UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL

DESENVOLVIMENTO DE ALGORITMOS DE SUBESTRUTURAÇÃO PARA ELEMENTOS FINITOS

Misael Luis Santana Mandujano
Orientador: **Prof. Dr. Philippe R. B. Devloo**
DISSERTAÇÃO DE MESTRADO

Campinas - SP, Brasil
1997

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL

DESENVOLVIMENTO DE ALGORITMOS DE SUBESTRUTURAÇÃO PARA ELEMENTOS FINITOS

Misael Luis Santana Mandujano
Orientador: **Prof. Dr. Philippe R. B. Devloo**

Dissertação de mestrado apresentada à Faculdade de Engenharia Civil como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Civil.

Área de Concentração: Estruturas

Campinas - SP, Brasil
1997

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA CIVIL

DESENVOLVIMENTO DE ALGORITMOS DE SUBESTRUTURAÇÃO PARA ELEMENTOS FINITOS

Misael Luis Santana Mandujano.

Dissertação de Mestrado defendida e aprovada, em 24 de março de 1997, pela Banca Examinadora constituída pelos professores:

Prof. Dr. Philippe Remy Bernard Devloo, presidente
FEC - UNICAMP

Prof. Dr. José Luiz Antunes de Oliveira e Sousa
FEC - UNICAMP

Prof. Dr. Alvaro Luiz Gayoso de Azeredo Coutinho
COPPE-UFRJ

Atesto que esta é a versão definitiva da dissertação

Mãe, há golpes na vida tão fortes como os açoites de carrascos irados, e a sua morte foi o pior desses golpes.

A minha mãe Victoria (in memoriam), a meu pai Luis e a minha esposa Cláudia.

AGRADECIMENTOS

À FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), pela concessão da bolsa de estudos, pela reserva técnica que possibilitou a compra de um computador para o desenvolvimento deste trabalho e apoio financeiro à pesquisa;

Ao Professor Philippe R. B. Devloo, pela orientação, pelo incentivo e pelo entusiasmo;

À Faculdade de Engenharia Civil, pela estrutura oferecida;

Aos funcionários da secretaria da Coordenação de Pós-Graduação, principalmente à Paula, pelo auxílio.

Às secretárias e professores do Departamento de Construção Civil, pela amizade demonstrada;

À minha querida esposa e colega de curso Cláudia, pela compreensão, auxílio e incentivo.

RESUMO

Na engenharia muitos fenômenos físicos são modelados usando equações diferenciais. O método dos elementos finitos é bastante eficiente para resolver numericamente estas equações. Muitas aplicações levam a sistemas com grande quantidade de equações e incógnitas (problemas de grande escala), que para serem resolvidos necessitam de técnicas especializadas. A ferramenta mais atual para resolver problemas de grande escala é o processamento em paralelo. Essa ferramenta é usada em conjunto com a técnica da subestruturação, que consiste em dividir o domínio do problema, gerando uma malha que é chamada de malha grossa. Usando a malha grossa monta-se um sistema de equações que é chamado de sistema reduzido. Este sistema é resolvido usando o método do gradiente conjugado pré-condicionado. Este trabalho tem duas partes: a primeira é implementar a técnica de subestruturação usando o paradigma da programação orientada a objetos. A segunda parte propõe a construção de um pré-condicionador resultante da mudança de bases dos contornos das subestruturas para bases hierárquicas.

ABSTRACT

In Engineering many physical phenomena are modeled using differential equations. The finite element method is very efficient to solve these equations numerically. Many applications lead to very large problems, whose solution requires specialized methods. The most recent tool for solving large scale problems is the parallel processing. This tool is used together with the substructuring technique, which consists in splitting the domain of the problem, generating a coarse mesh. The coarse mesh is assembled into a system of equations named the reduced system. This system is solved by using the pre-conditioned conjugate gradient. This work includes two parts: the first one is the implementation of substructuring using the object oriented programming paradigm. The second part is devoted to the construction of a pre-conditioner resulting from the modification of the shape functions on the contours of the substructures into hierarchical basis functions.

SUMÁRIO

1. INTRODUÇÃO

A evolução acelerada do *hardware* e *software* nos anos noventa tem permitido explorar novos tipos de cálculos que antes eram impossíveis ou muito difíceis de se realizar. No âmbito do desenvolvimento de *software*, um dos maiores progressos foi a introdução da filosofia da programação orientada a objetos. Neste trabalho, esta nova filosofia de programação foi utilizada na implementação da subestruturação dentro da análise estrutural.

A subestruturação vem se mostrando o melhor caminho para a solução em paralelo de grandes sistemas de equações lineares e não lineares, que surgem quando problemas elípticos de elasticidade, dinâmica dos fluidos, e outros problemas de engenharia são discretizados por elementos finitos. Após a divisão do domínio em subdomínios, estes são distribuídos entre os processadores. As variáveis internas de cada subdomínio são reduzidas, obtendo-se um novo sistema referente às variáveis do contorno de cada subestrutura. Este novo sistema é cheio, pois é resultante de operações de multiplicação e adição de matrizes. Para este novo sistema, o método de resolução mais conveniente é o método do gradiente conjugado pré-condicionado, pois este é altamente paralelizável e a sua convergência pode ser acelerada com a utilização de pré-condicionadores.

Os termos subestruturação e subestrutura são equivalentes a decomposição de domínios e subdomínios, respectivamente. Em 1869, SCHWARZ, apud WIDLUND (1990), no estudo de problemas elípticos, desenvolveu o método da decomposição de domínios em sub-regiões para estabelecer a existência de funções harmônicas sobre regiões com contornos não suaves. Em 1963, nasceram os termos subestruturação e subestruturas, quando a restrição da capacidade de memória dos computadores e sua reduzida velocidade de processamento levaram PRZEMIENIECKI a usar a análise matricial de estruturas para dividir as estruturas em várias subestruturas para estudá-las separadamente.

Este trabalho compõe-se basicamente de duas partes: a primeira trata da implementação da subestruturação de forma consistente dentro do ambiente PZ que é orientado a objetos, viabilizando o processamento em paralelo de problemas de grande porte. A segunda parte propõe a construção de um pré-condicionador tomando como base a malha resultante da subestruturação. A técnica de subestruturação foi implementada de forma consistente pela criação da classe **TSuperEl**. Para implementação do pré-condicionador foram criadas duas classes: a classe **TModGridShape** e a classe **TBlockDiagonal**. Foram usadas bases hierárquicas entre as interfaces dos subdomínios. Seu desenvolvimento foi feito sobre domínios regulares e com subdomínios não sobrepostos uns aos outros.

Primeiro uma abordagem geral da aproximação de elementos finitos é apresentada. Em seguida uma descrição do ambiente PZ, que por ser um ambiente de desenvolvimento recente, não tem muita documentação. Esta descrição, portanto, tem o objetivo de servir como um guia de referência para os usuários desse ambiente. Segue a descrição da classe **TMatrix**, que gerencia o cálculo matricial, uma classe desenvolvida anteriormente pelo autor deste trabalho[1] Essa descrição é importante como documentação da classe **TMatrix**.

[1] SANTANA e DEVLOO (1995, 1996)

2. OBJETIVOS

O objetivo principal deste trabalho é a implementação do método de subestruturação dentro de um programa de elementos finitos, usando a filosofia da programação orientada a objetos. As classes resultantes deste esforço foram implementadas dentro do ambiente PZ, tendo em vista os seguintes objetivos específicos:

1. Implementar o algoritmo de sub-estruturação usando métodos diretos dentro de cada sub-domínio;
2. Utilização das equações de interface para a solução iterativa do sistema de equações;
3. Mudança de base dos nós (intersecção de 3 ou mais sub-domínios) dos sub-domínios para uma base hierárquica;
4. Implementação de um pré-condicionador para acelerar a convergência do método de gradiente conjugado pré-condicionado.

3. O MÉTODO DOS ELEMENTOS FINITOS: UMA ABORDAGEM GERAL

As equações diferenciais parciais, dependendo dos seus coeficientes, podem ser classificadas em: equações diferenciais elípticas, hiperbólicas ou parabólicas. Este trabalho está dirigido para o uso dos elementos finitos na aproximação de equações diferenciais elípticas. A maior parte dos problemas estudados pela engenharia passam pela solução de equações diferenciais deste tipo. Neste terceiro capítulo faz-se uma revisão dos principais conceitos da aproximação por elementos finitos.

3.3. REGULARIDADE: OS ESPAÇOS $C^m(\Omega)$ E $H^m(\Omega)$

Uma importante propriedade das soluções de problemas de valor de contorno do método dos elementos finitos é a regularidade, que é "o grau de suavidade" das funções que formam o espaço "solução".

A descrição de uma dada função como sendo "regular" ou "irregular" é muito vaga e qualitativa. É necessário quantificar, de alguma forma, a noção de "grau de regularidade" de uma função. Há uma maneira natural e elegante de fazê-lo, universalmente usada, que consiste em se identificar a classe de continuidade $C^m(\Omega)$ ou a classe de Sobolev $H^m(\Omega)$ à qual a função a ser aproximada pertence.

3.2. O ESPAÇO $C^m(\Omega)$

Suponha-se que Ω seja uma região limitada em R^3 e que $u=u(x,y,z)$ seja uma função de valores reais em Ω. Então u é dita pertencente à classe $C^m(\Omega)$ se u e todas suas derivadas parciais de ordem menor ou igual a m são contínuas em todo ponto (x,y,z) pertencente a Ω, onde m é um número inteiro maior ou igual a zero. Como definição de $C^m(\Omega)$ pode ser usada a seguinte notação:

$$C^m(\Omega) = \left\{ u = u(x,y,z), (x,y,z) \in \Omega | u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}, \frac{\partial^2 u}{\partial x^2}, \frac{\partial^2 u}{\partial x \partial y}, \frac{\partial^2 u}{\partial x \partial z}, \ldots, \frac{\partial^2 u}{\partial z^2}, \ldots, \frac{\partial^m u}{\partial x^m}, \right.$$

$$\left. \frac{\partial^m u}{\partial x^{m-1} \partial y}, \ldots, \frac{\partial^m u}{\partial z^m} \text{ são contínuas em } \Omega \subset R^3 \right\}$$

que diz que $C^m(\Omega)$ é um conjunto de funções cujo membro típico u tem como propriedade que u, $\partial u/\partial x, \ldots \partial^m u/\partial x^m$ são contínuas em uma região Ω contida em R^3".

A classe $C^m(\Omega)$ é um espaço linear de funções, ou seja, se $u \in C^m(\Omega)$ e $v \in C^m(\Omega)$, então $\alpha u + \beta v \in C^m(\Omega)$, para quaisquer escalares reais α e β.

3.3. AS FUNÇÕES L^2

Uma função $f: \Omega \rightarrow R$, não necessariamente contínua, é quadrado integrável se:

$$\int_\Omega f^2 dx < \infty$$

Classifica-se tais funções pela introdução de um conjunto denotado $L^2(\Omega)$, que consiste de classes de equivalência $[f]$ de funções definidas em Ω, que são quadrado integráveis.

Seja g uma função que difere de f em valor em um número finito de pontos, mas tal que as integrais de f^2 e g^2 em Ω_s resultam em valores numéricos idênticos para qualquer subdomínio. Estas funções são ditas equivalentes em termos dessa propriedade e pertencem à mesma classe de equivalência $[f]$ (ou $[g]$) em $L^2(\Omega)$ (ver esquema na FIGURA 3.1).

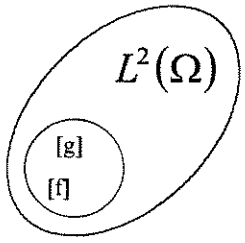

FIGURA 3.1- Funções equivalentes dentro do espaço $L^2(\Omega)$.

Em outras palavras, se duas funções f e g pertencem à mesma classe de equivalência, então:

$$\int_{\Omega} f^2 dx = \int_{\Omega} g^2 dx$$

e

$$\left(\int_{\Omega} |f - g|^2 dx \right)^{1/2} = 0$$

3.4. O ESPAÇO $H^m(\Omega)$

Uma função u pertence à classe $H^m(\Omega)$ se u e todas as suas derivadas parciais de ordem menor ou igual a m, sendo m um inteiro não negativo, são membros da classe $L^2(\Omega)$. Como definição, pode-se usar a seguinte notação:

$$H^m(\Omega) = \left\{ u | u, \frac{\partial u}{\partial x}, \ldots, \frac{\partial^m u}{\partial z^m} \in L^2 \right\}$$

Observa-se que as classes $H^m(\Omega)$ são uma maneira mais generalizada e natural de se quantificar a suavidade ou a regularidade de funções do que as classes $C^m(\Omega)$. A classe $H^m(\Omega)$, assim como a classe $C^m(\Omega)$, também é um espaço linear de funções, ou seja, se $u \in H^m(\Omega)$ e $v \in H^m(\Omega)$, então $\alpha u + \beta v \in H^m(\Omega)$, para quaisquer escalares reais α e β.

3.5. FORMULAÇÃO FRACA

Para mostrar como se constrói a forma variacional do problema, ou forma fraca, será usada como exemplo a seguinte equação diferencial, que mais adiante será referida como problema exemplo:

$$-a_{11}\frac{\partial}{\partial x}\left[\frac{\partial u(x,y)}{\partial x}\right]-a_{22}\frac{\partial}{\partial y}\left[\frac{\partial u(x,y)}{\partial x}\right]+a_{00}u(x,y)=f(x,y) \qquad (3.1)$$

sendo: $(x,y)\in\Omega\subset R^2$

Entende-se que o contorno $\partial\Omega$ de Ω é naturalmente dividido em duas partes, $\partial\Omega_1$ e $\partial\Omega_2$, tal que $\partial\Omega=\partial\Omega_1\cup\partial\Omega_2$, no qual dois tipos distintos de condições de contorno são respectivamente aplicados. Em $\partial\Omega_1$, admite-se que condições de contorno homogêneas *essenciais (ou condição de Dirichlet)* sejam impostas na forma:

$u=0$ em $\partial\Omega_1$

Em $\partial\Omega_2$, são impostas condições de contorno *naturais (ou condição de Neumann)* do tipo:

$\partial u/\partial n = g$ em $\partial\Omega_2$

onde n é a direção normal ao contorno.

Observa-se que a solução u dessa equação diferencial tem que ser diferenciável pelo menos duas vezes. O primeiro passo para calcular a forma fraca é calcular o resíduo $r(x, y)$, através de:

$$r(x,y)=-a_{11}\frac{\partial}{\partial x}\left[\frac{\partial u(x,y)}{\partial x}\right]-a_{22}\frac{\partial}{\partial y}\left[\frac{\partial u(x,y)}{\partial y}\right]+a_{00}u(x,y)-f(x,y)=0$$

Esse resíduo é multiplicado por uma função ponderadora v, também chamada de função teste, suficientemente suave, tal que o produto rv seja também integrável. A

função v ainda satisfaz a condição de Dirichlet homogênea com respeito $\partial\Omega_1$. Do anterior, tem-se:

$$\int_{\Omega}\left\{-a_{11}\frac{\partial}{\partial x}\left[\frac{\partial u(x,y)}{\partial x}\right]-a_{22}\frac{\partial}{\partial y}\left[\frac{\partial u(x,y)}{\partial y}\right]\right\}v dxdy+\int_{\Omega}a_{00}u(x,y)v dxdy-\int_{\Omega}f(x,y)v dxdy=0 \tag{3.2}$$

Utilizando-se a regra da derivada de um produto de funções, segue que:

$$\frac{\partial}{\partial x}\left(v\frac{\partial u}{\partial x}\right)=\frac{\partial v}{\partial x}\frac{\partial u}{\partial x}+v\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)$$

ou $$-v\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)=\frac{\partial}{\partial x}\left(v\frac{\partial u}{\partial x}\right)=\frac{\partial v}{\partial x}\frac{\partial u}{\partial x}$$

A substituição desta última expressão em (3.2), e a aplicação do teorema da divergência, leva a:

$$\int_{\Omega}\left\{a_{11}\frac{\partial v}{\partial x}\frac{\partial u}{\partial x}+a_{22}\frac{\partial v}{\partial y}\frac{\partial u}{\partial y}+a_{00}uv\right\}dxdy=\int_{\Omega}f(x,y)v dxdy+\int_{\partial\Omega}\left\{a_{11}\frac{\partial u}{\partial n}v+a_{22}\frac{\partial u}{\partial n}v\right\}dxdy \tag{3.3}$$

Cabe mencionar que aplicar em forma seqüencial a fórmula da derivada de um produto de funções e o teorema da divergência é equivalente a aplicar a integração por partes.

O lado direito da expressão (3.3) também pode ser escrito em termos do operador bilinear simétrico, expresso por:

$$B(u,v)=\int_{\Omega}\left\{a_{11}\frac{\partial v}{\partial x}\frac{\partial u}{\partial x}+a_{22}\frac{\partial v}{\partial y}\frac{\partial u}{\partial y}+a_{o}uv\right\}dxdy \tag{3.4}$$

No contexto de problemas de engenharia, a quantidade $B(u,v)$ é algumas vezes referida como o *trabalho virtual* do problema, tendo em vista sua correspondência com o trabalho virtual da mecânica clássica.

Uma notação mais geral para $B(u,v)$ é a seguinte:

$$B(u,v)=\int_{\Omega}\left(a^{11}_{mm}\frac{\partial^m u}{\partial x_1^m}\frac{\partial^m v}{\partial x_1^m}+a^{11}_{mm-1}\frac{\partial^m u}{\partial x_1^m}\frac{\partial^{m-1} v}{\partial x_1^{m-1}}+\ldots+a^{NN}_{mm}\frac{\partial^m u}{\partial x_N^m}\frac{\partial^m v}{\partial x_N^m}+\ldots+a^{NN}_{oo}uv\right)$$

Nesta última expressão as quantidades a^{ij}_{kl} são funções dadas de posição em Ω. De forma geral, $B(u,v)$ contem todas as possíveis combinações dos produtos das derivadas de u com relação a x_1, x_2, ..., x_N de ordem m ou menor, com as derivadas de v de ordem m ou menor.

Retomando-se a equação diferencial exemplo (3.1) e substituindo-se (3.4) em (3.3) tem-se que:

$$B(u,v)=\int_{\Omega} f(x,y)v\,dxdy+\int_{\partial\Omega}\left\{a_{11}\frac{\partial u}{\partial n}+a_{22}\frac{\partial u}{\partial n}\right\}v\,dxdy \qquad (3.5)$$

Nesta expressão ainda falta especificar um espaço apropriado para as funções v. Se u é uma função admissível[1], então a escolha de $u = v$ também é válida. Isto conduz a que v seja quadrado integrável. Fazendo-se as devidas considerações para a_{22}, a_{11} e a_{00}, pode-se escrever:

$$\int_{\Omega}\left[\left(\frac{\partial v}{\partial x}\right)^2+\left(\frac{\partial v}{\partial y}\right)^2+v^2\right]dxdy \le B(v,v)<\infty$$

Se a primeira derivada é quadrado integrável, então ela existe. Esta condição leva as funções u e v a pertencer à classe H^1.

Se esta última expressão vale para todas as funções teste v suficientemente suaves, e se a solução u também é suficientemente suave, a solução de (3.5) também é a solução de (3.1).

Isto possibilita fazer o enunciado variacional do problema exemplo:

---

[1] O espaço das funções admissíveis abrange as funções que podem constituir uma solução para o problema, e será visto mais adiante.

Encontrar u que pertença a H^1 tal que:

$$B(u,v) = \int_{\Omega} f(x,y)v dxdy + \int_{\partial\Omega} \left\{ a_{11}\frac{\partial u}{\partial n} + a_{22}\frac{\partial u}{\partial n} \right\} v dxdy \quad \forall \; v \in H^1 \tag{3.6}$$

e que cumpra com as condições de contorno:

$u = 0$ em $\partial\Omega_1$

e $\partial u / \partial n = g$ em $\partial\Omega_2$.

(lembre-se que $v=0$ em $\partial\Omega_1$)

O nome formulação fraca vem do fato de que a formulação abaixa a ordem de diferenciação da solução. Segundo a equação diferencial inicial, a solução *u* deveria pertencer ao espaço das funções duas vezes diferenciável, ou seja, *u* deveria pertencer a C^2. Porém, depois de se aplicar a integração por partes, *u* pode pertencer ao espaço de funções que são diferenciáveis apenas uma vez, desde que sejam quadrado integráveis.

3.6. CONDIÇÕES DE CONTORNO ESSENCIAIS E NATURAIS

As *condições essenciais de contorno,* também conhecidas como condições de Dirichlet, envolvem condições sobre as derivadas de ordem 0, 1, ..., *m*-1, e são usadas para definir o espaço *H* de funções admissíveis, como um subespaço de $H^m(\Omega)$. No exemplo, a condição essencial de contorno é a condição $u = 0$ em $\partial\Omega_1$ (m=1).

As *condições naturais de contorno*, também conhecidas como condições de Neumann, envolvem condições sobre as derivadas de ordem *m*, *m* +1, ..., 2*m* -1, e entram na sentença do problema variacional de valor de contorno. No exemplo, é a condição $\partial u / \partial n = g$ em $\partial\Omega_2$.

As *condições de contorno mistas* são o caso mais geral, pois são uma combinação linear das condições de contorno de Neumman e Dirichlet. Para o caso de m=1, tem-se que a condição de contorno mista pode ser escrita como:

$$\alpha \frac{\partial u(x_0, y_0)}{\partial n} + \beta u(x_0, y_0) = \varphi_0$$

3.7. O ESPAÇO DAS FUNÇÕES ADMISSÍVEIS

A estrutura do espaço H das funções admissíveis deve conter as funções que satisfaçam as condições de contorno essenciais e que sejam suaves o suficiente para que o problema variacional (3.6) faça sentido. No caso do problema exemplo conforme já foi analisado anteriormente, o espaço admissível é H^1.

Demonstra-se em ODEN (1979) que para os problemas elípticos de segunda ordem o espaço admissível é um subespaço $H^1(\Omega)$ tal que:

$$H = \left\{ v \in H^1(\Omega) | \, v = 0 \ em \ \partial\Omega_1 \right\}$$

3.8. APROXIMAÇÕES DE GALERKIN

De forma geral, uma aproximação de Galerkin de (3.6) é obtida apresentando-se o problema variacional em um sub-espaço dimensionalmente finito H^h do espaço das funções admissíveis H. Especificamente, procura-se u_h em H^h tal que:

$$B(u_h, v_h) = \int_\Omega f v_h dx + \int_{\partial\Omega_2} g v_h ds \qquad \forall v_h \in H^h \qquad (3.7)$$

O índice h aqui é uma notação para explicar a dependência do espaço de discretização de elementos finitos.

A aproximação de Galerkin u_h da solução u tem a forma:

$$u_h(\vec{x}) = \sum_{j=1}^{N} \alpha_j \phi_j(\vec{x}) \qquad (3.8)$$

onde N é a dimensão de H^h, α_j são constantes desconhecidas e ϕ_i são funções base linearmente independentes, cujas combinações lineares formam H^h . Estas funções também são chamadas de "funções de forma". As condições essenciais sobre $\partial\Omega_1$ são satisfeitas por u_h, uma vez que $H^h \subset H$. Assim, o problema (3.6) é equivalente ao seguinte sistema linear:

$$\sum_{j=1}^{N} K_{ij}\alpha_j = F_i, \qquad i = 1,2,\ldots,N$$

onde K_{ij} e F_i são entradas da matriz de rigidez e vetor de carga, respectivamente:

$$\left.\begin{aligned} K_{ij} &= B(\phi_i, \phi_j) \\ F_i &= \int_{\Omega} f\phi_i dx + \int_{\partial\Omega_2} g\phi_i ds \\ i &\geq 1;\ j \leq N \end{aligned}\right\}$$

3.9. O MÉTODO DOS ELEMENTOS FINITOS

3.9.3. DISCRETIZAÇÃO DO DOMÍNIO

O método dos elementos finitos fornece uma técnica geral e sistemática para a construção das funções base ϕ_i. O domínio Ω é substituído por uma coleção Ω_h de E domínios Ω_e, que são os elementos finitos:

$$\Omega_h = \bigcup_{e=1}^{E} \Omega_e \qquad \Omega_i \cap \Omega_j = \varnothing \quad \forall i, j$$

Em duas dimensões esta discretização é feita de tal forma que cada elemento geométrico Ω_e é de topologia triangular ou quadrilateral, com nós geométricos sobre seus vértices.

3.9.2. TRANSFORMAÇÃO GEOMÉTRICA

No domínio discretizado, aplica-se a formulação variacional a cada elemento. Para o problema exemplo tem-se que:

$$B_e(u_h^e, v_h^e) = \int_{\Omega_e} f v_h^e dx + \int_{\partial\Omega_e} g v_h^e ds \qquad e = 1,2,\ldots E$$

onde: $$B_e(u_h^e, v_h^e) = \int_{\Omega_e} \left\{ a_{11} \frac{\partial u_h^e}{\partial x} \frac{\partial v_h^e}{\partial x} + a_{22} \frac{\partial u_h^e}{\partial y} \frac{\partial v_h^e}{\partial y} + a_{00} uv \right\} dxdy$$

e $$\int_{\partial\Omega_e} g v_h ds = \int_{\partial\Omega_e} \left\{ a_{11} \frac{\partial u_h^e}{\partial n} + a_{22} \frac{\partial u_h^e}{\partial n} \right\} v_h ds$$

O cálculo das integrais no sistema original (x,y) pode ser facilitado pela mudança de variáveis de tal forma que os limites das integrais sejam de fácil aplicação. Esta mudança de variáveis tem a função de relacionar o elemento no sistema (ξ,η) com o elemento no sistema (x,y), segundo um mapeamento T_e, ou seja, a transformação T_e: $\hat{\Omega}_e \rightarrow \Omega_e$ é tal que:

$$T_e(\xi, \eta) = (x, y)$$

O elemento $\hat{\Omega}_e$ é chamado de elemento mestre, e o elemento Ω_e é chamado de elemento deformado. A

FIGURA 3.2 mostra o mapeamento linear para um elemento triangular.

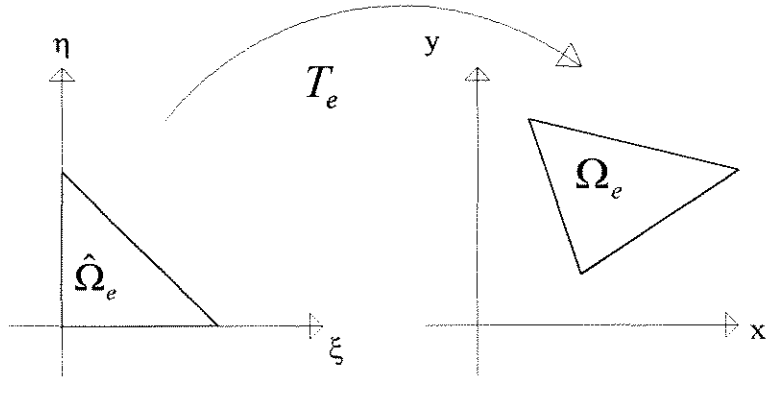

FIGURA 3.2- Mapeamento linear de um elemento triangular.

Qualquer integral sobre o domínio deformado pode ser transformada em uma integral sobre o elemento mestre. Utilizando a seguinte regra de transformação:

$$\int_{\Omega_e} g(x,y)\,dx\,dy = \int_{\Omega_e} g(x(\xi\eta), y(\xi\eta))\det(J)\,d\xi\,d\eta$$

onde:

$J = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial x}{\partial \eta} \\ \dfrac{\partial y}{\partial \xi} & \dfrac{\partial y}{\partial \eta} \end{bmatrix}$ é a matriz jacobiana da transformação.

e *det(J)* é o determinante da matriz jacobiana.

Depois da mudança de variáveis, o operador bilinear tem a seguinte forma:

$$B_e(u_h^e, v_h^e) = \int_{\Omega_e} \left\{ a_{11} \frac{\partial u(\xi,\eta)_h^e}{\partial x} \frac{\partial v(\xi,\eta)_h^e}{\partial x} + a_{22} \frac{\partial u(\xi,\eta)_h^e}{\partial y} \frac{\partial v(\xi,\eta)_h^e}{\partial y} + a_{00} u(\xi,\eta) v(\xi,\eta) \right\} \det(J)\,d\eta\,d\xi \tag{3.9}$$

O mapeamento T_e deve ter como propriedade que cada ponto do elemento mestre seja o mapeamento de um único ponto do elemento deformado. Esta propriedade implica que o jacobiano da transformação seja diferente de zero, o que é equivalente a dizer que a transformação inversa existe.

A construção desses mapeamentos é baseada em polinômios completos de grau *p*. DHATT e TOUZOT (1982) explicam amplamente este tipo de construção.

3.9.3. INTERPOLAÇÃO

Depois da transformação geométrica é mais conveniente aproximar *u* no elemento mestre, de tal forma que:

$$u_h|_{\Omega_e}(x,y) = u_h(\xi(x,y), \eta(x,y)) = \sum_{j=1}^{N_e} u_j \psi_j(\xi,\eta)$$

uma vez que a transformação T_e: $\hat{\Omega}_e \rightarrow \Omega_e$ é tal que:

$$T_e(\xi,\eta) = (x,y)$$

Com a reformulação do problema variacional (3.9) com $v^e_h = 0$ e $u_h(\xi, \eta)$, para cada elemento pode-se escrever:

$$B_e(\psi_i, \psi_j)_{\hat{\Omega}_e} u_j = \int_{\hat{\Omega}_e} f\psi_i \, det(J)dx + \int_{\partial\hat{\Omega}_e} g\psi_i \, det(J_1)ds, \qquad (3.10)$$

ou:

$$\sum_{j=1}^{N_e} k_{ij} u_j^e = f_i^e - \sigma_i^e, \qquad (3.11)$$

onde:

$$k_{ij}^e = \int_{\hat{\Omega}} \left\{ a_{11} \frac{\partial \psi_i}{\partial x} \frac{\partial \psi_j}{\partial x} + a_{22} \frac{\partial \psi_i}{\partial y} \frac{\partial \psi_j}{\partial y} + a_{00} \psi_i \psi_j \right\} \det(J) d\xi \, d\eta$$

$$f_i^e = \int_{\hat{\Omega}} f(\xi, \eta) \psi_i \det(J) d\xi \, d\eta$$

$$\sigma_i^e = \int_{\partial\hat{\Omega}} \left\{ a_{11} \frac{\partial u}{\partial x} n_x + a_{22} \frac{\partial u}{\partial y} n_y \right\} \psi_i \det(J) ds$$

k^e_{ij} é conhecida como matriz de rigidez local, f^e_i é chamado de vetor local de forças e σ^e_j é o "vetor de fluxo"que representa a contribuição as condições naturais no contorno do elemento Ω_e devido a $a_{11} \frac{\partial u}{\partial x} n_x + a_{22} \frac{\partial u}{\partial y} n_y$.

Renumerando-se os índices na equação (3.11), que correspondem à numeração "local a cada elemento", de tal forma a corresponder ao esquema de numeração usado para identificar os nós u_i da malha inteira dos elementos finitos, pode-se formar o sistema global de equações:

$$\sum_{j=1}^{E} k_{ij}^e u_j = \sum_{j=1}^{E} (f_i^e + \sigma_i^e) \qquad i = 1,2,\ldots,E$$

Este sistema, dependendo do tipo da equação diferencial, pode resultar numa matriz de rigidez simétrica ou não. No caso do problema exemplo, a matriz de rigidez é simétrica. Como exemplo de equação diferencial que resulta em uma matriz de rigidez não simétrica tem-se a equação $u'' + u' = f$. Na construção da formulação fraca, o termo u' não vai ser "absorvido" pela integração por partes. Isso pode ser visto na expressão seguinte, onde $k_{ij}^e \neq k_{ji}^e$:

$$k_{ij}^e = \int_{\hat{\Omega}} \left\{ \frac{\partial \psi_i}{\partial x} \frac{\partial \psi_j}{\partial x} + \frac{\partial \psi_i}{\partial x} \psi_j \right\} \det(J) d\xi d\eta \qquad (3.12)$$

Nota-se que a transformação geométrica e a interpolação não necessariamente usam as mesmos polinômios. Dependendo dessa relação define-se os elementos iso-paramétricos, pseudo-paramétricos e sub-paramétricos.

Elementos iso-paramétricos: São chamados assim os elementos que usam polinômios de transformação idênticos aos polinômios de interpolação.

Elementos pseudo-paramétricos: São aqueles elementos em que os polinômios de transformação são diferentes dos polinômios de interpolação, mas que usam os mesmos monômios na construção destes, ou seja os polinômios são do mesmo grau.

Elementos sub-paramétricos: São os elementos que usam polinômios de transformação de grau menor que os polinômios de interpolação. Existem também os elementos super-paramétricos que são o caso oposto.

3.10. ADAPTATIVIDADE

Embora na literatura sejam encontrados estimadores de erro *a priori*, na prática é muito difícil, e alguma vezes impossível, julgar antecipadamente a precisão da aproximação, para assim construir uma malha apropriada. Existe a necessidade de que

programas de elementos finitos sejam capazes de controlar o erro de aproximação por um procedimento de retroalimentação. Isso pode ser feito num ciclo de análise, que se inicia com o cálculo do erro da aproximação. Como segundo passo, o espaço de aproximação é estendido de acordo com a distribuição do erro, e por último o problema é re-analisado, iniciando-se um novo ciclo se for necessário.

Uma medida para o erro usada com freqüência é a norma da energia. O método dos elementos finitos calcula uma função aproximada u_h, que pertence ao espaço das funções admissíveis. Demonstra-se que a função u_h é aquela que minimiza a norma da energia do erro, ou seja:

$$\|u - u_h\|_{E(\Omega)} = \underbrace{min}_{\tilde{u} \in H} \|u - \tilde{u}\|_{E(\Omega)}$$

Uma demonstração dessa relação pode ser encontrada em SZABÓ e BABUSKA (1991). A norma da energia do erro é definida por $\|u\|_E = B(u,u)^{1/2}$.

Por exemplo, no caso da equação $\nabla[K\nabla u(x,y)] - bu(x,y) = f(x,y)$, o erro da norma da energia é dado por:

$$\|u\|_E = \left(\int_\Omega K(\nabla u)^2 + bu^2 d\Omega \right)^{1/2} = B(u,u)^{1/2} \qquad (3.13)$$

3.13. VERSÃO h DOS ELEMENTOS FINITOS

Um aspecto inerente a todo sistema adaptativo é a sua capacidade de reanálise. Em termos de aproximação de elementos finitos, esta pode ser vista como uma extensão do espaço de aproximação que pode ser implementada de várias maneiras. Uma dessas maneiras é a versão h , em que a malha de elementos finitos é refinada localmente. A técnica mais comum é partir de uma malha grossa e refinar os elementos localmente.

As malhas geradas a partir do refinamento h são propícias à utilização da técnica *multigrid*, em que se parte de uma malha fina para malhas cada vez mais grossas

aplicando-se algum procedimento iterativo, até chegar a uma malha bem grosseira. Este processo é conhecido como restrição. Com a aproximação encontrada na malha grossa se faz o caminho inverso até se chegar à malha original. Este processo é conhecido como interpolação.

Na literatura existe a sugestão para se reconstruir a malha a cada refinamento ao invés de refinar os elementos localmente (RANK, 1993). Este método é chamado de triangulação. Também tem-se versão *r* que consiste no reposicionamento dos nós.

3.12. A VERSÃO p

Quando se trata da versão *h*, implicitamente é assumido que o grau de interpolação seja o mesmo para toda a malha. Usualmente, aproximações de ordem mais baixa são usadas, com o grau do polinômio igual a p=1 ou p=2. Uma forma totalmente diferente de se alcançar a adaptatividade em aproximações de elementos finitos é usar para todos os passos da análise a mesma malha de elementos finitos, mas aumentar (localmente ou globalmente) a ordem das funções de forma.

Segundo SZABÓ e BABUSKA (1991), funções de base hierárquicas para a versão tipo *p* foram construídos primeiramente por PEANO em 1975. Ainda na mesma referência encontra-se uma descrição detalhada de como se constroem bases hierárquicas para o caso bidimensional e tridimensional.

Para o caso unidimensional, uma família inteira de funções de forma é obtida pela definição de uma seqüência de polinômios P_i com grau i tal que a função seja nula no contorno de cada elemento. Essa última condição pode ser expressa por:

$$\left\{P_i(\varsigma) | i \geq 2, \varsigma \in [-1,1], P_i(-1) = P_i(1) = 0\right\}$$

Uma possível escolha para P_i é dada pela integral de polinômios de Legendre.

$$P_i(\xi) = \int_{t=-1}^{\xi} \frac{1}{2^{n-1}(n-1)!} \frac{d^n\left(x^2-1\right)^n}{dx^n} dt \qquad (3.14)$$

Na FIGURA 3.3 mostra-se a base hierárquica gerada pela expressão anterior.

A FIGURA 3.4 mostra a base padrão. Obsevando-se as FIGURAS 3.3 e 3.4, percebe-se que a diferença entre as bases hierárquicas e as bases padrão é que no caso das bases hierárquicas todas as funções de forma de ordem mais baixa são mantidas na base de ordem mais alta, o que não acontece no caso das bases padrão, em que a base de ordem p só tem funções de ordem p.

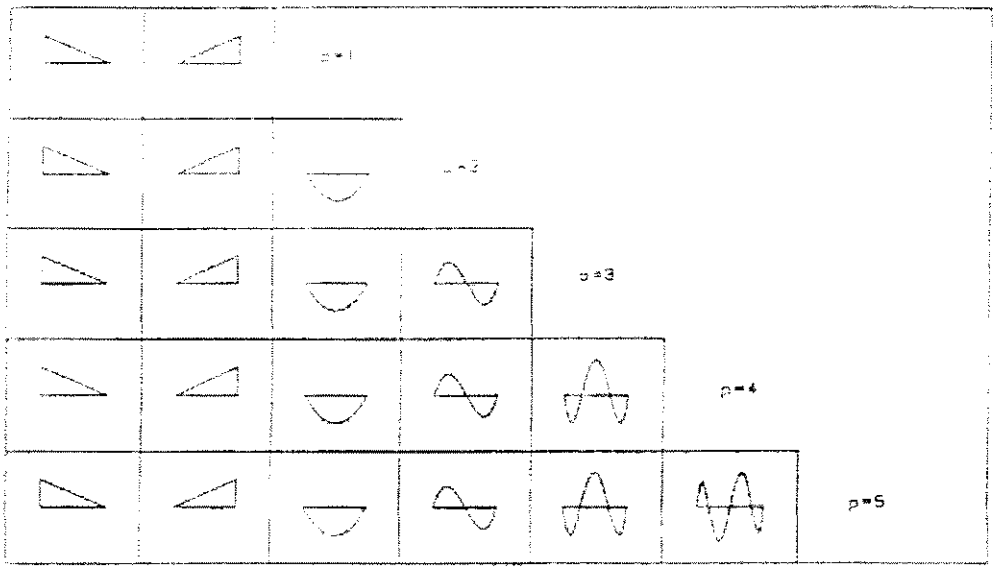

FIGURA 3.3- Funções de base hierárquicas em uma dimensão.

FONTE: RANK, 1993, p. 5.

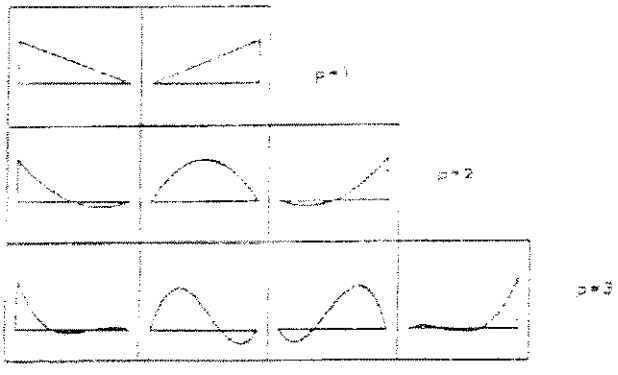

FIGURA 3.4- Funções de base padrão em uma dimensão.

FONTE: RANK, 1993, p. 5.

As bases hierárquicas têm uma conseqüência imediata sobre a estrutura da matriz de rigidez do elemento. Se as equações são ordenadas de tal forma que todas as formas lineares abrangem do numero 0 até n_1, e as formas quadráticas abrangem de n_1+1 até n_2, da mesma forma para a função cúbica e assim por diante, as matrizes de rigidez terão a forma mostradas na FIGURA 3.5.

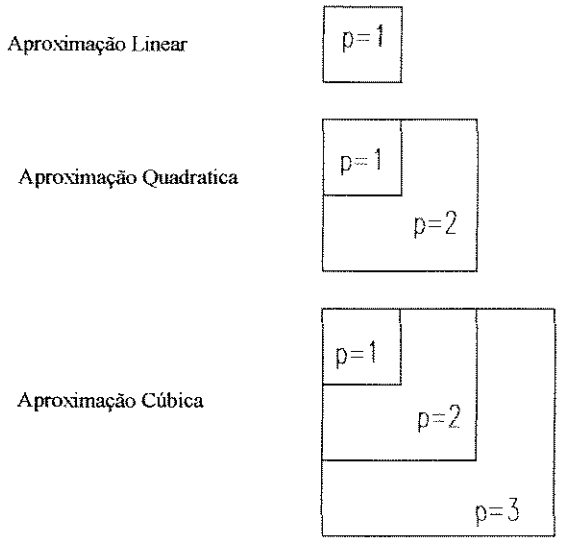

FIGURA 3.5- A matriz de rigidez para bases hierárquicas.

