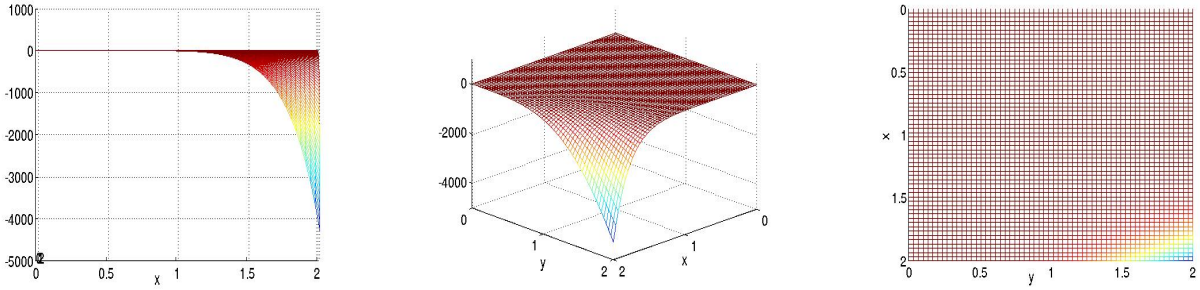
$$f(x,y,z) = \exp(\pi x) \sin\left(\frac{\pi y}{4}\right) (x-1)^2 y^2 (z - xy) \tag{4.12}$$

Os testes são computados utilizando-se malhas estruturadas e não-estruturadas, que são apresentadas nas Figuras 4.4 e 4.5. Para todos os testes, os arquivos de entrada usam base polinomial de Lagrange, pontos de colocação igualmente espaçados, integração numérica de Gauss-Legendre e solução de sistema linear pelo método de Gauss.



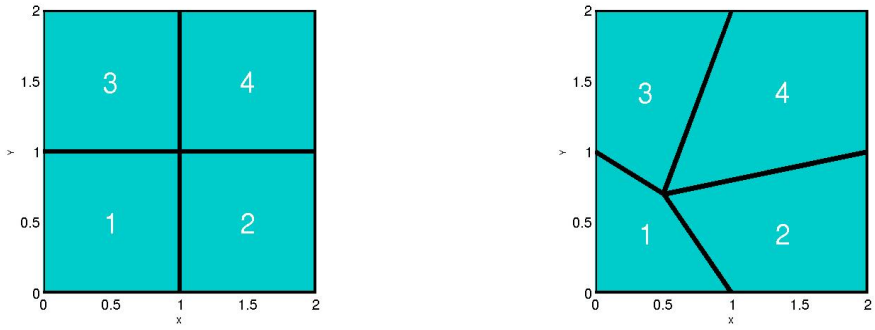
(a) (b) (c)

Figura 4.2: Função 2D utilizada.



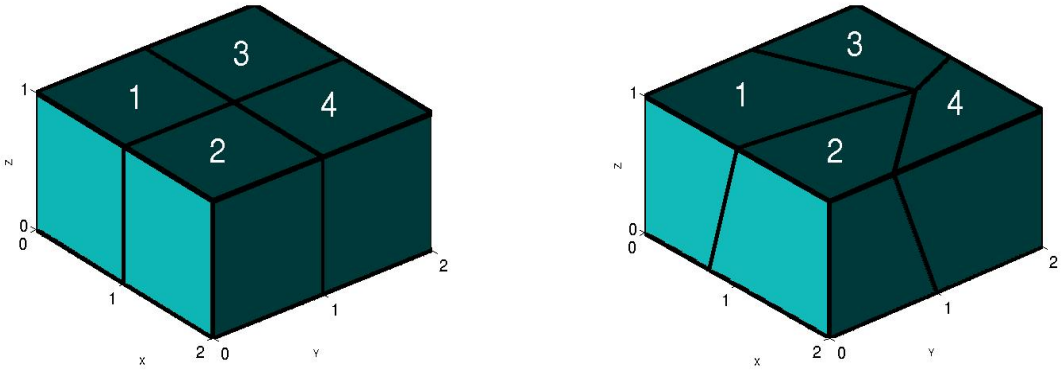
(a) (b) (c)

Figura 4.3: Solução para o problema de projeção 3D plotada com $z = 0$.



(a) Malha estruturada. (b) Malha não-estruturada.

Figura 4.4: Quadrado - Malha estruturada e não-estruturada.



(a) Malha estruturada. (b) Malha não-estruturada.

Figura 4.5: Hexaedro - Malha estruturada e não-estruturada.

4.2.1 Malha de Quadrados

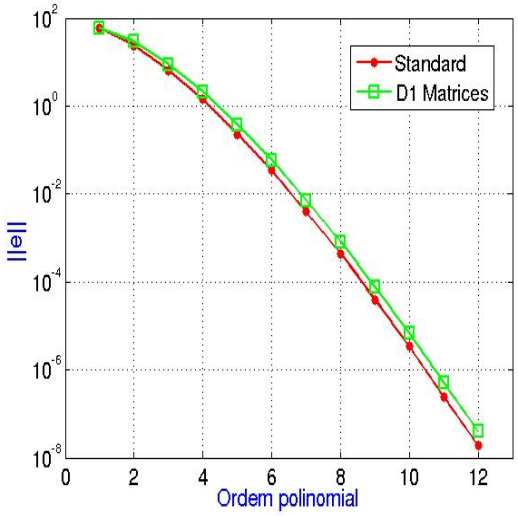
Os testes foram feitos na malha 2D com 4 elementos, apresentada na Figura 4.4. Variou-se o grau do polinômio de 1 a 15. Primeiramente, compara-se o erro na norma L_2 entre os procedimentos de cálculo D1_MATRICES e STANDARD, quando plotados em função da ordem do polinômio na Figura 4.6(a), e em função dos graus de liberdade na Figura 4.6(b). Os dois procedimentos apresentam um erro similar e como esperado convergem de forma exponencial.

O custo computacional entre as estratégias de cálculo (STANDARD e D1_MATRICES) é apresentado na Figura 4.7. O tempo de execução e a memória consumida foram avaliados para todo processo de execução do hp^2FEM do *solver* de projeção (ver Diagramas 3.19 e 3.20) para malhas uniformes com p-uniforme. Para esses testes, foram utilizadas as funções ***gettimeofday()*** e ***getrusage()*** da biblioteca GNU C (LOOSEMORE *e outros*, 2012). Assim, calcula-se o tempo de execução e o valor máximo de memória consumida do processo executado para cada grau polinomial. Esses valores são apresentados na Tabela 4.1.

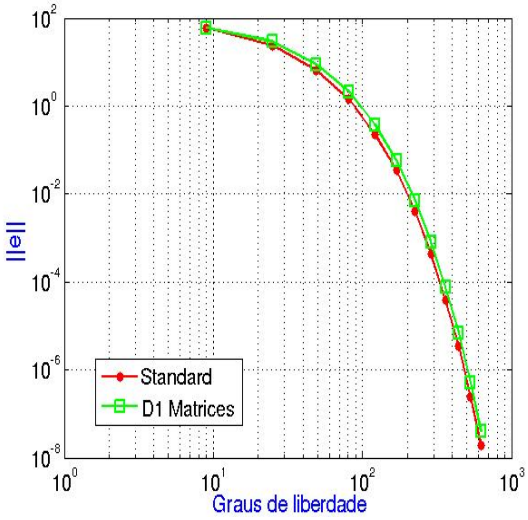
A análise do custo computacional foi realizada por meio de regressão polinomial sobre as funções de tempo de execução e memória consumida para malhas de quadrado e hexaedro. Nos gráficos das Figuras 4.7(a) e 4.7(b), as barras representam os tempos de execução e memória (kilobytes) consumida. As linhas em vermelho e verde são os ajustes realizados buscando a função polinomial que se aproxima às funções de tempo e memória calculados. E por fim, a linha em amarelo é variação entre as duas funções de ajuste polinomial para os procedimentos STANDARD e D1_MATRICES.