4. O AMBIENTE PZ

4.1. VISÃO GERAL

Existem muitos programas de elementos finitos em diferentes campos de aplicação, tais como Engenharia Mecânica, Engenharia de Petróleo, Engenharia Civil, Engenharia de Materiais, Aeronáutica, Biomecânica, etc. Todos eles implementam dentro de seus códigos procedimentos que descrevem o processo de aproximação por elementos finitos, porém, não abstraem os conceitos da formulação matemática dessa aproximação. Esses conceitos são três:

1. definição do domínio;
2. o espaço das funções de interpolação;
3. definição da equação diferencial que governa o problema e suas condições de contorno associadas.

A maioria dos programas implementa essa formulação usando os seguintes "blocos estruturados":

- Discretização do domínio: o domínio matemático é discretizado por uma malha de elementos gerando uma lista de nós e elementos. Também nesta parte são incluídos os coeficientes da equação diferencial e é definido o esquema de interpolação, ficando caracterizado o tipo de problema e o espaço de aproximação.
- Identificação das condições de contorno: as condições de contorno são identificadas e aplicadas aos nós.
- Cálculo: é feita a computação das matrizes locais, montagem do vetor de forças (*right hand side*) e aplicação das condições de contorno do elemento.
- Construção do sistema e solução: onde é montada a matriz global, o sistema a ser resolvido, e são aplicadas as condições de contorno. Também é escolhido o tipo de método que será empregado para solucionar o sistema (decomposição LU, LDL^t ou Cholesky).

Estes "blocos-estruturados" na maior parte dos *softwares* são implementados utilizando a linguagem FORTRAN. Esta filosofia de programação estruturada dificulta a manutenção de projetos de *software* de grande porte, ou seja, quanto mais geral, mais complexo se torna o programa. Toma-se como base de comparação o FORTRAN porque a maioria dos códigos da computação científica é escrita nessa linguagem, embora nos *softwares* de escopo comercial a programação orientada a objetos (especialmente a linguagem C++) esteja sendo bastante utilizada.

O ambiente PZ foi desenvolvido por DEVLOO (1993) usando o paradigma da programação orientada a objetos e permite atingir um alto nível de abstração no desenvolvimento de programas de elementos finitos. Central nesta filosofia de programação está o conceito de classe, que une dados e operações sobre eles mesmos, permitindo a criação de novos tipos de dados.

Este ambiente está edificado sobre um conjunto de classes-base virtuais, que definem tipos que representam os três conceitos, mencionados anteriormente, de uma aproximação por elementos finitos. Nesses tipos estão abstraídos a malha, o elemento, o nó e o material, e a estes estão associadas operações de transformação de sistemas de coordenadas, integração numérica, construção de sistema de equações, etc. Todos esses objetos são armazenados em árvores binárias e não em listas, uma vez que a busca numa árvore binária é mais rápida que a busca em listas com grande quantidade de dados (LEA, 1992). Cada um desses objetos tem identificadores que são chamados de Ids, e que não necessariamente formam uma seqüência numérica.

O ambiente PZ fornece uma estrutura de classes orientada para simular problemas de mecânica computacional e incorporar a maioria de algoritmos de elementos finitos (inclusive adaptatividade e paralelismo), sem tornar o programa necessariamente ineficiente. Este conjunto de classes abstrai de uma maneira eficiente a formulação dos elementos finitos. Atualmente, o ambiente contém elementos uni-dimensionais e bi-dimensionais triangulares e quadriláteros. As interpolações são de ordem arbitrária, utilizando-se funções hierárquicas de interpolação.

Os modelos implementados são pórticos bi-dimensionais, treliças tri-dimensionais, problema de Poisson, elasticidade bi-dimensional, placas, membranas e escoamento em meios porosos com interação entre a pressão nos poros e deformação elástica da rocha. Qualquer outro sistema de equações bi-dimensional linear pode ser facilmente incorporado.

Neste trabalho serão descritas de forma breve as classes principais do ambiente, pois a descrição de todas as classes de forma detalhada seria muito extensa. Além disso, o ambiente PZ utiliza, além das classes implementadas nele mesmo, muitas classes do GNU C++ Library (LEA, 1992) e da biblioteca Templates (BARRET, BERRY, CHAN et al., 1994). Nessa descrição serão encontrados muitos objetos dessas bibliotecas[1]. Para maiores detalhes sugere-se consultar as respectivas referências.

4.2. ESTRUTURA DE CLASSES DO AMBIENTE PZ

Um ambiente orientado a objetos apresenta-se ao usuário como um conjunto de classes. Este conjunto de classes deve seguir uma filosofia de resolução de problemas tal que o usuário, familiarizado com o ambiente, procure a classe que resolva um dado problema. Na determinação da aproximação de uma equação diferencial existem três passos:

- a definição da geometria do domínio;

[1] Por exemplo, vetor de inteiros (**TIntVec**), vetores de dupla precisão (**TDoubleVec**), vetor de ponteiros (**VoidPtrVec**), mapeamento de ponteiros (**TVoidPtrMap**), etc.

- o espaço de interpolação;
- a definição dos coeficientes da equação diferencial e/ou condições de contorno.

Na filosofia da programação orientada a objetos, as classes são desenvolvidas para abstrair a definição e o comportamento dos objetos que formam um problema. Seguindo essa filosofia, o ambiente PZ contem três principais conjuntos de classes com finalidades distintas:

- classes para aproximação da geometria do problema;
- classes para definição do espaço de interpolação;
- classes para definição da(s) equação(ões) diferencial(ais) e as condições de contorno.

Dentro desse ambiente, existe uma classe chamada de **TAnalysis**, que monta o sistema global de equações, calcula a largura da banda, etc., interagindo esses três tipos de classes. As classes são chamadas a partir de um programa principal, constituindo as partes de discretização, modelagem computacional e definição de condições de contorno.

Um mesmo domínio pode ser discretizado por elementos quadrilaterais ou triangulares, mas segundo o material e a equação diferencial, o comportamento do modelo é diferente. Este ambiente permite que sejam usados diferentes materiais e graus de interpolação, para um mesmo domínio. Isto permite uma variedade de análises muito rica, sem precisar desenvolver versões particulares para cada tipo de modelo.

A FIGURA 4-1 mostra esquematicamente a seqüência de chamadas dos conjuntos de classes do ambiente PZ.

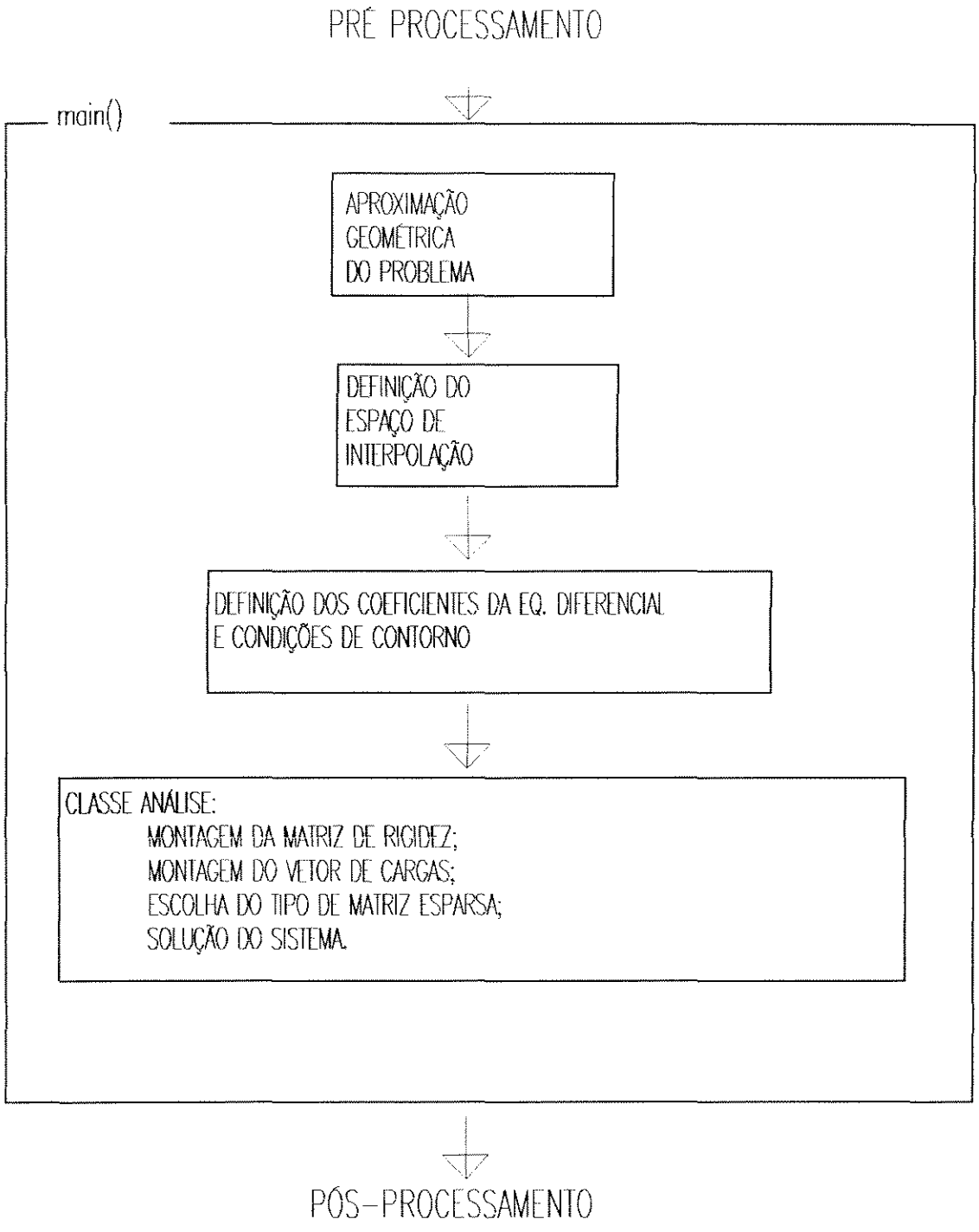

FIGURA 4-1- Esquema mostrando a sequência de chamadas as classes do ambiente PZ.

4.2.1. PRÉ-PROCESSAMENTO E PÓS-PROCESSAMENTO

No que se refere a pré-processamento, o ambiente PZ tem uma classe **TModulef** que lê arquivos gerados pelo programa MODULEF[2]. Existe ainda a classe **TGenGrid,** que gera malhas retangulares. Estas duas classes trabalham com elementos triangulares e retangulares.

Para pós-processamento, existem as classes **TDXGrafGrid**, **TMVGrafGrid** e **TV3DGrafGrid,** que criam arquivos de dados que podem ser interpretados pelos programas DATA-EXPLORER, MVIEW e VIEW3D[3] respectivamente. Para maiores informações do DATA-EXPLORER ver IBM CORPORATION (1994) e para o MVIEW ver CENPES-PETROBRÁS/PUC-RIO (1995).

4.2.2. APROXIMAÇÃO DA GEOMETRIA DO PROBLEMA

O domínio da equação diferencial é discretizado por elementos finitos. Cada elemento define um espaço parametrizado uni-, bi- ou tri-dimensional. Esta parametrização é expressa por um mapeamento entre a configuração no domínio (configuração deformada) e um *elemento mestre*. Este mapeamento é definido por funções de interpolação lagrangeanas lineares e quadráticas. A classe principal da aproximação da geometria é a classe **TGeoGrid** (malha geométrica). Esta classe contém árvores binárias de ponteiros que apontam para objetos do tipo **TCosys** (sistema de coordenadas), **TGeoNod** (nó geométrico) e **TGeoEl** (elemento geométrico).

4.2.2.1. TGeoGrid

A classe **TGeoGrid** é um conjunto de elementos e nós geométricos que define a geometria do domínio da equação diferencial. Nota-se que, caso a malha seja refinada de modo adaptativo, o elemento mãe e seus sub-elementos estão incluídos na árvore binária dos elementos (ver FIGURA 4-2).

---

[2] Modulef é uma biblioteca de programas para elementos finitos, ver SIMULOG (1995).

[3] Este é um programa de visualização desenvolvido por RIBEIRO [1996]

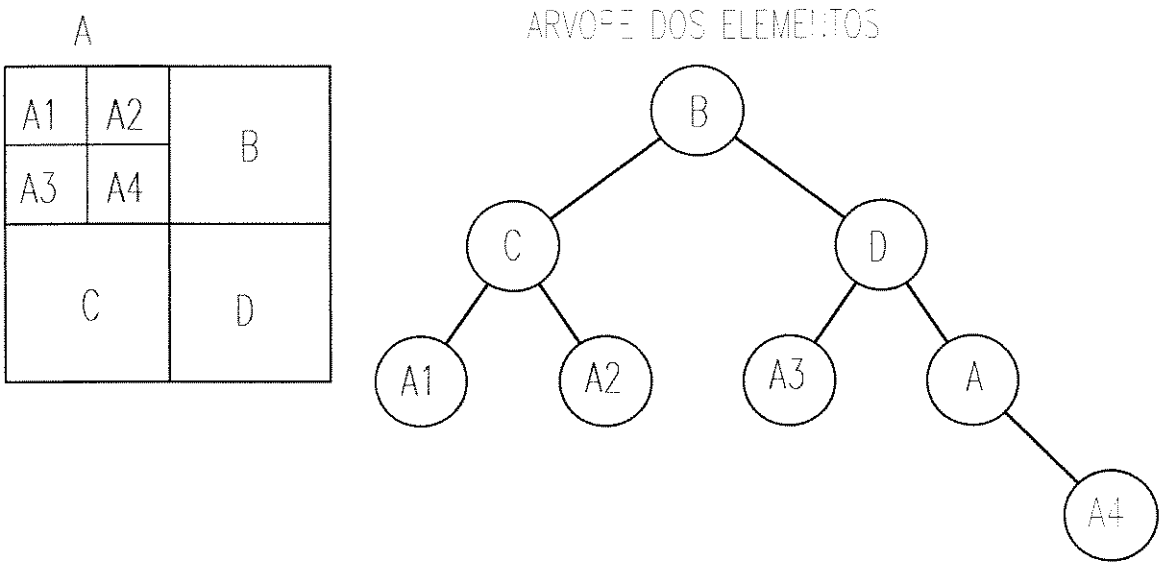

FIGURA 4-2- Árvore binária dos elementos de uma malha.

Esta classe tem as seguintes características:

- proporciona ao usuário a árvore binária de elementos geométricos e a de nós geométricos;
- proporciona a árvore binária dos nós do contorno do domínio e a árvore binária dos elementos do contorno do domínio;
- inclui método para cálculo da vizinhança dos elementos.

Supõe-se que várias aproximações possam ser feitas com base em uma única malha geométrica. Este seria o caso de algoritmos adaptativos em que a seqüência de malhas são armazenadas separadamente. Em seguida é apresentada uma descrição das variáveis membro e das funções membro desta classe.

VARIÁVEIS MEMBRO

char *fChecked*: É verdadeiro se todos os nós estiverem definidos; esta variável é inicializada somente após a função Check(ostream & err = cerr) ter sido chamada;

char *fName[63]*: Nesta variável é guardado o nome da malha geométrica;

int *fNDim*: Guarda a dimensão da topologia do problema;

TVoidPtrMap *fElementMap*: Árvore binária de ponteiros para os elementos do **TGeoGrid**;

TVoidPtrMap *fNodeMap*: Árvore binária de ponteiros para os nós do **TGeoGrid**;

TVoidPtrMap *fElBndCondMap*: Árvore binária de ponteiros para os elementos com condições de contorno;

TVoidPtrMap *fNodBndCondMap*:Árvore binária de ponteiros para os nós com condições de contorno.

MÉTODOS DO TIPO PÚBLICO

static TGeoGrid **gCurrent*: Ponteiro para um objeto do tipo **TGeoGrid**; é usado para apontar a malha que está sendo construída. Na forma "natural" de um objeto **TGeoGrid**, este ponteiro tem valor nulo;

TGeoGrid(int nd = 3): Construtor da classe, tem como parâmetro a dimensão da topologia da malha; a dimensão padrão é 3. Todas as listas são inicializadas com dimensão zero; todos os ponteiros apontam para NULL, e somente a variável *fNDim* é inicializada com nd;

TGeoGrid (TGeoGrid & gr): Construtor de cópia, ou seja, constrói o objeto corrente com todas as características da malha gr;

void CleanUp(): Apaga todas as alocações e referências contidas na malha.

MÉTODOS PARA ACESSO ÀS VARIÁVEIS MEMBRO

int NumNodes(): Retorna o número de nós que a malha possui;

int NumElem(): Retorna o número de elementos que a malha possui;

char IsChecked(): É verdadeiro se a malha geométrica tiver sido verificada;

void SetName(char *nm): É alterada a variável membro *fName* (nome da malha) para um novo nome, dado através de nm;

char* Name(): Retorna o nome do objeto corrente;

TVoidPtrMap &ElementMap(): Retorna uma árvore binária de ponteiros para os elementos do objeto tipo **TGeoGrid**;

TVoidPtrMap &NodeMap(): Retorna uma árvore binária de ponteiros para os nós do objeto tipo **TGeoGrid**;

TVoidPtrMap &ElementBcMap(): Retorna uma árvore binária de ponteiros que apontam para os elementos que têm condições de contorno associadas;

TVoidPtrMap &NodeBcMap(): Retorna uma árvore binária de ponteiros que apontam para os nós que tem condições de contorno associadas.

FUNÇÕES UTILITÁRIAS

void ResetReference(): Anula a referência da classe dos elementos e nós. Diferentes malhas computacionais (**TCompGrid**) podem ser criadas com base numa malha geométrica (**TGeoGrid**), porém somente um **TCompGrid** está sendo apontado por um **TGeoGrid**; se esta função é invocada, vai anular as referências dos elementos e nós, ou seja, a malha geométrica não aponta para nenhuma malha computacional;

char Check(ostream & err = cerr): Verifica se todos os nós estão definidos;

void Print(ostream & out = cout): Imprime na saída out todas as características do **TGeoGrid**, como nome , nós, elementos, etc;

TGeoNod* FindNode(Long nid): Retorna um ponteiro para um objeto do tipo **TGeoNod**; este objeto representa o nó identificado com *Id*=nid. Retorna NULL caso o nó não exista;

TGeoNod* FindNode(DoubleVec &co): Retorna um ponteiro para um objeto do tipo **TGeoNod**; este objeto representa o nó mais próximo da coordenada dada pelo vetor co do tipo DoubleVec (esta é uma classe da biblioteca GNU). Retorna NULL caso o nó não exista;

TGeoEl *FindElem(Long elid): Retorna um ponteiro para um objeto do tipo **TGeoEl**, este objeto representa o elemento identificado com *Id*=nid. Retorna NULL caso o elemento não exista;

TCosys *FindCosys(Long cosysid): Retorna um ponteiro para um objeto do tipo

TCosys; este objeto representa o sistema de coordenadas identificado com *Id*=cosyid. Retorna NULL caso o sistema não exista;

void BuildConnectivity(): Constrói a conectividade dos elementos dentro da malha.

void Swap(TGeoNod *np1, TGeoNod *np2): substitui o nó apontado por np1 pelo nó apontado por np2;

void GetNodePtr(int nno, LongAVec& nos, VoidPtrVec &nodep): Copia os primeiros nno ponteiros identificados pelos valores contidos em nos para o vetor de ponteiros nodep;

void GetBoundaryElements(int NodFrom, int NodTo, VoidPtrVec &ElementVec, TIntVec &Sides): Retorna no vetor de ponteiros ElementVec os ponteiros que estão apontando para os elementos que estão no contorno do domínio, localizados entre os nós NodFrom e NodTo, contados no sentido anti-horário. Este método usa a conectividade dos elementos. O método BuildConnectivity inicializa a informação de conectividade dentro da malha.

void BoundBox(TMatrix &bbox): Faz uma caixa em volta do domínio geométrico, conforme esquematizado na FIGURA 4-3; isso é feito encontrando-se as coordenadas máximas e mínimas do domínio.

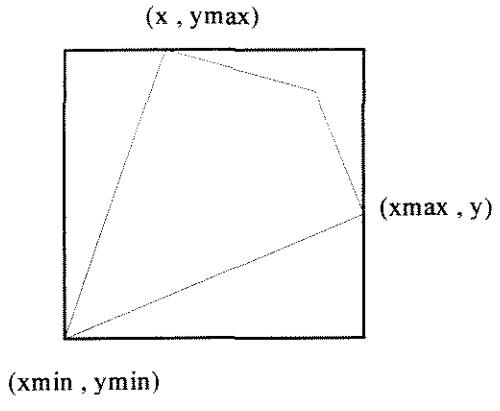

FIGURA 4-3- Limites definidos pela função BoundBox.

APRIMORAMENTOS

No que se refere ao método **Check()**, além de verificar se todos os elementos estão definidos na malha, ele poderia verificar se todas as condições de contorno estão associadas a nós e elementos, e se todos os nós definidos nos elementos estão dentro do objeto **TGeoGrid**.

4.2.2.2. TGeoEl

Esta classe define o mapeamento entre o elemento deformado e o elemento mestre. **TGeoEl** contém dados para identificar o vizinho do elemento, identificar o número de referência da condição de contorno e identificar o número de referência do material. Seus principais métodos são:

- método para cálculo do Jacobiano do mapeamento;
- método para dividir o elemento em sub-elementos;
- método para criação de um elemento computacional baseado no elemento geométrico atual.

São derivadas da classe **TGeoEl** as classes **TGeoEl1d**, **TGeoElQ2d** e **TGeoElT2d,** que implementam respectivamente o elemento geométrico uni-dimensional, o quadrilátero bi-dimensional e o triângulo bi-dimensional (ver FIGURA 4-4).

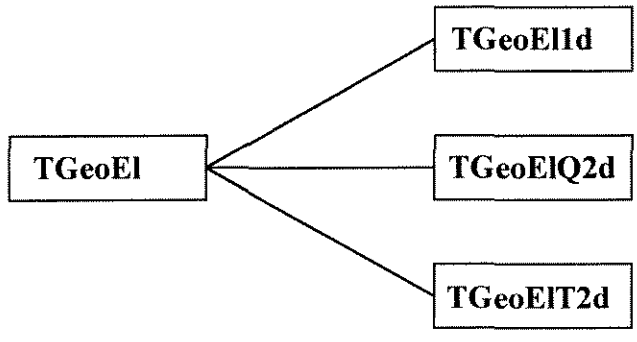

FIGURA 4-4- A classe do elemento geométrico e suas derivadas.

VARIÁVEIS MEMBRO

TGeoGrid *fGrid: Ponteiro para referenciar a malha à qual o objeto corrente pertence;

Long fId: Índice para o número do elemento;

int fNumNodes: Número de nós do elemento;

Long fMatIndex: Índice para tipo de material do elemento;

VoidPtrVec fNodep: Vetor de ponteiros que referência aos nós do elemento;

int fNumSides: Número de arestas do elemento;

VoidPtrVec fConnect: Vetor de ponteiros referenciados aos elementos que estão conectados ao objeto corrente;

LongVec fSide: Lados pelos quais os elementos são conectados; caso o elemento seja de fronteira, esta variável contém o número de referência da condição de contorno;

TCompEl *fReference: Ponteiro para um elemento computacional que foi criado segundo o elemento geométrico corrente;

VoidPtrVec *fSubEl: Vetor de ponteiros para sub-elementos. Os sub-elementos de um elemento não são diferenciados dos outros dentro do **TGeoGrid**; os sub-elementos também entram na lista de elementos do **TGeoGrid**.

MÉTODOS DO TIPO PROTEGIDO

void SetNumNodes(int nn): Inicializa com nn o número de nós do elemento;

void SetNumSides(int ns): Inicializa com ns o número dos lados do elemento;

void DefineConnectivity(LongVec &np): Inicializa o vetor de ponteiros fNodep com os ponteiros contidos em np. Deve-se ter cuidado com o vetor np, pois não existe forma de se verificar se os ponteiros passados em np são realmente ponteiros para objetos do tipo **TGeoNod** (nós geométricos);

void Shape(double x, int num, TFMatrix &phi, TFMatrix &dphi): Calcula as num funções de forma unidimensionais no ponto x. O valor da função é retornada em phi e o valor da derivada em dphi. As funções padrão desta classe base são funções Lagrangeanas lineares e quadráticas em uma dimensão;

Para num = 2:

$$phi = \begin{bmatrix} \frac{1-\xi}{2} \\ \frac{1+\xi}{2} \end{bmatrix} \qquad dphi = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}$$

Para num=3:

$$phi = \begin{bmatrix} \frac{\xi(\xi-1)}{2} \\ 1-\xi^2 \\ \frac{\xi(\xi+1)}{2} \end{bmatrix} \qquad dphi = \begin{bmatrix} \xi - \frac{1}{2} \\ -2\xi \\ \xi + \frac{1}{2} \end{bmatrix}$$

MÉTODOS DO TIPO PÚBLICO

TGeoEl(long Id): Construtor da classe, tem como argumento o Id do elemento. Todas as variáveis são inicializadas com zero e os ponteiros ficam apontando para NULL, ou seja, o elemento existe com *fId* = Id, porém, não tem lados, nem material, etc;

TGeoEl(TGeoEl & el): Constrói o objeto corrente com as mesmas características de el.

MÉTODOS PARA ACESSO ÀS VARIÁVEIS MEMBRO

TGeoGrid *Grid(): Retorna um ponteiro para a malha (**TGeoGrid**) ao qual o elemento pertence;

void SetGrid(TGeoGrid *gr): Deixa a variável *fGrid* apontada para gr; com isso dizemos que o elemento corrente pertence à malha apontada por gr;

Long Id(): Retorna o *id* do elemento;

int NumberOfNodes(): Retorna o número de nós do elemento;

TGeoNod* NodePtr(int i): Retorna um ponteiro do tipo **TGeoNod**. Este ponteiro aponta para o nó com numeração i;

Long MaterialNumber(): Retorna o Id do material, onde o elemento corrente está indexado;

TCompEl *Reference(): Retorna um ponteiro para o elemento computacional, que o elemento corrente está apontando;

virtual TCompEl *CreateCompEl(): Método que cria um elemento computacional baseado no elemento geométrico corrente;

short NumSides(): Retorna o número de lados do elemento;

virtual int NumSideNodes(): Retorna o número de nós por lado. Por exemplo, num elemento retangular, se for bi-linear retorna 2, e se for quadrático retorna 3 (ver FIGURA 4-5);

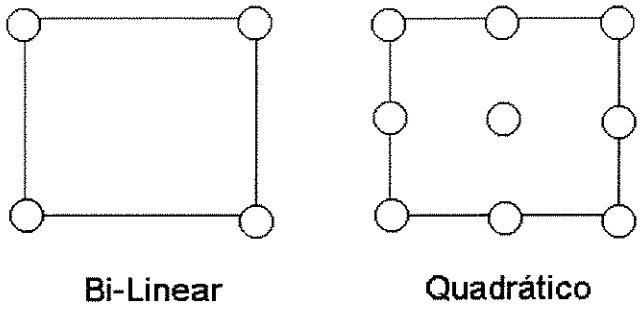

FIGURA 4-5- Número de nós por lado.

virtual TGeoNod *SideNode(int side, int node)=0: Retorna um ponteiro ao nó node do lado side (ver FIGURA 4-6);

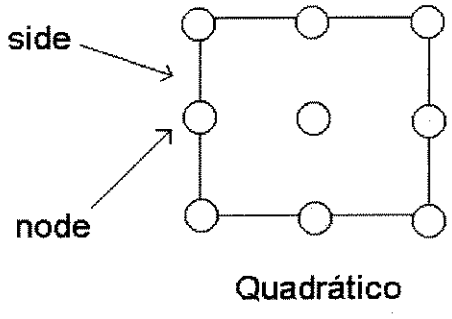

FIGURA 4-6- Acesso ao no *node* do lado *side*.

long SideNodeId(int side, int nodenum): Retorna o Id do nodenum do lado side;

int WhichSide(TLongVec &SideNodeIds): Retorna o lado que contém os nós referenciados por SideNodeIds;

void SetMaterial(Long matind): Retorna o índice do material do elemento;

void SetNodePtr(int i, TGeoNod* np): Define o nó i para ser np;

void SetSide(int i, int siden): Serve para mudar a numeração do lado i para siden; caso siden menor que zero, identifica o número de referência da condição de contorno.

void SetConnectivity(int i, TGeoEl* el): Define o elemento que esta ao longo do lado i igual ao elemento el;

TGeoEl *Neighbour(short is): Retorna um ponteiro para o elemento que é vizinho ao lado is do elemento corrente;

short NeighbourSide(short is): Retorna o número do lado do elemento vizinho que está conectado ao lado is do elemento corrente (ver FIGURA 4-7);

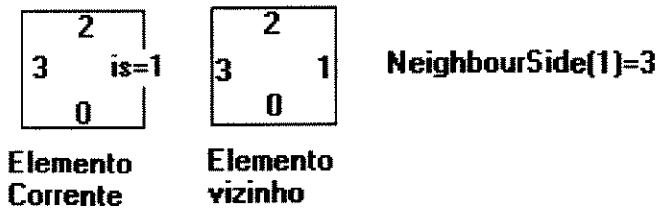

FIGURA 4-7- O lado do elemento vizinho conectado ao lado 1 do elemento corrente é 3.

short Bc(short side): Retorna o número de referência da condição de contorno do lado side.

FUNÇÕES UTILITÁRIAS

void SwapNode(TGeoNod* gp1, TGeoNod* gp2): Substitui o nó apontado pelo gp1 pelo nó apontado pelo gp2. Esta mudança é efetiva somente se o gp1 existe no elemento;

void SwapMaterial(Long mat1, Long mat2): Substitui o índice de material para mat2. Isso somente acontece se o índice *fMatIndex* do elemento corrente é igual a mat1;

void Print(ostream & out = cout): Imprime as características do elemento corrente, como o id do elemento, numero de nós, a numeração dos nós, o índice do material, elementos conectados, etc.

void SetReference(TCompEl *elp): Deixa a variável fReference apontando para elp. Isso quer dizer que o elemento geométrico corrente é o mapeamento do elemento computacional elp.

void ClearReference(): Deixa a variável fReference apontando para NULL, ou seja, depois disto, o elemento corrente não tem elemento computacional recíproco.

friend ostream& operator<<(ostream &s,TGeoEl &el): Sobrecarga do operador <<, usado para imprimir as características do elemento geométrico na saída definida por s.

void Center (double *Cent): Retorna no argumento Cent os valores das coordenadas do centro geométrico do elemento;

virtual void Divide(VoidPtrVec &pv): Divide o elemento e coloca os elementos resultantes dentro do vetor pv;

TGeoEl *SubEl(short is): Retorna um ponteiro ao subelemento is;

virtual void Jacobian(DoubleAVec &coordinate, TFMatrix &jac, TFMatrix &axes): Retorna em jac o cálculo do jacobiano no ponto definido por coordinate; e em axes a direção dos eixos ao qual o jacobiano se refere.

virtual void X(DoubleAVec &coordinate, DoubleAVec &result): Nesta função é transformada a coordenada de um ponto no elemento mestre (coordinate), para o valor correspondente no sistema global (result) (ver FIGURA 4-8).

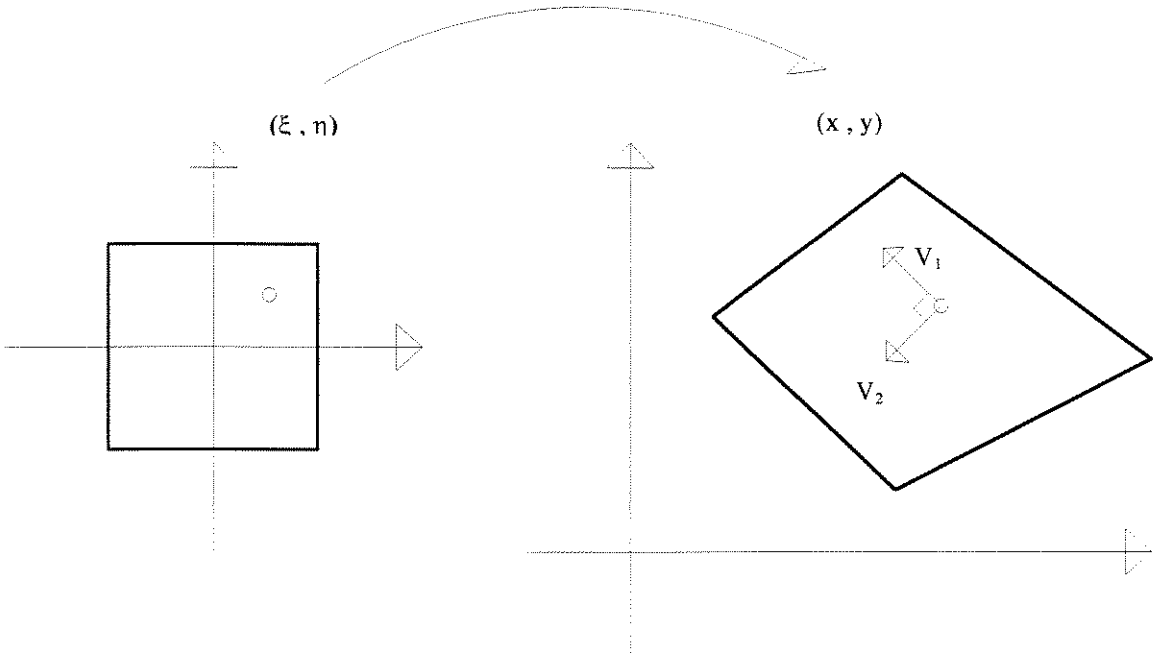

FIGURA 4-8- Mapeamento entre o elemento mestre e o sistema de coordenadas cartesianas.

4.2.2.3. TGeoNod

Esta classe define o nó geométrico, e contém as coordenadas de um nó no espaço euclidiano tri-dimensional além de um ponteiro para um sistema de coordenadas. As coordenadas passadas para construir um objeto deste tipo são coordenadas globais. O construtor tem como valor padrão coordenadas tri-dimensionais. Se forem passadas coordenadas com dimensão maior, o sistema não faz as verificações necessárias.

VARIÁVEIS MEMBRO

char fDefined: É um *flag* que indica se a malha foi verificada ou não. Tem os valores de TRUE ou FALSE;

Long fId: É o Id do nó;

double fCoord[3]: Aqui são armazenadas as coordenadas dos nós. Pode-se observar que esta limitado a três dimensões;

TCosys *fSys: Ponteiro para o sistema de referência;

TDofNod *fDofNod: Ponteiro para um objeto **TDofNod**. Um objeto do tipo **TDofNod** gerencia o grau de liberdade do nó;

MÉTODOS DO TIPO PÚBLICO

TGeoNod(long id, int ndim = 3, double *xp = NULL, TCosys *ref = NULL): Construtor da classe, tem como argumentos o Id do nó, a dimensão do problema passado em ndim e as coordenadas da malha dadas em xp, sendo que estas devem estar definidas em coordenadas globais. Se é passado um ponteiro com valor NULL, o nó fica com a variável fDefined igual a zero, ou seja, o nó existe mas não está inicializado. Como último argumento temos o tipo de sistema de referência adotado (**TCosys**). O argumento ndim tem como valor padrão igual a três, que é o seu valor máximo;

GeoNod(TGeoNod & nn): Construtor de cópia. Constrói o objeto corrente com as caraterísticas de nn;

void Define(int ndim, double *xp = NULL, TCosys *ref = NULL): Podemos mudar as características do objeto corrente; isso somente é possível se o ponteiro xp inicializa o nó;

Long Id(): Retorna o Id do nó;

char IsDefined(): É verdadeiro se o nó estiver definido;

void SetNodeNumber(Long i): Muda o Id do nó para o valor de i;

double Coord(int i): Retorna a cordenada x_i;

void SetCoord(double *x, int dim = 3): Se o nó for definido, as coordenadas são substituídas pelos valores do vetor x até a posição dim. Esta variável pode ter 3 como valor máximo;

void SetCosys(TCosys* csp): Deixa a variável fSys apontando para csp, ou seja, o sistema de referência do nó muda para csp;

void SetReference(TDofNod *dof): Deixa a referência para o correspondente **TDofNod**; este objeto contém a informação dos graus de liberdade de um nó geométrico;

void ResetReference(): Deixa a variável fDofNod apontando para NULL, ou seja, o nó fica sem sistema de coordenadas (perigoso);

TDofNod *Reference(): Retorna o ponteiro armazenado em fDofNod;

void Print(ostream & out = cout): Imprime as características do objeto corrente na saída indicada por out;

4.2.2.4. TCosys

Esta é uma classe base de onde se derivam as classe TCartsys, TCylinsys, TEsfersys, que descrevem coordenadas cartesianas cilíndricas e esféricas respectivamente. Isto deixa à disposição diferentes tipos de sistemas de coordenadas para os sistemas globais e locais (ver FIGURA 4-9).

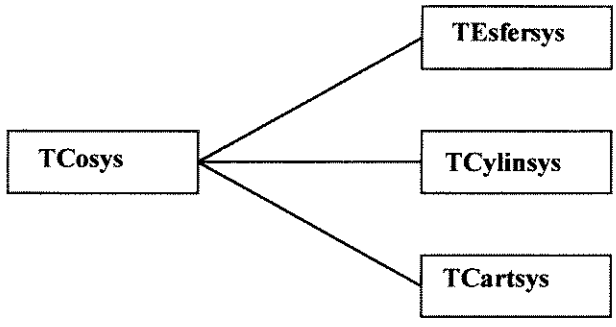

FIGURA 4-9- Sistemas de coordenadas derivados da classe Tcosys.

Nenhum tipo de verificação é feito sobre as referências circulares. Se este tipo de referência existir, o programa entrará num *loop* infinito. Um exemplo seria o caso de um sistema s_0 estar referenciado para o sistema s_1, e s_1 estar referenciado para s_0.

TIPO ENUMERADO

enum gCosysType {cartesian,cylindric,esferic}: Esta variável enumerada serve como *flag* para identificar os sistemas de coordenadas, cartesianas (gCosysType=0), cilíndricas (gCosysType=1) e esféricas (gCosysType=2).

VARIÁVEIS MEMBRO

int fNumber: Índice do sistema de coordenada;

Float fOrigin[3]: Origem do sistema de coordenadas expresso em coordenadas cartesianas;

TCosys *fReference: Ponteiro para o sistema de coordenadas;

Float fTr[3][3]: Armazena os vetores unitários expressos em coordenadas cartesianas. Em cada linha são armazenadas as componentes dos vetores unitários **u**, **v**, **w** respectivamente, segundo a FIGURA 4-10. Estes vetores são relativos ao sistema de coordenadas de referência.

$$fTr = \begin{bmatrix} u1 & u2 & u3 \\ v1 & v2 & v3 \\ w1 & w2 & w3 \end{bmatrix}$$

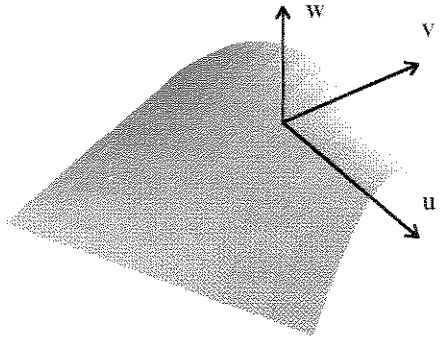

FIGURA 4-10- Vetores unitários de coordenadas curvilíneas na classe TCosys.

MÉTODOS DO TIPO PRIVADO

void Normalise(Float t[3]): Deixa o vetor passado em t3 unitário;

void GetNormal(Float vec1[3], Float vec2[3], Float norm[3]): Encontra a normal (norm) do plano definido pelos vetores vec1 e vec2;

Float Determinant(Float t[3][3]): Calcula o determinante da base definida por t;

virtual void ToCart(Float point[3]): Transforma o ponto point para o sistema cartesiano.

virtual void FromCart(Float point[3]): Transforma o ponto point do sistema cartesiano para o sistema corrente.

MÉTODOS DO TIPO PÚBLICO

TCosys(int num, TCosys* ref = NULL, Float *org = NULL): Este construtor tem como argumentos o Id do objeto (num), a referência do sistema (ref) e a origem do sistema (org);

virtual int Type(): Retorna o tipo do sistema de coordenadas corrente;

void Print(ostream& out = cout): Imprime as características do sistema de coordenadas, como o Id do objeto, tipo de coordenadas, sistema ao qual está referenciado, origem do sistema, etc;

int Id(): Retorna o Id do sistema corrente;

void SetReference(TCosys *co, Float *org = NULL): Muda o sistema de referência para co e a origem para org. Este último parâmetro é expresso no sistema de coordenadas de referência corrente;

void SetOrigin(Float org[3]): Muda a origem do sistema de referência segundo org. Este tem que ser expresso no sistema de referência corrente;

void SetOrigin(FloatAVec& org): Muda a origem de sistema de referência segundo org. Este tem que ser expresso no sistema de referência corrente. É semelhante ao anterior mas com argumento de outro tipo;

void Reset().-Deixa fTr=I, *fReference*=NULL e origem =0;

$$fTr = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

void SetAxes(Float t[3][3]): Substitui os valores dos vetores unitários (fTr) atuais pelos valores de t;

$$fTr = \begin{bmatrix} t00 & t01 & t02 \\ t10 & t11 & t12 \\ t20 & t21 & t22 \end{bmatrix}$$

void SetAxes(Float x[3],Float z[3]): Define o eixo y perpendicular a x e z;

void SetAxes(FloatAVec& x, FloatAVec& z): Define o eixo *y* perpendicular a x e z;

void ToReference(Float point[3]): Transforma point do sistema corrente para o sistema de referência;

void FromReference(Float point[3]): Transforma point do sistema de referência para o sistema corrente;

void ToGlobal(Float point[3]): Transforma point do sistema corrente para o sistema global;

void FromGlobal(Float point[3]): Transforma point do sistema global para o sistema corrente;

void ToReference(FloatAVec& point): Transforma point do sistema corrente para o sistema de referência;

void FromReference(FloatAVec& point): Transforma point do sistema de referência para o sistema corrente;

void ToGlobal(FloatAVec& point): Transforma point do sistema corrente para o sistema global;

void FromGlobal(FloatAVec& point): Transforma point do sistema global para o sistema corrente;

4.2.2.5. TGeoNodBc

A classe **TGeoNodBc** define a condição de contorno aplicada a um nó geométrico. Este objeto é definido como um *struct*, por isso ele é acessível por qualquer método ou função externa. Dentro dessa estrutura está declarado um ponteiro para **TGeoNod**. Deve-se ter cuidado com isso, pois não há forma de se verificar se o objeto nó que **TGeoNodBc** está apontando existe ou foi destruído. Sua implementação é a seguinte:

```
struct TGeoNodBc {
     TGeoNod         *fNod;
     int             fId;
     TGeoNodBc(TGeoNod *nodin, int numberin){
     fNod = nodin;
     fId = numberin;
     }
};
```

4.3. DEFINIÇÃO DO ESPAÇO DE APROXIMAÇÃO

A distinção entre o método de elementos finitos e a aproximação de Rayleigh Ritz é a forma sistemática com que o método de elementos finitos define as funções de interpolação. No ambiente PZ, as funções de interpolação são definidas com base em polinômios ortogonais. Isto significa que as funções de interpolação são efetivamente dissociadas das funções de mapeamento. As funções de interpolação são hierárquicas, o que torna disponível a adaptividade tipo *p*, que permite aumentar o grau de interpolação se, depois de uma análise do erro da energia, este não for considerado satisfatório. Esta variação do grau de interpolação pode resultar em elementos iso-paramétricos, sub-paramétricos ou super-paramétricos.

São chamados de elementos computacionais e nós computacionais os elementos e nós sobre os quais são definidos as funções de interpolação. A cada elemento computacional, associa-se um elemento geométrico. Este calcula o mapeamento entre o elemento deformado e o elemento mestre, e o elemento computacional calcula as funções de forma e suas derivadas nos pontos de integração. Assim o elemento computacional utiliza a informação do elemento geométrico para calcular a matriz de rigidez do elemento. Na FIGURA 4-11 é esquematizada a relação entre a malha computacional e a malha geométrica.

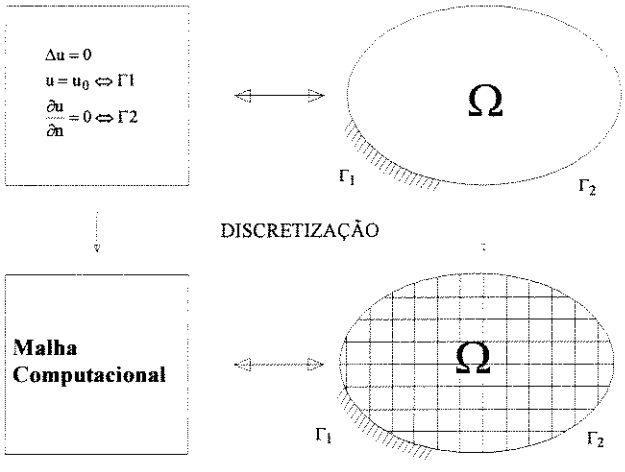

FIGURA 4-11- Discretização do problema usando a malha computacional.

A seguir são descritas as classes base que abstraem o espaço de aproximação.

4.3.1.1.TCompGrid

É um conjunto de elementos e nós computacionais, materiais e condições de contorno. O usuário tem acesso direto às árvores binárias da classe. Ao contrário da malha geométrica, a malha computacional particiona o domínio, ou seja, nesta malha acontece o cálculo discreto dos elementos finitos. Além dos métodos de acesso e auxílio, a malha proporciona métodos para :

- calcular o número de equações do modelo;
- calcular a largura de banda do sistema;
- renumerar os nós seguindo o algoritmo de Cuthill-McKee[4];
- calcular o tamanho dos blocos do sistema de equações (ver definição das classes de objetos tipo TMatrix);
- montar o sistema de equações globais e o vetor de carga;
- carregar um vetor de solução nos graus de liberdade.

No método dos elementos finitos, a geometria do domínio e o cálculo discreto estão relacionados pelo cômputo do jacobiano em cada elemento. Por esta razão toda malha computacional é derivada de uma malha geométrica. Nada impossibilita que possam existir vários níveis de malhas computacionais derivadas de uma única geometria (malha geométrica). Isto torna possível a adaptatividade tipo *h*, ou seja, o refinamento parcial ou total do domínio dependendo do erro da norma de energia. No processo de cálculo, apenas uma malha computacional é a "malha corrente". Esta informação é implementada por um ponteiro definido dentro da classe **TGeoGrid,** que aponta para uma malha computacional (**TCompGrid**), ou seja, este ponteiro indica que nível de malha está sendo calculado. Na FIGURA 4-12 podemos ver vários níveis de refinamento para um objeto **TGeoGrid.**

---

[4] O algoritmo de Cuthill-McKee pode ser encontrado em CUTHILL e MCKEE (1969).

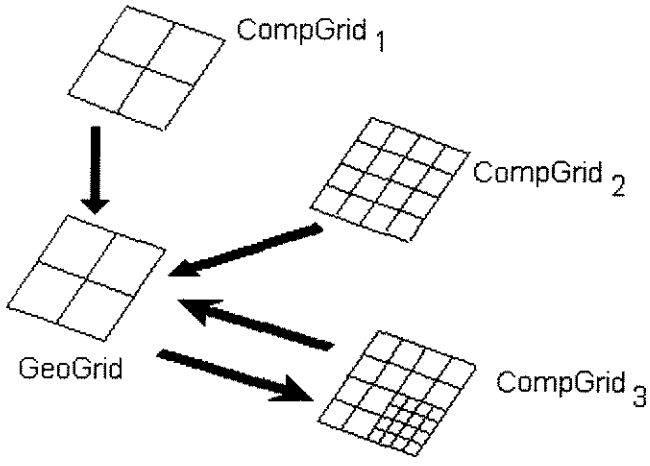

FIGURA 4-12- Reciprocidade entre a malha computacional e malha geométrica.

VARIÁVEIS MEMBRO

TGeoGrid *fReference: Toda malha computacional é construída com base numa malha geométrica; a variável fReference aponta para a malha geométrica correspondente;

char fName[127]: Nome da malha, para identificação do modelo;

char fChecked: É TRUE se a malha já tiver sido verificada;

TVoidPtrMap fElementMap: Árvore binária de ponteiros para elementos;

TVoidPtrMap fNodeMap: Árvore binária de ponteiros para nós;

TVoidPtrMap fMaterialMap: Árvore binária de ponteiros para materiais;

TVoidPtrMap fBndCondMap:Árvore binária de ponteiros para as condições de contorno.

TVoidPtrMap fElBndCondMap: Lista de elementos com condições de contorno associadas;

TVoidPtrMap fNodBndCondMap: Lista de nós com condições de contorno associadas;

MÉTODOS DO TIPO PÚBLICO

static TCompGrid *gCurrent: Variável do tipo global, que aponta para a malha atualmente sendo construída;

TCompGrid(TGeoGrid* gr): Construtor da classe; tem como argumento uma malha geométrica, que é usada como base para construir a malha computacional;

TCompGrid(TCompGrid &gr): Construtor de cópia; serve para construir a malha computacional corrente com base em gr;

void CleanUp(): Limpa todas as estruturas alocadas dinamicamente;

void SetgCurrent(): Deixa a variável gCurrent apontando para si mesma.

MÉTODOS PARA ACESSO ÀS VARIÁVEIS MEMBRO

int NumNodes(): Retorna o número de nós da malha;

int NumElem(): Retorna o número de elementos da malha;

int NumMat(): Retorna a quantidade de elementos da malha;

int NumBc(): Retorna a quantidade de condições de contorno definidas na malha;

int NumElBc(): Retorna o número de elementos que têm condições de contorno associadas;

int NumNodBc(): Retorna o número de nós com condições de contorno associadas;

void SetName(char *nm): Atribui à malha o nome nm;

char* Name(): Retorna o nome da malha.

MÉTODOS PARA ACESSO ÀS ÁRVORES BINÁRIAS

TVoidPtrMap &ElementMap(): Retorna uma referência à árvore binária que contém ponteiros para os elementos da malha computacional;

TVoidPtrMap &NodeMap(): Retorna uma referência à árvore binária que contém ponteiros para os nós da malha computacional;

TVoidPtrMap &ElementBcMap(): Retorna uma referência à árvore binária que contém ponteiros para os elementos com condição de contorno associadas;

TVoidPtrMap &NodeBcMap(): Retorna uma referência à árvore binária que contém ponteiros para os nós com condição de contorno associadas;

TVoidPtrMap &MaterialMap(): Retorna uma referência para a árvore binária que contém ponteiros para os materiais da malha;

TVoidPtrMap &BndCondMap(): Retorna uma lista de ponteiros; esta lista aponta para objetos do tipo **TBndCond**. Um objeto desse tipo abstrai as condições de contorno definidas no problema. No esquema da FIGURA 4-11, estas condições seriam Γ_1 e Γ_2.

FUNÇÕES UTILITÁRIAS

void Print(ostream & out = cout): Imprime as características da malha computacional, na saída definida por out;

TDofNod* FindNode(Long nid): Procura e retorna um ponteiro para o nó que tenha o Id igual a nid. Retorna NULL caso o nó não tenha sido encontrado;

TMaterial* FindMaterial(Long nm): Procura e retorna um ponteiro para o material com o número nm;

TCompEl *FindElement(Long elnumber): Procura e retorna um ponteiro para o elemento computacional com o número elnumber;

TCoSys *FindCoSys(Long coordnumber): Procura e retorna um ponteiro para o sistema de referência com número coordnumber;

TBndCond* FindBc(Long nm): Procura e retorna um ponteiro para a condição de contorno com número nm;

TGeoGrid *ReferenceGrid(): Retorna um ponteiro para a malha geométrica correspondente;

void LoadReferences(): Deixa a malha computacional corrente referenciada pela correspondente malha geométrica. Sabe-se que muitas malhas computacionais podem ser criadas tomando como base uma malha geométrica, porém, somente uma delas é referenciada pela malha geométrica;

virtual void ComputeNodElCon(): Carrega o número de elementos conectados por cada nó;

void ComputeNodeSequence(long ElementId): Renumera os nós segundo o algoritmo de Cuthill-McKee, começando com o elemento ElementId;

void BoundingBox(TMatrix &bbox): Constrói um caixa em torno de um sub-domínio. Esta caixa é construída segundo as coordenadas máximas e mínimas do domínio.

ROTINAS CIENTÍFICAS

long NumEquations(): Retorna o número de variáveis do sistema global;

long BandWidth(): Retorna a banda do sistema;

void InitializeBlock(TBlock &bl): Inicializa o objeto bl com o tamanho de blocos dos nós computacionais;

void Assemble(TBlock &block, TMatrix &rhs): Monta a matriz de rigidez na matriz referenciada por block e o vetor de carga em rhs;

void AutoBuild(): Cria os elementos computacionais, segundo o grau de liberdade dos nós;

int Consolidate(): Constrói as condições de contorno para os nós;

int Check(): Verifica a consistência da estrutura de dados;

char IsChecked(): É verdadeiro se a malha tiver sido verificada;

void LoadSolution(TBlock &sol): Cada nó computacional reserva espaço para armazenar a solução corrente; LoadSolution copia a solução nos nós computacionais.

4.3.1.2.TCompEl

A classe **TCompEl** define o comportamento que um elemento computacional precisa implementar para enquadrar-se no ambiente PZ. A maioria da implementação desses métodos é feita nas classes derivadas. Caso a classe derivada não implemente algum desses, o método implementado pela classe **TCompEl** resultará em mensagem de aviso e/ou saída do programa. Em terminologia de programação orientada a objetos, a classe **TCompEl** é definida como abstrata. O comportamento da classe **TCompEl** é amplo, e inclui os seguintes métodos:

- definição da ordem de interpolação nas várias direções. Isso define o comportamento da adaptatividade tipo *p*;
- definição da regra de integração para o elemento. Como padrão, o elemento utiliza a regra suficiente para integração do quadrado das funções de forma que incorpora. Este método permite a incorporação da sub-integração;
- cálculo da função de forma em um dado ponto paramétrico. Este método é auxiliar para o cálculo da matriz de rigidez;
- cálculo da matriz de rigidez do elemento;
- aplicação da condição de contorno sobre a matriz de rigidez;
- projeção do fluxo associado à lei de conservação sobre o espaço de interpolação;
- cálculo do erro de aproximação baseado na função interpolada do fluxo (implementação do estimador de Zienkiewicz e Zhu);
- montagem da estrutura de dados para eliminar os graus de liberdade dos nós internos do elemento. Este método é implementado apenas na classe **TSuperEl**.

Da classe **TCompEl** são derivadas as classes **TCompEl1d**, **TCompEl2d**, **TCompElQ2d** e **TCompElT2d**, que implementam o comportamento para aproximações uni-dimensionais, bi-dimensionais, bi-dimensionais quadrilaterais e bi-dimensionais triangulares respectivamente. Também disponível está a classe **TElBiot2d** derivada de **TCompElQ2d** (ver FIGURA 4-13).

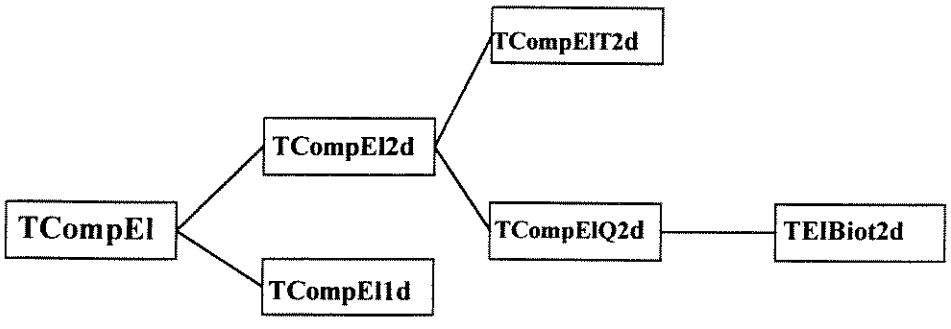

FIGURA 4-13- A classe TCompel e suas classes derivadas.

A adaptatividade tipo *p* é possível porque os elementos computacionais usam bases hierárquicas para a interpolação. É necessário descrever as funções de forma que o ambiente PZ usa, pois são originais e portanto não se encontram na literatura. As bases hierárquicas são construídas utilizando funções de forma, que por sua vez são implementadas tendo como base os polinômios de Chebyshev e polinônios lagrangeanos lineares. Estes polinômios são de fácil implementação computacional.

Para fins de informação a fórmula de recorrência dos polinômios de Chebyshev é demonstrada a seguir.

Seja $x = \cos\theta$, com $-1 \le x \le 1$ e $-\pi \le \theta \le \pi$. Seja também $T_n(x)$ definido por:

$$T_n(x) = \cos(n\theta) = \cos(n \; arc\cos x)$$

Com $n=0$ e $n=1$ obtém-se:

$$T_0(x) = \cos(0\theta) = 1$$
$$T_1(x) = \cos(1\theta) = \cos(\theta) = x$$

Aplicando-se as relações trigonométricas para ângulos múltiplos, tem-se:

$$T_{n+1}(x) = cos(n+1)\theta = cos(n\theta + \theta) = cos n\theta \; cos\theta - sen n\theta \; sen\theta$$
$$T_{n-1}(x) = cos(n-1)\theta = cos(n\theta - \theta) = cos n\theta \; cos\theta + sen n\theta \; sen\theta$$

Somando-se $T_{n+1}(x)$ e $T_{n-1}(x)$ segue que:

$$T_{n+1}(x) + T_{n-1}(x) = 2 cos n\theta \; cos\theta$$

Utilizando-se $T_n(x) = \cos(n\theta)$ e $x = cos\theta$, pode ser obtida a seguinte fórmula de recorrência:

$$T_{n+1}(x) = 2x\,T_n(x) - T_{n-1}(x)$$

Com essa fórmula pode-se obter os polinômios de grau n+1 com base no polinômios de grau n e n-1. Na FIGURA 4-14 são mostradas as expressões e gráficos dos polinômios até grau 5.

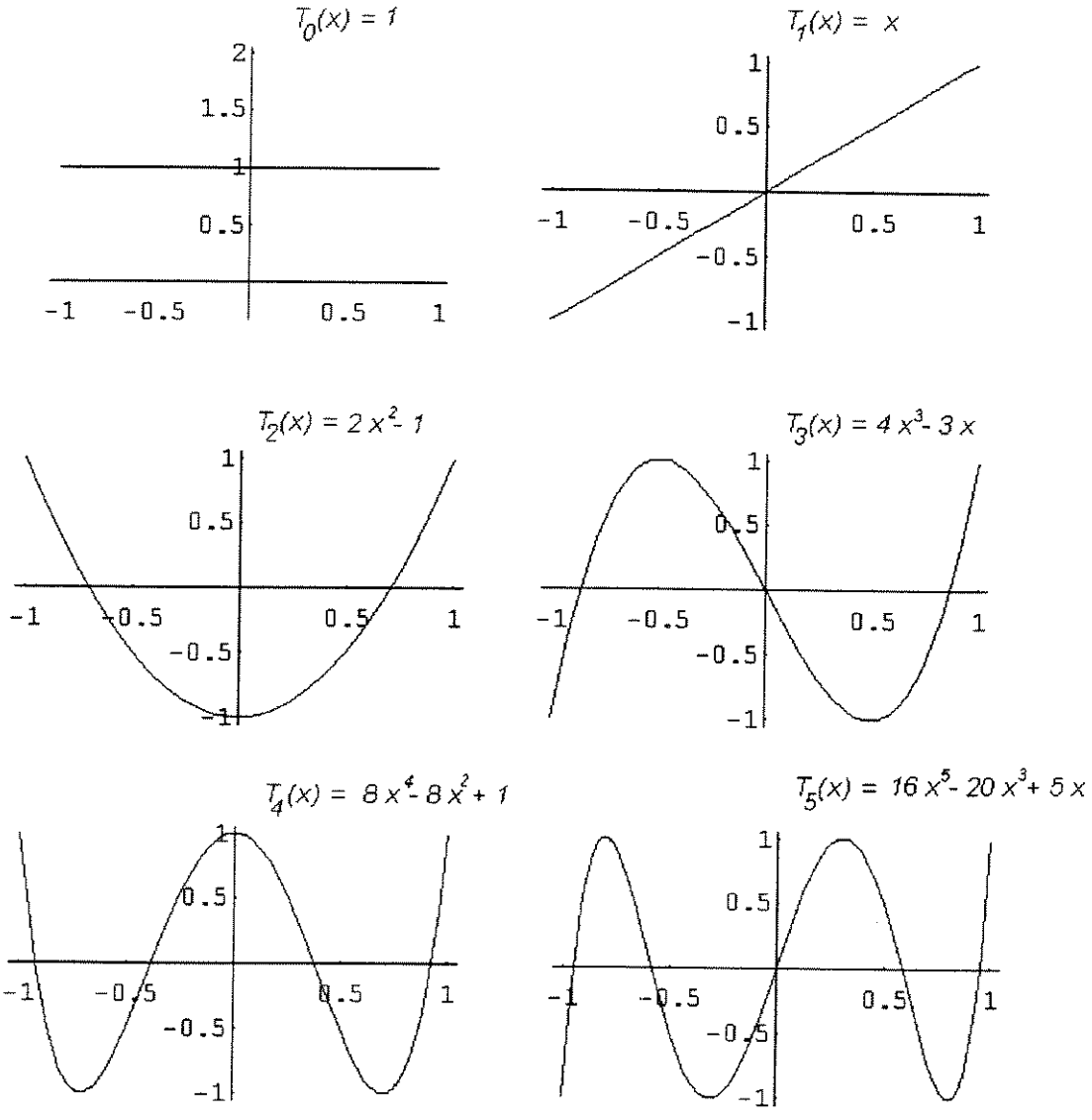

FIGURA 4-14- Polinomios de Chebyshev.

Para os elementos computacionais unidimensionais, o cálculo das funções de forma segue o seguinte algoritmo:

1. construção de uma função quadrática multiplicando-se as duas primeiras funções lagrangeanas lineares;

2. as funções de forma de maior grau são calculadas a partir da multiplicação da função quadrática por uma função de Chebyshev. Por exemplo, a função de quarta ordem é obtida pela multiplicação da função quadrática obtida no primeiro passo por um polinômio de Chebyshev de segundo grau.

Na FIGURA 4-15 são mostradas funções de forma $\psi^p(\xi)$ até ordem 7 resultantes do algoritmo acima.

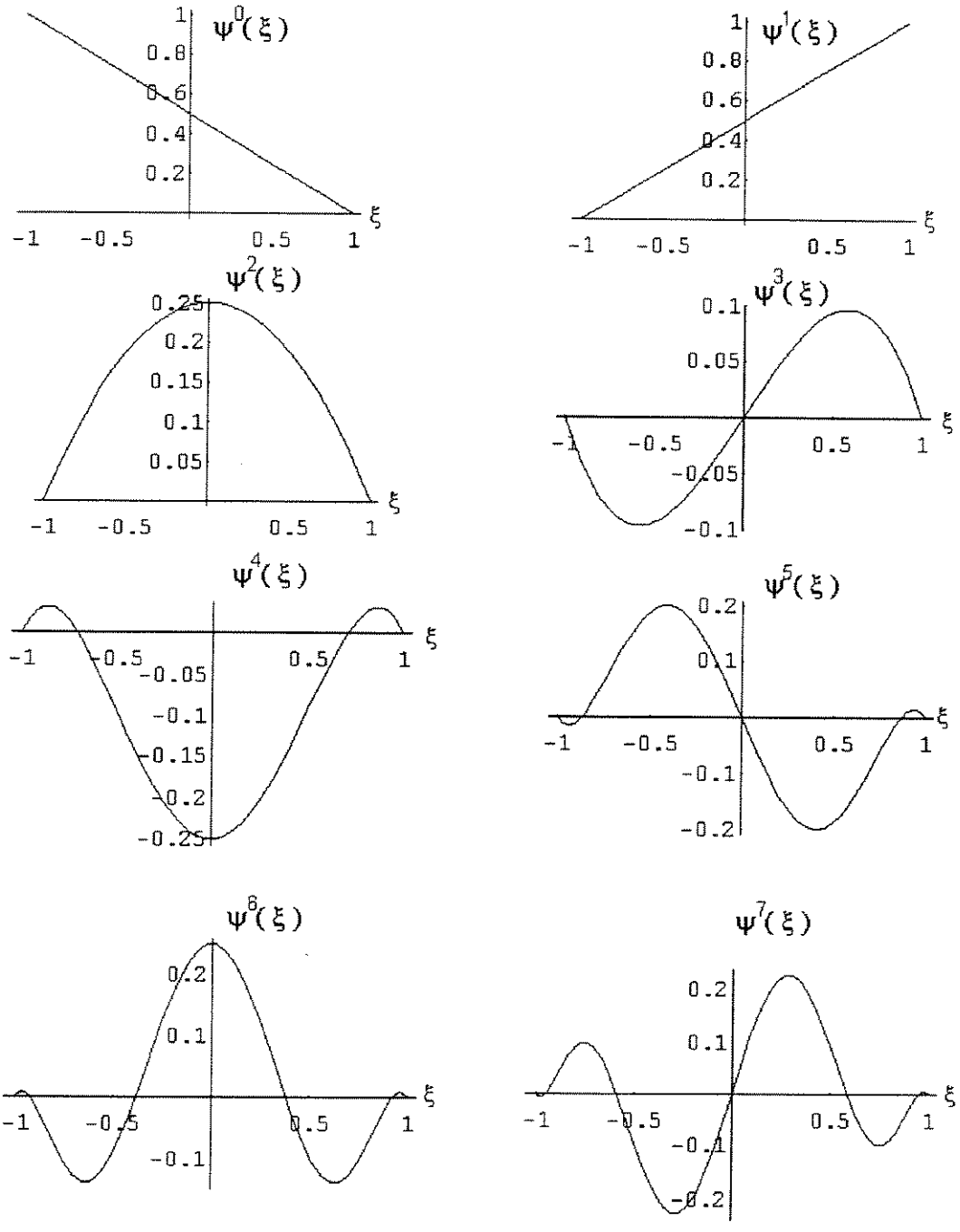

FIGURA 4-15- Funções de forma para elementos computacionais lineares.

Para elementos computacionais quadrados o procedimento é muito semelhante. O cálculo das funções de forma relativas à aresta *lado*$_i$ (ver FIGURA 4-16) segue o seguinte algoritmo:

1. construção da função quadrática segundo a expressão:

$$\psi_i^2(\xi,\eta)=\psi^2(\xi)\psi^0(\eta)$$

2. a construção de funções de forma de ordem maior segue a seguinte expressão:

$$\psi_i^p(\xi,\eta)=\psi_i^2(\xi,\eta)\mathrm{T}_{p-2}(\xi),$$

onde $\mathrm{T}_p(\xi)$ são os polinômios de Chebyshev.

Algumas destas funções são mostradas na FIGURA 4-16.

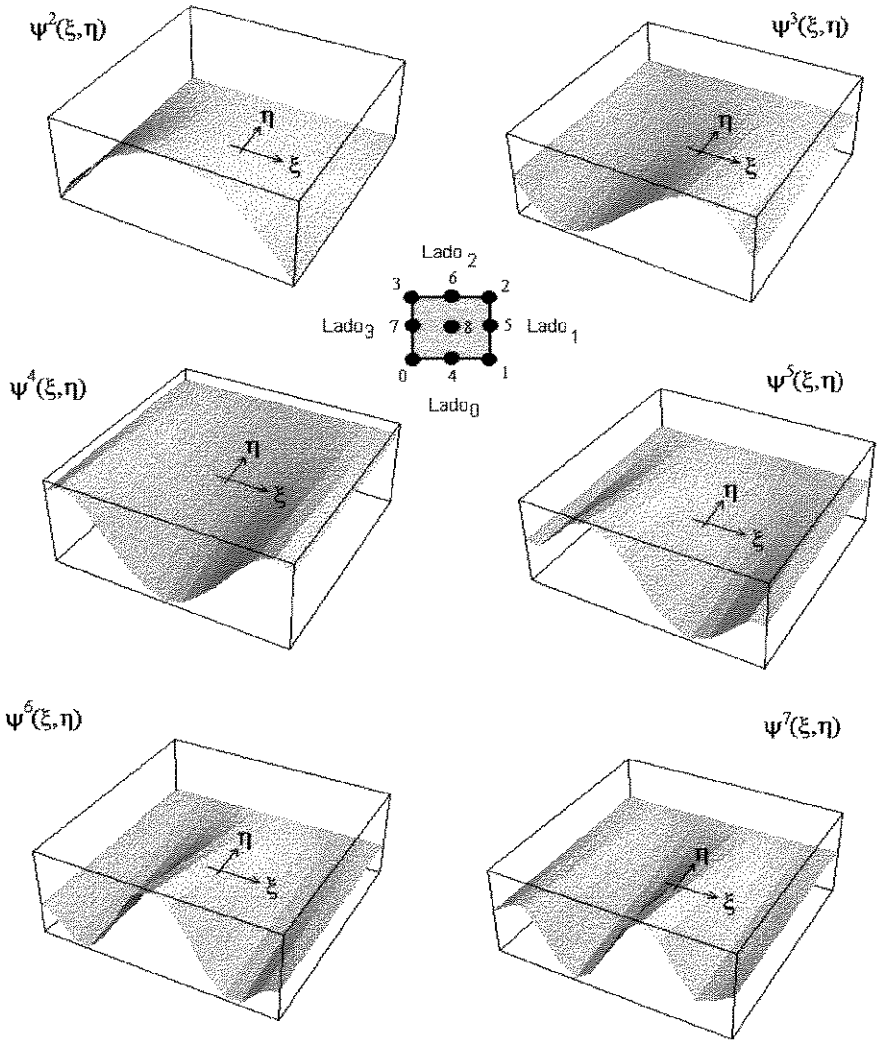

FIGURA 4-16- Funções de forma para elementos quadrados.

VARIÁVEIS MEMBRO

Long fId: Armazena o Id da classe;

TCompGrid *fCGrid: Ponteiro para referenciar a malha computacional à qual o objeto corrente está referenciado.

VARIÁVEIS DO TIPO PROTEGIDO

TGeoEl* fReference: Referência para elemento geométrico;

TCompEl(): Construtor da classe.

VARIÁVEIS DO TIPO PÚBLICO

TCompEl(TCompEl & el): Construtor de cópia; constrói o elemento corrente com as caraterísticas de el.

MÉTODOS PARA ACESSO ÀS VARIÁVEIS MEMBRO

Long Id(): Retorna o Id (índice de numeração) do elemento;

TGeoEl* Reference(): Retorna um ponteiro para um elemento geométrico (**TGeoEl**);

virtual Long MaterialNumber(): Retorna o índice do material, referente ao objeto corrente;

virtual int NumNodes(): Retorna o numero de nós do elemento;

virtual TDofNod *Node(int i): Retorna um ponteiro para o i-ésimo nó (no_i);

virtual TDofNod *SideNode(short node, short side): Retorna um ponteiro para o nó que está situado no lado side. O valor de node pode ser 0, 1 ou 2, segundo a convenção da FIGURA 4-17.

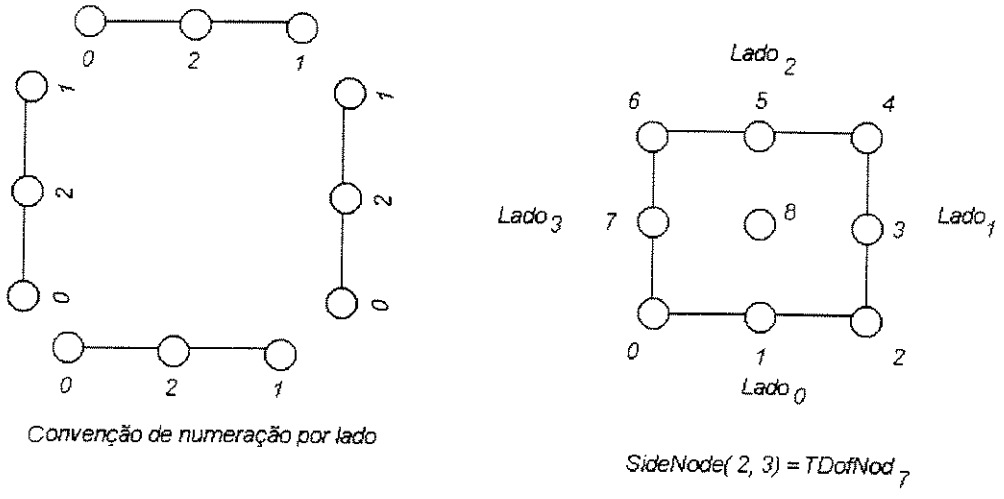

FIGURA 4-17- Ponteiro para o nó da aresta $lado_i$.

MÉTODOS QUE MODIFICAM OS DADOS DO TIPO PRIVADO

void SetId(Long sn): Armazena o Id do elemento;

long CreateElementId(): Cria um Id que é único dentro da malha;

virtual void SetMaterial(TMaterial *): Esta função permite indicar que o material do elemento computacional é igual ao objeto apontado pelo argumento;

virtual char *ValidMat(): Retorna o nome do material do qual a classe **TMaterial** deve ser derIvada. Pode-se citar um contra-exemplo: se estiverem sendo usados elementos triangulares, um material não válido seria **TMat1dLin** (material linear unidimensional);

void SetGrid(TCompGrid *cgr): Define que o elemento corrente pertence à malha computacional cgr;

TCompGrid *Grid(): Retorna um ponteiro para a malha computacional à qual o elemento pertence.

FUNÇÕES DE AJUDA

TCompEl *Connect(short iside): Retorna um ponteiro para o elemento computacional que esta conectado ao longo do lado iside;

short SideConnect(short iside): Retorna o número do lado do elemento vizinho que esta conectado com o lado iside do elemento corrente;

virtual void SetInterpolationOrder(ShortAVec &ord): Com este método indica-se a ordem do polinômio de interpolação;

virtual int PreferredSideOrder(int iside): Retorna a ordem de interpolação do polinômio que está ao longo do lado iside;

short Bc(short iside): Retorna o índice da condição de contorno ao longo do lado iside;

virtual void Print(ostream & out = cout): Imprime as características do elemento corrente, na saida definida por out;

friend ostream& operator<<(ostream &s,TCompEl &el): Sobrecarga do operador << , que imprime as características do elemento geométrico (**TCompEl**) na saída definida por s.

ROTINAS CIENTÍFICAS

void Chebyshev(double x, int num, TFMatrix & phi, TFMatrix & dphi, Long *id): Calcula o valor da função de forma unidimensional no ponto x; num indica o número de funções a serem calculadas. Caso id[0]>id[1], o sinal das funções de ordem ímpar e maiores que dois é invertido;

virtual void Shape(double x, int num, TMatrix &phi, TMatrix &dphi): Retorna o valor das funções de forma e de suas derivadas em phi e dphi respectivamente. O número de funções é num, avaliado no ponto x;

virtual void CalcStiff(TElementMatrix &ek, TElementMatrix &ef): Calcula a matriz de rigidez local em ek e o vetor de cargas local em ef;

virtual void ApplyBc(TElementMatrix &ek, TElementMatrix &ef, TBndCond &bc, int lado): Aplica as condições de contorno na matriz de rigidez ek e no vetor de forças ef; estas condições são dadas por bc;

virtual void ProjectFlux(TElementMatrix &M, TElementMatrix &N): Projeta a função de fluxo sobre o espaço de aproximação do elemento finito;

virtual void EvaluateError(void (*fp)(DoubleAVec &loc, DoubleAVec &val, TMatrix &deriv),double &true_error, double &L2_error, TMatrix *flux, double &estimate): Calcula os erros definidos pela norma da energia (true_error) e pela norma da média quadrática (L2_error). Esses valores são dados por:

$$e = (\textit{solução exata}) - (\textit{solução aproximada})$$

$$\textit{Norma da Energia} = \textit{EnergiaL} = \left\{ \int_0^1 \left[(e')^2 + (e)^2 \right] \right\}^{\frac{1}{2}}$$

$$\textit{Norma da Média Quadrática} = \textit{NormaL} = \left\{ \int_0^1 e^2 \right\}^{\frac{1}{2}}$$

O argumento estimate retorna o cálculo do erro baseado no fluxo:

$$\int_\Omega f\left(\sigma_{calc} - \sigma_{\text{int}erp}\right) d\Omega$$

O argumento (*fp)(DoubleAVec &loc, DoubleAVec &val, TMatrix &deriv) representa um ponteiro para uma função, que retorna o valor exato da solução da equação diferencial. De fato, a solução exata está representada dentro da função apontada por fp. No argumento loc são passadas as variáveis da função, ou seja, x0, x1 e x2. O valor da função é retornado em val e o da derivada em deriv.

virtual void Solution(DoubleAVec &qsi, int var, DoubleAVec &sol, int &numvar): Retorna o valor da solução no vetor sol. Esta solução corresponde à variável com índice var. O ponto onde a solução é avaliada é dado por qsi e o número de variáveis armazenadas em sol é retornado em numvar. Se estiver sendo estudando um problema bi-dimensional de elasticidade com três variáveis, ***deslocamento***, ***pressão*** e ***tensão***, no ponto $p=(\xi_0,\eta_0)$ no caso do ***deslocamento***, a chamada será: Solução(p,0,sol,numvar), onde sol retornará $Deslocamento_x$ e $Deslocamento_y$ e numvar seria igual a dois. Ainda no mesmo ponto no caso da ***pressão*** a chamada será: Solução(p,1,sol,numvar). O argumento sol retornará somente um e numvar será igual a um.

virtual void ReduceInternalNodes(): Este método é usado para a decomposição de domínios. Ela deixa "invisíveis" os nós internos de um sub-domínio (ver **TSuperEl**).

4.3.1.3. TDofNod

Esta classe deixa disponíveis informações como o número da equação associada ao grau de liberdade de cada nó, o número de elementos conectados ao nó e os valores da solução atual. Um objeto **TDofNod** nem sempre está referenciado a um nó geométrico (**TGeoNod**).

VARIÁVEIS MEMBRO

TGeoNod *fReference: Ponteiro para nó geométrico (**TGeoNod**);

long fNodeId: Armazena o Id da classe;

long fBlockNumber: Armazena o número do objeto do *bloco* associado ao nó (ver classe **TBlock**);

DoubleAVec fVar: Vetor das variáveis nodais;

int fNumElCon: Número de elementos conectados ao nó corrente;

static Long gNodeCounter: Contador usado para escolher um Id único para o nó. Cada vez que um nó é criado, este contador é incrementado, mas se o nó deixar de existir na malha, este contador não decresce.

MÉTODOS DO TIPO PÚBLICO

TDofNod(int ndof, TGeoNod *ref = NULL): Construtor da classe;
Long BlockNumber(): Retorna o número do **TBlock** relacionado ao objeto corrente;
void SetBlockNumber(Long i): Coloca i como número do bloco;
Long Id(): Retorna o Id da classe.

FUNÇÕES DE AJUDA

double Val(int i): Retorna o valor da i-ésima variável var_i.
double &operator()(int i): Sobrecarga do operador *( )* para retornar o valor da i-ésima variável var_i;
int NDof(): Retorna o número de graus de liberdade do nó corrente (número de graus de liberdade);
void SetNDof(short newsize): Muda o número de graus de liberdade associados ao nó;
void SetVal(int i, double x): Deixa a variável ***i*** com o valor de x (fVar[i]=x);
TGeoNod *Reference(): Retorna um ponteiro para o nó geométrico que o objeto corrente está referenciando;
void Load(): Coloca a referência do objeto corrente dentro de um nó geométrico;
void Print(ostream & out = cout): Imprime as características da classe;
void ResetElCon(): Atribui ao número de elementos conectados ao nó o valor zero;
void IncrementElCon(): Incrementa em 1 o número de elementos conectados ao nó;
int NumElCon(): Retorna o número de elementos conectados ao nó;

4.3.1.4. TDofNodBc

A classe **TDofNodBc** associa o grau de liberdade do nó às suas condições de contorno. Estas condições de contorno podem ser condições de Dirichlet, carregamento pontual ou condições mistas. Este objeto foi definido como um *struct* e por isso suas variáveis e métodos estão disponíveis para qualquer classe. Sua implementação é mostrada abaixo.

```
struct TDofNodBc {
      TDofNod *fNod;
      TBndCond *fBc;

TDofNodBc(TDofNod *nd, TBndCond *b){
      fNod = nd;
      fBc = b;
}

int Check() {
      return (!fNod) || (!fBc);
}
};
```

4.4. DEFINIÇÃO DOS COEFICIENTES DA EQUAÇÃO DIFERENCIAL E CONDIÇÕES DE CONTORNO.

No conceito do programa, a matriz de rigidez é vista como a integral de uma função cujos argumentos são os valores das funções de forma e de suas derivadas, e cujo resultado é um valor matricial. A classe que implementa esta função é a classe ***TMaterial.*** Este nome foi escolhido porque é nesta classe que são armazenados os coeficientes de material correspondentes a um problema de mecânica computacional.

4.4.1. TMaterial

A principal função desta classe é calcular a contribuição para a matriz de rigidez dos nós internos e das condições de contorno. Tendo em vista que as respostas de materiais uni-dimensionais é diferente da resposta de materiais bi-dimensionais (um utiliza uma derivada, o outro utiliza duas derivadas), foram incluídos métodos para verificar se o objeto pertence a uma classe derivada apropriada. **TMaterial** é uma classe que serve de base para **TMat1dLin** e para **TMat2dLin**, e esta última classe serve de base para **TMatBiot** e **TPMat**.

VARIÁVEIS MEMBRO

Long fId: Id da classe.

MÉTODOS DO TIPO PÚBLICO

TMaterial(Long identificador): Construtor, inicializa o Id da classe com identificador;

TMaterial (TMaterial & nn): Construtor de cópia;

virtual TBndCond *CreateBc(Long id, int typ, TFMatrix &val1, TFMatrix &val2): Cria a condição de contorno a ser identificada com id. Esta condição é do tipo typ, e os valores são dados em val1 e val2;

virtual char *Name(): Retorna o nome da classe derivada da qual Name() está sendo invocada. Se este método está sendo chamado da classe base (**TMaterial**) é retornado "no_name";

virtual short DerivedFrom(char *Name): Retorna 1 caso a classe seja derivada ou igual a classe indicada por Name;

Long Id(): Retorna o Id da classe;

virtual short NumberOfFluxes(): Retorna o número de fluxos. Por exemplo, no caso de um problema de elasticidade, o fluxo tem três valores.

virtual short NumVariables(): Retorna o número de variáveis de estado. No caso da elasticidade tri-dimensional tem-se 3 deslocamentos, dx, dy e dz. Já num problema de fluxo de calor tem-se a temperatura como única variável;

virtual void SetForcingFunction(void (*fp)(DoubleAVec &loc, DoubleAVec &result)): Permite definir um vetor de carga sob a forma de função;

virtual void Print(ostream & out = cout): Imprime as características do material, na saída definida por out;

virtual int VariableIndex(char *name): Retorna o índice da variável com nome name;

virtual void Solution(TFMatrix &Sol, TFMatrix &DSol, int var, DoubleAVec &Solout, int &numvar): Retorna a solução associada com o índice var , dado os valores de estado e suas derivadas em Sol e DSol. O índice var é obtido com VariableIndex ("*nome_da_variavel*").