O tempo de execução e o consumo de memória obtidos entre D1_MATRICES e STANDARD são próximos para polinômios de ordem mais baixa. Porém, para ordens polinomiais maiores, a função obtida pela regressão polinomial cresce mais para o procedimento STANDARD. Logo, o grau de ajuste polinomial para a função tempo de execução foi de 5 para STANDARD e 4 para D1_MATRICES e para a função de consumo de memória, 7 e 5 para STANDARD e D1_MATRICES, respectivamente.

Outro caso avaliado é visto na Figura 4.8, que apresenta a avaliação do erro na norma L_2 entre malha uniforme e não-estruturada para intervalo de 9 a 625 graus de liberdade. Nesse caso, o erro calculado foi próximo nos dois tipos de malha.

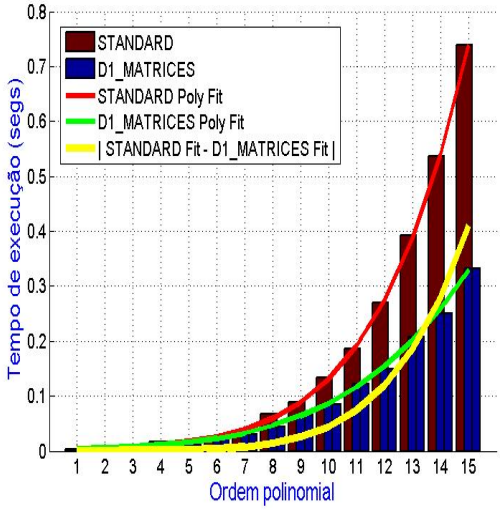


(a) Erro da norma L_2 X Ordem polinomial.

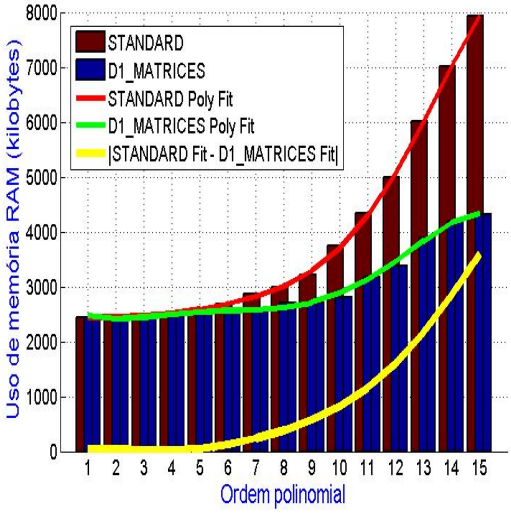


(b) Erro da norma L_2 X Graus de liberdade.

Figura 4.6: Quadrado - Erro da norma L_2 entre os procedimentos D1_MATRICES e STANDARD.



(a) Tempo de execução.



(b) Uso de memória (kilobytes).

Figura 4.7: Quadrado - Tempo de execução e consumo de memória entre D1_MATRICES e STANDARD.

Tabela 4.1: Quadrado - Avaliação computacional.

Grau Polinomial	Tempo de execução (segs)		Consumo de memória (kilobytes)	
	STANDARD	D1_MATRICES	STANDARD	D1_MATRICES
1	0.003020	0.002873	2432	2448
2	0.004644	0.004258	2452	2452
3	0.008870	0.010320	2472	2460
4	0.016553	0.010009	2512	2476
5	0.015002	0.013507	2580	2500
6	0.023676	0.026433	2688	2532
7	0.036647	0.030434	2860	2612
8	0.066724	0.043895	2988	2696
9	0.088096	0.065344	3220	2712
10	0.134300	0.084408	3744	2816
11	0.186133	0.122620	4340	3176
12	0.269977	0.148911	4996	3368
13	0.392978	0.208086	6016	3888
14	0.536603	0.250939	7012	4188
15	0.738450	0.332220	7924	4312

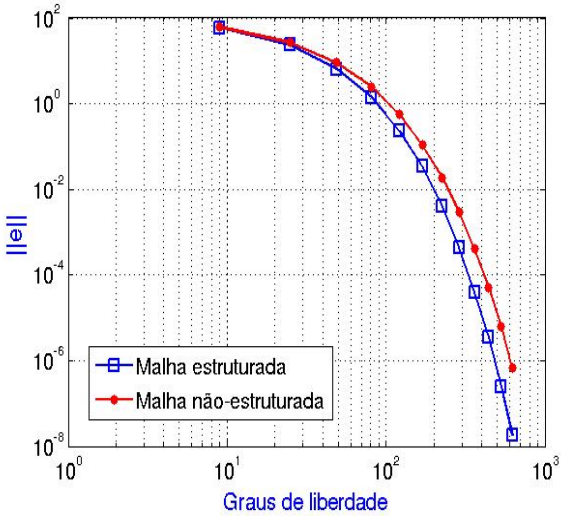


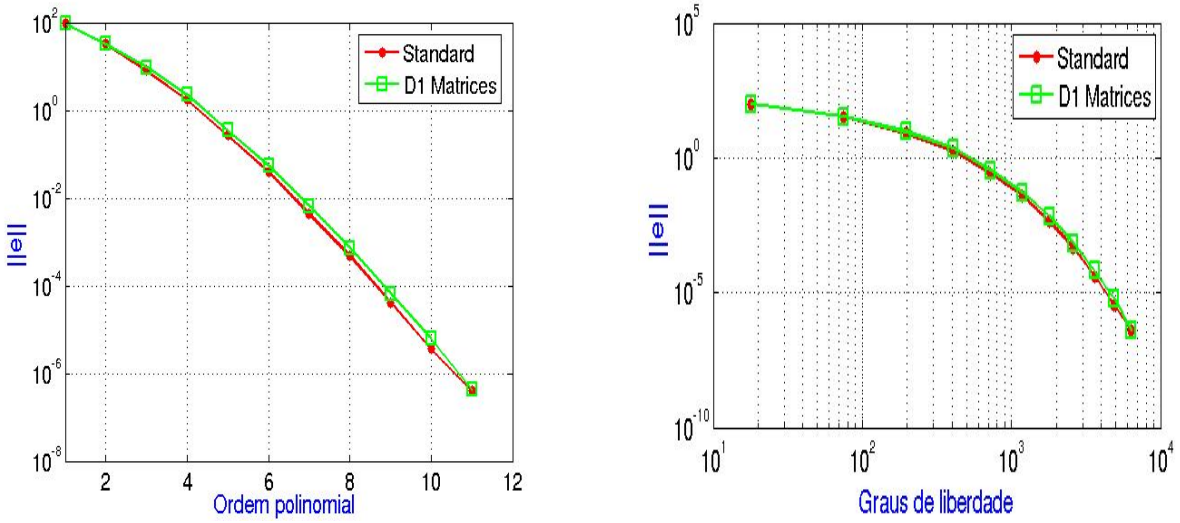
Figura 4.8: Quadrado - Erro da norma L_2 entre tipos de malhas.

4.2.2 Malha de Hexaedros

Os casos anteriores da Seção 4.2.1 também foram avaliados para elementos hexaedros. Os resultados do erro na norma L_2 assemelham-se aos resultados dos elementos quadrados (Figura 4.9). Outro caso analisado, a comparação entre malha estruturada e não-estruturada obtêm a convergência do erro de forma similar e exponencial (Figura 4.11).

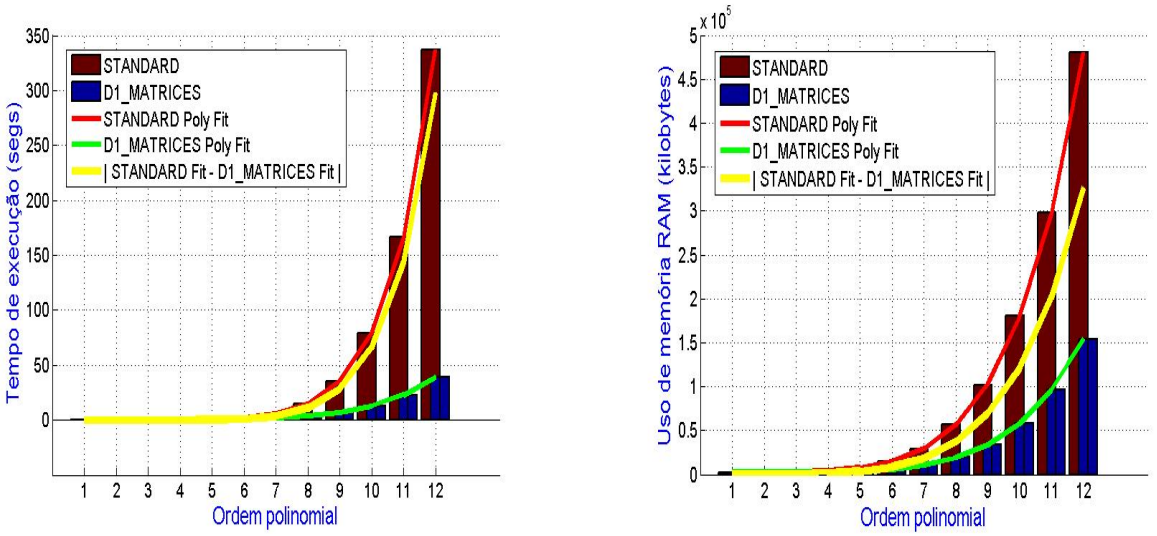
Para a avaliação do custo computacional, apresentam-se os valores de consumo de memória e tempo de execução na Tabela 4.2 e aplica-se o mesmo procedimento de regressão polinomial usado na Seção 4.2.1. Assim, de forma semelhante à Figura 4.7, a Figura 4.10 descreve o custo computacional para a malha de hexaedros.

Assim como no caso anterior (Seção 4.2.1), o procedimento STANDARD obtém maior custo computacional avaliado sobre a malha uniforme com p-uniforme. A função polinomial obtida pelo ajuste da função do tempo de execução é de grau 7 para STANDARD e 5 para D1_MATRICES. Logo, obteve-se uma maior diferença no tempo de execução entre esse procedimentos para elementos hexaedros do que em relação a elementos quadrados. Para a avaliação do consumo de memória, os graus da função de ajuste polinomial são de 7 e 6 para STANDARD e D1_MATRICES, respectivamente.



(a) Erro da norma L_2 X Ordem polinomial. (b) Erro da norma L_2 X Graus de liberdade.

Figura 4.9: Hexaedro - Erro da norma L_2 entre os procedimentos D1_MATRICES e STANDARD.



(a) Tempo de execução.

(b) Uso de memória (kilobytes).

Figura 4.10: Hexaedro - Tempo de execução e consumo de memória entre D1_MATRICES e STANDARD.

Tabela 4.2: Hexaedro - Avaliação computacional.

Grau Polinomial	Tempo de execução (segs)		Consumo de memória (kilobytes)	
	STANDARD	D1_MATRICES	STANDARD	D1_MATRICES
1	0.005406	0.030677	2468	2468
2	0.015912	0.018219	2584	2508
3	0.049234	0.044774	2996	2632
4	0.164143	0.122129	4220	2852
5	0.549411	0.326865	7540	4088
6	1.681481	0.767674	14552	5952
7	5.336972	1.742116	28904	10656
8	14.411422	3.473562	56480	19172
9	35.352515	6.938881	101868	33736
10	79.008939	12.892254	180152	58080
11	166.510947	22.722333	297308	95972
12	336.605528	39.144736	480016	153708

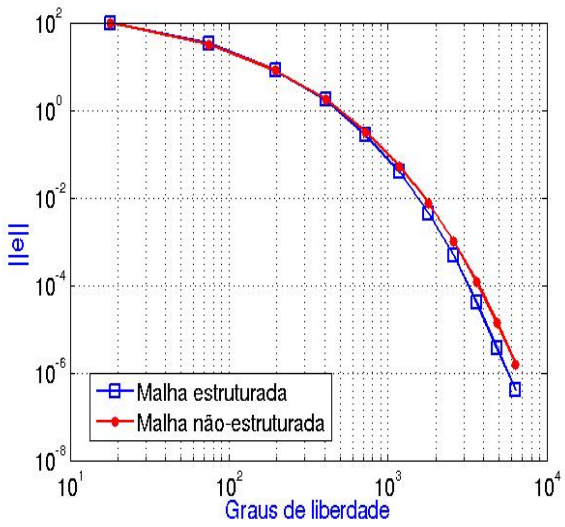


Figura 4.11: Hexaedro - Erro da norma L_2 entre tipos de malhas.

4.3 Estratégia p-Não-Uniforme

As funções (4.11) e (4.12) apresentam um gradiente elevado em uma das extremidades de seus domínios. Dessa forma, espera-se obter o resultado aproximado da melhor solução p-uniforme, aplicando o melhor grau de solução de uma malha p-uniforme apenas nas extremidades e diminuindo o grau nas outras regiões das funções, onde a variação em x e y é quase constante.

A avaliação do caso p-não-uniforme descrito na Seção 2.3.4 é feita através da comparação entre os erros da norma L_2 p-uniforme e p-não-uniforme. Como pode ser visto nas Tabelas 4.3 e 4.4, um determinado grau polinomial é definido em cada elemento. Assim, para os 4 elementos quadrados e hexaedros, determinaram-se diferentes ordens polinomiais aproximando a ordem polinomial p-uniforme que obtêm o melhor resultado.

Tabela 4.3: Quadrado - variação polinomial.