Observação: Suponha que queira-se obter a pressão no ponto $p=(\xi_0, \eta_0)$.O procedimento para obter este valor é o seguinte:

Primeiro é obtido o índice da variável com

var=material.VariableIndex ("*pressao*")

Como segundo passo chama-se a função Solution(p,var,sol,numvar) do objeto tipo **TCompel**. O valor da pressão é retornado em sol. Em numvar é retornado o número de variáveis (neste caso é um porque a solução requerida é escalar). O objeto **TCompel**. calcula o valor e as derivadas da(s) função(ões) no ponto p e passa estes valores para função Solution do objeto do tipo **TMaterial**.

4.4.1.1.TBndCond

Implementa a definição da condição de contorno. Os objetos do tipo **TBndCond** não implementam a condição de contorno, apenas armazenam os dados utilizados pelos objetos do tipo **TMaterial** para a aplicação das mesmas. Para garantir a compatibilidade com o objeto do tipo **TMaterial**, o construtor é acessível apenas aos objetos deste tipo.

VARIÁVEIS MEMBRO

As seguintes declarações deixam as variáveis membro desta classe acessíveis para as classes do tipo **TMaterial**:

friend class TMat1dLin;

friend class TMat2dLin;

friend class TMatBiot;

friend class TPMat;

As variáveis membro para esta classe são:

long fNumber: Número da condição de contorno (Γ_1, Γ_2, Γ_3.....Γ_n);

int fType: Tipo da condição de contorno (Dirichlet, Neuman, Mixto);

TFMatrix fBcVal1: Valor da primeira condição de contorno;

TFMatrix fBcVal2: Valor da segunda condição de contorno;

TMaterial *fMatPtr: Retorna um ponteiro para o objeto **TMaterial** que criará a condição de contorno.

MÉTODOS DO TIPO PRIVADO

TBndCond(long number, int type, TFMatrix &val1, TFMatrix &val2): Construtor da classe;

TBndCond(TBndCond & bc): Construtor de cópia.

MÉTODOS DO TIPO PÚBLICO

long Number(): Retorna o número da condição de contorno;

int Type(): Retorna o tipo da condição de contorno;

TFMatrix &Val1(): Retorna o primeiro valor da condição de contorno;

TFMatrix &Val2(): Retorna o primeiro valor da condição de contorno;

TMaterial *MatPtr(): Retorna um ponteiro para o material que criará a condição de contorno;

static TBndCond *CreateNodBc(int num, int type, TFMatrix &val1, TFMatrix &val2): Retorna um objeto do tipo **TBndCond**. Este objeto é a condição de contorno com número num, é do tipo type, e tem os valores val1 e val2. O objeto retornado por este método não está associado a nenhum tipo de material;

void Print(ostream & out = cout): Imprime as características do objeto na saída definida por out;

4.5. MONTAGEM E SOLUÇÃO DO SISTEMA DE EQUAÇÕES - CLASSE TAnalysis

Um ambiente orientado a objetos fornece um conjunto de classes que o usuário usará para implementar suas próprias aplicações. Dentro do ambiente PZ existe uma classe chamada de **TAnalysis**, que tem como objetivo principal chamar de forma ordenada os métodos das classes do ambiente PZ segundo a sequência do cálculo. Esta classe chama os métodos da malha geométrica e malha computacional para calcular o número de equações do sistema, a largura da banda, escolher o tipo de matriz mais conveniente para a matriz de rigidez global, resolver o sistema, etc.

VARIÁVEIS MEMBRO

TGeoGrid *fGeoGrid: Malha geométrica;

TCompGrid *fCompGrid: Malha computacional;

TMatrix *fStiffness: Matriz de rigidez global. Deve-se notar que ela está apontando a classe base **TMatrix**, o que nos permite inicializá-la com os diferentes tipos de matrizes disponíveis no **TMatrix**;

TFMatrix *fRhs: Vetor de carga global do sistema de equações, a ser montado por esta classe;

TFMatrix *fSolution: Nesta matriz é armazenada a solução do sistema linear montado nesta classe.

TBlock *fBlock: Este objeto do tipo **TBlock** é usado para montar a matriz de rigidez. Cada bloco está associado ao número de graus de liberdade de cada nó.

void (*fExact)(DoubleAVec &loc, DoubleAVec &result, TMatrix &deriv): Esta função é a solução exata do problema; é usada para calcular a norma da energia e a norma da média quadrática.

TMatrixSolver fSolver;: Esta variável é usada para que seja possível gerenciar com mais facilidade os diferentes tipos de métodos diretos e indiretos que a classe **TMatrix** implementa.

MÉTODOS DO TIPO PÚBLICO

TAnalysis(TCompGrid *calcgrid): O objeto do tipo **TAnalysis** é inicializado com a malha geométrica na qual são feitos todos os cálculos;

void SetBlockNumber(): As equações da malha são renumeradas segundo o algoritmo de Cuthill-McKee;

long NumEquations(): Retorna o número de equações do sistema;

long BandWidth(): Calcula a largura da banda do sistema;

int BlockBandWidth(): Calcula a largura da banda do sistema em blocos;

void SetMatrix(TMatrix *stiff): Inicializa a matriz global do sistema com stiff.

void Assemble(): Monta a matriz de rigidez global;

void Solve(): Inverte a matriz de rigidez;

virtual void Run(VoidPtrVec &scalnames, VoidPtrVec &vecnames, char *plotfile, ostream &out = cout): Chama a sequência apropriada de métodos para construir a solução a cada espaço de tempo (montagem do sistema global de equações, solução do sistema, chamada às interfaces com os pós-processadores). Nos argumentos scalnames e vecnames são passados os nomes das variáveis escalares e numéricas respectivamente; se estas listas estão vazias, as classes que montam a interface com os pós-processadores não são ativadas;

void LoadSolution(): Carrega a solução dentro da malha computacional;

void SetExact(void (*f)(DoubleAVec &loc, DoubleAVec &result, TMatrix &deriv)): Deixa a variável membro *fExact* apontando para *f*;

void PostProcess(DoubleAVec &loc, ostream &out = cout): Imprime o cálculo da norma da energia, norma da média quadrática e o erro baseado no fluxo;

TMatrixSolver & Solver(): Devolve uma referência para um objeto do tipo **TMatrixSolver**; isso permite manipular todas as possibilidades implementadas para a solução de sistemas de equações, seja por métodos diretos ou iterativos;

4.6. PROGRAMA EXEMPLO

A seguir é apresentado um programa que utiliza as classes do ambiente PZ. Um programa principal - *main()* - precisa de dois arquivos de dados, um para a leitura da malha gerada pelos pré-processadores e outro para a leitura das condições de contorno. Esta é uma abordagem muito natural pois permite aplicar diferentes condições de contorno para uma mesma geométria. Neste exemplo, esses arquivos são "chave.msh" e "chave.mat", que correspondem ao teste da página 107, porém para um domínio sem subestruturação. Uma explicação sobre este programa é apresentada depois do código.

4.6.1. CÓDIGO EXEMPLO

```
1.#include "pzgrid.h"        //Malha geométrica
2.#include "pzcgrid.h"       //Malha computacional;
3.#include "analysis.h"       //Classe TAnalysis
4.#include "tsbndmat.h"      //Matriz banda simétrica
5.#include "modulef.h"       //Classe que le uma malha gerada pelo Modulef
6.#include "pzgeoel.h"       //Elemento geométrico
7.#include "compel2d.h"      //Elemento computacional bi-dimensional.
8.#include "elasmat.h"       //Material elastico
9.#include "voidvec.h"       //vetor de ponteiros

10.void ReadMaterialBC(char *filename, TCompGrid &c, TAnalysis * &an);
11.void main() {
12.     TGeoGrid *g = new TGeoGrid;
13.     TCompGrid *c = new TCompGrid(g);
14.     TGeoGrid::gCurrent = g;
15.     TCompGrid::gCurrent = c;
16.     TModulef arq("chave.msh");
17.     arq.Read(*c);
18.     TAnalysis *a = 0;
```

```
19.    ReadMaterialBC("chave.mat",*c,a);
20.    VoidPtrVec scalnames(4),vecnames(4);
21.    scalnames[0] = "Pressure";
22.    scalnames[1] = "MaxStress";
23.    scalnames[2] = "SigmaX";
24.    scalnames[3] = "SigmaY";
25.    vecnames[0] = "Displacement";
26.    vecnames[1] = "PrincipalStresses 1";
27.    vecnames[2] = "PrincipalStresses 2";
28.    vecnames[3] = "Displacement6";
29.    nt numeq = (int) a->NumEquations();
30.    a->SetBlockNumber();
31.    Int band = (int) a->BandWidth();
32.    TSBMatrix *s = new TSBMatrix(numeq,band);
33.    a->SetMatrix(s);
34.    a->Solver().SetDirect(ECholesky);
35.    a->Run(scalnames,vecnames,"sample.dx");
36.    delete g;
37.    delete c;
38.    delete a;
39.}
```

4.6.2. EXPLICAÇÃO DO CÓDIGO EXEMPLO

Nas linhas 1 a 9 estão incluídos os cabeçalhos (*headers*). No código exemplo cada *header* tem sua explicação ao lado como comentário. A linha 10, da função `void ReadMaterialBC(char *filename, TCompGrid &c, TAnalysis * &an)`, será explicada com mais detalhes no ANEXO.

A seguir, cada linha do código será explicada separadamente:

•Linha 12 : Declaração de um ponteiro para malha geométrica;

•Linha 13 : Declaração de um ponteiro para malha computacional;

•Linha 14 : A variável global gCurrent fica apontando para a malha geométrica g;

•Linha 15 : A variável global gCurrent fica apontando para a malha computacional c;

•Linha 16 : É criado o objeto arq do tipo **TModulef**. Este tipo de objeto permite ler malhas geradas pelo programa MODULEF;

•Linha 17 : São criados nós e elementos geométricos segundo a malha gerada pelo programa MODULEF. Os nós e elementos criados estão referenciados pela malha computacional c;

•Linha 18 : Declaração de um ponteiro nulo do tipo **TAnalysis**;

•Linha 19 : Leitura das condições de contorno do arquivo "chave.mat". Estas condições de contorno são aplicadas aos elementos computacionais de c. A Seguir é criado um objeto **TAnalysis** tendo como base a malha c. Este objeto **TAnalysis** é apontado pelo ponteiro a;

•Linha 20 : Declaração dos vetores de ponteiros que guardam os nomes das variáveis escalares (scalnames(4)) e vetoriais (vecnames(4));

Da linha 21 a 28 são passadas os nomes das variáveis escalares e vetoriais;

•Linha 29 : Cálculo do número de equações do sistema global ou tamanho do sistema de equações;

•Linha 30 : A função SetBlockNumber() renumera a malha segundo o algoritmo de Cuthill-McKee;

•Linha 31 : Cálculo do tamanho da banda do sistema global;

•Linha 32 : Criação dinâmica de uma matriz banda simétrica s. Podem ser usados diferentes tipos derivados da classe **TMatrix**. Como exemplo, podem ser citados:

```
TSkylMatrix *s = new TSkylMatrix(numeq);//matriz skyline simétrica
TSSpMatrix *s = new TSSpMatrix(numeq);//matriz esparsa simétrica
```

•Linha 33 : A matriz **s** está sendo colocada dentro do objeto da classe **TAnalysis**. Isso permite montar o sistema global de equações com a esparsidade de **s** (matriz banda simétrica). Pode-se usar qualquer tipo de matriz derivada da classe **TMatrix** como argumento de SetMatrix(s), mas tem que ser concordante com o tipo de problema que está sendo aproximado. Como exemplo, tem-se que a equação $u'' + u' = f(x)$ não resulta em um sistema simétrico na aproximação por elementos finitos, ou seja, não se pode usar as matrizes simétricas da classe **TMatrix** para o sistema global.

•Linha 34 : Nesta linha escolhe-se o método a ser usado para resolver o sistema de equações. Neste exemplo está sendo usado o método de Cholesky. Esta linha pode ser mudada por qualquer uma das seguintes linhas:

```
a->Solver().SetDirect(ELU);//decomposicao LU
a->Solver().SetDirect(ELDLt);//decomposicao LDLt
a->Solver().SetJacobi(30, 1E-8, 0);//30 iteracoes de Jacobi
a->Solver().SetSOR(10, 0.5, 1E-8, 0);//10 iteracoes de SOR
a->Solver().SetSSOR(20, 1.3, 1E-8,0);//20 iteracoes de SSOR
```

•Linha 35 : Esta função monta o sistema global e o resolve segundo o metódo da linha 34. O primeiro argumento passa o nome das variáveis escalares de acordo com as linhas 21, 22, 23 e 24. Os nomes das variáveis vetoriais (linhas 25, 26, 27 e 28) são passadas no segundo argumento. O terceiro argumento é usado para criar um arquivo de interface onde serão escritos os resultados para um pós-processamento. O tipo de formato que este arquivo terá depende da extensão de seu nome. Se a extensão for "plt" será criado um formato para leitura no VIEW3D. No caso da extensão "pos" cria-se um formato para leitura no MVIEW, e se a extensão for "dx" (como neste exemplo) será criado um formato para o DATA EXPLORER. Se um dos vetores com os nomes das variáveis é vazio, ele não cria nenhum arquivo de interface.

•Nas linhas 36, 37 e 38 são apagados a malha geométrica, a malha computacional e o objeto da classe **TAnalysis** respectivamente.

5. CLASSE TMatrix ORIENTADA A OBJETOS

5.1. VISÃO GERAL

Existem atualmente muitas bibliotecas que manipulam matrizes com muita eficiência, tais como LAPACK, LINPACK, EISPACK[1]. Porém, essas bibliotecas foram concebidas com uma filosofia estrutural, o que as torna difíceis de serem manipuladas e estendidas. Suas novas versões "orientadas a objeto", como por exemplo LAPACK++, são adaptações das versões estruturadas e não incorporam as vantagens oferecidas por um projeto orientado a objetos. Outro inconveniente dessas bibliotecas é usar muitos argumentos na chamada de suas funções, tornando seu uso tedioso.

O objetivo da criação da classe **TMatrix** foi o desenvolvimento de uma biblioteca de uso geral, orientada a objetos, que:

- manipule matrizes com diferentes padrões de armazenamento;
- seja fácil de usar e portátil para diferentes plataformas;
- que use os conceitos de encapsulamento, herança, abstração de dados;
- que seja, sobretudo, de fácil manutenção.

Os cálculos de elementos finitos geram sistemas lineares de equações [K][u]=[F], onde a matriz de rigidez (K) pode ser simétrica ou não-simétrica, dependendo da equação diferencial modelada. Ainda segundo a estrutura da malha, K pode ser melhor armazenada em estrutura de Banda ou Skyline (ver FIGURA 5-1). Isso mostra a necessidade de ter uma biblioteca que manipule diferentes tipos de matrizes.

---

[1] Maiores informaçoes sobre essas bibliotecas podem ser encontradas em ANDERSON et al. (1995) e BUNCH et al. (1979).

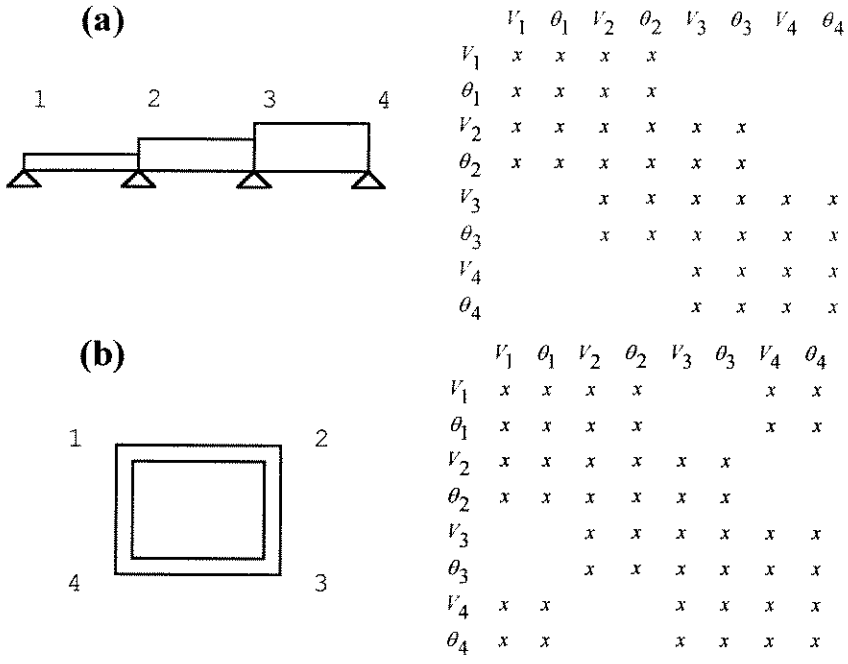

FIGURA 5-1- Exemplos de matriz de rigidez: (a) banda e (b) skyline.

A biblioteca chamada de ***TMatrix*** contém as seguintes matrizes:

1. matrizes não simétricas:
    - matriz cheia;
    - matriz banda;
    - matriz esparsa;
2. matrizes simétricas:
    - matriz cheia;
    - matriz banda;
    - matriz skyline;
    - matriz esparsa;
3. condensação estática.

5.2. ESTRUTURA DAS CLASSES TMatrix

Uma matriz é um arranjo ordenado de dados que permite diferentes tipos de operações sobre o mesmo. Com a intenção de abstrair a idéia de matriz, existe uma classe base que define um conjunto de métodos padrão para todos os tipos de matrizes. Porém, cada um destes métodos poderá ter uma implementação apropriada segundo o tipo de matriz. Esta biblioteca possui vários tipos de matrizes[2], cada uma com uma alocação de memória apropriada, assim como algoritmos especializados para manipular estas alocações. Estas particularidades são ou não implementadas nas classes derivadas. Existe ainda outra classe derivada que serve de classe base para as matrizes do tipo simétricas. Estas duas classes base são do tipo virtual, isto é, não fazem nenhum tipo de alocação, apenas perfilam o comportamento das classes filhas.

Na FIGURA 5-2 pode-se observar a estrutura desta classe. Tem-se ainda que o usuário é livre para derivar suas próprias classes.

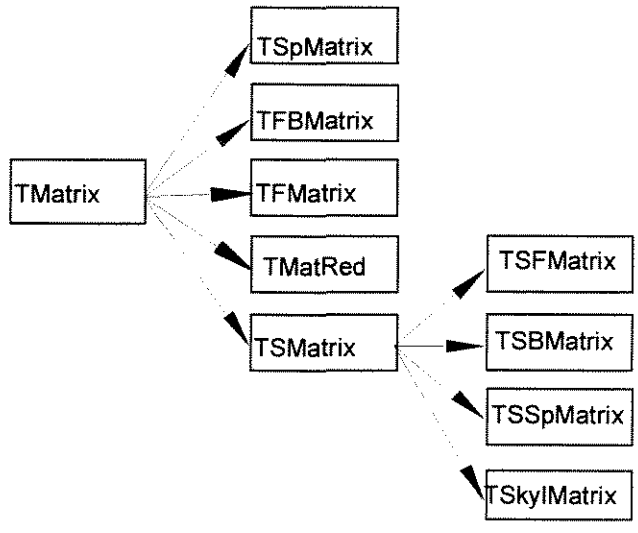

FIGURA 5-2- A estrutura das classes derivadas de TMatrix.

[2] A classe TMatrix é a classe base dessa biblioteca, que contém diferentes tipos de matrizes derivadas. Daqui em diante, o termo as classes TMatrix estará se referindo à biblioteca inteira.

Exemplos de utilização desta biblioteca podem ser obtidas na pagina internet http://www.cenapad.unicamp.br/~phil/matrix.htlm.

5.3. A CLASSE BASE TMatrix

Esta é a classe base da qual os diferentes tipos de matrizes foram derivadas. Nesta classe define-se o comportamento que será implementado pelas classes derivadas. Na classe **TMatrix** também estão definidos vários tipos de *flags*, que contêm informações sobre o "estado" da matriz, como por exemplo, se a matriz foi decomposta e qual decomposição foi utilizada. A obtenção deste dado é justificável, porque a matriz resultante de uma decomposição (Cholesky, LU, LDLt, etc.) é guardada na própria matriz.

Também foi admitido que qualquer operação algébrica entre matrizes retorna uma matriz cheia. Esta limitação cobre 99% dos casos e evita a implementação de um grande número de métodos virtualmente inúteis (por exemplo multiplicação matriz "skyline" com matriz esparsa, banda com "skyline", etc.).

Várias operações com sub-matrizes foram implementadas, como por exemplo colocar uma sub-matriz dentro de uma matriz, copiar uma sub-matriz para uma matriz e somar uma sub-matriz com uma matriz. As mesmas operações com sub-matrizes também poderiam ser feitas com objetos do tipo **TBlock**, porém foram implementadas para matrizes para que o programador não precise definir um objeto do tipo **TBlock** se for realizar uma operação simples com uma sub-matriz.

A classe **TMatrix** não aloca memória para armazenar os elementos, é uma classe virtual. Mesmo assim, utilizando os métodos GetVal e PutVal (ou o operador () sobrecarregado), vários métodos de decomposição foram implementadas. Isso implica que qualquer classe derivada da classe **TMatrix** dispõe de um conjunto de algoritmos de decomposição. Porém, para melhorar a eficiência da implementação, a maioria dos métodos de decomposição são redefinidos nas classes derivadas. As decomposições

simétricas, Cholesky, LDLt, usam a parte inferior da diagonal da matriz, assumindo que a matriz que a está chamando é simétrica.

Existem três métodos que precisam ser implementados nas classes derivadas. Estes são:

virtual void PutVal(int row, int col, REAL value).

-Torna o elemento (row,col) igual a value;

virtual REAL &GetVal(int row,int col).

-Retorna uma referência ao elemento (row,col);

virtual void Mult(TFMatrix &matin, TFMatrix &matout, int opt).

-Este método implementa a multiplicação entre a matriz corrente e matin; o resultado desta operação é retornado em matout. A opção opt indica se a matriz corrente ou sua transposta será usada na multiplicação.

Quando uma classe é derivada da classe **TMatrix**, o usuário tem a opção de definir PutVal e GetVal e/ou Mult. Dependendo do método a ser implementado, esta classe derivada pode ganhar certa funcionalidade ou não. Em um certo nível de abstração, uma matriz é uma transformação de um vetor em R^n em outro vetor em R^m. Assim, é suficiente definir esta transformação e a matriz existirá sem a capacidade de retornar e colocar elementos dentro de si mesma. A matriz que implementa esta transformação em Mult apenas pode ser invertida somente com métodos iterativos. Classes que implementam PutVal e GetVal não precisam implementar Mult, porque a classe **TMatrix** implementa este comportamento. Não obstante, para aumentar o desempenho da classe derivada pode-se redefinir Mult tirando vantagem da esparsidade da matriz. A chamada de um método que use PutVal e GetVal dentro de uma classe em que não tenham sido implementadas, pode causar erro e a saída da execução. A seguir serão apresentadas as classes derivadas da classe **TMatrix**.

5.3.1. TFMatrix

Implementa o esquema de armazenamento de matrizes cheias. Os elementos desta matriz são alocados dinamicamente e armazenados num vetor. Este tipo de armazenamento inclui matrizes quadradas e retangulares.

O armazenamento dos elementos da matriz é feito num vetor, e são guardados por coluna. A transformação desta alocação é dada por *i+j*Num_Colunas*, conforme esquema mostrado na FIGURA 5-3.

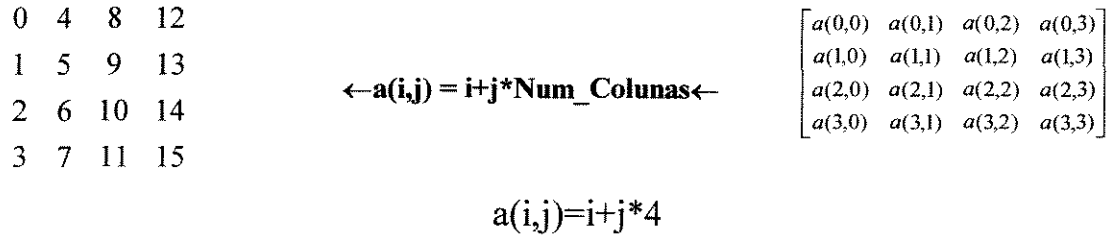

FIGURA 5-3- Esquema da alocação da classe TFMatrix.

Este objeto permite usar um "pedaço" de memória dada pelo usuário para alocar seus elementos. Isto é possível pelo fato que os elementos são alocados num vetor de *doubles*. Porém o tamanho da memória fornecida pelo usuário tem que ser coerente com o tamanho requerido pela matriz.

5.3.2. TFBMatrix

Esta classe gerencia matrizes do tipo banda não simétrica, porém trabalha somente com matrizes quadradas. A correspondência entre a posição dos elementos na matriz e a posição dos mesmos na memória alocada é dada pela seguinte expressão:

a(i,j)=elemento_na_memoria[tamanho_da_Banda(2*i+1)+j]*

Na elaboração da classe para este tipo de matriz, foi admitido que seriam manipuladas apenas matrizes quadradas. Como argumentos do construtor tem-se a dimensão da matriz e o tamanho da banda. Assim, uma matriz quadrada com dimensão 6 e banda 2 terá que ser declarada como *TFBMatrix a(6,2).* Na FIGURA 5-4 mostra-se a correspondente alocação de memória para esta declaração.

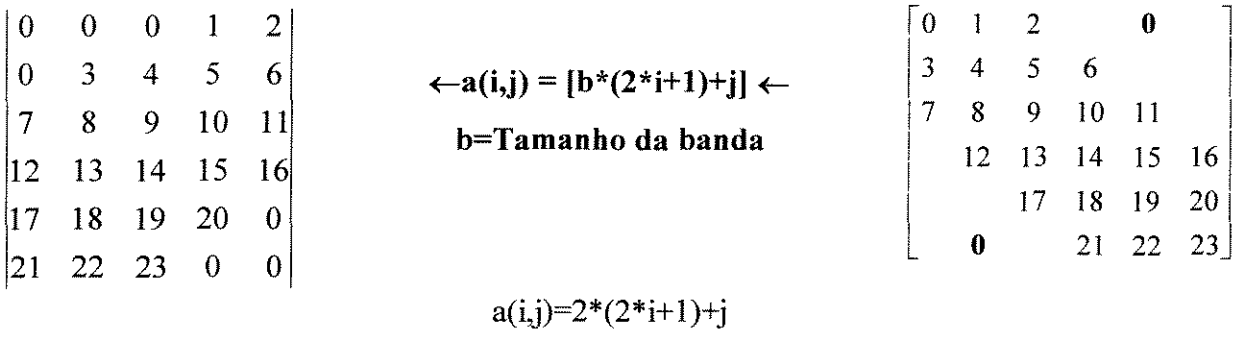

FIGURA 5-4- Esquema da alocação da classe TFBMatrix.

Ao se estudar a quantidade de alocação requerida para uma matriz, tem-se que para uma matriz com dimensão igual a n e banda igual a b são alocados $(2*b+1)*n$ elementos. Para uma matriz cheia quadrada, o número de elementos é igual a n^2. Se for definida a memória poupada como sendo a diferença entre uma alocação tipo banda e uma alocação tipo matriz cheia, pode-se escrever a seguinte expressão:

$$memória\_poupada = n^2 - (2*b+1)*n > 0$$

Resolvendo-se a expressão acima, tem-se que $b < (n+1)/2$, ou seja, só é possível realmente poupar memória se o tamanho da banda for menor que a metade da dimensão da matriz.

Assim, não é vantajoso do ponto de vista da alocação de memória usar um objeto do tipo **TFBMatrix** se a banda for maior que a metade da dimensão da matriz.

5.3.3. TSpMatrix

Esta classe utiliza esquema de armazenamento do tipo esparsa não simétrica. Para a alocação dos elementos deste tipo de matriz foi criada a classe **TLink**, que implementa uma lista ligada com número indefinido de elementos. Na classe **TLink** são feitas operações de inserir, adicionar, remover e retornar elementos da lista. O tempo de busca de um elemento da matriz é otimizado pela utilização da função GetLast, que retorna o último elemento pedido. Este artifício aumenta o desempenho, principalmente na decomposição de matrizes.

Por exemplo, na lista ligada da FIGURA 5-5, no caso de se estar trabalhando com o elemento 6, e for necessário usar o elemento anterior, que é o elemento 5, seria necessário percorrer a lista inteira para retorná-lo. Mas com a função GetLast, retorna-se o elemento 5 imediatamente, sem a necessidade de percorrer toda a lista. Quando o elemento anterior não existir na alocação de memória, é retornado zero.

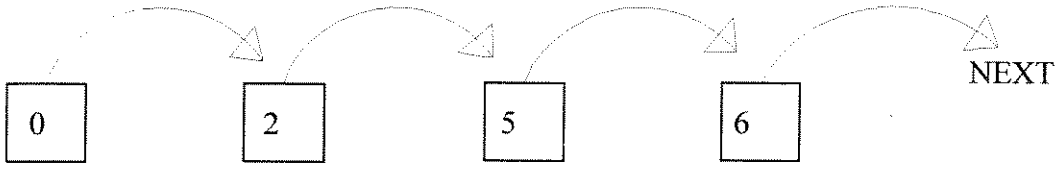

FIGURA 5-5- Lista ligada.

Em seguida é mostrado um exemplo de alocação de memória para uma matriz de 5x8, mostrada na FIGURA 5-6.

$$\begin{bmatrix} X & X & 0 & X & X & 1 & X & X \\ X & X & X & 2 & X & X & X & X \\ X & X & X & X & X & X & X & X \\ X & 3 & X & X & X & X & X & X \\ X & X & X & 4 & 5 & X & X & 6 \end{bmatrix}$$

FIGURA 5-6- Exemplo de uma matriz esparsa.

Esta matriz de 40 elementos só tem 7 elementos alocados, de acordo com o esquema das seguintes listas ligadas, mostrado na FIGURA 5-7. Os números inferiores indicam a posição na lista.

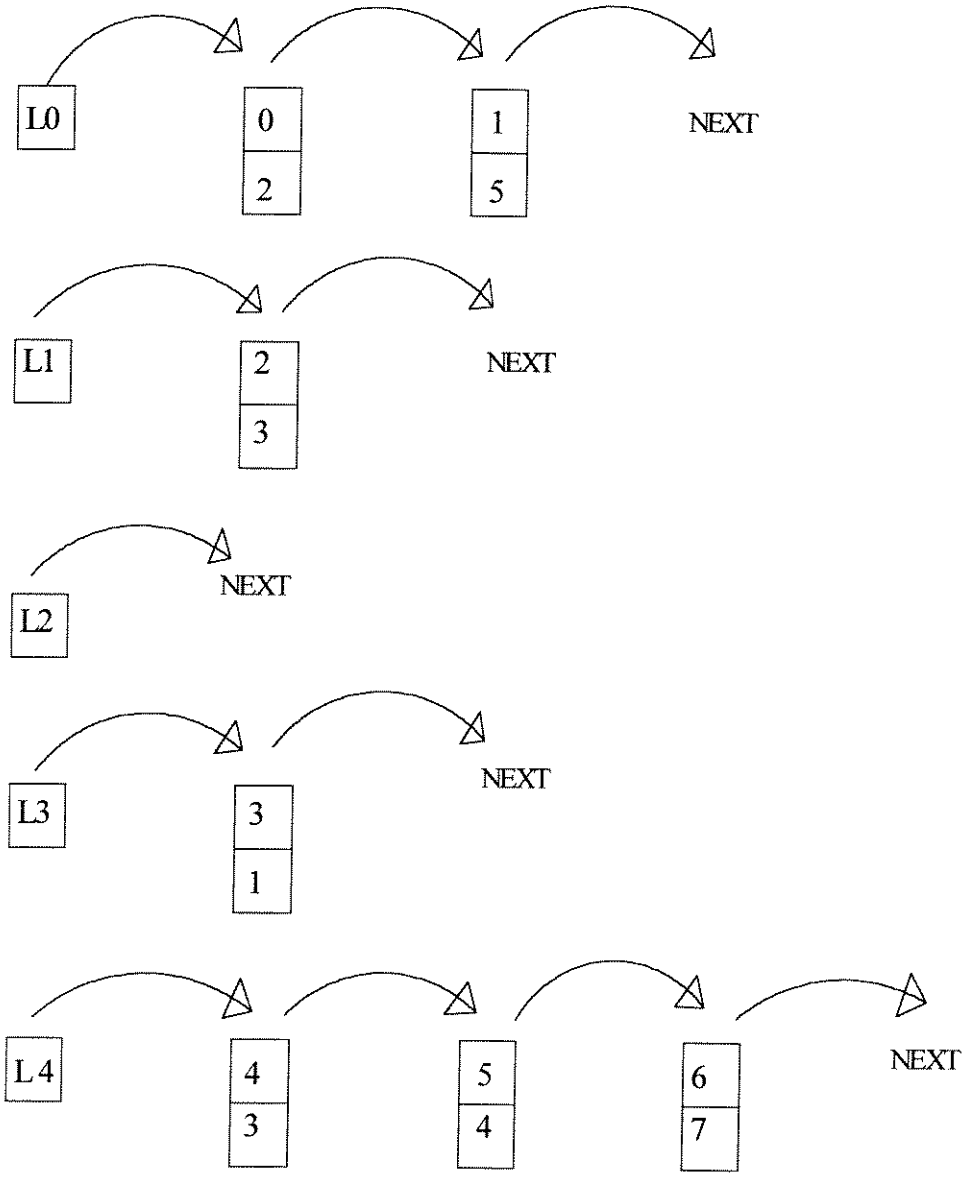

FIGURA 5-7- Esquema de alocação dos elementos da matriz esparsa.

Na FIGURA 5-7, L0, L1, L2, L3, e L4 são ponteiros que apontam para o primeiro elemento de cada linha. Assim, quando uma matriz é declarada, simplesmente deixa-se os ponteiros apontando para os primeiros elementos de cada linha, sem ser feita nenhuma alocação. Os elementos vão sendo alocados segundo vão sendo criados. Não é verificado se o elemento é diferente de zero antes de ser alocado, pois isso diminuiria o desempenho da alocação, e além disso, não há necessidade de alocar elementos nulos em uma matriz esparsa.

A entrada e a saída de elementos da matriz podem ser efetivadas através das funções Put e Get, respectivamente (que verificam se os elementos manipulados pertencem ao domínio da matriz) ou PutVal e GetVal, que não fazem nenhum tipo de verificação. Também poderia ser usada a função Input da classe **TMatrix**, que faz uma varredura da matriz inteira para ler os elementos. Porém, isso não teria sentido, já que uma matriz esparsa contem apenas alguns elementos diferentes de zero.

Esta matriz pode ser quadrada ou não, por isso, para declarar uma matriz deste tipo, é passado o número de linhas e o número de colunas da matriz. Para o exemplo anterior, tem-se a declaração *TSpMatrix a(5,8)*.

5.3.4. TMatRed

Esta classe é usada para condensação estática. Maiores informações encontram-se no capítulo "Subestruturação".

5.3.5. TSimMatrix

Para a implementação de matrizes simétricas foi derivada da classe base **TMatrix** a classe **TSimMatrix**. Esta classe também é uma classe base, na qual foram definidas todas as operações permitidas entre matrizes simétricas e matrizes não simétricas. As matrizes simétricas são as seguintes: **TSFMatrix** (*Symmetric Full Matrix*), **TSBMatrix**

(*Symmetric Banded Matrix*), **TSSpMatrix** (*Symmetric Sparse Matrix*) e **TSSkylMatrix** (*Symmetric Sky-Line Matrix*). Dentro da classe **TSimMatrix** estão implementadas operações de álgebra matricial e métodos para resolução de sistemas de equações usando as funções GetVal e PutVal. Para as matrizes simétricas, estão implementados dois métodos de resolução de sistemas, a decomposição de Cholesky, só se aplica a matrizes definidas positivas, e a decomposição LDLt, que se aplica a matrizes não definidas positivas também.

5.3.6. TSFMatrix

Esta classe gerencia matrizes do tipo simétricas cheias. A correspondência entre um elemento da matriz e um elemento alocado é dado pela seguinte expressão:

elemento_a(i,j)=alocação[i(i+1)/2+j]*

A FIGURA 5-8 mostra esquematicamente as alocações da matriz *A(4,4)*.

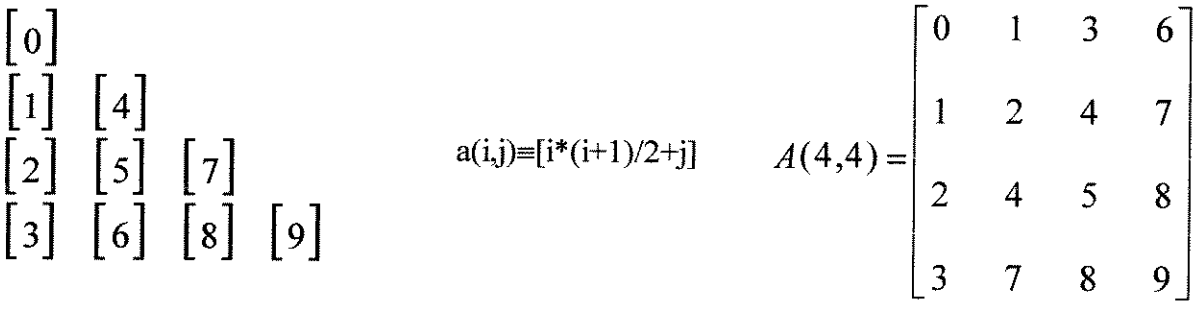

FIGURA 5-8- Esquema de alocação da classe TSFMatrix.

O construtor de uma matriz simétrica cheia tem apenas um argumento, pois toda matriz simétrica é quadrada. Dessa forma, ao invés de se declarar *TSFMatrix a(4,4)*, declara-se *TSFMatrix a(4)*.

5.3.7. TSBMatrix

Esta classe implementa o esquema de armazenamento para matrizes de banda simétrica. Para cada elemento na diagonal aloca-se dinamicamente um vetor do tamanho igual ao da banda da matriz. Isto foi feito com a intenção de se ganhar um pouco mais de desempenho na decomposição de matrizes. Os algoritmos usados para a decomposição por Cholesky e LDLt trabalham por colunas, portanto, uma alocação por colunas facilita o retorno da posição de cada elemento.

A localização do elemento em cada coluna é dada por uma simples operação de subtração entre a posição da coluna e a posição do elemento da diagonal, conforme mostrado na FIGURA 5-9. Por exemplo, para o elemento *a(3,4)* tem-se que a posição do mesmo é 4-3=1, no vetor correspondente à coluna 4.

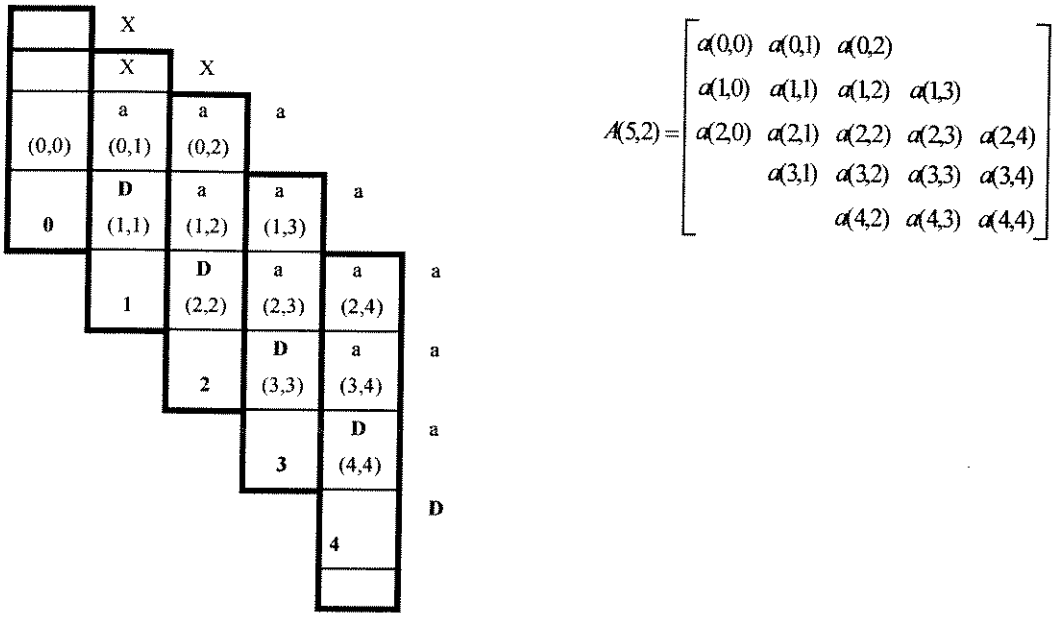

FIGURA 5-9- Esquema da alocação da classe TSBMatrix.

5.3.8. TSSpMatrix

Esta classe gerencia matrizes simétricas do tipo esparsa. A filosofia utilizada para a alocação deste tipo de matriz é igual à da classe **TSpMatrix** (matrizes esparsas não

simétricas), porém nesta classe as listas ligadas partem dos elementos da diagonal da matriz.