Tipo de variação polinomial	Elem. 1	Elem. 2	Elem. 3	Elem. 4	$\|e\|$ da norma L_2
1	13	13	13	13	1.8846e-08
2	12	12	13	13	1.9661e-08
3	11	11	13	13	8.2871e-08
4	12	12	12	13	1.9693e-08
5	11	11	11	13	8.4874e-08
6	11	11	12	13	8.2876e-08
7	11	12	12	13	1.9863e-08
8	10	11	12	13	8.4268e-08

Nas Figuras 4.12 e 4.13, observa-se que o erro na norma L_2 obtido para p-uniforme é representado por uma linha vermelha. As colunas em azul, numeradas de 1 a 8, indicam o tipo de variação polinomial p-não-uniforme especificadas nas Tabelas 4.3 e 4.4, variando-se o grau do polinômio dos elementos em até -3. Para essa variação, o erro obtido é próximo ao do procedimento p-uniforme, estando dentro da mesma ordem de grandeza.

Tabela 4.4: Hexaedro - variação polinomial.

Tipo de variação polinomial	Elem. 1	Elem. 2	Elem. 3	Elem. 4	$\|e\|$ da norma L_2
1	11	11	11	11	3.7999e-07
2	10	10	11	11	6.3310e-07
3	9	9	11	11	6.0686e-06
4	10	10	10	11	6.4565e-07
5	9	9	9	11	6.6834e-06
6	9	9	10	11	6.0697e-06
7	9	10	10	11	6.7297e-07
8	8	9	10	11	6.1115e-06

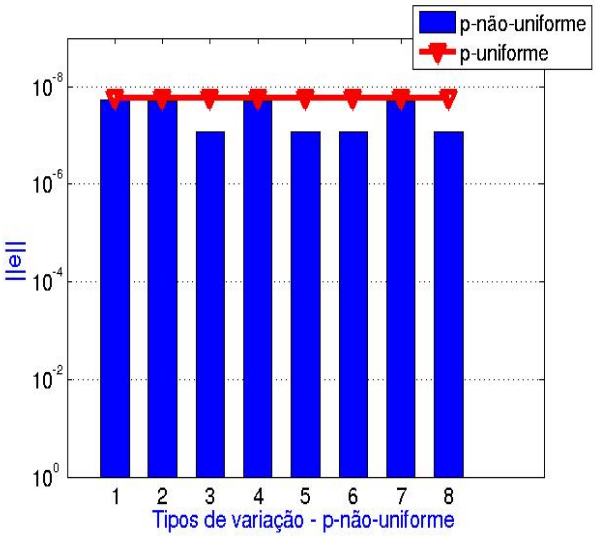


Figura 4.12: Quadrado - comparação entre p-não-uniforme e p-uniforme.

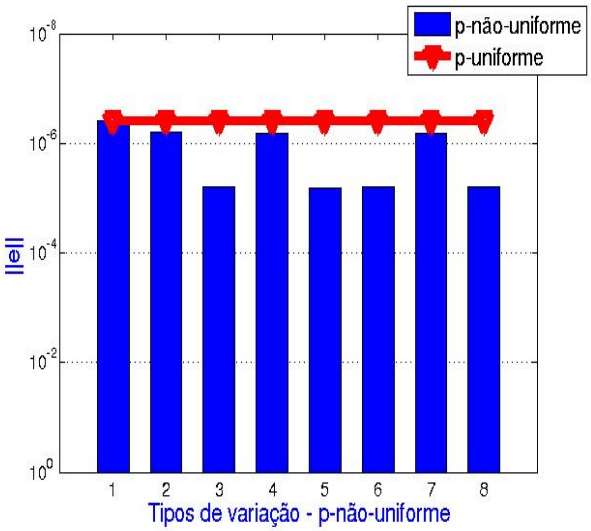


Figura 4.13: Hexaedro - comparação entre p-não-uniforme e p-uniforme.

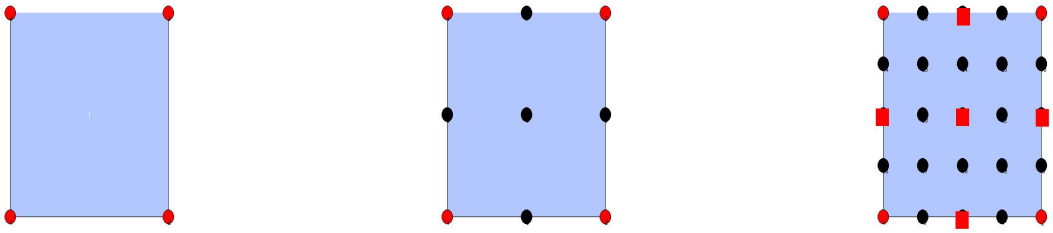
4.4 Interpolação entre as Malhas de Pós-Processamento e Solução

Como exemplificado na Seção 2.3.3, é possível interpolar os resultados da malha de solução na malha de pós-processamento, quando as duas possuem ordens polinomiais diferentes. Tal procedimento permite uma maior flexibilidade na visualização dos resultados (dependendo do programa a ser usado para essa finalidade). Os testes apresentados consideram que a malha de pós-processamento tem o grau polinomial menor que a malha de solução.

As Figuras 4.14 e 4.15 representam a diferença de interpolação entre as malhas. Para as Figuras 4.14(a) e 4.14(b), emprega-se a interpolação da malha de pós-processamento com grau 1 e solução com grau 2. Para as Figuras 4.14(b) e 4.14(c), a malha de pós-processamento é de grau 2 e solução de grau 4. O mesmo comportamento pode ser percebido na Figura 4.15 nos hexaedros.

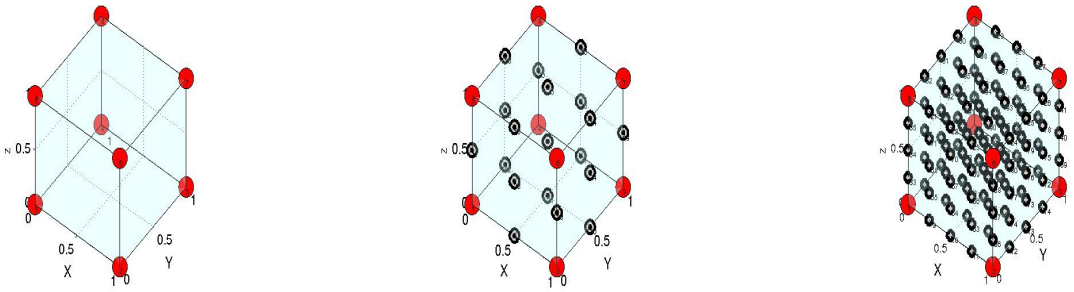
Os resultados obtidos nessa seção visam a comparação das soluções em cada grau de liberdade, entre as malhas de pós-processamento e de solução. Dessa forma, espera-se obter resultados conforme descrito na Seção 2.3.3. A Tabela 4.5 exibe a solução para cada grau de liberdade de um elemento quadrado, cuja malha de pós-processamento foi gerada com grau 2 e a de solução com grau 4. Os quatro primeiros coeficientes são iguais, assim como para os graus de liberdade de 5 a 9 da malha de pós-processamento, os quais correspondem aos graus de liberdade 6, 9, 12, 15 e 21 de solução.

O mesmo comportamento é obtido para os 4 elementos hexaedros, como observado na Tabela 4.6. Os resultados para os primeiros graus de liberdade são os mesmos entre as malhas, cujas ordens são 2 para malha de solução e 1 na malha de pós-processamento.



(a) $p = 1$. (b) $p = 2$. (c) $p = 4$.

Figura 4.14: Quadrado - interpolação entre as malhas de pós-processamento e solução para os graus (1,2) e (2,4).



(a) $p = 1$. (b) $p = 2$. (c) $p = 4$.

Figura 4.15: Hexaedro - interpolação entre as malhas de pós-processamento e solução para os graus (1,2) e (2,4)

Tabela 4.5: Quadrado - solução obtida nas malhas de solução e pós-processamento.

Graus de liberdade	Malha de solução com grau 4	Graus de liberdade	Malha de pós-processamento com grau 2
1	**7.2624e-01**	1	**7.2624e-01**
2	**-2.8905e-04**	2	**-2.8905e-04**
3	**1.9843e-05**	3	**1.9843e-05**
4	**-4.9856e-02**	4	**-4.9856e-02**
5	*3.2087e-01*		
6	**9.8224e-02**	5	**9.8224e-02**
7	*1.2493e-02*		
8	*-3.4816e-04*		
9	**-3.3858e-04**	6	**-3.3858e-04**
10	*-1.8431e-04*		
11	*-8.5766e-04*		
12	**-6.7431e-03**	7	**-6.7431e-03**
13	*-2.2028e-02*		
14	*4.6308e-01*		
15	**8.5068e-01**	8	**8.5068e-01**
16	*8.7476e-01*		
17	*3.8650e-01*		
18	*1.1831e-01*		
19	*1.5048e-02*		
20	*3.7586e-01*		
21	**1.1506e-01**	9	**1.1506e-01**
22	*1.4634e-02*		
23	*2.0460e-01*		
24	*6.2632e-02*		
25	*7.9662e-03*		

Tabela 4.6: Hexaedro - solução obtida nas malhas de solução e pós-processamento.

Graus de liberdade	Malha de solução com grau 2	Malha de pós-processamento com grau 1
1	**-7.5091e-03**	**-7.5091e-03**
2	**4.2716e-02**	**4.2716e-02**
3	**5.1340e-01**	**5.1340e-01**
4	**-9.0252e-02**	**-9.0252e-02**
5	**-1.7511e-02**	**-1.7511e-02**
6	**2.0729e-02**	**2.0729e-02**
7	**5.5045e-02**	**5.5045e-02**
8	**-2.9876e-01**	**-2.9876e-01**
9	*3.1578e-02*	
10	*3.8909e-02*	
11	*3.7954e-01*	
12	*-6.8399e-03*	
13	*-7.8627e-03*	
14	*-2.5229e-02*	
15	*-4.4270e-01*	
16	*-3.6017e-02*	
17	*-1.2510e-02*	
18	*3.1722e-02*	
19	*2.8422e-01*	
20	*-1.9451e-01*	
21	*2.8764e-02*	
22	*-2.1428e-02*	
23	*1.1858e-02*	
24	*6.8399e-03*	
25	*-3.1578e-02*	
26	*-8.6292e-02*	
27	*-2.8764e-02*	

5 Considerações Finais

5.1 Conclusões

Baseados em tomadas de decisões no percurso de desenvolvimento, pode-se dizer que a arquitetura de *software* hp^2FEM provê generalização, flexibilidade e modularização para as funcionalidades do MEF-AO, as quais foram implementadas sobre o paradigma de orientação a objeto em C++. A estratégia de desenvolvimento *bottom-up* (descrita na Seção 3.1.1) proporcionou a validação das funcionalidades dos módulos de forma mais consistente e precisa. Apesar de consumir boa parte do tempo nas etapas de modelagem, implementação e testes em camadas mais baixas, essa metodologia proporcionou ganho de tempo em testes das camadas superiores.

Dentre os principais conceitos de engenharia de *software* para obter o propósito geral da arquitetura, destacam-se o uso de encapsulamento, métodos virtuais e generalização de objetos. Isso pode ser visto no Capítulo 3 e percebido de forma mais clara em duas partes do *software* hp^2FEM. A primeira é nos módulos base, onde é possível escolher diferentes tipos de métodos para interpolação polinomial e solução de sistemas lineares. A segunda através da camada de aplicações, que permite aos usuários adicionar rotinas de soluções ao MEF-AO de forma flexível, aproveitando rotinas já desenvolvidas.

Para a validação da arquitetura, foram realizados testes de unidades, gerando entradas específicas em cada módulo. Importantes tomadas de decisões sobre a arquitetura foram realizadas durante a fase de testes de integração. Nesse processo, além de comprovar que a arquitetura planejada permitia flexibilidade de alteração, alguns acoplamentos foram identificados, proporcionando melhorias nos requisitos previamente apresentados.

Para as etapas de desenvolvimento do *software* foi fundamental o uso do ambiente de ferramentas proposto. Entre os principais aspectos do *framework*, destaca-se a forma precisa para detectar vazamento de memória e acesso indevido nas regiões de memória. Outros recursos proporcionados foram a facilidade na modelagem, geração de código a partir do modelo e o suporte de ferramentas na implementação do analisador simbólico-numérico, fundamental para aplicação de expressões simbólicas ao MEF-AO.

Para o teste funcional da arquitetura proposta, foi considerado o problema de projeção com as abordagens de soluções propostas. As estratégias implementadas, além de proporcionarem metodologias de solução ao algoritmo do MEF-AO, viabilizaram melhor desempenho da arquitetura hp^2FEM.

Na busca de otimização na modelagem do MEF-AO, foram obtidos bons resultados com aplicação do procedimento de cálculo usando matrizes com funções 1D para solução do sistema da expressão (4.7) para malhas uniformes com p-uniforme. O consumo de memória e tempo de execução foram reduzidos de forma considerável. Nesse caso, é possível dizer que a estratégia de construção das matrizes de massa e vetor de carga a partir de matrizes unidimensionais reduz de maneira significativa o custo computacional.

Por outro lado, a construção dessas matrizes utilizando as próprias funções de forma de elementos 2D ou 3D aumenta o custo (procedimento padrão). Pelos resultados obtidos, conclui-se que para um maior número de elementos ou alto grau polinomial, a diferença do custo computacional entre os dois procedimentos pode ser ainda maior.

Para a abordagem de interpolação entre malha de pós-processamento e solução, demonstrou-se na Seção 4.4 que o algoritmo de interpolação aplicado mantém a consistência dos dados. Assim, na aplicação de um grau polinomial menor no pós-processamento em relação à malha de solução, observou-se que os valores nos pontos da malha de solução permaneceram os mesmos para a malha de pós-processamento.

Essa estratégia de aplicação de variação polinomial não uniforme comprova outra característica de flexibilidade do hp^2FEM, o qual é possível gerar funções para diferentes ordens polinomiais simultaneamente. O mesmo ocorre na aplicação de graus polinomiais diferentes entre os tipos de malhas do hp^2FEM (malhas de entrada, solução, mapeamento e pós-processamento, assim como exemplificado na Seção 2.3.3).