A declaração de uma matriz do tipo **TSSpMatrix** requer apenas um argumento, ou seja, a dimensão da matriz, uma vez que a matriz é simétrica e, portanto, quadrada.

5.3.9. TSkylMatrix

Esta classe gerencia matrizes do tipo "skyline". Quando um objeto do tipo **TSkylMatrix** é criado, são alocados vetores de tamanho igual a 1 sobre os elementos da diagonal. Isso é conseguido com auxílio de uma classe que foi denominada **TColuna**. A classe **TColuna** gerencia a alocação de um vetor redimensionável, sendo possível apenas incrementar o número de elementos do vetor e nunca diminuir, pois seria dispendioso em termos de tempo construir um outro vetor de tamanho menor, copiar os elementos do vetor um a um e eliminar o vetor anterior.

Um objeto do tipo **TColuna** guarda três informações importantes: os elementos da coluna (Vet), o tamanho da coluna (VetSize) e o tamanho máximo da coluna (Size), que depende da posição na diagonal onde a coluna será alocada. Os elementos da matriz são retornados segundo a seguinte expressão:

$$A(i,j)\ \forall\ i<j\ \begin{cases} 0.0 & se\ Dj.Size \le (j-i) \\ Dj.Vet(j-i) & se\ Dj.Size > (j-i) \end{cases}$$

5.4. GERENCIAMENTO DOS METODOS DE SOLUÇÃO DE SISTEMAS LINEARES

Conforme já foi mencionado, a classe **TMatrix** define um comportamento que as classes derivadas precisam implementar ou não dependendo de suas características próprias. A classe **TMatrix** declara e implementa diferentes métodos para a resolução de sistemas lineares, que as classes derivadas podem redefinir ou não, com a intenção de melhorar sua eficiência. Além dos métodos para solução de sistemas lineares definidos e

implementados, esta classe possui uma interface para os arquivos *header* da biblioteca TEMPLATES[3]. Como tal, a classe **TMatrix** define e implementa os seguintes métodos:

métodos diretos:

- decomposição de LU;
- decomposição de Cholesky;
- decomposição de LDLt;

métodos iterativos:

- método de Jacobi;
- SOR (*Successive Overrelaxation Method*);
- SSOR (*SymmetricSuccessive Overrelaxation Method*);
- CG (*Conjugate Gradient*) pré-condicionado;
- Método GMRES (*Generalized Minimal Residual Method*) pré-condicionado.

O pré-condicionador do CG e do GMRES é implementado como um objeto da classe **TSolver**. Geralmente é um método direto ou iterativo aplicado na mesma matriz objeto. Para facilitar o gerenciamento desta enorme quantidade de combinações, foi criada separadamente uma classe **TSolver**, que associa um objeto do tipo **TMatrix** a um método de solução. Como exemplo, pode-se imaginar o GMRES sendo condicionado por duas iterações do GC que por sua vez é pré-condicionado por uma iteração do SSOR.

5.5. TRATAMENTO DE BLOCOS.

A classe **TBlock** implementa uma divisão lógica da matriz em blocos de tamanho variável. O construtor tem três argumentos: um ponteiro para um objeto do tipo **TMatrix**, um inteiro que é o número de blocos na diagonal da matriz, e outro que é a dimensão de cada bloco na diagonal. A classe **TBlock** não modifica a matriz. Ela aumenta a capacidade de endereçar os elementos da matriz por blocos.

---

[3] Sobre a biblioteca TEMPLATES, ver BARRET et al. (1994).

O objeto do tipo **TBlock** contem duas informações básicas: a posição do bloco na diagonal e sua dimensão. A FIGURA 5-10 é um exemplo de uma matriz dividida em três blocos de diferentes dimensões: o bloco 0, de dimensões 2×2, o bloco 1, de 3×3, e o bloco 2 de 2×2.

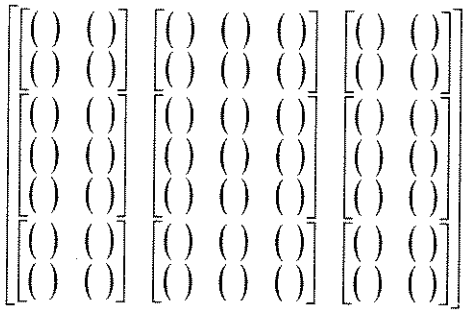

FIGURA 5-10- Esquema de matriz dividida em blocos.

As informações básicas são suficientes para calcular também a posição e a dimensão de qualquer bloco fora da diagonal, de acordo com o esquema mostrado na FIG. 11. Por exemplo, o bloco cuja posição é (1,2) tem dimensão 3x2 (no esquema, a linha 1 corresponde à dimensão 3, e a coluna 2 corresponde à dimensão 2).

Posição na Diagonal	0	1	2
Dimensão	2	3	2

FIGURA 5-11- Informações contidas no objeto Tblock.

Este construtor inicialmente só permite que os blocos da diagonal sejam todos do mesmo tamanho. Por exemplo, se forem feitas as seguintes declarações:

```
TFMatrix A(5,5)      //declaração de uma matriz cheia
TBlock b(&A,3,2)     //declaração do bloco b com 3 blocos na diagonal
                     //cada bloco com dimensão 2x2
                     //este bloco esta apontando para a matriz A
```

esquematicamente, será obtido o arranjo mostrado na FIGURA 5-12.

$$\begin{bmatrix} \begin{bmatrix} A(0,0) & A(0,1) \\ A(1,0) & A(1,1) \end{bmatrix} & \begin{bmatrix} A(0,2) & A(0,3) \\ A(1,2) & A(1,3) \end{bmatrix} & \begin{bmatrix} A(0,4) & A(0,5) \\ A(1,4) & A(1,5) \end{bmatrix} \\ \begin{bmatrix} A(2,0) & A(2,1) \\ A(3,0) & A(3,1) \end{bmatrix} & \begin{bmatrix} A(2,2) & A(2,3) \\ A(3,2) & A(3,3) \end{bmatrix} & \begin{bmatrix} A(2,4) & A(2,5) \\ A(3,4) & A(3,5) \end{bmatrix} \\ \begin{bmatrix} A(4,0) & A(4,1) \\ A(5,0) & A(5,1) \end{bmatrix} & \begin{bmatrix} A(4,2) & A(4,3) \\ A(5,2) & A(5,3) \end{bmatrix} & \begin{bmatrix} A(4,4) & A(4,5) \\ A(5,4) & A(5,5) \end{bmatrix} \end{bmatrix}$$

FIGURA 5-12- Arranjo em blocos obtido para a matriz exemplo A.

Agora, se na segunda declaração acima, os dois últimos argumentos do construtor não forem passados, o construtor assumirá o número de blocos na diagonal igual ao número de elementos na diagonal, ou seja, assumirá blocos de dimensão 1x1. Se forem feitas as declarações:

```
TFMatrix A(8,8)     //declaração de uma matriz cheia
TBlock b(&A)        //declaração do bloco b sem especificar
                    //o número de blocos nem o tamanho dos blocos
```

o arranjo dos blocos na matriz será o mostrado na FIGURA 5-13.

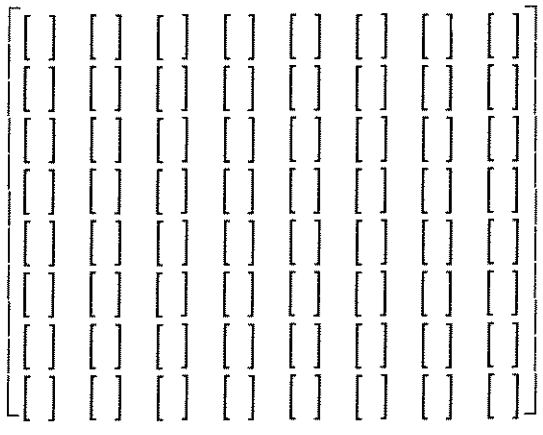

FIGURA 5-13- Novo arranjo obtido para a matriz exemplo.

Este tipo de "visualização" da matriz pode ser modificado utilizando-se duas funções. A primeira modificação, o número de blocos na diagonal, pode ser realizada com a função SetBlocks, que tem como argumento o novo número de blocos que o objeto do tipo **TBlock** terá.

Outra função utilizada para mudar o objeto do tipo **TBlock** é Set, que utiliza três argumentos. O primeiro argumento fornece a posição na diagonal do bloco a ser modificado, o segundo argumento, o novo tamanho do bloco a ser modificado; por último, com o terceiro argumento é fornecida a posição relativa do primeiro elemento do bloco, com respeito à sua posição na diagonal da matriz.

Em seguida mostra-se como modificar o arranjo dos blocos da FIGURA 5-12.

```
b.SetBlocks(4);        //modificação do número de blocos na diagonal
b.Set(0,2,0);          //o bloco 0 tem dimensão 2
                       //e o primeiro elemento do bloco está na
                       //posição 0 da diagonal da matriz.
b.Set(1,3,2);          //o bloco1 tem dimensão 3
                       //e o primeiro elemento do bloco está na
                       //posição 2 da diagonal da matriz.

b.Set(2,1,5);          //o bloco 2 tem dimensão 1
                       //e o primeiro elemento do bloco está na
                       //posição 5 da diagonal da matriz
b.Set(3,2,6);          //o bloco 3 tem dimensão 2
                       //e o primeiro elemento do bloco está na
                       //posição 6 da diagonal da matriz
```

Esquematicamente, será obtido o novo arranjo mostrado na FIGURA 5-14.

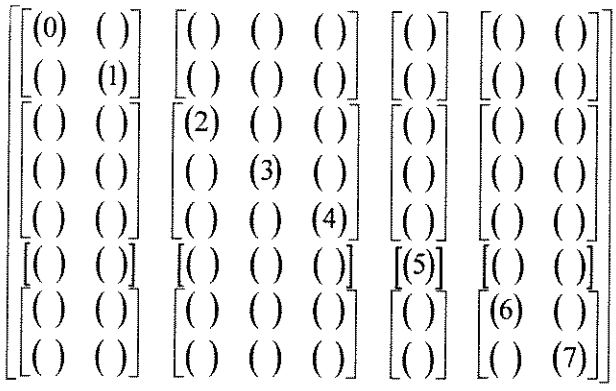

FIGURA 5-14- Novo arranjo para a matriz exemplo.

Uma observação importante é que esta modificação tem que ser de uma maneira ordenada, ou seja, primeiro tem que ser mudado o número de blocos, depois o tamanho do primeiro bloco, depois o tamanho do segundo bloco, e assim por diante até o último bloco.

O mesmo procedimento anterior pode ser feito usando-se a função SetAll(dimensions) e a função Resequence(start). A primeira função passa para o objeto do tipo **TBlock** as dimensões dos blocos na diagonal através do vetor de inteiros ***dimensions***. A segunda função faz a seqüência dos blocos a partir da posição start (este argumento tem como padrão a posição 0).

Observa-se que a matriz não foi modificada; ela continua existindo na mesma forma com que foi criada.

Outras funções também permitem que os diferentes blocos sejam copiados para outras matrizes. Também é permitida a operação de igualar dois objetos do tipo **TBlock** e de extrair um dado bloco da matriz. Todas essas e outras funções estão documentadas no arquivo *header* da classe.

6. SUBESTRUTURAÇÃO

6.1. CONCEITOS BÁSICOS

A modelagem de um problema de grande escala pode ser organizada pela discretização da equação diferencial em elementos finitos dentro de subdomínios. A idéia central é dividir o problema para resolvê-lo, i. é., "dividir para conquistar".

Em cada subdomínio tem-se nós internos e nós de interface (ou de fronteira). Entende-se por nós internos aqueles que só pertencem a um subdomínio, e nós de fronteira aqueles que pertencem a mais de um subdomínio. AINSWORTH (1996) mostra uma visão variacional da condensação estática. A seguir, usa-se essa técnica para reduzir os nós internos de cada subdomínio sobre seus nós de interface num problema de Dirichlet para uma equação uniformemente elíptica de segunda ordem sobre um domínio poligonal (R^2).

Propõe-se resolver o seguinte problema:

$$\begin{aligned} Lu &= f && em\ \Omega, \quad \Omega \subset R^2 \\ u &= 0 && em\ \partial\Omega \end{aligned}$$

onde:

$$Lu = \sum_{i,j=1}^{2} \frac{\partial}{\partial x_i}\left(a_{ij} \frac{\partial u}{\partial x_i} \right)$$

A formulação fraca deste problema consiste em encontrar $u \in H_0^1(\Omega)$ tal que:

$$a_\Omega(u,v) = f(v) \qquad \forall v \in H_0^1(\Omega)$$

Nessa expressão $H_0^1(\Omega)$ é um subespaço de funções que tem valores de contorno nulos e com sua primeira derivada quadrado integrável. Este subespaço pertence ao espaço de Sobolev $H^1(\Omega)$.

O domínio é dividido em subdomínios que, por sua vez, são divididos em elementos. Admite-se que as funções de forma são regulares e que os subdomínios não são superpostos.

Seja $V^H(\Omega) \subset H_0^1(\Omega)$ e $V^h(\Omega) \subset H_0^1(\Omega)$ espaços de funções hierárquicas sobre os subdomínios e sobre os elementos respectivamente. Então a formulação fraca pode ser escrita na seguinte forma discreta:

$$a_\Omega(u_h, v_h) = f(v_h) \qquad \forall v \in V^h(\Omega)$$

Com a introdução das funções de base $\{\varphi_i\}$ do espaço V^h é possível usar as funções de forma $u_h = \sum x_i \varphi_i$ e $v_h = \sum x_j \varphi j$ para formar o seguinte sistema de equações:

$$a_\Omega(\varphi_i, \varphi_j) x_i = f(\varphi_j)$$ [1]

Uma vez que a integral sobre todo o domínio Ω pode ser escrita como a soma de integrais sobre subdomínios, a matriz de rigidez pode ser calculada como a soma das integrais sobre os subdomínios. Sobre cada subdomínio s pode-se definir o seguinte problema:

$$a_\Omega^s(\varphi_i, \varphi_j) x_i^s = f^s(\varphi_j)$$

As funções de forma $\{\varphi_i\}$ de cada subdomínio são divididas em dois conjuntos. Um conjunto é das funções de base $\{\varphi_i\}$, $i \in \mathrm{I}$, apoiadas no interior do subdomínio. Outro conjunto é das funções de base $\{\varphi_i\}$, $i \in \tilde{\mathrm{I}}$, que tem valores diferentes de zero em pelo menos numa das interfaces $\partial\Omega_H$ do subdomínio. Na FIGURA 6-1 é mostrado um exemplo com dois subdomínios usando funções lineares de interpolação.

---

[1] Na engenharia $a_\Omega(\varphi_i, \varphi_j)$ é conhecida como matriz de rigidez e $f(\varphi_j)$ vetor de cargas, devido a sua origem na análise matricial das estruturas.

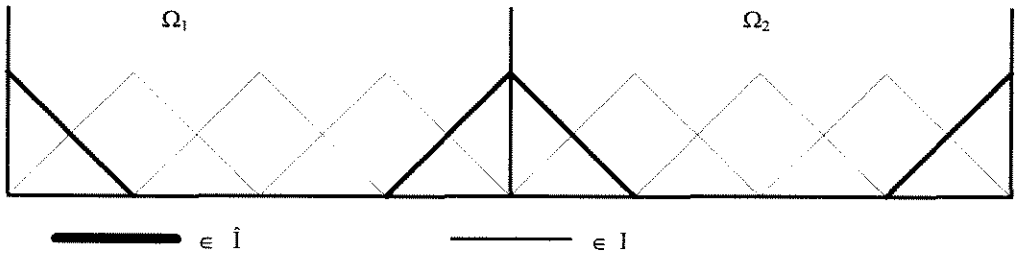

FIGURA 6-1- Divisão das funções de forma em dois conjuntos.

A condensação estática dos graus de liberdade dos nós internos a cada subdomínio pode ser interpretada como uma mudança de base. As funções $\{\varphi_i\}, i \in \tilde{I}$ das interfaces são substituídas por novas funções $\{\omega_i\}, i \in \tilde{I}$, que são ortogonais às funções, $\{\varphi_i\}, i \in I$ e têm os mesmos valores que as funções $\{\varphi_i\}, i \in \tilde{I}$ nas interfaces. Estas duas condições podem ser escritas respectivamente como:

$$a_\Omega^s(\omega_i, \varphi_j) = 0 \quad \forall \; j \in I$$

$$\omega_i = \varphi_i \qquad sobre \;\; \partial\Omega_H$$

Para maior facilidade de manipulação, as funções de forma serão agrupadas em vetores segundo:

$$\varphi_I = [\varphi_i]_{i \in I}, \qquad \varphi_{\tilde{I}} = [\varphi_i]_{i \in \tilde{I}}, \qquad \omega_{\tilde{I}} = [\omega_i]_{i \in \tilde{I}}$$

As funções φ_I permanecem inalteradas e as novas funções $\omega_{\tilde{I}}$ são obtidas subtraindo-se $T\varphi_I$ das funções $\varphi_{\tilde{I}}$, ou seja:

$$\omega_{\tilde{I}} = \varphi_{\tilde{I}} - T\varphi_I \qquad\qquad \varphi_I = \varphi_I$$

sendo T uma matriz apropriada. As expressões anteriores podem ser escritas numa forma matricial:

$$\begin{bmatrix} \varphi_I \\ \omega_{\tilde{I}} \end{bmatrix} = \begin{bmatrix} I & 0 \\ -T & I \end{bmatrix} \begin{bmatrix} \varphi_I \\ \varphi_{\tilde{I}} \end{bmatrix}$$

A matriz da transformação de bases é a seguinte:

$$\mathrm{P}=\begin{bmatrix} I & 0 \\ -T & I \end{bmatrix}$$

Escrevendo-se $a_{\Omega}^{s}\left(\varphi_i,\varphi_j\right)$ em blocos, tem-se:

$$\begin{bmatrix} a_{\mathrm{II}} & a_{\mathrm{I}\tilde{\mathrm{I}}} \\ a_{\tilde{\mathrm{I}}\mathrm{I}} & a_{\tilde{\mathrm{I}}\tilde{\mathrm{I}}} \end{bmatrix}$$

A condição $a_{\Omega}^{s}\left(\omega_i,\varphi_j\right)=0 \quad \forall\ j \in \mathrm{I}$ indica que a matriz P é a transformação ortogonal dos nós internos com os nós de interface, ou seja, diagonaliza a matriz $a_{\Omega}^{s}\left(\varphi_i,\varphi_j\right)$. Usando-se esta condição tem-se:

$$\begin{bmatrix} I & 0 \\ -T & I \end{bmatrix}\begin{bmatrix} a_{\mathrm{II}} & a_{\mathrm{I}\tilde{\mathrm{I}}} \\ a_{\tilde{\mathrm{I}}\mathrm{I}} & a_{\tilde{\mathrm{I}}\tilde{\mathrm{I}}} \end{bmatrix}\begin{bmatrix} I & -T \\ 0 & I \end{bmatrix}=\begin{bmatrix} d_{\tilde{\mathrm{I}}\tilde{\mathrm{I}}} & 0 \\ 0 & d_{\mathrm{II}} \end{bmatrix}$$

ou:

$$\begin{bmatrix} a_{\mathrm{II}} & a_{\mathrm{I}\tilde{\mathrm{I}}}-a_{\mathrm{II}}T \\ a_{\tilde{\mathrm{I}}\mathrm{I}}-Ta_{\mathrm{II}} & a_{\tilde{\mathrm{I}}\tilde{\mathrm{I}}}-Ta_{\mathrm{I}\tilde{\mathrm{I}}}-a_{\tilde{\mathrm{I}}\mathrm{I}}T+Ta_{\mathrm{II}}T \end{bmatrix}=\begin{bmatrix} d_{\tilde{\mathrm{I}}\tilde{\mathrm{I}}} & 0 \\ 0 & d_{\mathrm{II}} \end{bmatrix}$$

Igualando-se as matrizes fora da diagonal a zero, encontra-se a expressão de T:

$T=a_{\tilde{\mathrm{I}}\mathrm{I}}\,a_{\mathrm{II}}^{-1}$.

Também tem-se que $a_{\mathrm{I}\tilde{\mathrm{I}}}=a_{\tilde{\mathrm{I}}\mathrm{I}}$ é uma condição necessária para diagonalizar a matriz $a_{\Omega}^{s}\left(\varphi_i,\varphi_j\right)$, i. é., a matriz P não é a transformação ortogonal de a_{Ω}^{s} se esta não é simétrica.

A matriz resultante da diagonalização é:

$$\begin{bmatrix} a_{\mathrm{II}} & 0 \\ 0 & s \end{bmatrix}$$

A matriz $s = a_{\tilde{I}\tilde{I}} - a_{\tilde{I}I} a_{II}^{-1} a_{I\tilde{I}}$ é chamada de complemento de Shur[2].

Escrevendo-se o sistema de equações do subdomínio em forma de blocos, tem-se a seguinte expressão:

$$\begin{bmatrix} a_{II} & a_{I\tilde{I}} \\ a_{\tilde{I}I} & a_{\tilde{I}\tilde{I}} \end{bmatrix} \begin{bmatrix} x_I \\ x_{\tilde{I}} \end{bmatrix} = \begin{bmatrix} f_I \\ f_{\tilde{I}} \end{bmatrix}$$

Aplicando a transformação de bases obtemos o seguinte sistema:

$$\begin{bmatrix} I & 0 \\ -T & I \end{bmatrix} \begin{bmatrix} a_{II} & a_{I\tilde{I}} \\ a_{\tilde{I}I} & a_{\tilde{I}\tilde{I}} \end{bmatrix} \begin{bmatrix} x_I \\ x_{\tilde{I}} \end{bmatrix} = \begin{bmatrix} I & 0 \\ -T & I \end{bmatrix} \begin{bmatrix} f_I \\ f_{\tilde{I}} \end{bmatrix}$$

ou:

$$\begin{bmatrix} a_{II} & a_{I\tilde{I}} \\ 0 & s \end{bmatrix} \begin{bmatrix} x_I \\ x_{\tilde{I}} \end{bmatrix} = \begin{bmatrix} f_I \\ f_{\tilde{I}} - a_{I\tilde{I}} a_{II} f_I \end{bmatrix}$$

Este problema pode ser resolvido em dois passos:

1.Cálculo dos graus de liberdade dos nós externos ao subdomínio:

$$[s]\,[x_{\tilde{I}}] = [f_{\tilde{I}} - a_{I\tilde{I}} a_{II} f]$$

2.Cálculo dos graus de liberdade dos nós internos ao subdomínio:

$$[a_{II}]\,[x_I] = [f_{\tilde{I}} - a_{I\tilde{I}} x_{\tilde{I}}]$$

Com cada sistema do passo 1, constrói-se um sistema de equações global que permitirá calcular os graus de liberdade de todas as interfaces $\partial\Omega_i$ do domínio inteiro, expresso por:

$$A x_I = F$$

---

[2] Ver CHAN et al. (1990), p. 7.

onde: $A = \sum_{i}^{\Omega_n} s_i$

$$F = \sum_{i}^{\Omega_n} \left[f_{\tilde{\Gamma}} - a_{\Pi\tilde{\Gamma}} a_{\Pi} f \right]_i$$

Este novo sistema é chamado de sistema reduzido ou sistema de interface, e pode ser resolvido por métodos iterativos pré-condicionados ou diretos. Depois de se ter calculado os graus de liberdade dos nós de interface, passa-se a calcular os graus de liberdade dos nós internos segundo o passo 2.

Um dos maiores desafios da computação paralela é resolver o sistema global de equações com o menor custo de comunicação entre processadores. Em LE TALLEC, SALTEL e VIDRASCU (1994, p. 139), encontra-se o seguinte comentário com respeito ao sistema global sem subestruturação:

> *"Este sistema freqüentemente é muito mal condicionado. O cálculo da matriz de rigidez e a solução do sistema por métodos diretos é extremamente dispendioso em tempo e memória. O desenvolvimento de algoritmos paralelos efetivos para resolver este problema é um desafio. Métodos de decomposição de domínios, que misturam métodos diretos para as matrizes locais e métodos iterativos para as globais de interface, resultam numa resposta muito boa para este desafio."*

A subestruturação usa inicialmente métodos diretos em cada subdomínio e métodos iterativos para resolver o sistema reduzido. Nota-se que a redução estática pode ser feita em paralelo em cada subdomínio e que o sistema reduzido pode ser resolvido com o gradiente conjugado pré-condicionado, que é altamente paralelizável.

A seguir mostra-se a implementação computacional da subestruturação desenvolvida neste trabalho.

6.2. IMPLEMENTAÇÃO COMPUTACIONAL

6.2.1. CONDENSAÇÃO ESTÁTICA: A CLASSE TMatRed

Numa modelagem orientada a objetos de problemas que usam partições de domínios em subdomínios não sobrepostos, é necessário que exista um objeto que seja capaz de eliminar os nós internos destes subdomínios. Este objeto reduz o problema original para o problema de interface que é resolvido por métodos iterativos.

A classe **TMatRed** (matriz reduzida), desenvolvida neste trabalho, implementa sistemas de equações quadrados simétricos ou não simétricos com vetor de carga. Oferece ao usuário a opção de dividir as equações em dois conjuntos: um conjunto de equações internas e outro de equações externas. Num estado inicial, um objeto do tipo **TMatRed** tem um comportamento de um objeto do tipo **TMatrix**. A transição desse estado para o de uma matriz condensada reduz o conjunto de equações internas sobre o conjunto de equações externas.

Nesta classe a matriz é dividida em quatro submatrizes, que são implementadas por quatro ponteiros. A primeira submatriz da diagonal, $[K_{II}]$, aponta para um objeto do tipo **TMatrix,** e as outras apontam para **TFMatrix** (matriz cheia), conforme mostra a FIGURA 6-2.

Nesta classe o vetor de carga também é dividido em duas submatrizes, sendo que cada uma aponta para objetos do tipo **TFMatrix** (ver FIGURA 6-2).

$$\begin{bmatrix} [*\,\text{TMatrix}\ K_{II}] & [*\,\text{TFMatrix}\ K_{IC}] \\ [*\,\text{TFMatrix}\ K_{CI}] & [*\,\text{TFMatrix}\ K_{CC}] \end{bmatrix} \begin{bmatrix} [*\,TFMatrix\, U_I] \\ [*\,TFMatrix\, U_C] \end{bmatrix} = \begin{bmatrix} [*\,TFMatrix\, f_I] \\ [*\,TFMatrix\, f_C] \end{bmatrix}$$

FIGURA 6-2- Submatrizes de um objeto TMatRed.

Da FIGURA 6-2 define-se a condensação dos graus de liberdade U_I sobre os graus de liberdade U_C com as seguintes operações matriciais:

$$[K_{CC}]_{RED} = [K_{CC}] - [K_{CI}][K_{II}]^{-1}[K_{IC}]$$
$$[f_C]_{RED} = [f_C] - [K_{CI}][K_{II}]^{-1}[f_I]$$

As matrizes $[K_{CC}]_{RED}$ e $[f_C]_{RED}$ são armazenadas em $[K_{CC}]$ e $[f_C]$ respectivamente. O estado reduzido de um objeto tipo **TMatRed** é mostrado na FIGURA 6-3. Nesse estado a matriz K_{II} está decomposta ($[K_{II}]_{DECOMP}$), e as matrizes resultantes dessa decomposição estão armazenadas na própria matriz K_{II}.

Também em f_I é armazenado $[K_{II}]^{-1}[f_I]$, e em $[K_{IC}]$ é armazenado $[K_{II}]^{-1}[K_{IC}]$ (ver FIGURA 6-3).

$$\begin{bmatrix} [K_{II}]_{DECOMP} & [K_{II}]^{-1}[K_{IC}] \\ [K_{CI}] & [K_{CC}]_{RED} \end{bmatrix} \begin{bmatrix} [U_I] \\ [U_C] \end{bmatrix} = \begin{bmatrix} [K_{II}]^{-1}[f_I] \\ [f_C]_{RED} \end{bmatrix}$$

FIGURA 6-3- Objeto TMatRed no estado reduzido.

As operações de redução permitem que possa ser definido o seguinte sistema reduzido:

$$[K_{CC}]_{RED}[U_C] = [f_C]_{RED}$$

Poderiam ser utilizados tipos de matrizes com esparsidades especiais para representar as submatrizes. Por exemplo, no caso da matriz de rigidez do subdomínio ser uma matriz tipo banda, $[K_{IC}]$ e $[K_{CI}]$ serão matrizes triangulares inferior e triangular superior de um tipo especial, onde os elementos não nulos iriam até uma subdiagonal, conforme mostra a FIGURA 6-4. Ainda na mesma figura observa-se que $[K_{II}]$ e $[K_{CC}]$, dependendo da largura da banda, podem ser matrizes em banda ou não. Porém, toda esta esparsidade é perdida no cálculo de $[K_{CC}]_{RED}$ e $[f_C]_{RED}$, pois são realizadas operações de

multiplicação e subtração antes do armazenamento do resultado destas operações em $[K_{CC}]$ e $[f_C]$. Com isso conclui-se que não é vantajoso usar matrizes especiais para armazenar $[K_{IC}]$, $[K_{CI}]$ e $[K_{CC}]$.

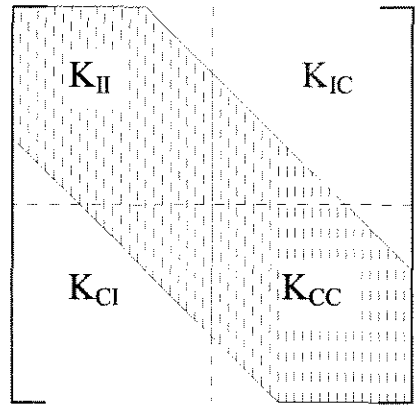

FIGURA 6-4- Matriz de rigidez local ao subdominio banda.

OBSERVAÇÕES

1. A matriz $[K_{II}]$ é decomposta apenas uma vez. Os cálculos de $[K_{II}]^{-1}[F_I]$ e de $[K_{II}]^{-1}[K_{IC}]$ equivalem a resolver $[K_{II}][y_1]=[F0]$ e $[K_{II}][y_2]=[K_{IC}]$;
2. O ponteiro $*[K_{II}]$ para um objeto do tipo **TMatrix** pode ser inicializado com diferentes tipos de matrizes simétricas ou não simétricas, como matriz banda, "skyline", etc;
3. As observações acima indicam que a liberdade da escolha de $[K_{II}]$ permite usar a otimização incorporada nas matrizes derivadas de **TMatrix,** no que se refere a resolução de equações, já que a submatriz $[K_{II}]$ precisa ser invertida.

6.2.2. SUBESTRUTURAÇÃO: A CLASSE TSuperEl

O comportamento de um subdomínio é de malha e elemento. Tem-se um comportamento de malha, pois existem nós internos num subdomínio. Por outro lado a própria malha também é um elemento dentro de uma malha de subdomínios. Para abstrair este comportamento híbrido de elemento e malha, foi desenvolvida neste trabalho a classe **TSuperEl** (superelemento), que é derivada das classes **TCompEl** e **TCompGrid**.

Uma vez que um objeto do tipo **TSuperEl** é elemento, ele pode pertencer a uma malha que seja outro **TSuperEl**, o que possibilita criar super-elementos dentro de outros super-elementos. A FIGURA 6-5 mostra um exemplo em que a malha é dividida em quatro super-elementos, S0, S1, S2 e S3, sendo que o super-elemento S1 contém outros quatro super-elementos (Ss0, Ss1, Ss2, Ss3).

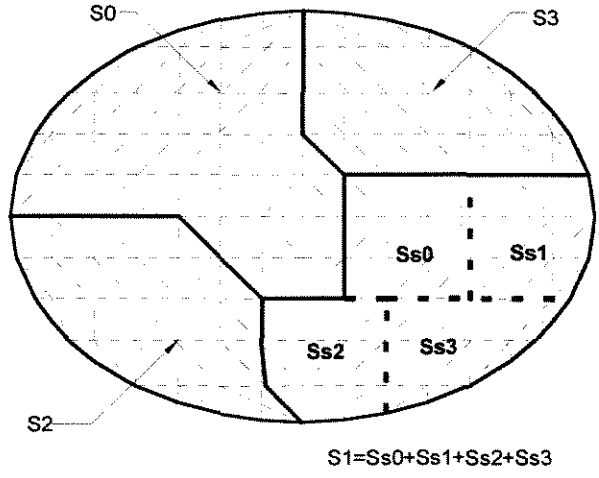

FIGURA 6-5- Malha com quatro super-elementos.

Conforme já foi mencionado, a classe **TSuperEl** abstrai dois estados do comportamento de subdomínios. Inicialmente tem-se o estado não condensado, em que a malha à qual o objeto pertence vê o super-elemento como um elemento com grande número de nós e equações. O outro estado é o condensado, em que o super-elemento torna visível apenas os nós com conectividade externa. Neste estado, a malha à qual o super-elemento pertence vê apenas os nós de interface do subdomínio. A transição de um estado para outro é feita pelo método **ReduceInternalNodes()**, método da classe **TCompEl**, mas implementado apenas na classe **TSuperEl**.

Na criação da classe **TSuperEl**, os métodos e características da classe **TCompGrid** foram herdados sem modificação. Já o comportamento de elemento proveniente da classe **TCompEl** teve que ser implementada internamente, especialmente no que se refere à montagem da matriz local de cada subdomínio e a redução dos seus nós internos. O objeto do tipo **TSuperEl** tem como variável membro um ponteiro para

um objeto do tipo **TMatRed**, que implementa todas as operações algébricas necessárias para a redução estática do sistema de equações.

Após serem reduzidos os nós internos, a malha vê somente os nós de fronteira do super-elemento. Isso permite montar um sistema de equações de forma "convencional", que obviamente permitirá somente o cálculo dos nós de interface. O sistema reduzido (ou problema de interface) é expresso por:

$$\sum_{s=1}^{Ns} K^{s}{}_{\mathrm{RED}}\, U_{\mathrm{interface}} = \sum_{s=1}^{Ns} F^{s}{}_{\mathrm{RED}}$$

Este sistema ainda pode ser grande, no âmbito do processamento em paralelo. A eficiência da resolução deste sistema por métodos diretos não necessariamente aumentará com o número de processadores.

Segundo FARHAT (1994, p. 145), KEYES (1992, p. 293) comenta sobre a decomposição de domínios (DD) o seguinte:

> *"Para muitos desses algoritmos e para problemas suficientemente grandes, o custo de cada iteração pode ser diminuído quase linearmente com o número de computadores disponíveis. Entretanto, uma vez que incrementar o número de subdomínios é a maneira mais simples de incrementar o grau de paralelismo para o método de DD, a característica mais interessante dos algoritmos de DD para processamento paralelo pesado é que o número de iterações não cresce significativamente com o número de subdomínios".*

Isto define claramente a escalabilidade da decomposição de domínios (ou subestruturação). Ainda na mesma publicação, FARHAT (1994, p. 142) faz uma análise de um algoritmo para resolver matrizes "skyline" usando processamento paralelo em métodos diretos, apresentado por FARHAT e WILSON (1988) e conclui que:

> *"A análise da escalabilidade apresentado aqui mostra que o paralelismo em métodos diretos para resolver sistemas "skyline" está limitado pelo problema da banda e não pelo tamanho do problema, e que estes métodos diretos não são escaláveis quando são aplicados a problemas de análise estrutural."*

Do anteriormente exposto, justifica-se o uso de métodos iterativos, tais como o gradiente conjugado pré-condicionado (quando o sistema é simétrico).

6.3. TESTES NUMÉRICOS

Inicialmente foram comparados os resultados do mesmo problema mas utilizando-se malhas sem subdomínios e malhas com subdomínios regulares (primeiro e segundo testes). Em seguida foram usados domínios irregulares para problemas elásticos (terceiro teste), tendo sido os resultados coerentes com os esperados. Todos esses testes foram realizados num computador PC-Pentium com 133 Mhz de processamento.

6.3.1. PRIMEIRO TESTE

Como primeiro teste foi resolvido um problema com condições de contorno de Dirichlet num domínio regular (ver FIGURA 6-6). Na primeira rodada foram usados apenas 144 elementos triangulares com uma aproximação quadrática. Na segunda também foram usados 144 elementos mas foram utilizados 4 subdomínios, tendo sido novamente usada uma aproximação quadrática.

Na última rodada foram usados 1024 elementos e o domínio foi dividido em 256 subdomínios, tendo cada subdomínio 4 elementos. Estas malhas foram geradas pela classe **TGenGrid**, que gera domínios retangulares e os divide em subdomínios.

O problema resolvido foi o seguinte:

$$-\Delta u(x,y)+1600u(x,y)=+1600(x^3y+xy^3) \text{ em } \Omega,$$

com u=0 em Γ_1 Γ_2 Γ_3 Γ_4 .

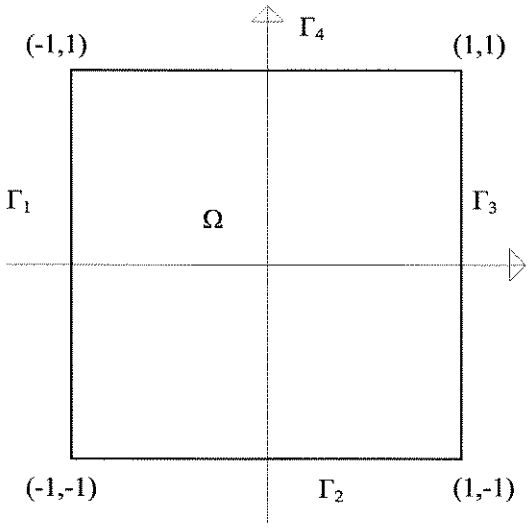

FIGURA 6-6- Primeiro teste numérico.

Primeira execução: Foi usada aproximação quadrática, numa malha de 144 elementos (169 equações). O elemento mestre foi um elemento triangular, tendo sido obtido o resultado mostrado na FIGURA 6-7.

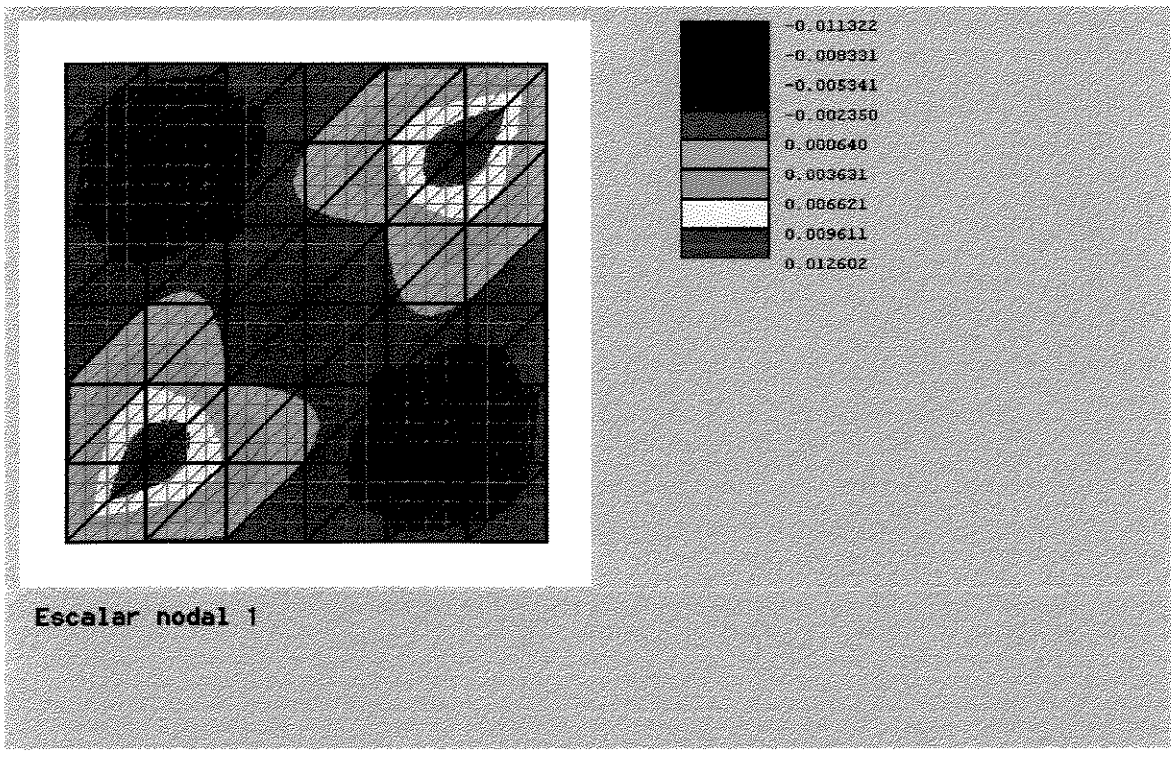

FIGURA 6-7- Resultado obtido na primeira execução.

Segunda execução: Foi usada aproximação quadrática, numa malha de 144 elementos e quatro superelementos, de 64, 16, 16 e 4 elementos. O elemento mestre foi um elemento triangular, tendo sido obtido o resultado mostrado na FIGURA 6-8.