Em relação ao resultado do erro obtido, observa-se que com a redução do grau de alguns elementos da malha é possível chegar à resultados semelhantes. As funções (4.11) e (4.12) apresentam comportamentos suaves em alguns intervalos, sendo assim não é necessário utilizar o mesmo número de pontos de interpolação em toda malha para aproximar a solução. Em problemas mais complexos com malhas maiores, espera-se que possa ser configurado uma variação de grau polinomial maior no mesmo tipo de malha com o objetivo de obter redução nos custos computacionais.

5.2 Propostas para Trabalhos Futuros

As sugestões de continuação desse trabalho podem ser dadas a seguir:

- Aprimoramento das configurações de entrada do hp^2FEM, atrelando ao código um gerador de malhas de elementos finitos.
- Implementação de estimadores de erro e estratégias de refinamento da solução.
- Análise de desempenho ou requisitos não funcionais através do uso de ferramentas de perfilamento, otimizando módulos do hp^2FEM.
- Desenvolvimento de módulo para uso de interfaces gráficas em atividades de pós-processamento e análise de resultados.
- Implementação de código otimizado para execução paralela. Esse tipo de implementação, poderá ser definido de forma híbrida com o uso de memória distribuída e compartilhada, proporcionando estratégias diferentes para cada módulo do *software*. Nesse caso, pode-se usar MPI (GROPP *e outros*, 1999) (para o uso de memória distribuída) para distribuição de soluções de estruturas maiores do código. Além disso, é possível o uso de programação com múltiplas *threads* ou em GPU (Graphics Processing Unit) para memória compartilhada usando a manipulação e operações sobre estruturas menores como produto matricial.

Referências

Necktar++ - spectral/hp element framework. abril 2012.
URL: *http://www.nektar.info/*

ANZT, H.; AUGUSTIN, W.; BAUMANN, M.; BOCKELMANN, H.; GENGENBACH, T.; HAHN, T.; HEUVELINE, V.; KETELAER, E.; LUKARSKI, D.; OTZEN, A.; RITTERBUSCH, S.; ROCKER, B.; RONNAS, S.; SCHICK, M.; SUBRAMANIAN, C.; WEISS, J.P. e WILHELM, F. Hiflow3 – a flexible and hardware- aware parallel finite element package. **Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)**, 2010.

AZEVEDO, A.F.M. **Método dos Elementos Finitos**. Faculdade de Engenharia da Universidade do Porto, Porto, Portugal, 1 ed., Abril 2003.

BANGERTH, W.; HARTMANN, R. e KANSCHAT, G. deal.ii — a general purpose object oriented finite element library. **ACM Trans. Math. Softw.**, v. 33, n. 4, agosto 2007.
URL: *http://doi.acm.org/10.1145/1268776.1268779*

BARGOS, F. F. **Implementação de Elementos Finitos de Alta Ordem baseado em Produto Tensorial**. 2009. Dissertação (Mestrado). UNICAMP - Universidade Estadual de Campinas.

BASTIAN, P.; ENGWER, C.; BLATT, M.; KLÖFKORN, R.; SANDER, O.; DEDNER, A. e OHLBERGER, M. **The Distributed and Unified Numerics Environment (DUNE) Grid Interface HOWTO.** Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Universität Heidelberg, Heidelberg, Germany, Maio 2006.

BITTENCOURT, M.L. Using C++ templates to develop finite element classes. **Engineering Computations**, v. 17, n. 7, 775–788, 2000.

BITTENCOURT, M.L. **Análise computacional de estruturas com aplicação do Método de Elementos Finitos**. Editora da Unicamp, 2010.

BROWN, C.A.E.A. Valgrind. Agosto 2011.
URL: *http://valgrind.org/*

CLEMENTS, P.; BACHMANN, F.; BASS, L.; GARLAN, D.; IVERS, J.; LITTLE, R.; MERSON, P.; NORD, R. e STAFFORD, J. **Documenting software architectures : views and beyond**. Pearson Education, Inc., 2nd ed., 2010.

COMSOL, I. Comsol multiphysics. 2012.
URL: *http://www.comsol.de/products/multiphysics/*

DANTAS, B. A. **Um framework para análise sequencial e em paralelo de sólidos elásticos pelo método dos elementos finitos**. 2006. Dissertação (Mestrado). Universidade Federal de Mato Grosso do Sul.

DEITEL, P.J. e DEITEL, H.M. **C++ How To Program**. Prentice Hall, 6 ed., 2008.

FERNANDES, F. G. **Um arcabouço para verificação automática de modelos UML**. 2011. Dissertação (Mestrado).

FU, C. Parallel computing for finite element structural analysis on workstation cluster. **International Conference on Computer Science and Software Engineering**, pp. 291–294, 2008.

FURLAN, F. A. C. **Métodos locais de integração explícito e implícito aplicados ao método de elementos finitos de alta ordem**. 2011. Dissertação (Mestrado). UNICAMP - Universidade Estadual de Campinas.

GARLAN, D. e SHAW, M. **An Introduction to Software Architecture**. World Scientific Publishing Company, 1994.

GRADSHTEYN, I.S.; RYZHIK, I.M.; JEFFREY, A. e ZWILLINGER, D. **Table of Integrals, Series, and Products, Sixth Edition**. Academic Press, 6th ed., agosto 2000.

GROPP, W.; LUSK, E. e SKJELLUM, A. **Using MPI, 2nd Edition: Portable Parallel Programming with the Message Passing Interface**. MIT Press, novembro 1999.

GUIMARÃES, A.C.S. e FEIJÓO, R.A. O sistema ACDP. Relatório de Pesquisa e Desenvolvimento 027/89, LNCC, Rio de Janeiro, 1989.

HAVEN, H. **Metamill 6.0: User's Guide**. Metamill Software, Julho 2011.

HEESCH, D.V. **Doxygen - Manual for version 1.8.1.1**. Doxygen, Junho 2012.

HEITLINGER, P. **Guia prático da XML**. Inova, 2001.

HUGHES, T.J.R. **The Finite Element Method Linear Static and Dynamic Finite Element Analysis**. 2000.

KARNIADAKIS, G.E. e S., S. **Spectral/hp Element Methods for Computational Fluid Dynamics.** Oxford University Press, 2005.

KAWABATA, C.L.O.; VENTURINI, W.S. e CODA, H.B. Desenvolvimento e implementação de um método de elementos finitos paralelo para análise não linear de estruturas. **Cadernos de Engenharia de Estruturas, São Carlos.**, v. 11, 151–155, 2009.

KNOW, Y.W. **The finite element method**. CRC Press, 1997.

LOOSEMORE, S.; STALLMAN, R.M.; MCGRATH, R.; A., O. e DREPPER, U. **The GNU C Library Reference Manual**. Free Software Foundation, Inc., for version 2.16 ed., 2012.

MCCORMICK, C.W. **The NASTRAN User's Manual**. NASA - National Aeronautics and Space

Administration, Scientijic and Technical Information Division, Office of Technology Utilization, National Aeronautics and Space Administration, Washington, USA., 1970.

MELO, A.C. **Desenvolvendo aplicações com UML 2.2 Do conceitual à implementação**. BRASPORT Livros e Multimidia Ltda., 2010.

MEUER, H.; STROHMAIER, E.; DONGARRA, J. e SIMON, H. Top 500 supercomputer sites @ONLINE. fevereiro 2012.
URL: *http://www.top500.org/*

MIANO, M. G. V. **Tensorização de Matrizes de Rigidez Unidimensionais para Quadrados e Hexahedros Usando o Método de Elementos Finitos de Alta Ordem**. 2009. Tese (Doutorado). UNICAMP - Universidade Estadual de Campinas.

OLIVEIRA, M. L. S. **Modelagem de arquitetura de software orientada a aspectos com UML 2.0**. 2007. Dissertação (Mestrado). IME - Instituto Militar de Engenharia.

OMG. **MOF 2.0/XMI Mapping Specification**. OMG - Object Management Group, 2.4.1 ed., Agosto 2011.

RUBIRA, C.M.F. e BRITO, P.H.S. **Introdução à Análise Orientada a objetos**. Instituto de Computação - Universidade Estadual de Campinas, Campinas, SP, 2007.

SANTOS, P.R. **Compiladores: ferramentas de desenvolvimento**. Universidade Técnica de Lisboa, 4 ed., 2006.

SILVA, C. A. C. **Otimização estrutural e análise de sensibilidade orientadas por objetos.** 1997. Dissertação (Mestrado). UNICAMP - Universidade Estadual de Campinas.

STALLMAN, R.M.; MCGRATH, R. e SMITH, P.D. **The Gnu Make Manual**. Free Software Foundation, free software foundation ed., July 2010.

STOLARSKI, T.; NAKASONE, Y. e YOSHIMOTO, S. **Engineering Analysis with ANSYS Software**. Butterworth-Heinemann, fevereiro 2007.

SUMMERFIELD, M. **Advanced Qt Programming: Creating Great Software with C++ and Qt 4**. Prentice Hall Press, Upper Saddle River, NJ, EUA, 1st ed., 2010.

VANDEVOORDE, D. e JOSUTTIS, N. **C++ templates: the complete guide.** Pearson Education, Inc., 2003.

VASCONCELOS, A.M.L.; ROUILLER, A.C.; MACHADO, C.A.F. e MEDEIROS, T.M.M. **Introdução à engenharia de software e à qualidade de software.** Fundação de Apoio ao Ensino, Pesquisa e Extensão - FAEPE - Universidade Federal de Lavras - UFLA, 2006.

VOS, P. E. J. **From h to p efficiently: Optimising the implementation of spectral/hp Element Methods**. 2011. Tese (Doutorado). University of London.

ANEXO A Arquivos de Entrada do hp^2FEM

Os dados apresentados a seguir correspondem os arquivos de entrada utilizados para testar o problema de projeção para quatro elementos quadrados.

A.1 Arquivo .fem

O arquivo a seguir apresenta as configurações para uma malha estruturada.

```
*TITLE
Title
*

*ELEMENT_GROUPS
1
1   4   SQUARE  1   4

*INCIDENCES
1 1 2 5 4
2 2 3 6 5
3 4 5 8 7
4 5 6 9 8

*DIMENSION
2

*COORDINATES
9
1   0   0
2   1   0
3   2   0
4   0   1
5   1   1
6   2   1
7   0   2
8   1   2
9   2   2
```

```
*GM_KEYPOINT
0

*GM_LINE
0 0

*GM_SURFACE
0 0

*GM_VOLUME
0 0

*END
#
# Developed by Gilberto Luis Valente da Costa
# e-mail: betogil@gmail.com
```

A.2 Arquivo .def

O arquivo a seguir apresenta as configurações para uma malha uniforme e p-uniforme.

```
*POST_PROCESSOR
4 GID

*SOLUTION_DIRECTIVES
*SOLVER_TYPE
ELEMENT_ELEMENT

*ANALYSIS_TYPE
PROJECTION  CONSIST

*SOLUTION_ALGORITHM
LINEAR_SOLUTION

*LINEAR_SOLUTION_METHOD
GAUSS

*OPERATOR_TYPE
SYMMETRIC

*THEORETICAL_SOLUTION
L2 1 1 P s exp(3.1416*X)*sin((3.1416/4)*Y)*((X-1)^2)*(Y^2)


*GROUP_PARAMETERS_1

*MESH_TYPE_1
UNIFORM pUNIFORM

*SOLUTION_ORDER_1
4

*DOFS_GROUP_1
1 P 0 1 P 2 QX QY 0 0 0

*MATERIAL_1
HOOKE 2100 0.3 1.0E-06 1
```

```
*GEOMETRIC_PROPERTIES_1
1 1 1

*MAPPING_1
ISOPARAMETRIC

*KINEMATICS_1
INFINITE MATERIAL TOTAL_LAGRANGIAN

*PROBLEM_PARAMETERS_1
POISSON D1_MATRICES CONSIST CONSISTENT

*INTERPOLATION_1
STAND NODAL NON_HIERARCHICAL STANDARDLAGRANGE

*COLLOCATION_POINTS_1
EQUALLY_SPACED

*INTEGRATION_POINTS_1
GAUSS_LEGENDRE 2

*DOF
1
P
0
1
P
2
QX QY
0
0
0

*HOMOGENEOUS_DIRICHLET_BC
*HDBC_NODES
0
*HDBC_EDGES
0
*HDBC_FACES
0
*HDBC_KEYPOINTS
```

```
0
*HDBC_LINES
0
*HDBC_SURFACES
0

*NON_HOMOGENEOUS_DIRICHLET_BC
*NHDBC_NODES
0
*NHDBC_EDGES
0
*NHDBC_FACES
0
*NHDBC_KEYPOINTS
0
*NHDBC_LINES
0
*NHDBC_SURFACES
0


*LOAD_SETS
1

*LOAD_SET_1

*NODAL_LOADS_1
0
*KEYPOINT_LOADS_1
0
*EDGE_LOADS_1
0
*LINE_LOADS_1
0
*FACE_LOADS_1
0
*SURFACE_LOADS_1
0
*BODY_LOADS_1
4
1 1 P s exp(3.1416*X)*sin((3.1416/4)*Y)*((X-1)^2)*(Y^2)
2 1 P s exp(3.1416*X)*sin((3.1416/4)*Y)*((X-1)^2)*(Y^2)
```

```
3 1 P s exp(3.1416*X)*sin((3.1416/4)*Y)*((X-1)^2)*(Y^2)
4 1 P s exp(3.1416*X)*sin((3.1416/4)*Y)*((X-1)^2)*(Y^2)
*VOLUME_LOADS_1
0
*EDGE_PRESSURE_1
0
*FACE_PRESSURE_1
0
*LINE_PRESSURE_1
0
*SURFACE_PRESSURE_1
0

*END
#
# Developed by Gilberto Luis Valente da Costa
# e-mail: betogil@gmail.com
```