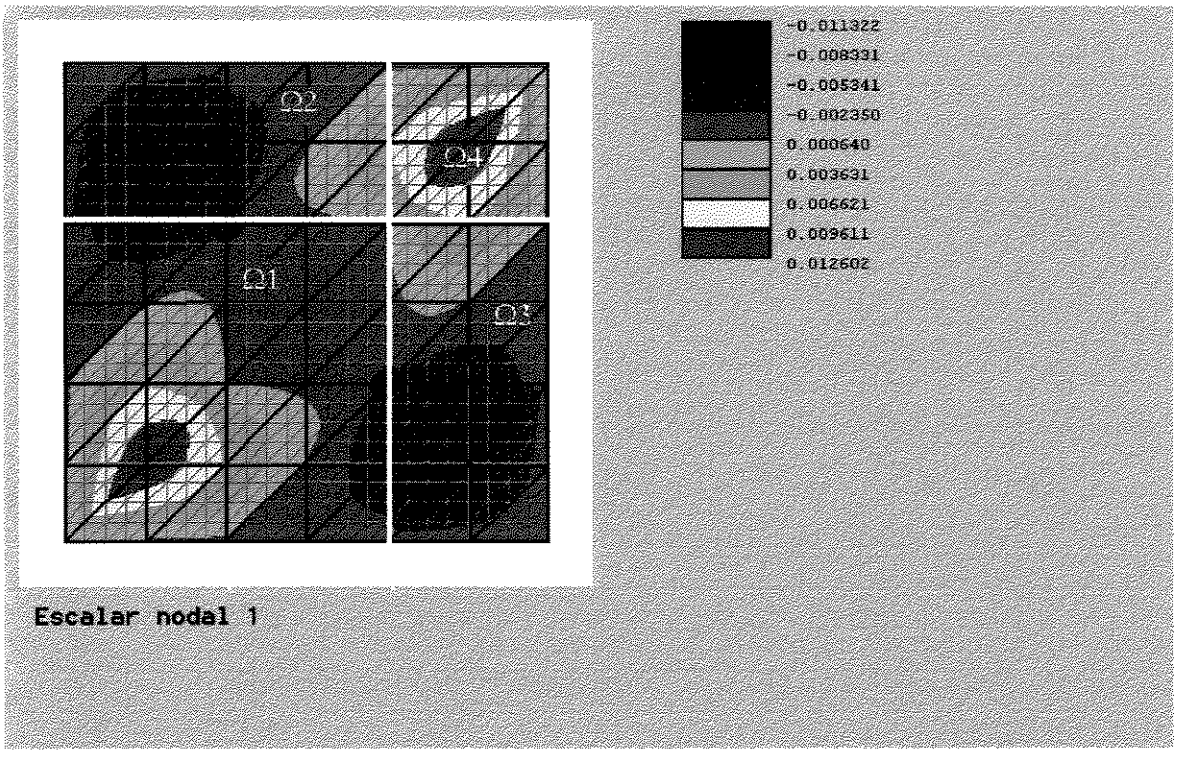

FIGURA 6-8- Resultado obtido na segunda execução.

Terceira execução: foi usada uma malha de 1024 elementos (9409 equações) com superelementos de 4 elementos, tendo sido o elemento mestre um elemento quadrado. Foi usada aproximação cúbica. O resultado obtido foi o mostrado na FIGURA 6-9.

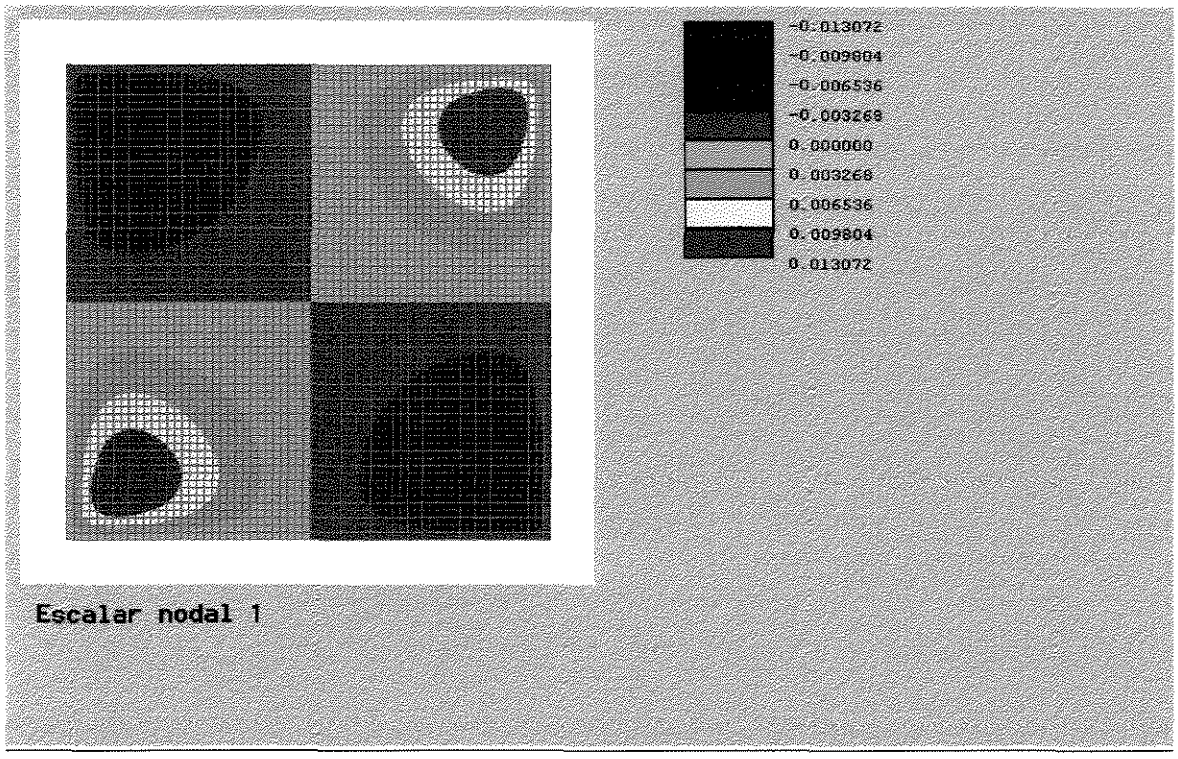

FIGURA 6-9-Resultado obtido na terceira execução.

6.3.2. SEGUNDO TESTE

Conhecendo-se a função solução, foi criada uma equação diferencial, foi calculada a solução numérica desta equação diferencial, e em seguida foram comparadas as soluções. O gráfico da função solução está mostrado na FIGURA 6-10.

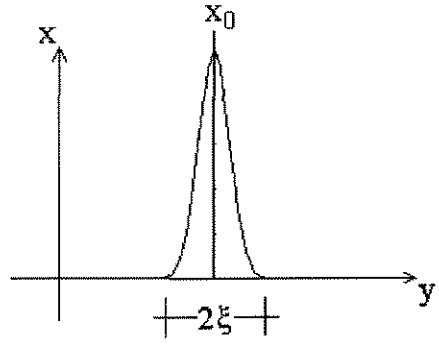

FIGURA 6-10- Função exata usada no segundo teste.

Esta função é governada pela seguinte equação:

$$v(x, x_0, \xi) = -e^{\frac{-3(\xi - x_0)^2}{\xi^2}} + e^{\frac{-3(x - x_0)^2}{\xi^2}}$$

Observa-se que esta função é par, e tem a forma de um sino. Sua abertura é representada por 2ξ, a variável x_0 representa o deslocamento do eixo de simetria com relação à origem das coordenadas. Colocando-se o eixo de simetria na reta $y=0$ e uma abertura de 2 (ξ=1), obtém-se a seguinte expressão:

$$v(x,0,1) = -e^{-3} + e^{-3x^2}$$

O gráfico desta função é o mostrado na FIGURA 6-11.

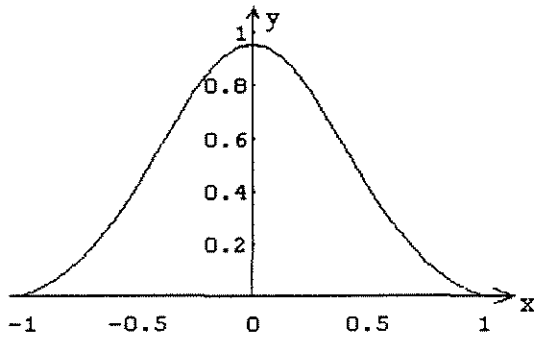

FIGURA 6-11- Representação gráfica da função exata para x_0 =0 e ξ=1.

Ainda com essa expressão, pode ser definida uma função bidimensional, dada por:

$$u(x, y) = v(y,0,1) \cdot v(x,0,1) = \left(-e^{-3} + e^{-3x^2}\right)\left(-e^{-3} + e^{-3y^2}\right)$$

O gráfico bidimensional desta função pode ser visto na FIGURA 6-12.

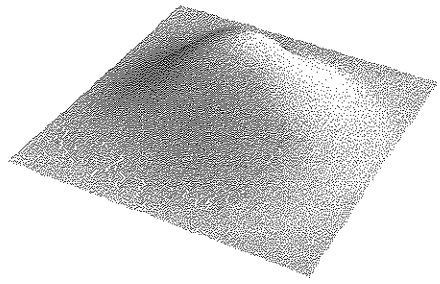

FIGURA 6-12- Gráfico da função exata bidimensional.

Com esta função pode-se definir o problema-teste:

$$-\Delta u(x,y) = \frac{6\left(-e^{-3}+e^{-3x^2}\right)}{e^{3y^2}} + \frac{6\left(-e^{-3}+e^{-3y^2}\right)}{e^{3x^2}} - \frac{36\left(e^{-3}+e^{-3y^2}\right)x^2}{e^{3x^2}} - \frac{36\left(e^{-3}+e^{-3x^2}\right)y^2}{e^{3y^2}} \text{ em } \Omega,$$

com $u = 0$ em Γ .

Para este teste utilizou-se o mesmo domínio que o teste anterior. O mesmo foi discretizado utilizando-se 400 elementos quadrados. Esses elementos foram agrupados em quatro subdomínios de 100 elementos cada um. Foi usada uma interpolação de grau 6 (14.641 equações). O resultado numérico gerou o gráfico mostrado na FIGURA 6-13. Segundo a análise das FIGURAS 6-12 e 6-13, pode-se considerar o resultado deste teste como satisfatório

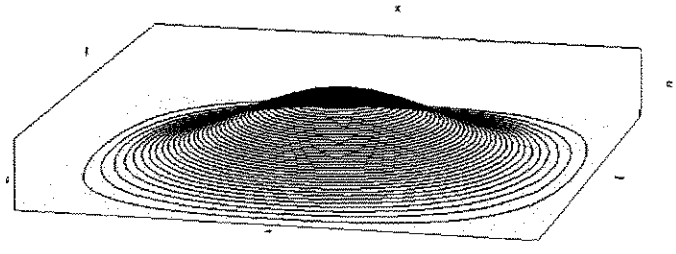

FIGURA 6-13- Gráfico gerado pela aproximação numérica.

6.3.3. TERCEIRO TESTE

Como terceiro teste, foi resolvido um problema elástico, cuja formulação variacional pode ser expressa por:

$$a_\Omega(u_h, v_h) = f(v_h) \qquad \forall v \in V^h(\Omega)$$

onde: $a_\Omega(u,v) = \int_\Omega E^{ijkl} \frac{\partial u_i}{\partial x_j} \frac{\partial v_k}{\partial x_l} s$

$$f(v) = \int_\Omega f_i v_i \, dx_1 dx_2 + \int_{\partial\Omega_1} S_i v_i d$$

E^{ijkl} é o tensor das constantes elásticas de Hooke. Este tensor de quarta ordem satisfaz a condição de simetria $E^{ijkl} = E^{jikl} = E^{ijlk} = E^{klij}$. Os vetores dos deslocamentos no sistema cartesiano são $u = (u_1, u_2)$, $v = (v_1, v_2)$. f_i é a força de volume, no caso, o peso próprio, e S_i é a força aplicada ao corpo na porção $\delta\Omega_1$ do contorno $\partial\Omega = \overline{\partial\Omega_1} \cup \overline{\partial\Omega_2} \cup \overline{\partial\Omega_3}$ conforme mostra a FIGURA 6-14.

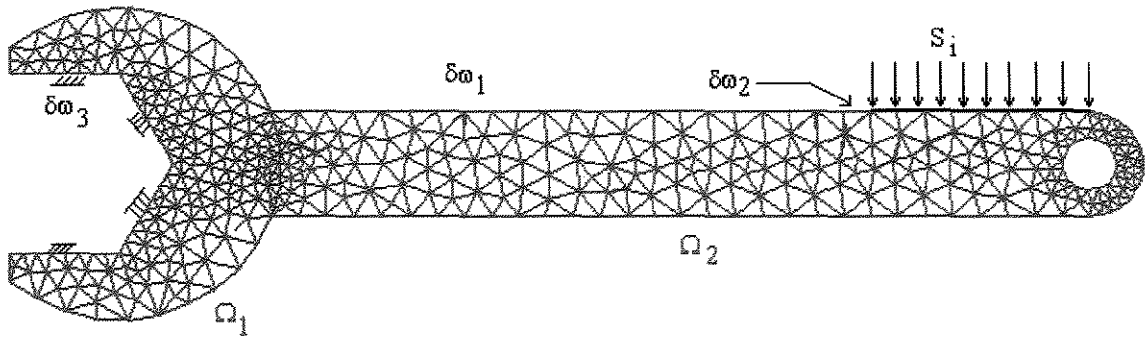

FIGURA 6-14- Problema elástico com dois subdomínios.

Para esta simulação numérica foi considerado que o material tem um comportamento isotrópico. Foi desprezado o peso próprio. Foram usados o módulo de

Young E=3234 N/m^2 e o coeficiente de Poisson μ=0,3. A força aplicada S_i tem intensidade de 400 N/m.

Na FIGURA 6-15 mostra-se o resultado referente à pressão no material. Observa-se que a figura mostra uma forte tração na região em preto e uma compressão na região vermelha, o que é coerente com o tipo de solicitação aplicada. A simulação com e sem subestruturação levaram ao mesmo resultado numérico.

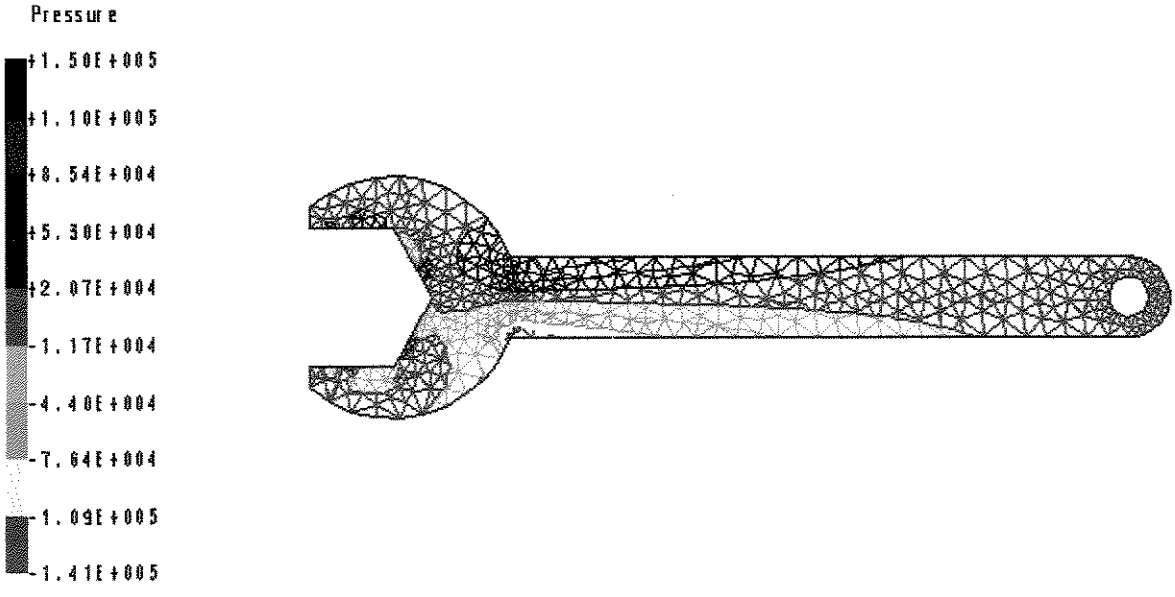

FIGURA 6-15- Resultado numérico obtido para o problema elástico.

7. MÉTODOS ITERATIVOS APLICADOS AO PROBLEMA REDUZIDO

7.1. MÉTODO DO GRADIENTE CONJUGADO

O método do gradiente conjugado é usado para resolver o problema reduzido. Por isso, neste capítulo, faz-se uma abordagem da teoria apresentada por AXELSSON (1984), tendo como objetivo uma revisão dos conceitos básicos referentes a esse método.

7.1.1. O FUNCIONAL QUADRÁTICO

Seja x_k uma aproximação do sistema linear $Ax=b$, onde A pertence a $R^{n\times n}$, definida positiva, e b pertence a R^n. O resíduo correspondente a esta aproximação é dado por:

$$r_k = b - Ax_k \tag{7.1}$$

Sabe-se que, se A é definida positiva, então sua inversa também é. Seguindo-se esse raciocínio, $r_k^t A^{-1} r_k$ também resulta num número maior ou igual a zero. Então tem-se que:

$$c^2 = r_k^t A^{-1} r_k \tag{7.2}$$

e que, se c=0, implica que r=0.

A substituição da equação (7.1) em (7.2) leva a:

$$c^2 = x_k^t A x_k - 2x_k^t b + \underbrace{b^t A^{-1} b}_{c_1}$$

Fazendo-se $\phi = c^2 - c_1$ na expressão anterior, segue que:

$$\phi(x_k) = \frac{x_k^t A\, x_k}{2} - b^t x_k \tag{7.3}$$

A expressão (7.3) é conhecida como o funcional quadrático.

Observa-se que ϕ é função de x_k e o seu gradiente é dado por:

$$\nabla\phi(x_k) = Ax_k - b \qquad (7.4)$$

Segundo a expressão (7.4), o valor mínimo de ϕ é encontrado em $x_k = A^{-1}b$, ou seja, minimizar ϕ e resolver $Ax = b$ são problemas equivalentes.

Tratando-se de um método iterativo, em cada iteração percorrem-se os pontos $\{x_1 \ldots x_k\}$ segundo as direções $\{p_1, \ldots . p_k\}$. Esse percurso converge para a solução se as direções $\{p_1, \ldots . p_k\}$ minimizam (7.3) a cada iteração, e se no ponto x_k cumpre-se $\min \phi(x) = \phi(x_k)$.

7.1.2. MINIMIZAÇÃO DO FUNCIONAL QUADRÁTICO SEGUNDO AS DIREÇÕES P_k

Seja P_k o conjunto dos espaços vetoriais formados pela combinação linear de $\{p_1, \ldots . p_k\}$, ou seja, $P_k = span[p_1, p_2, \ldots . p_k]$[1]. O ponto x_k pertence a este espaço, uma vez que é obtido a partir de P_k. Daí tem-se que:

$$x_k = \underbrace{y_1 p_1 + \ldots\ldots y_{k-1} p_{k-1}}_{P_{k-1}} + \alpha_k p_k$$

ou:
$$x_k = \underbrace{P_{k-1} Y}_{x_{k-1}} + \alpha_k p_k \qquad (7.5)$$

Então:

$$\phi(x_k) = \phi(P_{k-1} Y + \alpha_k p_k)$$

ou:
$$\phi(x_k) = \phi(P_{k-1} Y) + \frac{\alpha_k^2}{2} p_k^t A p_k - \alpha_k p_k^t b + \alpha_k Y P_{k-1}^t A p_k$$

Minimizando-se $\phi(x_k)$, tem-se que:

---

[1] O span[v1, v2, v3, ...vn] representa o espaço vetorial gerado pela combinação linear dos vetores { v1, v2, v3, ...vn }.

$$\min \phi(x_k) = \min[\phi(P_{k-1}Y) + \underbrace{(\frac{\alpha_k^2}{2} p_k^t A p_k - \alpha_k p_k^t b)}_{\psi(\alpha)} + (\alpha_k Y P_{k-1}^t A p_k)] \tag{7.6}$$

No primeiro termo da equação (7.6), a expressão $\phi(P_{k-1}Y)$ é igual a $\phi(x_{k-1})$, que já é conhecido. O segundo termo $\psi(\alpha)$ varia quadraticamente em α, e o seu valor mínimo é obtido com $\frac{\partial \psi}{\partial \alpha} = 0$, o que leva a:

$$\alpha_k = \frac{p_k^t b}{p_k A p_k} \tag{7.7}$$

O terceiro e o último termo da equação (7.6) dependem da combinação linear de todas as direções anteriores $YP_{k-1}^t A$, o que torna o cálculo de α_k muito custoso. A direção atual (p_k) ideal para se alcançar a minimização é aquela que anula $\alpha_k Y P_{k-1}^t A p_k$. Com esse objetivo, a direção atual tem que ser A conjugada[2] com todas as direções anteriores. Em outras palavras, o vetor p_k tem que ser perpendicular ao espaço gerado por $\{Ap_1, Ap_2, ...Ap_{k-1}\}$, ou melhor, p_k tem que pertencer ao espaço $span[Ap_1, Ap_2, ...Ap_{k-1}]^{\perp}$, conforme o esquema mostrado na FIGURA 7-1).

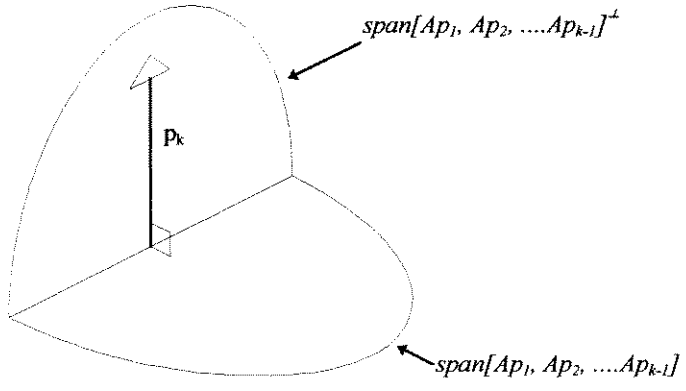

FIGURA 7-1- p_k pertencente ao espaço span[Ap1, Ap2, ...Apk-1]⊥.

[2] Definição: u e v são A-conjugadas se $uAv=0$, $<u,v>_A$.

Observações:

1. A multiplicação de $x_k = x_{k-1} + \alpha_k p_k$ por A resulta em:

$$r_k = r_{k-1} - \alpha_k A p_k \tag{7.8}$$

2. Caso p_k é A conjugado com as direções anteriores demonstra-se que:

$p_k^t b = p_k^t r_{k-1}$, o que leva à redefinição de α segundo:

$$\alpha_k = \frac{p_k^t r_{k-1}}{p_k A p_k}$$

Prova:

Usando-se a relação (7.1), tem-se as seguintes expressões:

$$b = r_{k-1} - A x_{k-1}$$

$$p_k^t b = p_k^t r_{k-1} - p_k^t A x_{k-1} \tag{7.9}$$

Como x_{k-1} *pertence a* P_{k-1}, *este é conjugado com* p_k^t, *ou seja,* $p_k^t A x_{k-1} = 0$. *A equação (7.9) então fica:*

$$p_k^t b = p_k^t r_{k-1} \tag{7.10}$$

Substituindo-se esta última igualdade em (7.7) tem-se uma nova expressão:

$$\alpha = \frac{p_k^t r_{k-1}}{p_k^t A p_k} \tag{7.11}$$

3. Os resíduos são mutuamente ortogonais[3], assim:

$$r_k^t r_{k-1} = 0$$

4. As direções p_j^t e r_j são ortogonais, ou melhor, $p_j^t r_j = 0$:

Prova:

Como $x_j \in P = span[p_1, p_2, \ldots\ldots p_j]$, *então pode-se escrever:*

$x_j = PY$.

---

[3] Ver o teorema 10.2.3 de GOLUB e VAN LOAN (1990, p. 521).

Daí segue que:

$$\min \phi(x_k) = \min \phi(PY)$$

$$\phi(PY) = \frac{1}{2} y^t p_j^t A p_j y - y^t p_j^t b$$

O valor mínimo desta função é encontrado a partir de:

$$p_j^t A p_j y = p_j^t b$$

$$p_j^t (\underbrace{A p_j y - b}_{r_j}) = 0$$

$$p_j^t r_j = 0 \qquad (7.12)$$

7.1.3. DETERMINAÇÃO DA DIREÇÃO CONJUGADA

Demonstra-se que o vetor p_k pode ser escrito como a soma do vetor r_{k-1} com outro vetor y_{k-1} pertencente ao espaco $AP_{k-1} = span[Ap_1, Ap_2, ...Ap_{k-1}]$. Isso satisfaz a condição das direções conjugadas $y_{k-1}^t A p_k = 0$. Já que y_{k-1}^t tem que ser perpendicular a p_k então é conveniente escolher o vetor y_{k-1} como sendo aquele que seja paralelo a p_{k-1}, porque p_{k-1} já é conhecido. Ou seja, escolhe-se $y_{k-1} = \beta_k p_{k-1}$ (ver FIGURA 7-2).

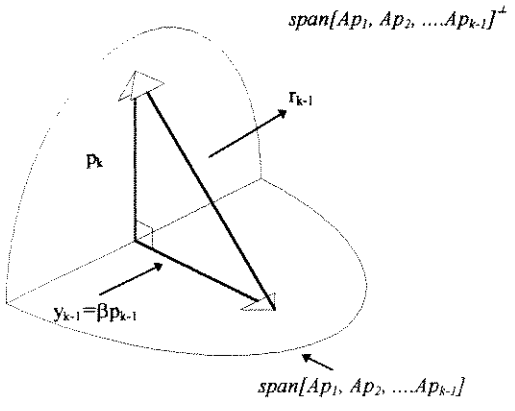

FIGURA 7-2- Relação entre os vetores p_k, p_{k-1} e r_{k-1}.

A consideração anterior permite escrever:

$$p_k = r_{k-1} - \beta_k p_{k-1} \quad [4] \tag{7.13}$$

A multiplicação desta última equação por $p_{k-1}A$ pela direita resulta em:

$$\underbrace{p_{k-1} A p_k}_{0} = p_{k-1} A r_{k-1} - \beta p_{k-1} A p_{k-1}$$

ou: $$\beta_k = -\frac{p_{k-1}^t A r_{k-1}}{p_{k-1}^t A p_{k-1}} \tag{7.14}$$

Até o momento, tem-se as seguintes equações:

$$x_k = x_{k-1} + \alpha_k p_k \tag{7.15}$$

$$\alpha_k = \frac{p_k^t r_{k-1}}{p_k^t A p_k} \tag{7.16}$$

$$p_k = r_{k-1} - \beta_k p_{k-1} \tag{7.17}$$

$$\beta_k = -\frac{p_{k-1}^t A r_{k-1}}{p_{k-1}^t A p_{k-1}} \tag{7.18}$$

Nota-se que o produto $p_k^t A$ aparece no cálculo de α_k β_k. Este fator ($p_k^t A$) pode ser eliminado utilizando-se as seguintes transformações.

Multiplicando-se a equação (7.13) por r_{k-1}^t tem-se a seguinte relação:

$$p_k r_{k-1}^t = r_{k-1} r_{k-1}^t - \beta_k p_{k-1}^t r_{k-1}^t$$

O segundo termo do segundo membro da equação acima é nulo, conforme a equação (7.12), portanto tem-se:

$$p_k r_{k-1}^t = r_{k-1} r_{k-1}^t$$

ou: $$r_{k-1}^t p_k = r_{k-1}^t r_{k-1} \tag{7.19}$$

---

[4] Uma demonstração detalhada desta expressão pode ser encontrada em GOLUB eVAN LOAN (1990, p. 520).

Esta última equação pode ser usada em (7.11) obtendo-se uma nova expressão para α_k:

$$\alpha_k = \frac{r_{k-1}^t r_{k-1}}{p_k A p_k} \tag{7.20}$$

A equação (7.8) para a posição k-1 é:

$$r_{k-1} = r_{k-2} - \alpha_{k-1} A p_{k-1}$$

A equação anterior multiplicada por r_{k-1}^t resulta em:

$$r_{k-1}^t r_{k-1} = \underbrace{r_{k-1}^t r_{k-2}}_{0} - \alpha_{k-1} \underbrace{r_{k-1}^t A p_{k-1}}_{p_{k-1}^t A r_{k-1}}$$

Daí, tem-se que:

$$\alpha_{k-1} = -\frac{r_{k-1}^t r_{k-1}}{p_{k-1}^t A r_{k-1}}$$

que, tendo em vista a equação (7.18), equivale a escrever:

$$\alpha_{k-1} = \frac{1}{\beta_k} \frac{r_{k-1}^t r_{k-1}}{p_{k-1}^t A p_{k-1}}$$

Igualando-se a expressão acima à equação (7.16) com *k=k-1,* consegue-se uma nova expressão para β_k, dada por:

$$\beta_k = \frac{r_{k-1}^t r_{k-1}}{r_{k-2}^t r_{k-2}} \tag{7.21}$$

7.2. ALGORITMO DO GRADIENTE CONJUGADO

As seguintes fórmulas, já vistas anteriormente, são usadas pelos algoritmos propostos na literatura.

$$\beta_k = \frac{r_{k-1}^t r_{k-1}}{r_{k-2}^t r_{k-2}} \tag{7.22}$$

$$p_k = r_{k-1} - \beta_k p_{k-1} \tag{7.23}$$

$$\alpha_k = \frac{r_{k-1}^t r_{k-1}}{p_k A p_k} \tag{7.24}$$

$$x_k = \underbrace{P_{k-1} Y}_{x_{k-1}} + \alpha_k p_k \tag{7.25}$$

$$r_k = r_{k-1} - \alpha_k A p_k \tag{7.26}$$

GOLUB e VAN LOAN (1996, p. 524) propõem o seguinte algoritmo:

k=0; x=0, r=b, $\rho_0 \|r\|_2^2$;
Enquanto $\sqrt{\rho_k} > \varepsilon \|b\|_2$ e $k < k_{max}$
    k=k+1;
    Se k=1
        p_k= r
    Caso contrário
        $\beta_k = \rho_{k-1} / \rho_{k-2}$
        $p_k = r + \beta_k p$
    Final do Se
    w=Ap
    $\alpha_k = \rho_{k-1} / p^t w$
    $x = x + \alpha_k p$
    $r = r - \alpha_k w$
    $\rho_k = \|r\|_2^2$
Fim do enquanto

7.3. CONVERGÊNCIA DO GRADIENTE CONJUGADO

Segundo AXELSSON (1984), a matriz A simétrica definida positiva tem as seguintes propriedades :

1) A tem todos seus autovetores positivos, sendo que estes autovetores podem ser iguais;

2) a norma da energia define-se como $\|x\|_A = x^t Ax$;

3) os autovalores de A cumprem o seguinte teorema:

Teorema:

Sejam $\{\lambda_1, \lambda_2, \lambda_3, \ldots\ldots \lambda_n\}$ *os autovalores de A. Então a convergência está limitada pela seguinte expressão:*

$$\|x_{k+1} - \bar{x}\|_A \leq \max_{1 \leq i \leq n} |1 + \lambda_i p_k(\lambda_i)| \|x_0 - \bar{x}\|_A \qquad \forall\, p_k(\lambda_i) \text{ polinomio de grau} \leq k$$

onde $\bar{x}$ *é a solução exata.*

Segundo esse teorema, se a matriz A tem autovalores múltiplos $\{\lambda_1, \lambda_2, \lambda_3, \ldots\ldots \lambda_n\}$ e se no ponto m-1, sendo m menor do que n, escolhemos $p_{m-1}(x)$ tal que:

$$\max_{1 \leq i \leq n} |1 + \lambda_i p_k(\lambda_i)| = 0$$

então $\|x_m - \bar{x}\|_A = 0$, ou seja, $x_m = \bar{x}$. Isso garante que este método converge em m iterações.

4) se A é uma matriz simétrica definida positiva e $\lambda_1 \geq \lambda_2 \geq \lambda_3 \ldots\ldots \lambda_n$ [5] e P_k é o espaço dos polinômios de grau k, pode-se escolher um polinômio $\bar{p}_k(x)$ que seja o mínimo entre todos os $\max_{x \in [\lambda_n, \lambda_1]} |1 + x p_k(x)|$. Isso pode ser expresso como:

---

[5] Dizer que uma matriz é definida positiva implica também que todos os seus autovalores são positivos

$$\max_{x\in[\lambda_n,\lambda_1]}\left|1+x\bar{p}_k(x)\right| = \min_{p_k\in P_k}\left\{\max_{x\in[\lambda_n,\lambda_1]}\left|1+xp_k(x)\right|\right\}$$

Pode-se provar, com as propriedades dos polinômios, que:

$$2\left[\frac{\sqrt{\frac{\lambda_1}{\lambda_n}}-1}{\sqrt{\frac{\lambda_1}{\lambda_n}}+1}\right]^k \le \max_{x\in[\lambda_n,\lambda_1]}\left|1+x\bar{p}_k(x)\right| \tag{7.27}$$

Assim tem-se um limite mais restrito para a convergência, dado por:

$$\left\|x_k-\bar{x}\right\|_A \le 2\left[\frac{\sqrt{\frac{\lambda_1}{\lambda_n}}-1}{\sqrt{\frac{\lambda_1}{\lambda_n}}+1}\right]^k \left\|x_0-\bar{x}\right\|_A \tag{7.28}$$

A partir da equação (7.28), observa-se que, se $\sqrt{\lambda_1/\lambda_n}$ tem um valor próximo de 1, a expressão $r=\sqrt{\lambda_1/\lambda_n}-1/\sqrt{\lambda_1/\lambda_n}+1$ resulta num valor quase nulo. Conseqüentemente o valor x_k também é muito próximo da solução exata. Sendo assim, com poucas k iterações, x_k converge para $\bar{x}$. No caso de $\sqrt{\lambda_1/\lambda_n}$ ser "muito maior" que 1, o valor de r resulta num valor muito próximo de 1. Assim, é necessário um valor muito grande de k para reduzir r, ou seja, a convergência é lenta. O mencionado no paragrafo anterior mostra-se esquematicamente Na FIGURA 7-3.

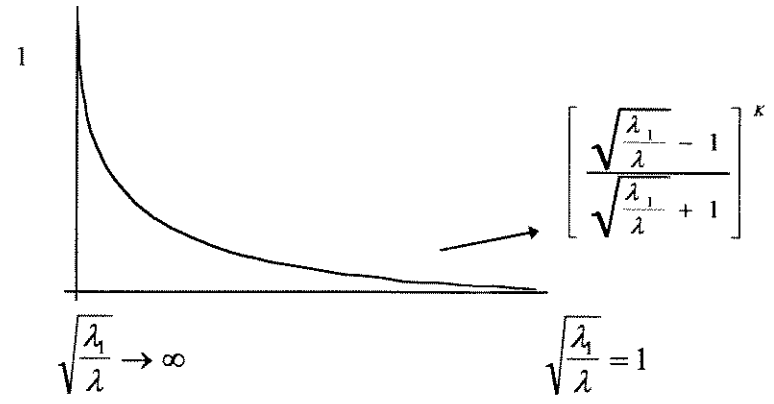

FIGURA 7-3- Variação de $r = \sqrt{\lambda_1 / \lambda_n} - 1 / \sqrt{\lambda_1 / \lambda_n} + 1$.

O próximo passo é saber quanto a relação λ_1 / λ_n influencia na convergência do gradiente conjugado. Com esse objetivo foi feita a seguinte análise. Seja *aprox* a aproximação obtida depois de *k* iterações, ou seja:

$$aprox = \left[\frac{\sqrt{\frac{\lambda_1}{\lambda_n}}-1}{\sqrt{\frac{\lambda_1}{\lambda_n}}+1}\right]^k$$

Isolando-se *k* da expressão acima tem-se que:

$$k = Log(aprox) / Log\left[\frac{\sqrt{\frac{\lambda_1}{\lambda_n}}-1}{\sqrt{\frac{\lambda_1}{\lambda_n}}+1}\right]$$

Na expressão acima, *aprox* é a aproximação desejada (neste exemplo será usado $aprox=10^{-5}$). A

FIGURA 7-4 representa o gráfico, em escala logarítmica, do número de iterações *k* em função de λ_1 / λ_n dentro do intervalo $[1,10^{10}]$. Observa-se nesse gráfico que para uma relação λ_1 / λ_n igual a 10^8, são necessárias 10,000 iterações, e para uma relação λ_1 / λ_n igual a 10^2, com 10 iterações chega-se à aproximação desejada (aprox=10^{-5}).

Isso mostra que a relação λ_1 / λ_n é um fator muito importante na convergência do gradiente conjugado.

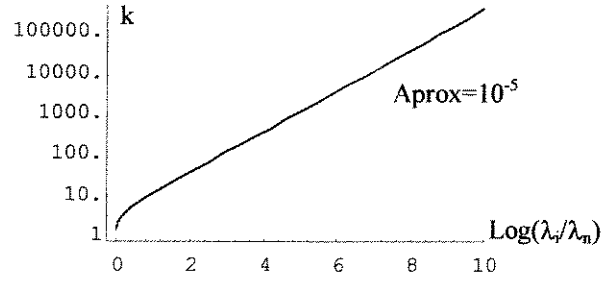

FIGURA 7-4- Número de iterações em função de λ_n.

Na literatura, $K_2(A) = \sqrt{\lambda_1 / \lambda_n}$ [6] é definido como sendo o "número de condicionamento espectral da matriz A" ou simplesmente "número de condicionamento da matriz A". Pode-se provar que as matrizes ditas "mal condicionadas" têm valores grandes de $K_2(A)$, enquanto que as matrizes com valores pequenos de $K_2(A)$ são "bem condicionadas". Existem ainda as matrizes ortogonais, que são "perfeitamente condicionadas", para as quais $K_2(Q)=1$. Do que foi mencionado anteriormente, conclui-se que dizer que a matriz A é mal-condicionada equivale a dizer que a mesma tem uma taxa de convergência lenta.

Os problemas de convergência lenta e mal condicionamento têm que ser evitados. A maneira mais comum é construir um sistema equivalente onde o número de condicionamento seja mais favorável. Esta técnica é chamada de pré-condicionamento.

---

[6] $K_p(A)$ define o numero de condicionamento para norma-p. Este numero mede a distancia relativa de norma-p entre a matriz A e o conjunto das matrizes singulares. Ver GOLUB e VAN LOAN (1996, p. 81).

7.4. GRADIENTE CONJUGADO PRÉ-CONDICIONADO

Do exposto anteriormente, a convergência de um sistema de equações lineares depende das características do número de condicionamento da matriz de rigidez. Para evitar esta limitação, a melhor forma de se resolver um sistema que não tenha um número de condicionamento "favorável", é construir outro sistema equivalente com um número de condicionamento mais adequado, ou seja, menor.

Seja o seguinte sistema linear de equações $Ax = b$, onde $A \in R^{n \times n}$. Pode-se efetuar sobre esse sistema as seguintes transformações:

$$\underbrace{C^{-1}AC^{-1}}_{\tilde{A}}\underbrace{Cx}_{\tilde{x}} = \underbrace{C^{-1}b}_{\tilde{b}}$$

ou: $$\tilde{A}\tilde{x} = \tilde{b} \qquad (7.29)$$

Já foi mostrado que resolver este sistema equivale a minimizar:

$$\phi(\tilde{x}) = \frac{\tilde{x}^t \tilde{A} \tilde{x}}{2} - \tilde{b}^t \tilde{x}$$

Para se ter certeza de que $\tilde{A}$ seja simétrica e definida positiva, é necessário escolher uma matriz C que seja também definida positiva. Ainda na matriz $\tilde{A}$ pode ser realizada mais uma transformação:

$$C^{-1}\tilde{A}C = C^{-1}\underbrace{C^{-1}AC^{-1}}_{\tilde{A}}C$$

$$C^{-1}\tilde{A}C = (C^{-1})^2 A$$

$$C^{-1}\tilde{A}C = (\underbrace{C^2}_{M})^{-1} A$$

$$C^{-1}\tilde{A}C = M^{-1}A$$

onde a matriz M é chamada de matriz pré-condicionadora. A seguir mostra-se que os autovalores da matriz $\tilde{A}$ são iguais aos da matriz $M^{-1}A$.

Se λ é um autovalor de $\widetilde{A}$, pela definição de autovalor, tem-se que:

$$\underbrace{\widetilde{A}}_{C^{-1}AC^{-1}} x = \lambda x$$

Fazendo-se $C^{-1}x = w$ segue que:

$$C^{-1}Aw = \lambda x$$

Multiplicando-se essa expressão por C^{t} resulta que:

$$\left(\underbrace{C^2}_{M}\right)^{-1} Aw = \lambda \underbrace{C^{-1}x}_{w}$$

Substituindo-se M e w, chega-se finalmente a:

$$M^{-1}Aw = \lambda w$$

Nota-se do anterior que o autovalor λ é comum para $\widetilde{A}$ e $M^{-1}A$, ou melhor, a matriz pré-condicionada $\widetilde{A}$ tem os mesmos autovalores de $M^{-1}A$. Com isso, consegue-se obter os autovalores de $\widetilde{A}$ calculando-se os autovalores de $M^{-1}A$, em outras palavras, os autovalores de $\widetilde{A}$ podem ser totalmente determinados, desde que $M^{-1}A$ seja conhecido.

É conveniente escolher a matriz M tal que seja parecida com A, assim, $M^{-1}A \cong I$. Desse modo o sistema equivalente ($M^{-1}A\ x = M^{-1}b$) estará muito próximo da solução, diminuindo o número de iterações.

O algoritmo do gradiente conjugado pré-condicionado aplica-se à matriz $\widetilde{A}$. Assim no produto $\widetilde{r}_{k-1}^{t}\widetilde{r}_{k-1}$ tem-se as seguintes transformações:

$$\widetilde{r}_{k-1}^{t}\widetilde{r}_{k-1} = \left(C^{-1}r_{k-1}\right)^{t}\left(C^{-1}r_{k-1}\right)$$

ou: $$\widetilde{r}_{k-1}^{t}\widetilde{r}_{k-1} = r_{k-1}^{t}C^{-1}C^{-1}r_{k-1} = r_{k-1}^{t}M^{-1}r_{k-1}$$

7.4.1. ALGORITMO DO GRADIENTE CONJUGADO PRÉ-CONDICIONADO

Na implementação computacional do gradiente conjugado pré-condicionado, inicialmente calcula-se $z_{k-1} = M^{-1} r_{k-1}$, e nos lugares do algoritmo em que se encontra $r_{k-1}^t M^{-1} r_{k-1}$, é usado $r_{k-1}^t z_{k-1}$.

A direção inicial associada a matriz $\widetilde{A}$ é $\widetilde{p} = C^{-1} p$. Porem no algoritmo utiliza-se a direção $p^{(1)} = z^{(0)}$ porque $\widetilde{p}^{(1)} = \widetilde{z}^{(0)}$ é igual a $C^{-1} p^{(1)} = C^{-1} z^{(0)}$ onde o termo C^{-1} pode ser eliminado levando a expressão $p^{(1)} = z^{(0)}$.

Na biblioteca TEMPLATES, BARRET et al. (1994, p.15) implementam o seguinte algoritmo para o gradiente conjugado pré-condicionado:

Calcular $r^0 = b - Ax^0$ para um valor inicial de x^0
Para i=1,2,3.....
    Resolver $M z^{i-1} = r^{i-1}$
    $\rho_{i-1} = r^{(i-1)t} z^{(i-1)}$
    Se i=1
        $p^{(1)} = z^{(0)}$
    Caso contrário
        $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$
        $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
    Final do Se
    $q^{(i)} = A p^{(i)}$
    $\alpha_i = \rho_{i-1} / p^{(i)t} q^{(i)}$
    $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
    Verifica a convergência e continua se necessário
Fim

8. CONSTRUÇÃO DO PRÉ-CONDICIONADOR BLOCO-DIAGONAL

Assume-se que após o domínio Ω ter sido dividido em elementos, as subestruturas Ω_i são formadas por conjuntos de elementos tal que não exista acoplamento entre subestruturas ($\Omega_i \cap \Omega_j = 0$). Na malha bidimensional de subestruturas (também conhecida como malha grossa) reconhece-se que os nós podem pertencer aos seguintes três conjuntos:

1. Os vértices: São os nós que pertencem a dois ou mais subdomínios e pertencem ao contorno do domínio.
2. As interfaces entre subestruturas: Nós que pertencem a dois subdomínios e não pertencem ao contorno do domínio..
3. Nós Internos: Nós que pertencem a uma subestrutura apenas.

Conforme já foi mencionado, as variáveis correspondentes aos nós internos a cada subdomínio são primeiro eliminadas. O sistema resultante desta redução envolve apenas variáveis das interfaces. Este sistema é resolvido pelo método do gradiente conjugado. Este trabalho usa esta sistemática, que também é comum aos artigos publicados por BJØRSTAD e WIDLUND (1986), BRAMBLE, PASCIAK e SCHATZ (1986), SMITH (1990), SMITH (1992), MANDEL (1990), LE TALLEC (1994) e AINSWORTH (1996). Todos estas publicações se diferenciam entre si pelo tipo de pré-condicionador implementado.

O sistema reduzido é dado por:

$$\underbrace{\sum_{s=1}^{Ns} K^{s}{}_{\mathrm{RED}}}_{\tilde{K}} \underbrace{U_{\mathrm{interface}}}_{Ui} = \underbrace{\sum_{s=1}^{Ns} F^{s}{}_{\mathrm{RED}}}_{\tilde{F}}$$

Onde $K^{s}_{RED\,ij}$ e $F^{s}_{RED\,i}$ são resultantes da condensação estática aplicada a cada subestrutura $s=\Omega_s$. Antes de montar a matriz $\widetilde{K}$[1], os vértices V são renumerados e em seguida os nós de interface B. Seja a matriz de rigidez composta por quatro blocos, conforme mostrado abaixo:

$$\widetilde{K}=\begin{bmatrix}\widetilde{K}_{VV} & \widetilde{K}_{VB}\\ \widetilde{K}_{BV} & \widetilde{K}_{BB}\end{bmatrix} \tag{8.1}$$

Uma das condições para que um pré-condicionador seja eficiente é que ele tem que ser parecido com a matriz pré-condicionada, tal que $\widetilde{K}M^{-1}=I$, onde M é a matriz pré-condicionadora. Atendendo a essa condição, para a matriz M inicialmente foi construído o pré-condicionador sem levar em conta o acoplamento existente entre os subdomínios, ou seja, foram usados somente os blocos $\widetilde{K}_{VV}$ e $\widetilde{K}_{BB}$, conforme mostrado abaixo:

$$M=\begin{bmatrix}\widetilde{K}_{VV} & 0\\ 0 & \widetilde{K}_{BB}\end{bmatrix} \tag{8.2}$$

Porém, esta técnica não traz bons resultados, pois as funções de forma de cada vértice tem apenas uma influência "local". Esta influência é uma aproximação isolada dos vértices que não tem acoplamento com outros nós do domínio. A FIGURA 8-1 mostra as funções de forma dos vértices da malha grossa.

---

[1] Esta-se usando $\widetilde{K}$ ao invés K porque a redução estática é interpretada como uma mudança de bases das funções de forma das interfaces ϕ para $\widetilde{\phi}$ (ver capitulo 6)

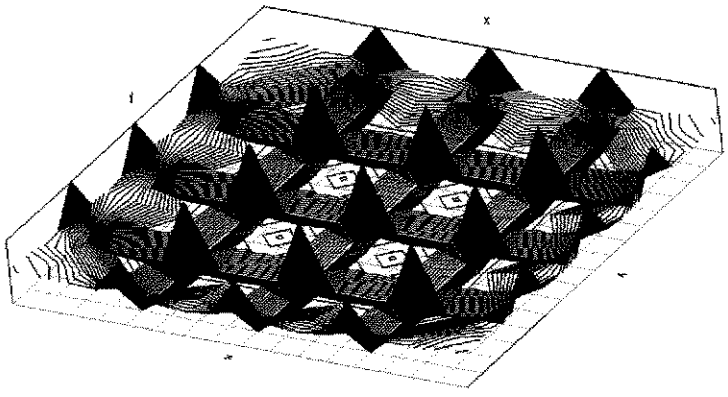

FIGURA 8-1- Funções de forma na malha dos subdomínios.

Do apresentado, conclui-se que caso as subestruturas têm poucos elementos, as funções de forma dos vértices aproximarão melhor o problema. No caso oposto, i. é., se dentro dos subdomínios existem muitos elementos, as funções de forma dos vértices não aproximarão bem o problema. Assim conclui-se que a eficiência deste pré-condicionador é inversamente proporcional ao número de elementos dentro de cada sub-domínio.

8.1. MUDANÇA DE BASE NAS INTERFACES

Para contornar o problema da influência local dos vértices implementou-se a mudança de bases das funções de forma relacionadas aos vértices. Com essa idéia procurou-se conseguir que as funções de forma dos vértices tivessem uma influência ao longo de cada interface. Como resultado dessa ampliação da influência dos vértices sobre a interface, o espaço formado pelas novas funções dos vértices e as funções das interfaces resulta numa base hierárquica.

A mudança para bases hierárquicas tem suas origens na técnica "multigrid". Em YSERENTANT (1988), criaram-se-subníveis de refinamento construindo-se sempre

duas novas funções de forma lineares a partir de uma outra que esta num nível superior imediato (ver FIGURA 8-2).

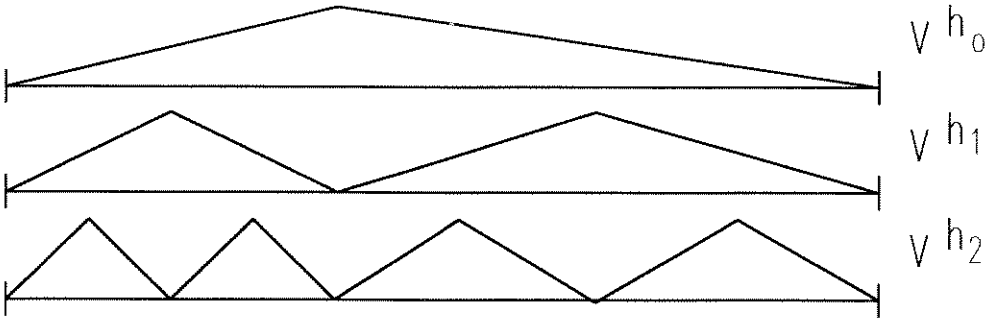

FIGURA 8-2- Mudança de bases implementada por YSERENTANT (1988).

SMITH e WIDLUND (1990) utilizaram a mesma técnica proposta por YSERENTANT (1988), mas a mudança de base acontece apenas nas interfaces dos subdomínios e não no domínio inteiro. MANDEL (1990) provou teoricamente que o pré-condicionador construído usando bases hierárquicas do tipo *p* também converge com taxas quase ótimas.

O pré-condicionador desenvolvido no presente trabalho usa os resultados apresentados por MANDEL (1990) e SMITH (1992), que usam como base os vértices da malha grossa para construir o pré condicionador. No presente trabalho, daqui em diante, a expressão "função" refere-se a "função de forma".

Seja V o espaço das funções de todo o domínio, constituído por três subespaços:

- O espaço $V_{harmônico}$ das funções referentes aos nós internos de cada subdomínio;
- O espaço $\widetilde{V}_B$ das funções referentes às interfaces;
- O espaço $\widetilde{V}_V$ das funções referentes aos vértices.

Encontra-se na literatura que as funções dos nós internos de cada subdomínio também são chamadas de funções harmônicas. Este nome tem sua origem na aplicação da técnica de subestruturação ao problema de Poisson. Depois de ser resolvido o problema de interface, os nós internos obedecem à seguinte equação (ver p. 91):

$$\left[a_{\mathrm{II}}\right]\left[x_{\mathrm{I}}\right]=\left[f_{\mathrm{I}}-a_{\mathrm{I\tilde{I}}}x_{\tilde{\mathrm{I}}}\right]$$

ou: $x_I = a_{II}^{-1} f_I - a_{II}^{-1} a_{I\tilde{I}} x_{\tilde{I}}$

O termo $-a_{II}^{-1} a_{I\tilde{I}} x_{\tilde{I}}$ é a influência das soluções dos nós de interface sobre os nós internos e equivale a resolver $\Delta u = 0$[2] dentro cada sub-domínio. Como o Laplaciano também é conhecido como operador harmônico, as funções internas ao subdomínio também são chamadas de funções harmônicas.

O espaço de funções de forma pode ser expresso como:

$V = V_{harmônico} \oplus \tilde{V}_B \oplus \tilde{V}_V$

Na FIGURA 8-3 são mostradas as funções de forma de ordem cúbica ao longo das interfaces que o ambiente PZ utiliza:

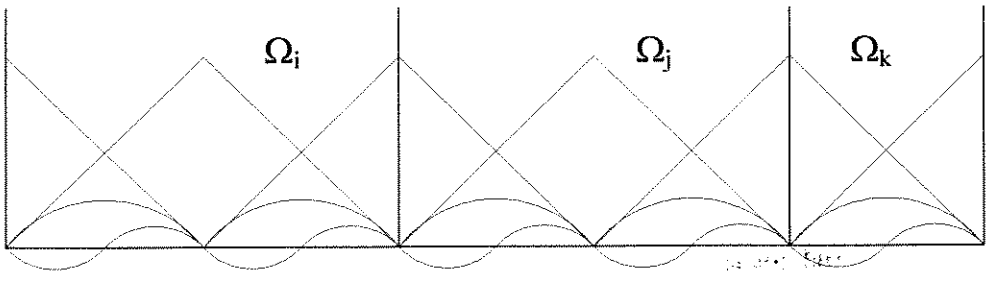

FIGURA 8-3- Funções hierárquicas de ordem cúbica numa interface.

[2] O que é equivalente a resolver $a_{II} y - a_{I\tilde{I}} x_{\tilde{I}} = 0$

Para obter o valor da função $\overline{\phi}_k$ sobre o nó k de tal maneira que tenha uma distribuição linear a longo da interface, usa-se a seguinte fórmula:

$$\overline{\phi}_k = \underbrace{\widetilde{\phi}_k\left(1-\frac{r_k}{L}\right)}_{f_k}$$

Onde r_k é a distância entre o vértice atual v^l e o nó k. A variável L representa a distância entre os vértices v^l-v^r da interface. A FIGURA 8-4 mostra esquematicamente como são definidas as distâncias r_k e L .

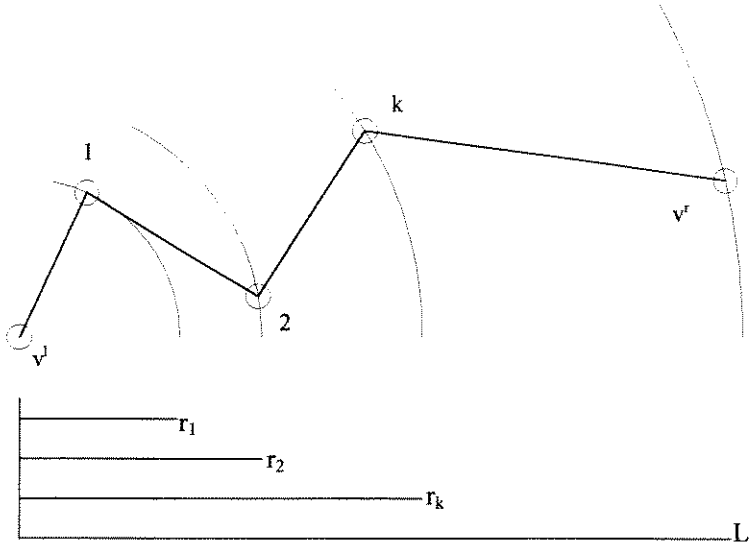

FIGURA 8-4- Relação entre as distância entre os nós da interface e os vértices.

A função $\widetilde{\phi}_k$ é a função de forma linear sobre o nó k. A FIGURA 8-5 mostra como o valor de $\overline{\phi}_k$ é distribuído linearmente ao longo da interface.

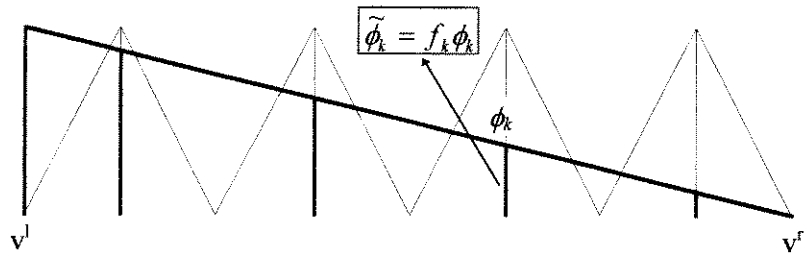

FIGURA 8-5- O valor da função $\overline{\phi}_k$ é proporcional a $\widetilde{\phi}_k$.

Depois da mudança de bases o espaço das funções de forma resulta em:

$$V = \mathrm{V}_{\text{harmonico}} \oplus \widetilde{V}_B \oplus \overline{V}_V$$

Na FIGURA 8-6 mostram-se as funções de forma da FIGURA 8-5 depois de distribuir linearmente as funções dos vértices.

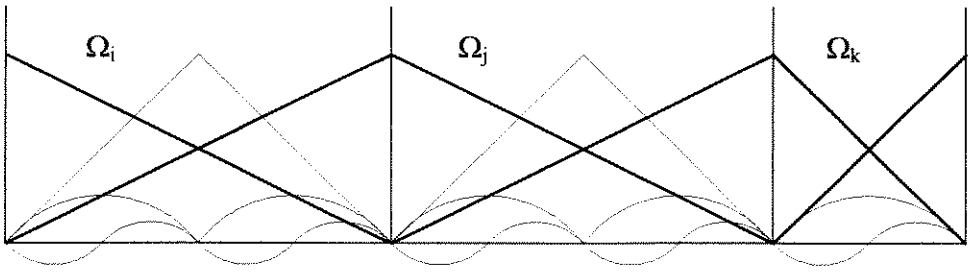

FIGURA 8-6- Funções de forma de ordem cúbicas depois da mudança de bases.

8.2. IMPLEMENTAÇÃO DA MUDANÇA DE BASE

Seja a seguinte matriz de rigidez para um subdomínio, tendo sido seus nós internos já reduzidos:

$$\widetilde{K}_{ij} = \iint \widetilde{\phi}_i \widetilde{\phi}_j d\Omega$$

Sejam $\overline{\phi}^l$ e $\overline{\phi}^r$ as funções de forma dos vértices "esquerdo" e "direito" respectivamente. Conforme já foi mostrado, as novas funções são obtidas pela soma das funções na interface multiplicadas por um fator f_k:

$$\overline{\phi}^l = \widetilde{\phi}_1 f_1^l + \widetilde{\phi}_2 f_2^l + \widetilde{\phi}_3 f_3^l \ldots\ldots \widetilde{\phi}_n f_n^l$$

$$\overline{\phi}^r = \widetilde{\phi}_1 f_1^r + \widetilde{\phi}_2 f_2^r + \widetilde{\phi}_3 f_3^r \ldots\ldots \widetilde{\phi}_n f_n^r$$

Os elementos da matriz de rigidez associadas com o vértice *l* podem ser escritos da seguinte maneira:

$$\overline{K}_{li} = \iint \overline{\phi}_l \widetilde{\phi}_i d\Omega \quad \text{para } i \neq l \text{ e}$$

Substituindo-se o valor das funções dos vértices segue que:

$$\overline{K}_{li} = \iint \left(\widetilde{\phi}_0 f_0^l + \widetilde{\phi}_1 f_1^l + \widetilde{\phi}_2 f_2^l \ldots\ldots\ldots\ldots\ldots + \widetilde{\phi}_n f_n^l\right) \widetilde{\phi}_i d\Omega \,^{3}$$

ou: $$\overline{K}_{li} = \iint (\widetilde{\phi}_0 \widetilde{\phi}_i f_0^l + \widetilde{\phi}_1 \widetilde{\phi}_i f_1^l + \widetilde{\phi}_2 \widetilde{\phi}_i f_2^l \ldots\ldots\ldots\ldots\ldots + \widetilde{\phi}_n \widetilde{\phi}_i f_n^l) d\Omega$$

A integral de cada produto $\widetilde{\phi}_j \widetilde{\phi}_i$ representa o elemento (j,i) da matriz reduzida $\widetilde{K}$. Pode-se escrever: $\overline{K}_{li}$ em função dos elementos de $\widetilde{K}$, então:

$$\overline{K}_{li} = f_0^l \widetilde{K}_{0i} + f_1^l \widetilde{K}_{1i} + f_2^l \widetilde{K}_{2i} \ldots\ldots\ldots\ldots\ldots + f_n^l \widetilde{K}_{ni}$$

ou em forma mais compacta:

$$\overline{K}_{li} = \sum_{j=0}^{j=n} f_j^l \widetilde{K}_{ji} \quad \text{para } i \neq l$$

Para os vértices tem-se que:

$$\overline{K}_{ll} = \iint \overline{\phi}_l \overline{\phi}_l d\Omega \text{ ou}$$

$$\overline{K}_{ll} = \iint \left(\widetilde{\phi}_0 f_0^l + \widetilde{\phi}_1 f_1^l + \widetilde{\phi}_2 f_2^l \ldots\ldots\ldots\ldots\ldots + \widetilde{\phi}_n f_n^l\right) \left(\widetilde{\phi}_0 f_0^l + \widetilde{\phi}_1 f_1^l + \widetilde{\phi}_2 f_2^l \ldots\ldots\ldots\ldots\ldots + \widetilde{\phi}_n f_n^l\right) d\Omega$$

---

[3] Nesta expressão *n* representa o número de nós na interface.

Essa expressão não foi demonstrada com detalhe porque o algoritmo explicado posteriormente não a utiliza.

As mesmas manipulações algébricas podem ser efetuadas para o vértice r (vértice direito):

$$\overline{K}_{ir} = \sum_{j=0}^{j=n} f_j^r \widetilde{K}_{ij} \quad \text{para } i \neq r$$

As transformações mostradas acima para a matriz de rigidez também são válidas para o vetor de carga. Mas neste caso apenas mudamos as linhas. Na implementação da mudança de base é necessário que os nós de interface tenham cinco informações:

1. a numeração local do nó corrente;
2. a numeração local do nó do extremo "direito";
3. a numeração local do nó do extremo "esquerdo";
4. o fator f^l que o nó corrente contribui para a função do vértice "esquerdo";
5. o fator f^r que o nó corrente contribui para a função do vértice "direito".

Com essas cinco informações, o algoritmo consiste em percorrer os nós de cada subdomínio e ir adicionando suas contribuições nos vértices extremos a cada interface. Isto cobre os elementos quadráticos que aparecem no calculo de $\overline{K}_{ll} = \iint \overline{\phi}_l \overline{\phi}_l d\Omega$. Cabe lembrar que nesse estágio, os nós "visíveis" em cada subdomínio são apenas os de interface e os nós dos vértices. Assim, nessa etapa o objeto do tipo **TSuperEl** tem um comportamento de elemento.

A mudança de bases foi implementada numa nova classe chamada **TModGridShape.** Esta classe é derivada da classe **TCompGrid** e tem como dado adicional uma árvore binária que para os nós de interface armazena as cinco informações mencionadas acima que são necessárias para implementar a mudança de base. A classe **TModGridShape** tem como função membro principal a função **Assemble** que constrói o sistema de equações global reduzido. Esta construção se faz percorrendo-se todos os elementos do domínio, sendo que em cada elemento tem-se três etapas:

•cálculo da matriz de rigidez local e do vetor de cargas nas bases padrão;

•Mudança de bases das matrizes de rigidez local e o vetor de carga local;

•Colocação das contribuições do elemento no sistema global..

Em Seguida, é mostrado o algoritmo que a função **Assemble** implementa detalhando-se a parte de mudança de bases.

iel = primeiro elemento da arvore binária de elementos
Enquanto *iel* exista;
Calcula a matriz de rigidez local *ek*;
Calcula o vetor de cargas local *fk*;
Para *in* = 0,1,2 ...numero de nós-1
Se o *nó(in)* não contribui para os vértices conclui-se este ciclo
middleeq = numeração local do *nó(in)*;
righteq = numeração local do vértice "direito" da interface;
leftid = numeração local do vértice "esquerdo" da interface;
leftval = contribuição f' de *nó(in)* para o vértice "esquerdo";
rightval = contribuição f' de *nó(in)* para o vértice direito;
Para *idf* = 0,1,2...graus de liberdade do *nó(in)*-1
Para *ieq* = 0,1,2,3...(numero de linhas de *ek*)-1

```
ek(ieq , lefteq+idf) += leftval* ek(ieq , middleeq+idf);
ek(ieq , righteq+idf) += righttval* ek(ieq , middleeq+idf);
ek(lefteq+idf , ieq) += leftval* ek(middleeq+idf , ieq);
ek(righteq+idf , ieq) += righttval* ek middleeq+idf , ieq);
```

Fim do ciclo *ieq*

```
ef (lefteq+idf , 0) += leftval* ef( middleeq+idf , 0);
ef( righteq+idf , 0) += righttval* ef( middleeq+idf , 0);
```

Fim do ciclo *idf*
Fim do ciclo *in*
Vai para o próximo *iel* da arvore binária

Coloca a contribuição de *ek* e *fk* no sistema global de equações
Fim do Enquanto

8.3. O PRÉ-CONDICIONADOR BLOCO-DIAGONAL COM BASE MODIFICADA

Após ter sido mudada a base dos vértices com a intenção de se estender a influência de suas funções, de maneira a permitir um acoplamento entre os vértices de subdomínios próximos, constrói-se o pré-condicionador. Este é construído sem levar em conta o acoplamento das funções $\overline{\phi}$ dos vértices com as funções $\widetilde{\phi}$ das interfaces. Conforme já foi mencionado anteriormente, os vértices são numerados inicialmente, e em seguida as interfaces. Sendo assim o pré-condicionador terá dois blocos na diagonal. O primeiro bloco refere-se aos vértices e o segundo bloco as interfaces.

$$M = \begin{bmatrix} \overline{K}_{VV} & 0 \\ 0 & \widetilde{K}_{BB} \end{bmatrix}$$

Cabe ressaltar que o segundo bloco $\widetilde{K}_{BB}$ na verdade é um conjunto de blocos D_i, onde cada bloco representa a matriz de rigidez local de uma interface na malha dos subdomínios, ou melhor, o bloco $\widetilde{K}_{BB}$ reúne as matrizes de rigidez das interfaces como mostra a FIGURA 8-7.

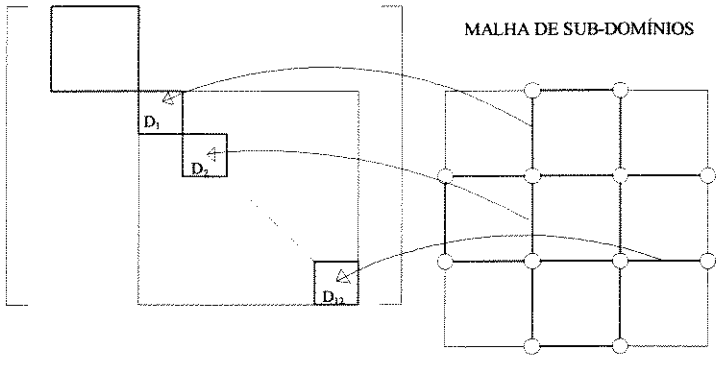

FIGURA 8-7- Localização das influências das interfaces na matriz M.

Para implementar um matriz que seja diagonal por blocos, foi criada a classe **TBlockDiagonal** derivada da classe **TMatrix** e da classe **TBlock**, para ter o comportamento de matriz em blocos.

Um objeto do tipo **TBlockDiagonal** precisa de duas informações para ser construído: o tipo de matriz que será o primeiro bloco que esta referido aos vértices dos sub-domínios, e as dimensões dos blocos referidos às interfaces dos subdomínios.

Estes dados inicializam as seguintes variáveis membro:

TMatrix *fDim0: Este ponteiro aponta para um objeto do tipo **TMatrix,** o que permite empregar qualquer classe derivada desta, i. é, qualquer tipo de matriz (skyline, banda, banda simétrica, matriz reduzida, etc.).

TFMatrix *fDimi: Com este ponteiro aloca-se dinamicamente um vetor de objetos tipo **TFMatrix** . Cada matriz desta alocação está referida à matriz resultante da interface entre dois vértices.

DoubleVec *fStorage: Neste vetor de variáveis de dupla precisão são armazenados os elementos das matrizes do tipo **TFMatrix**. Isto é possível porque foi acrescentada na classe **TFMatrix** a possibilidade dos elementos desta matriz serem armazenados num "pedaço" de memória indicado pelo usuário[4].

A FIGURA 8-8 mostra que os elementos dos blocos da diagonal correspondentes as interfaces são armazenados no vetor fStorage.

[4] Lembre-se que os elementos de um objeto **TFMatrix** são armazenados por colunas num vetor.

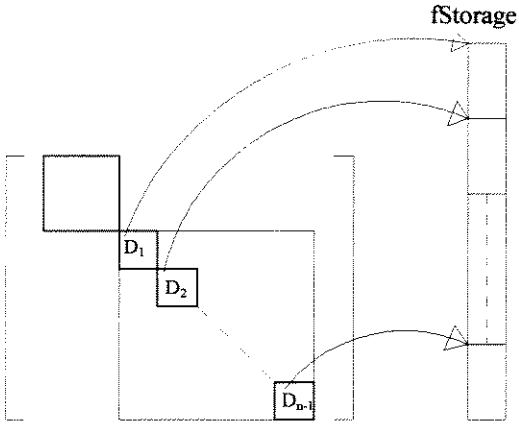

FIGURA 8-8- Armazenamento dos elementos da matriz tipo TFMatrix no vetor fStorage.

8.4. EQUIVALÊNCIA DO FATOR "ρ" COM A NORMA DO ERRO.

No cálculo do gradiente conjugado pré-condicionado define-se ρ como:

$$\rho = r^t z \qquad (8.3)$$

Se u_h é uma solução aproximada de Ku = f , então o resíduo está definido como $r_h = f - Ku_h$ e z = M^{-1} r, onde M é a matriz pré-condicionadora.

Se u é a solução exata, r_h também pode ser calculado como:

$$r_h = K(u - u_h)$$

A substituição desta última expressão em (8.3) leva a:

$$\rho = \left(u^t - u_h^t\right) K^t M^{-1} K\left(u - u_h\right)$$

Uma vez que o erro está definido como $e = u - u_h$, segue que:

$$\rho = e^t K^t M^{-1} K e \qquad (8.4)$$

Se for assumido que $M \cong K$, então $M^{-1}K \cong I$, o que permite escrever a expressão (8.4) como:

$$\rho \cong e^t Ke$$

que por definição é a norma do erro, ou melhor:

$$\rho \cong \left\| e^u \right\|_E^2 \qquad (8.5)$$

8.5. O PRÉ CONDICIONADOR COMO OPERADOR DIFERENCIAL

No algoritmo do gradiente conjugado da pagina 123 tem-se a linha :

Resolver M $z^{i-1}=r^{i-1}$.

Isto indica que o vetor z é calculado invertendo a matriz M. É fácil demonstrar que a inversão da matriz M resulta na inversão dos blocos $\overline{K}_{VV}$ e $\widetilde{K}_{II}$. Nota-se que o vetor z também esta dividido em dois blocos correspondentes aos vértices e às interfaces dos subdomínios, tal que o primeiro bloco esta relacionado com $\overline{K}_{VV}$ e o segundo bloco esta relacionado com $\widetilde{K}_{II}$

No bloco $\overline{K}_{VV}$ observa-se:

1. A inversão do bloco $\overline{K}_{VV}$ aproxima as funções de forma referentes aos vértices a uma função que não depende da malha. Pode-se afirmar isto porque a função que está sendo aproximada é a solução da equação diferencial $\Delta u_h = 0$ em cada subdomínio. As condições de contorno de essa equação diferencial são os valores das funções de forma associadas aos vértices. Estas condições de contorno sempre são as mesmas. Isto porque as funções associadas aos vértices sempre são distribuídas linearmente ao longo das interfaces, não importando o número de elementos que tenha o subdomínio. Na FIGURA 8-9 é mostrado que os vértices das malhas tem as mesmas funções de forma, para dois refinamentos diferentes.

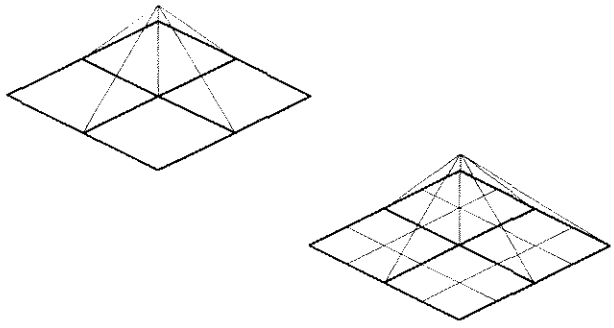

FIGURA 8-9 As funções de forma dos vértices são independentes do refinamento da malha

2. Seja a função $z_h^{n+1}(x,y)$ associado ao vetor z^{n+1} tal que:

   $$z_h^{n+1}(x,y) = \sum z_i^{n+1} \overline{\varphi}_i(x,y)$$

   Esta função converge para uma função $z^{n+1}(x,y)$ que é independente da malha. Isto porque segundo a observação 1 o problema variacional relacionado aos vértices é igual para qualquer malha e com as mesmas condições de contorno.

Para o bloco $\widetilde{K}_{II}$ tem se as seguintes observações:

3. A solução $z_h^{n+1}(x,y) = \sum z_i^{n+1} \varphi_i(x,y)$ calculada para os nós de interface equivale a resolver a equação diferencial com condição de Dirichlet no contorno sobre os dois domínios que ela separa (FIGURA 8-10). Estes valores no contorno são $z_h^n(x,y)$ ou seja correspondem ao valor obtido na iteração anterior. O dito anteriormente nos permite escrever:

$$a(z^{n+1}, \varphi) = f(\varphi)$$
$$z^{n+1}(s) = z^n(s) \qquad s \in \partial\, \Omega^s$$

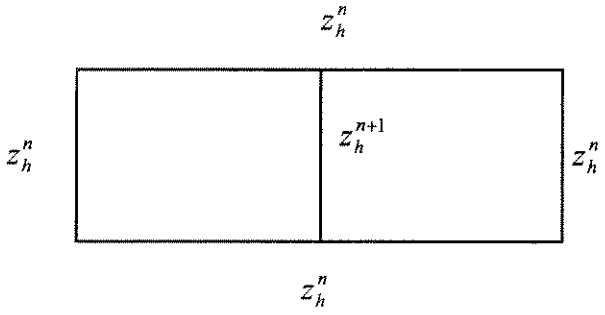

FIGURA 8-10 Problema de Dirichlet entre dois subdominios

Das observações anteriores pode se afirmar que este pré condicionador converge para um comportamento único independente do refinamento da malha.

8.6. TESTES NUMÉRICOS

Para realizar os testes numéricos sobre o pré-condicionador implementado, foi criada uma classe **TGenGrid** que, dados dois pontos (x_0,y_0) e (x_1,y_1) (FIGURA 8-11), constrói um domínio retangular. Os elementos do domínio são criados dividindo-se dois lados perpendiculares do retângulo em números dados de partições. Esta classe permite ainda criar superelementos, através do agrupamento de elementos criados da forma descrita.

Na FIGURA 8-11 é mostrado um exemplo de uma malha criada pela classe **TGenGrid**. Esta malha tem 5x4 elementos e tem superelementos de 2x2 elementos. Observa-se que como não puderam ser criados todos os superelementos de 2x2, esta classe criou automaticamente mais 2 superelementos de 1x2 elementos.

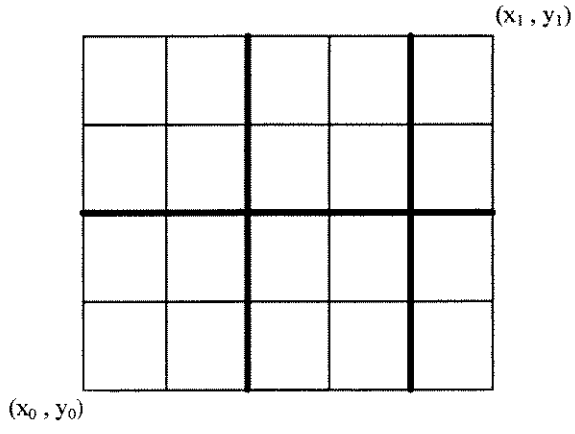

FIGURA 8-11 Exemplo de malha gerada pela classe TGemGrid

A seguir são descritos os testes realizados.

8.6.1. PRIMEIRO TESTE: COMPORTAMENTO DO PRÉ-CONDICIONADOR USANDO ADAPTATIVIDADE h

Para observar o comportamento do pré-condicionamento frente ao refinamento tipo *h* foram criados várias malhas de superelementos. Foi utilizado o problema de Poisson com condições de Dirichlet em todo o contorno do problema. Cada superelemento, que inicialmente tinha 4 elementos, foi refinado para 9 elementos, em

seguida para 16 elementos, e por último para 25 elementos. Todas estas aproximações usaram uma ordem de interpolação cúbica. Todas as malhas dos superelementos tiveram o comportamento mostrado no GRÁFICO 8-1.

Observa-se que nas iterações iniciais as curvas coincidem. Isso leva a supor que em maiores refinamentos encontram-se maiores pontos de coincidência. Essa afirmação pode ser verificada claramente no GRÁFICO 8-2.

A partir daí, pode-se dizer que para uma certa precisão requerida tem-se um refinamento "mínimo", tal que essa precisão é alcançada com o mesmo número de iterações, não importando se o refinamento é maior que o "mínimo". Por exemplo, nota-se no GRÁFICO 8-1 que uma precisão de duas casas decimais é alcançada em três iterações, não importando o refinamento dos subdomínios.

COMPORTAMENTO DO PRÉ CONDICIONADOR NA ADAPTATIVIDADE TIPO h

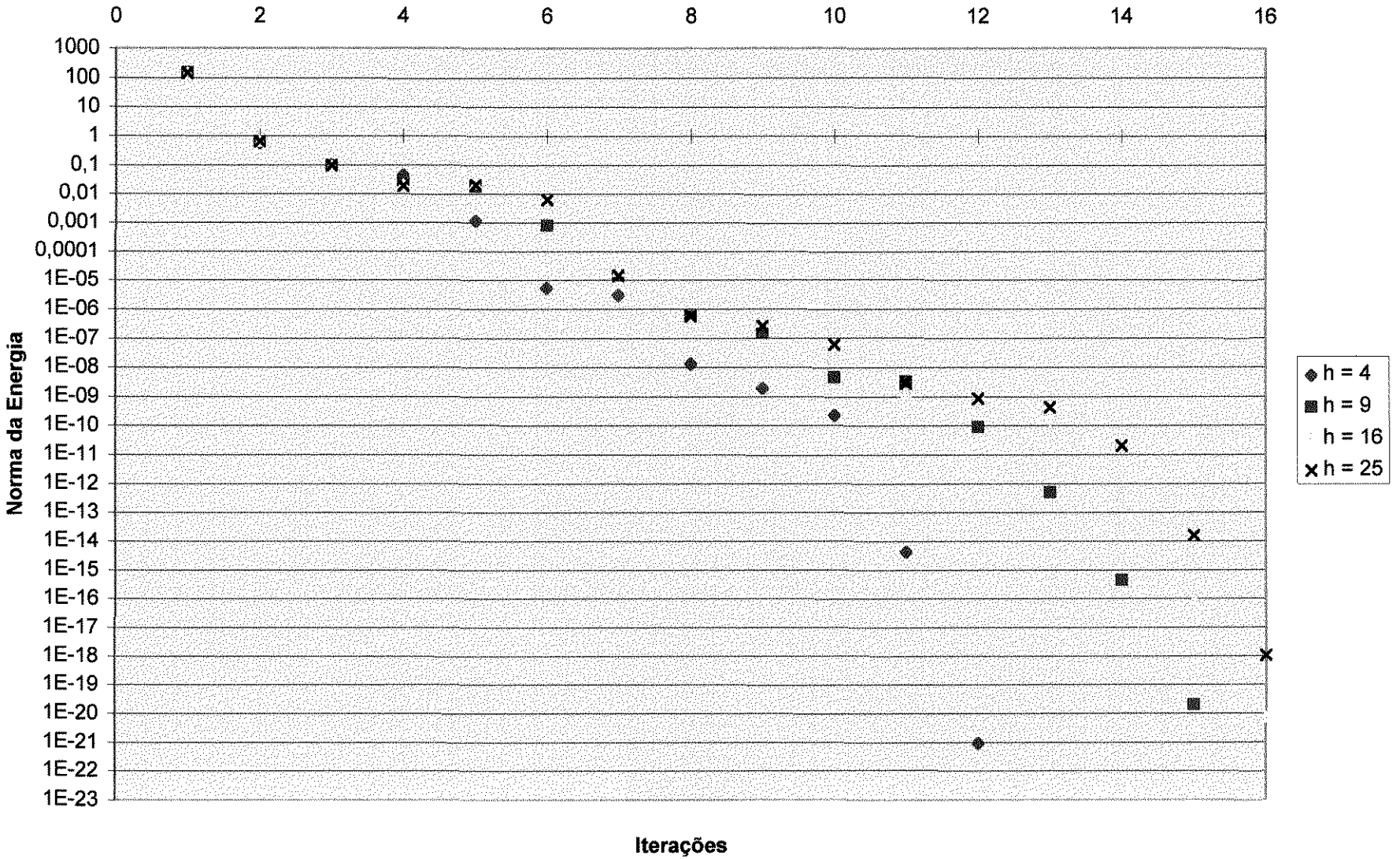

GRÁFICO 8-1

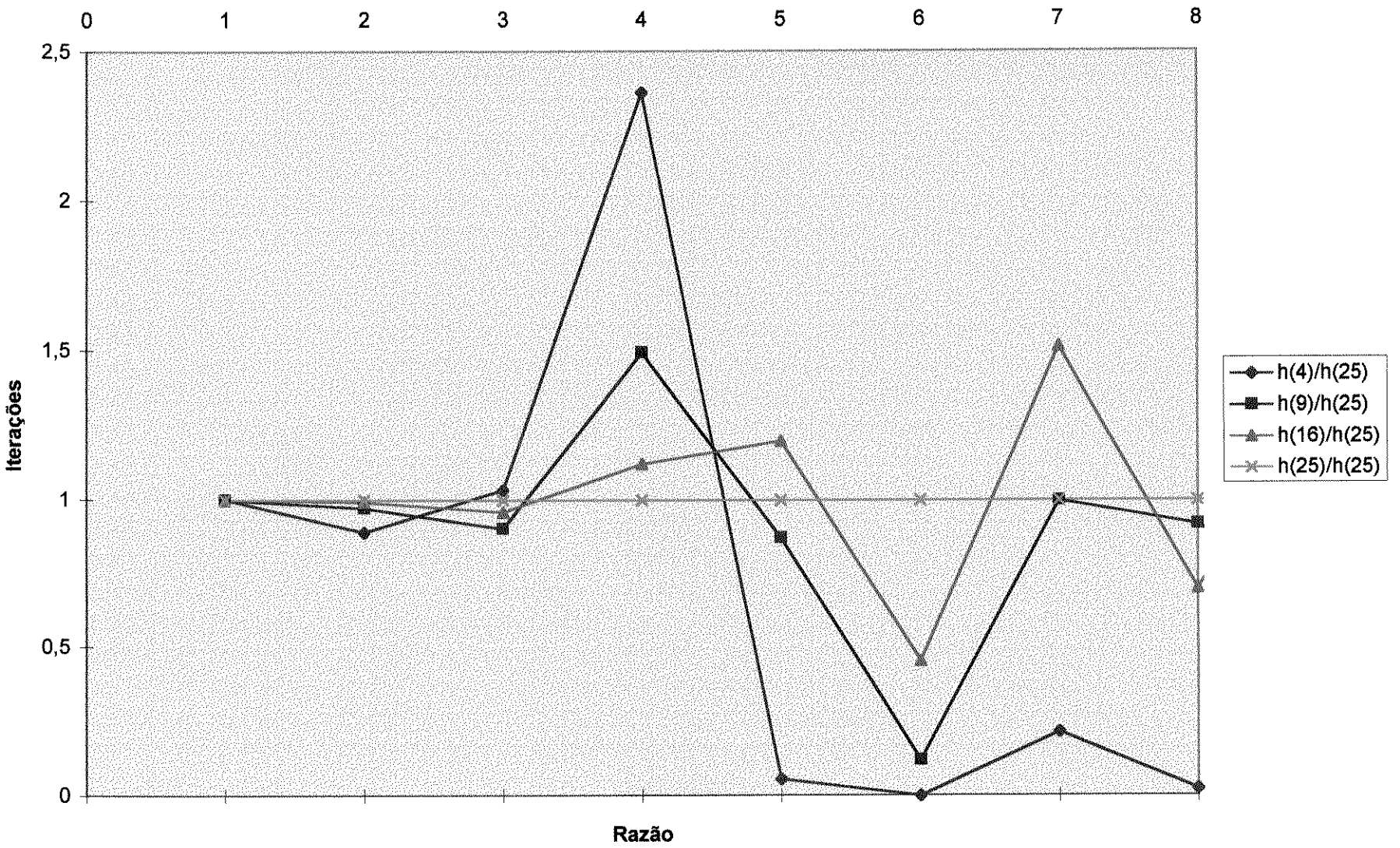
COMPARAÇÃO ENTRE OS REFINAMENTOS TESTADOS
0
1
2
3
4
5
6
7
8
2,5
2
1,5
1
0,5
0
Iterações
Razão
h(4)/h(25)
h(9)/h(25)
h(16)/h(25)
h(25)/h(25)

GRÁFICO 8.2

8.6.2. SEGUNDO TESTE: COMPORTAMENTO DO PRÉ-CONDICIONADOR USANDO ADAPTATIVIDADE p

Para observar o comportamento do pré-condicionamento frente à adaptatividade tipo p foi utilizado novamente o problema de Poisson com condições de Dirichlet em todo o contorno do problema. O domínio foi discretizado em 9 subdomínios, sendo que cada subdomínio foi dividido em 4 elementos. O grau de interpolação inicial foi cúbico, incrementando-se um grau a cada refinamento até o sétimo grau. Os resultados numéricos obtidos, mostrados no GRÁFICO 8-3 mostram que este tipo de aproximação tem um comportamento muito semelhante à aproximação tipo *h*. Para as primeiras iterações as curvas coincidem, e depois se distanciam. Da mesma forma que na adaptatividade tipo *h*, existe para uma aproximação dada o número de iterações que é independente do grau de interpolação. No GRÁFICO 8-4 comparam-se os graus de interpolação 3, 4, 5 e 6 com o grau 7, que foi o maior grau analisado no GRÁFICO 8-3. Observa-se no GRÁFICO 8-4 que realmente nas primeiras iterações o comportamento de qualquer ordem de interpolação é muito semelhante.