APÊNDICE A Arquivos do analisador simbólico-numérico

A.1 Arquivo do analisador léxico

Código A.1: Arquivo para geração de um analisador léxico

```
%{
#include "global.h"
#include "y.tab.h"

#include <stdlib.h>

#undef YY_INPUT
#define YY_INPUT(buf, result, max_size)      \
      {                                      \
          if (global_expr[0]=='\0')          \
              result = YY_NULL;              \
          else {                             \
              strcpy(buf, global_expr);      \
              result = strlen(global_expr)-1;\
      }                                      \
          global_expr[0]='\0';               \
      }
%}


white     [ \t]+
digit     [0-9]
integer   {digit}+
exponant  [eE][+-]?{integer}
real      {integer}("."{integer})?{exponant}?
variable  [a-z]


PLUS              "+"
MINUS             "-"
MULTIPLY          "*"
DIVIDE            "/"
POWER             "^"
SQRT              "sqrt"
```

```
35 SIN                  "sin"
36 COS                  "cos"
37 TAN                  "tan"
38 LOG                  "log"
39 LN                   "ln"
40 EXP                  "exp"
41 EQUAL                "="
42 LEFT_PARENTHESIS     "("
43 RIGHT_PARENTHESIS    ")"
44 NEW_LINE             "\n"
45 END                  ("end"|"END")
46
47
48 %%
49
50
51 {white}      { /* We ignore white characters */ }
52
53 {real}       {
54                     yylval = atof(yytext);
55                     return NUMBER;
56                 }
57
58 {variable}   {
59                     yylval = *yytext - 'a';
60                     return VARIABLE;
61                 }
62
63 {PLUS}                  {return PLUS; }
64 {MINUS}                 {return MINUS; }
65 {MULTIPLY}              {return MULTIPLY; }
66 {DIVIDE}                {return DIVIDE; }
67 {POWER}                 {return POWER; }
68 {SQRT}                  {return SQRT; }
69 {SIN}                   {return SIN; }
70 {COS}                   {return COS; }
71 {TAN}                   {return TAN; }
72 {LOG}                   {return LOG; }
73 {LN}                    {return LN; }
74 {EXP}                   {return EXP; }
75 {EQUAL}                 {return EQUAL; }
76 {LEFT_PARENTHESIS}     {return LEFT_PARENTHESIS; }
```

```
77 {RIGHT_PARENTHESIS}   {return RIGHT_PARENTHESIS; }
78 {NEW_LINE}             {return NEW_LINE; }
79 {END}                  {return END; }
80
81 %%
```

A.2 Arquivo do analisador sintático

Código A.2: Arquivo para geração de um analisador sintático

```
1  %{
2
3  #include "global.h"
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <math.h>
7  #include <string.h>
8  #include <ctype.h>
9
10 int yyerror(char*);
11
12 %}
13
14
15 %token   NUMBER
16 %token   VARIABLE
17 %token   PLUS MINUS MULTIPLY DIVIDE POWER SQRT
18 %token   SIN COS TAN
19 %token   LOG LN EXP
20 %token   EQUAL
21 %token   LEFT_PARENTHESIS RIGHT_PARENTHESIS
22 %token   NEW_LINE
23 %token   END
24
25 %left    EQUAL
26 %left    PLUS MINUS
27 %left    MULTIPLY DIVIDE
28 %left    SQRT
29 %left    SIN COS TAN
30 %left    LOG LN EXP
```

```
31 %left     NEG
32
33 %right    POWER
34
35 %start    Input
36
37
38 %%
39
40
41 Input:
42     /* Empty */
43     | Input Line
44     ;
45
46 Line:
47     NEW_LINE
48     | Expression NEW_LINE {
49           if (terminal) {
50               printf("Resultado: %lf\n",$1);
51           } else {
52               out_parse = $1;
53               return(0);
54           }
55          }
56     | VARIABLE EQUAL Expression { int ind = $1; sym[ind] = $3; }
57     | END { return(0); }
58     ;
59
60 Expression:
61     NUMBER      { $$ = $1; }
62     | VARIABLE { int ind = $1; $$ = sym[ind]; }
63     | Expression PLUS Expression      { $$ = $1 + $3;     }
64     | Expression MINUS Expression     { $$ = $1 - $3;     }
65     | Expression MULTIPLY Expression { $$ = $1 * $3;     }
66     | Expression DIVIDE Expression    { $$ = $1 / $3;     }
67     | MINUS Expression %prec NEG      { $$ = -$2;         }
68     | Expression POWER Expression     { $$ = pow($1,$3); }
69     | SQRT Expression     { $$ = sqrt($2);  }
70     | SIN Expression      { $$ = sin($2);   }
71     | COS Expression      { $$ = cos($2);   }
72     | TAN Expression      { $$ = tan($2);   }
```

```
73      | LOG Expression      { $$ = log10($2); }
74      | LN  Expression      { $$ = log($2);   }
75      | EXP Expression      { $$ = exp($2);   }
76      | LEFT_PARENTHESIS Expression RIGHT_PARENTHESIS { $$ = $2; }
77      ;
78
79
80  %%
81
82
83  int yyerror(char *s) {
84          printf("%s\n",s);
85  }
```

A.3 Arquivo de interface entre o hp^2FEM e o analisador simbólico-numérico

Código A.3: Arquivo de interface entre o *software* hp^2FEM e o analisador simbólico-numérico

```
1   %{
2
3   #include "global.h"
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <math.h>
7   #include <string.h>
8   #include <ctype.h>
9
10  int yyerror(char*);
11
12  %}
13
14
15  %token    NUMBER
16  %token    VARIABLE
17  %token    PLUS MINUS MULTIPLY DIVIDE POWER SQRT
18  %token    SIN COS TAN
19  %token    LOG LN EXP
20  %token    EQUAL
21  %token    LEFT_PARENTHESIS RIGHT_PARENTHESIS
22  %token    NEW_LINE
```

```
23 %token    END
24
25 %left      EQUAL
26 %left      PLUS MINUS
27 %left      MULTIPLY DIVIDE
28 %left      SQRT
29 %left      SIN COS TAN
30 %left      LOG LN EXP
31 %left      NEG
32
33 %right     POWER
34
35 %start     Input
36
37
38 %%
39
40
41 Input:
42     /* Empty */
43     | Input Line
44     ;
45
46 Line:
47     NEW_LINE
48     | Expression NEW_LINE {
49           if (terminal) {
50               printf("Resultado: %lf\n",$1);
51           } else {
52               out_parse = $1;
53               return(0);
54           }
55          }
56     | VARIABLE EQUAL Expression { int ind = $1; sym[ind] = $3; }
57     | END { return(0); }
58     ;
59
60 Expression:
61     NUMBER      { $$ = $1; }
62     | VARIABLE { int ind = $1; $$ = sym[ind]; }
63     | Expression PLUS Expression      { $$ = $1 + $3;     }
64     | Expression MINUS Expression     { $$ = $1 - $3;     }
```

```
65      | Expression MULTIPLY Expression { $$ = $1 * $3;     }
66      | Expression DIVIDE Expression   { $$ = $1 / $3;     }
67      | MINUS Expression %prec NEG     { $$ = -$2;         }
68      | Expression POWER Expression    { $$ = pow($1,$3); }
69      | SQRT Expression     { $$ = sqrt($2);  }
70      | SIN Expression      { $$ = sin($2);   }
71      | COS Expression      { $$ = cos($2);   }
72      | TAN Expression      { $$ = tan($2);   }
73      | LOG Expression      { $$ = log10($2); }
74      | LN  Expression      { $$ = log($2);   }
75      | EXP Expression      { $$ = exp($2);   }
76      | LEFT_PARENTHESIS Expression RIGHT_PARENTHESIS { $$ = $2; }
77      ;
78
79
80  %%
81
82
83  int yyerror(char *s) {
84          printf("%s\n",s);
85  }
```