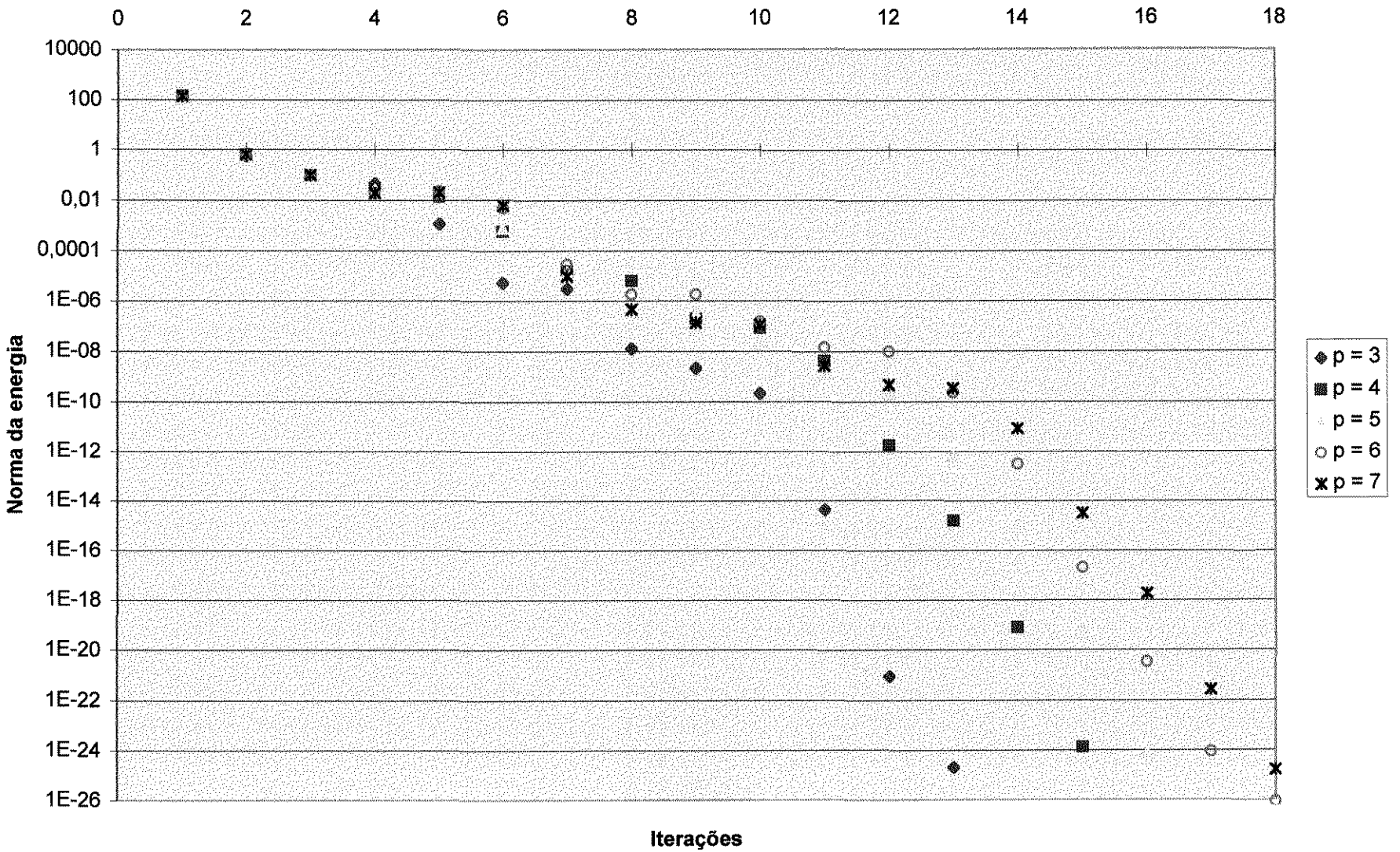
COMPORTAMENTO DO PRÉ CONDICIONADOR NA ADAPTATIVIDADE TIPO p
0
2
4
6
8
10
12
14
16
18
10000
100
1
0,01
0,0001
1E-06
1E-08
1E-10
1E-12
1E-14
1E-16
1E-18
1E-20
1E-22
1E-24
1E-26
Norma da energia
Iterações
p = 3
p = 4
p = 5
p = 6
p = 7

GRÁFICO 8.3

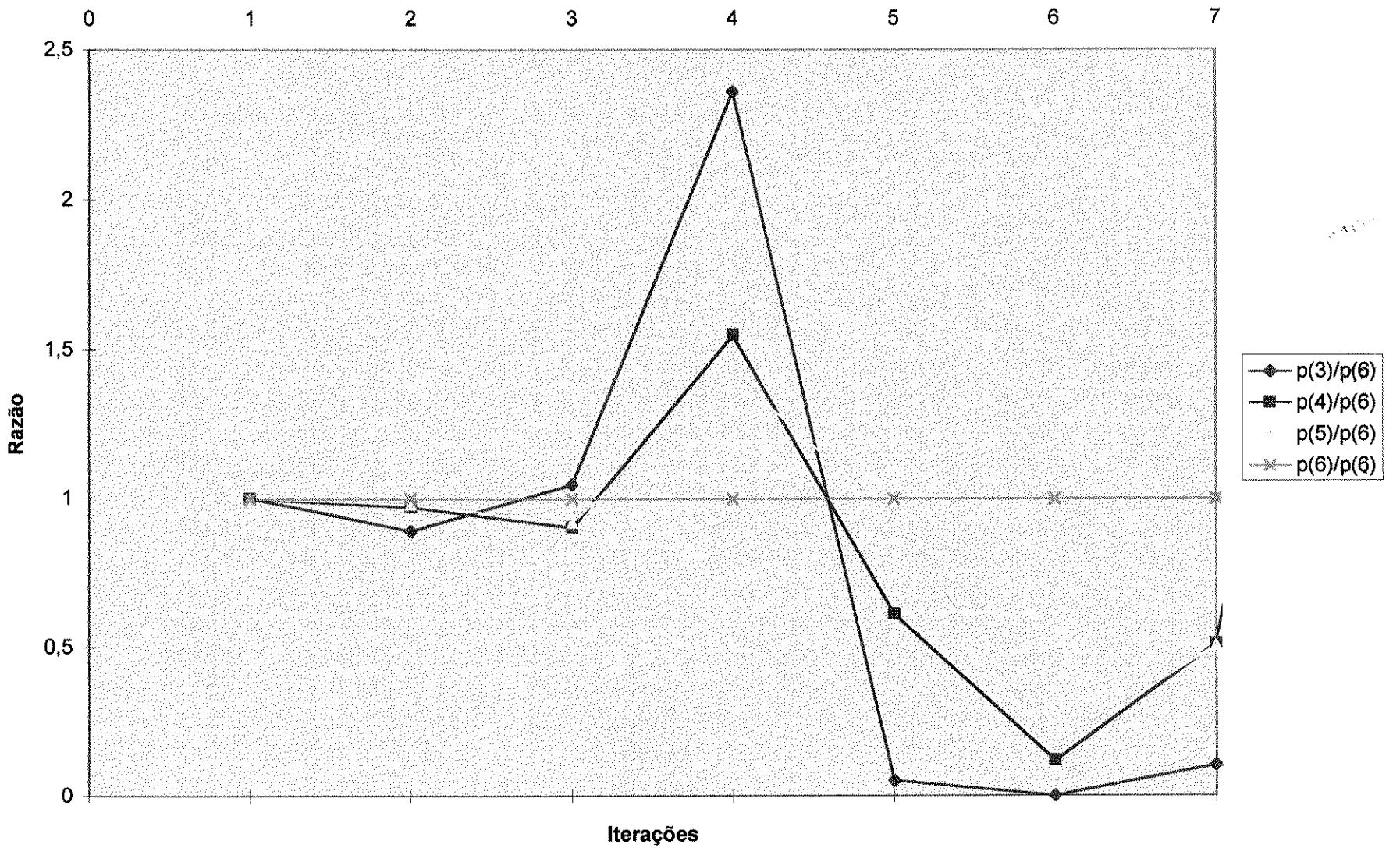
COMPARAÇÃO ENTRE OS DIFERENES GRAUS DE INTERPOLAÇÃO
0
1
2
3
4
5
6
7
2,5
2
1,5
1
0,5
0
Razão
Iterações
p(3)/p(6)
p(4)/p(6)
p(5)/p(6)
p(6)/p(6)

GRÁFICO 8.4

8.6.3. TERCEIRO TESTE: COMPARAÇÃO ENTRE AS ADAPTATIVIDADES TIPO h E p USANDO O PRÉ-CONDICIONADOR BLOCK-DIAGONAL

Conforme já foi mencionado, as adaptatividades *h* e *p*, ao que parece, têm os mesmos comportamentos quando usado o pré-condicionador Block-Diagonal. Para verificar esse fato, foi construído o GRÁFICO 8-5. Nesse gráfico foram incluídas as aproximações de sétima ordem e ainda as aproximações obtidas com 25 elementos por subdomínio. Nesse gráfico observa-se que o comportamento é idêntico. Como verificação do anterior tem-se o GRÁFICO 8-6 onde foi representado a razão entre o maior refinamento $h(25)$ com o maior grau de interpolação $p(7)$. Observa-se que a razão $h(25)/p(7)$ é igual a 1 nos primeiros pontos da curva. Do anterior pode-se concluir que para maiores ordens de iterpolação e maiores refinamentos, haverá maiores pontos de coincidência. Esse fato leva à conclusão de que essas adaptatividades têm um comportamento mais próximo entre si quanto maior é a ordem de interpolação, e assim o refinamento das adaptatividades *h* ou *p* é o mesmo.

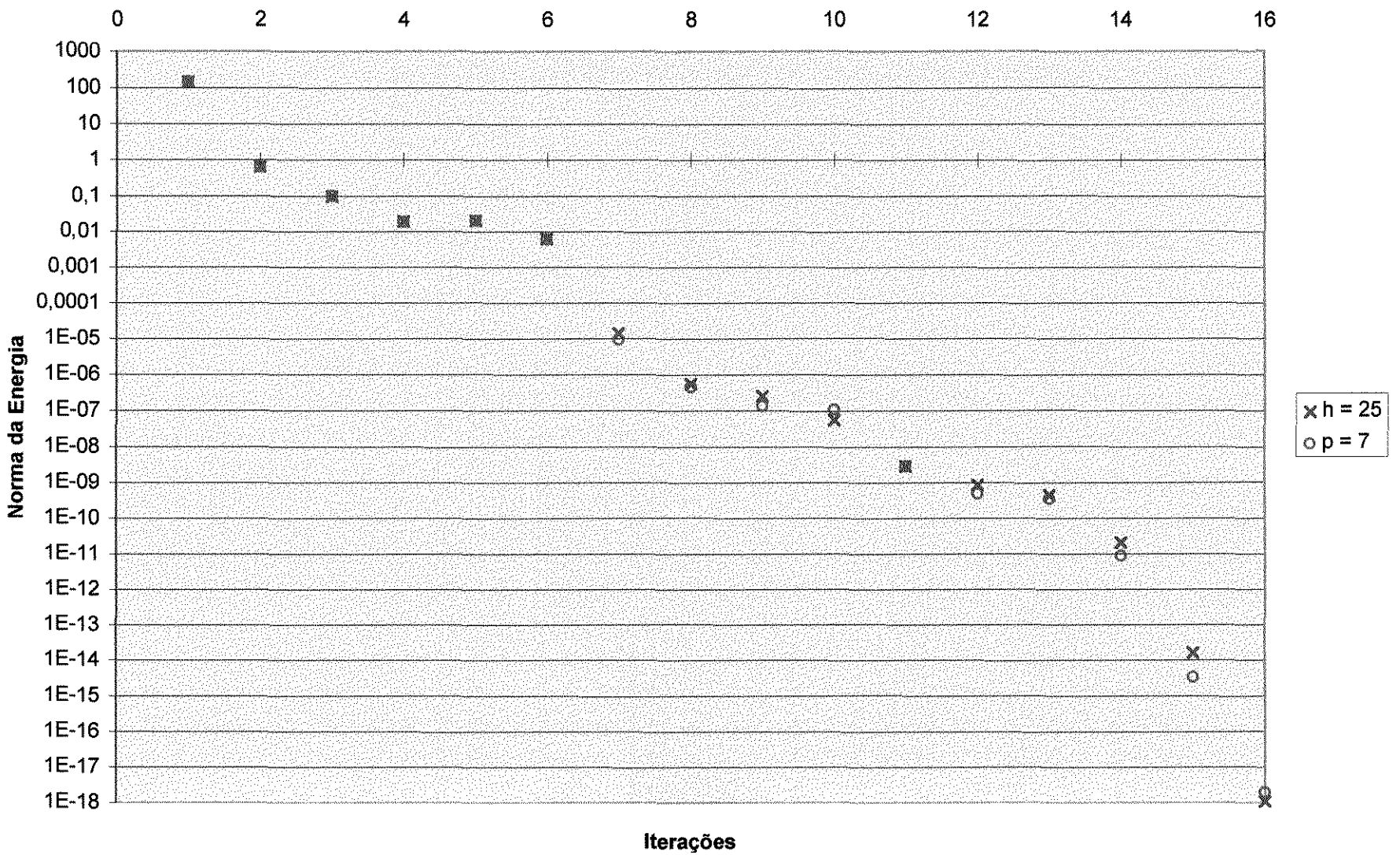
COMPARAÇÃO ENTRE AS ADAPTATIVIDADES TIPO p E h
0
2
4
6
8
10
12
14
16
1000
100
10
1
0,1
0,01
0,001
0,0001
1E-05
1E-06
1E-07
1E-08
1E-09
1E-10
1E-11
1E-12
1E-13
1E-14
1E-15
1E-16
1E-17
1E-18
Norma da Energia
Iterações
h = 25
p = 7

GRÁFICO 8.5

COMPARAÇÃO ENTRE O MAIOR REFINAMENTO h(25) E O MAIOR GRAU DE INTERPOLAÇÃO p(7)

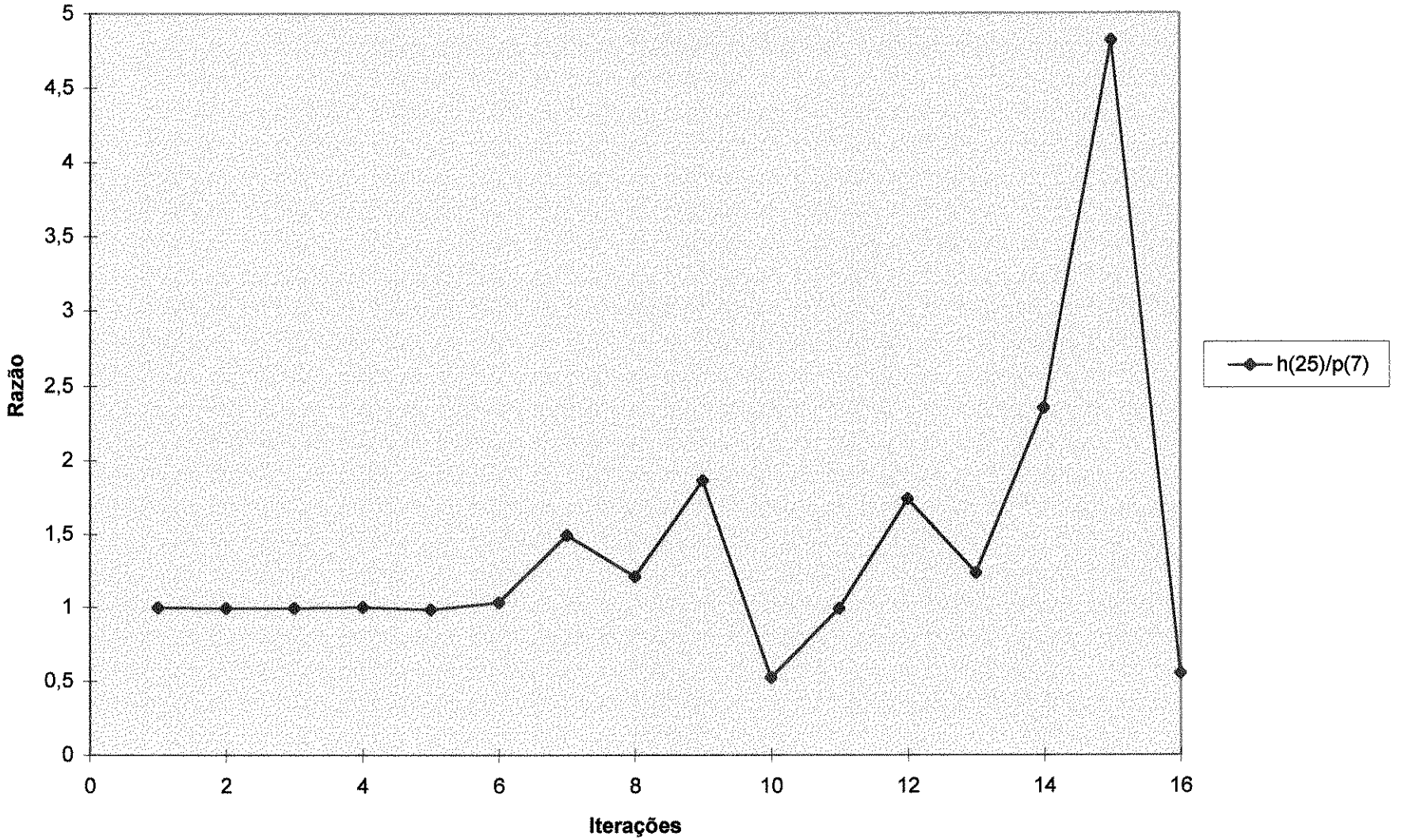

GRÁFICO 8.6

8.6.4. QUARTO TESTE: *MELHORAMENTO DO CONDICIONAMENTO DA MATRIZ DE RIGIDEZ*

Para calcular os autovalores máximos e mínimos das matrizes K [5] e $M^{-1}K$ [6] foram feitos os mesmos testes dos ítens 8.5.1 e 8.5.2. Os resultados obtidos são mostrados na TABELA 8-1. Pode-se observar que o número de condicionamento melhora bastante tanto para adaptatividade *h* e *p*. Este teste indica que este pré-condicionador é muito mais eficiente quando usada a adaptatividade tipo *p*. No caso em que foi usado um grau de interpolação de ordem 7, o número de pré-condicionamento foi diminuído de 899 para 19.

O melhoramento do condicionamento da matriz também teve bons resultados para o problema elástico descrito a seguir. Foi utilizado o domínio retangular dos primeiros testes, tendo como condições de contorno dois lados paralelos engastados e os outros livres, com uma carga uniformemente distribuída sobre um dos lados livres. Na TABELA 8-2 podem ser observados os resultados dos cálculos dos autovalores máximos e mínimos para a matriz K e $M^{1}K$. Igualmente ao teste anterior, estas matrizes correspondem às novas bases.

---

[5] Lembre-se que neste ponto do calculo a matriz K é da matriz reduzida com mudança de bases.

[6] A matriz pré-condicionadora com mudança de bases

PROBLEMA DE POISSON

	p=2				4 ELEMENTOS POR SUBDOMÍNIO			
	4	9	16	25	p=4	p=5	p=6	p=7
$\lambda(K)_{min}$	0.0820	0.0849	0.0858	0.0860	0.0177	0.0083	0.0054	0.0032
$\lambda(K)_{max}$	4.0570	3.8308	3.6476	3.5213	2.8707	2.8734	2.8737	2.8735
$\lambda(K)_{max}/\lambda(K)_{min}$	49.5014	45.1440	42.5270	40.9538	162.4348	346.0888	531.0021	899.3582
$\lambda(M^{-1}K)_{min}$	0.2263	0.1854	0.1623	0.1471	0.1584	0.1342	0.1181	0.1064
$\lambda(M^{-1}K)_{max}$	1.6780	1.7172	1.7409	1.7572	1.7194	1.7507	1.7726	1.7892
$\lambda(M^{-1}K)_{max}/\lambda(M^{-1}K)_{min}$	7.4140	9.2603	10.7293	11.9454	10.8527	13.0422	15.0148	16.8155

PROBLEMA ELÁSTICO

	p=2				4 ELEMENTOS POR SUBDOMÍNIO			
	4	9	16	25	p=4	p=5	p=6	p=7
$\lambda(K)_{min}$	0.0550	0.0551	0.0546	0.0535	0.0110	0.0052	0.0035	0.0020
$\lambda(K)_{max}$	4.2059	3.9870	3.7871	3.6433	3.0492	3.0465	3.0467	3.0465
$\lambda(K)_{max}/\lambda(K)_{min}$	76.4845	72.3790	69.3420	68.0445	278.3374	583.6257	861.4444	1501.7991
$\lambda(M^{-1}K)_{min}$	0.1782	0.1471	0.1294	0.1178	0.1428	0.1261	0.1061	0.0984
$\lambda(M^{-1}K)_{max}$	2.0771	2.0887	2.0953	2.0995	2.0908	2.0971	2.1037	2.1064
$\lambda(M^{-1}K)_{max}/\lambda(M^{-1}K)_{min}$	11.6545	14.2027	16.1899	17.8248	14.6399	16.6319	19.8197	21.3973

8.6.5. PONTOS DE SINGULARIDADE

Quando os testes numéricos do item 8.5.4 foram realizados, também foram "plotados" os espaços de autovetores gerados pelos autovalores mínimos e máximos das matrizes K e $M^{-1}K$, onde K é a matriz condensada correspondente às novas bases. A matriz M é a matriz pré-condicionadora também correspondente às novas bases.

O autovalor mínimo da matriz K em todos os testes mostrou que o espaço dos autovetores gerado pelo autovalor mínimo tem pontos de singularidade nos vértices dos subdomínios, como mostra a FIGURA 8-12.

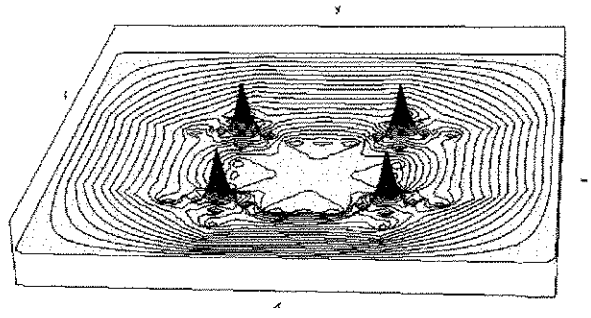

FIGURA 8-12- Espaço de autovetores gerado pelo autovalor mínimo da matriz K.

O autovalor mínimo da matriz $M^{-1}K$ em todos os testes mostrou que o espaço dos autovetores gerado pelo autovalor mínimo também tem pontos de singularidade nos vértices dos subdomínios, como mostra a FIGURA 8-13.

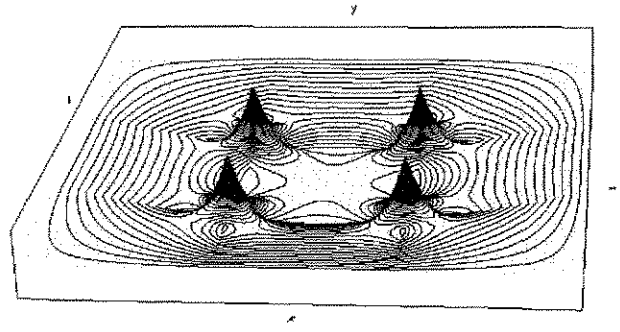

FIGURA 8-13- Espaço de autovetores gerados pelo autovalor mínimo da matriz $M^{-1}K$.

O autovalor máximo da matriz K em todos os testes mostrou que o espaço dos autovetores gerado pelo autovalor máximo tem uma singularidade nos vértices dos subdomínios. Nota-se que esta singularidade não é muito pronunciada, conforme se observa na FIGURA 8-14.

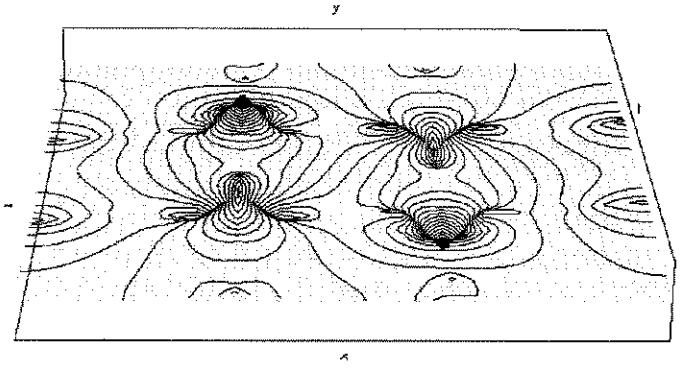

FIGURA 8-14- Espaço de autovetores gerado pelo autovalor máximo da matriz K.

O espaço de autovetores gerados pelo autovalor máximo da matriz $M^{-1}K$ tem pontos de singularidade nos vértices dos sub-domínios segundo pode se observar na FIGURA 8-15.

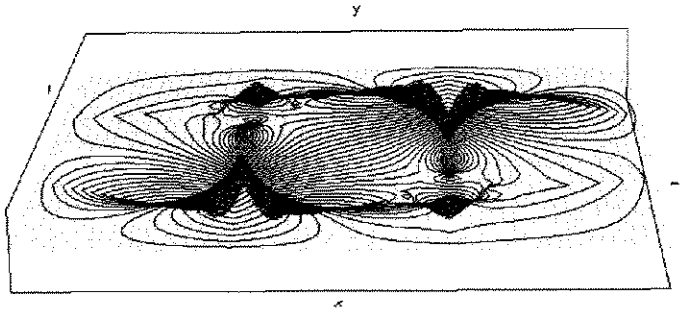

FIGURA 8-15- Espaço de autovetores gerado pelo autovalor máximo da matriz $M^{-1}K$.

Das observações anteriores pode-se concluir que o pré-condicionador *M* não consegue eliminar estes pontos de singularidade.

9. CONCLUSÕES

Este trabalho foi constituído de duas partes:

- A implementação da subestruturação dentro da filosofia orientada a objetos;
- Proposição e estudo de um pré-condicionador baseado na malha grossa. Este pré-condicionador é usado no método do Gradiente Conjugado pré-condicionado para resolver o sistema de equações resultante da subestruturação.

Na primeira etapa a técnica de sub-estruturação foi implementada de forma consistente no âmbito da filosofia de programação orientada para objetos. Duas classes formam a base desta implementação: a classe **TMatRed** e a classe **TSuperEl**.

A classe **TMatRed** representa um sistema de equações e permite particionar este sistema em dois grupos : as equações dependentes e as equações externas. Esta classe implementa as operações algébricas da condensação estática de sistemas de equações

A classe **TSuperEl** implementa a interface com o programa de elementos finitos: ela determina quais são os graus de liberdade internos e externos e apresenta uma interface consistente com a malha à qual pertence.

No âmbito do processamento foram viabilizadas as seguintes etapas:

- Cálculo da matriz de rigidez de cada subdomínio; dependendo do tamanho do problema, esta etapa pode ser realizada em diferentes processadores.
- Montagem do sistema para problema de interface $[K_{RED}][U]=[F_{RED}]$; este sistema ainda pode ser muito grande podendo ser resolvido com métodos iterativos que são os mais apropriados para processamento em paralelo.
- Um superelemento também pode formar a base para o estudo da técnica "multigrid", viabilizando a implementação do refinamento adaptativo em cada

subdomínio (superelemento). A principal motivação para o desenvolvimento deste trabalho foi o desenvolvimento de algoritmos paralelos para elementos finitos.

Na segunda parte do presente trabalho propôs-se usar um pré-condicionador construído a partir da malha grossa. Os testes realizados indicam que o comportamento deste pré-condicionador tem uma tendência a ter o mesmo comportamento para as adaptatividades tipo *p* e tipo *h* para ordens de interpolação altas e malhas com alto grau de refinamento. Também foi notado que o numero de pré-condicionamento é bastante reduzido melhorando o condicionamento da matriz de rigidez. Embora esta redução seja mais eficiente na adaptatividade tipo *p*, o efeito é o mesmo, pois a redução leva aos mesmos níveis de número de pré-condicionamento que os obtidos na adaptatividade tipo *h*. Observou-se que o pré-condicionador teve um bom desempenho para o problema elástico, à medida em que o número de condicionamento melhorou bastante.

Os autovetores correspondentes aos autovalores mínimos e máximos da matriz condensada apresentam pontos de singularidade nos vértices dos subdomínios. Estes pontos de singularidade são suavizados pelo pré-condicionador mas não são eliminados.

A mudança de base é fácil de se implementar, pois é realizada usando a matriz de rigidez e o vetor de carga local já calculadas nas bases padrões.

Em estudos posteriores podem ser feito um maior aprofundamento, testando-se problemas reais onde os subdominios sejam irregulares. Também poderia ser feito um estudo das singularidades detectadas nos autovalores..

Foi estudada a técnica de subestruturação desde seus conceitos básicos até sua implementação computacional, usando o paradigma da programação orientada a objetos.

10. *REFERÊNCIAS BIBLIOGRÁFICAS*

AINSWORTH, M. A hierarchical domain decomposition preconditioner for h-p finite element aproximation on locally refined meshes. *SIAM J. Sci. Comp.*, v. 17, n. 6, p.1395-1413, Nov. 1996.

ANDERSON, E. et al. *LAPACK users' guide*, Philadelphia: SIAM, 1995.

AXELSSON, D., BARKER, V. A. *Finite Element Solution of Boundary Value Problems;* Theory and Computation. Orlando: Academic Press, 1984. 432 p.

BARRET, R. et al. *TEMPLATES for the Solution of Linear Systems;* Building Blocks for Iterative Methods. Philadelphia: SIAM, 1994. 112 p. (e-mail: templates@cs.utk.edu).

BECKER, E. B., CAREY, G. F., ODEN, J. T. *Finite Elements;* An Introduction. Englewood Cliffs: Prentince-Hall, 1981. 258 p.

BJORSTAD, P. E., WIDLUND, 0. B. Iterative methods for the solution of elliptic problems on regions partitioned in substructures. *SIAM J. Numer Anal.*, v. 23, n. 6, p. 1097-1120, 1986.

BRAMBLE, J. H., PASIAK, J. E., SCHATZ, A. H. An Iterative Method for Elliptic Problems on Regions Partitioned into Substructures. *Math. Comput.*, v. 46, n. 174, p. 361-369, Apr. 1986.

CAREY, G. F., ODEN, J. T. *Finite Elements;* A Second Course. Englewood Cliffs: Prentice-Hall, 1983. 301 p.

CAREY, G. F., ODEN, J. T. *Finite Elements;* Mathematical Aspects. Englewood Cliffs: Prentice-Hall, 1983. 195 p.

CENPES-PETROBRÁS/PUC-RIO. *MVIEW Bidimensional Mesh View-Manual do Usuario.* Grupo de tecnologia em computação gráfica-TecGraf/PUC-Rio e Grupo de Geotecnia do SEDEN/DIPREX., 1995. 36p.

CHAN, T. et al. (Ed.) *Domain decomposition methods for partial differential equations.* Philadelphia: SIAM, 1990. 491 p.

DEVLOO, P. R. B. On the development of a finite element program based on the object oriented programming philodophy. In: OON-SKI PROCEEDINGS OF THE FIRST ANNUAL OBJECT ORIENTED NUMERICS CONFERENCE, 1993. *Proceedings...*Rogue Wave Software - SIAM., 1993. p.183-203.

DHATT, G., TOUZOT, G. The *Finite Element Method Displayed.* A willey-Interscience Publication, 1985. 510 p.

GOLUB, G. H., VAN LOAN, C. F. Matrix Camputations. London: The Jhons Hopkins University Press, 1989. 640 p.

IBM CORPORATION. IBM Visualization Data Explorer User's Guide. Thomas J. Watson Research Center, 1994.

KEYES, D. E., SAAD, Y., TRUHLAR, D. G. (Ed.) Domain-based parallelism and problem decomposition methods in computational science and engineering, Philadelphia: SIAM, 1995. 323 p.

LE TALLEC, P., SALTEL, E., VIDRASCU, E. M. Solving large scale structural problems on parallel computers using domain decomposition techniques, em Advances in parallel and vector processing for structural mechanics. In: ADVANCES IN PARALLEL AND VECTOR PROCESSING FOR STRUTURAL MECHANICS, 1994. Civil-Comp Ltd, Edinburg, Scotland. p.127-132.

LE TALLEC, P., SALTEL, E., VIDRASCU, E. M., Parallel domain decomposition algorithms for solving plate and shell problems. In: ADVANCES IN PARALLEL AND VECTOR PROCESSING FOR STRUTURAL MECHANICS, 1994, Edinburg, Scotland. *Proceedings...* Edinburg, Scotland: Civil-Comp. Ltd., 1994. p. 139-145.

LEA, D. *User's Guide to the GNU C++ Library.* Version 2.0. [s. l.] Free Software Foundation, 1992. 126 p. (e-mail: dl@g.oswego.edu).

MANDEL, J. Two-level domain decomposition preconditioning for the *p*-version finite element method in tree dimensions, *International Journal for Numerical Methods in Eng*ineering., v.29, p.1095-1108, 1990.

ODEN, J. T. *Applied Functional Analysis;* A First Course for Students of Mechanics and Engineering Science. Englewwod Cliffs: Prentice-Hall, 1979. 426 p.

ODEN, J. T., REDDY, J. N. *An Introduction to the Mathematical Theory of Finite Elements.* New York: John Wiley & Sons, 1976. 429 p.

PRZEMIENIECKI, J. S. Matrix strutural analysis of structuras. *Journal of the American Institute of Aeronautics and Astronautics*, v. 1, n. 1, p. 138-147, Jan. 1963.

RANK, E. Adaptivity and accuracy estimation for FEM and BIEM. In: BREBBIA, C. A., ALIABADI, M. H. (Ed.) *Adaptative Finite and Boundary Element Methods.* Southampton Boston: Computational Mechanics Publications/Elsevier Applied Science, 1993. p.1-43.

RIBEIRO, F. L. B .*Introdução à computação gráfica.* COPPE/UFRJ, 1996. 136 p.

SANTANA, M. L. M., DEVLOO, P. R. B. Object-oriented matrix classes. In: JOINT CONFERENCE OF ITALIAN GROUP OF COMPUTATIONAL MECHANICS AND IBERO-LATIN AMERICAN ASSOCIATION OF COMPUTATIONAL METHODS IN ENGINEERING, 1, 1996, Pádua, Itália. *Proceedings...* Pádua, Itália, 1996. p. 325-328.

SANTANA, M. L. M., DEVLOO, P. R. B. Desenvolvimento de classes de matrizes para aplicação no método dos elementos finitos - Relatório Final de Iniciação Cientifica, FAPESP processo N°: 93/0241-2, 1995. 95p.

SIMULOG. MUDULEF users guide n° 1. 180 p. 1995. (modulef@simulog.fr).

SMITH, B. F. An optimal domain decomposition preconditioner for the finite element solution of linear elaticity problems. *SIAM J. Sci. Stat. Comput.*, v. 13, n. 1, p. 364-378, Jan. 1992.

SMITH, B. F. A parallel implementation of an iterative substructuring algorithm for problems in three dimensions. *SIAM J. Sci. Comput.*, v. 14, n. 2, p. 406-423, Mar. 1992.

SMITH, B. F., WIDLUND, O. B. A Domain Decomposition Algorithm Using a Hierarchical Basis. *SIAM J. Sci. Stat. Comput.*, v. 11, p. 1212-1220, 1990.

SPENGERMANN, F., THIERAUF, G. A domain decomposition for large scale strutural optimization. In: ADVANCES IN PARALLEL AND VECTOR PROCESSING FOR STRUTURAL MECHANICS, 1994, Edinburg, Scotland. *Proceedings...* Edinburg, Scotland: Civil-Comp. Ltd., Edinburg, Scotland, 1994. p.159-164.

SZABÓ, B., BABUSKA, I. *Finite Element Analysis.* New York: John Wiley & Sons, 1991. 368 p.

WIDLUND O. B. Domain decomposition algorithms and the bicentennial of the French revolution (Introduction). In: CHAN, T. et al. (Ed.) *Domain decomposition methods for partial differential equations.* Philadelphia: SIAM, 1990. p. xv-xx.

YSERENTANT, H. Hierarchical Bases Give Conjugate Gradient Type Methods a Multigrid Speed of Convergence. *Applied Math. and Comput.*, New York, v. 19, p. 347-358, 1986.

YSERENTANT, H. On the Multi-Level Splitting of Finite Element Spaces. *Numer. Math.*, v. 49, p. 379-412, 1986.

ZEGLINSKI, G. W., HAN, R. P. S. Object oriented matrix classes for use in a finite element code using C++. *JNME*, v. 37, p. 3921-3937, 1994.

ZIENKIEWICZ, O. C. *The Finite Element Method in Engineering Science.* London: McGraw-Hill, 1971. 521 p.

ZIENKIEWICZ, O. C., MORGAN, K. *Finite Elements and Aproximation.* New York: John Wiley & Sons, 1983. 328 p.

11. BIBLIOGRAFIA

AINSWORTH, M. A hierarchical domain decomposition preconditioner for h-p finite element aproximation on locally refined meshes. *SIAM J. Sci. Comp.*, v. 17, n. 6, p.1395-1413, Nov. 1996.

ALVES FILHO, J. S. R., DEVLOO, P. R. B. Object Oriented Programming in Scientific Computing; The Beginning of a New Era. *Engineering Computations*, v. 8, p. 81-87. 1991.

ANDERSON, E. et al. *LAPACK users' guide*, Philadelphia SIAM, 1995.

AXELSSON, D., BARKER, V. A. *Finite Element Solution of Boundary Value Problems;* Theory and Computation. Orlando: Academic Press, 1984. 432 p.

BANK, R. E., DUPONT, T. F., YSERENTANT, H. The Hierarchical Basis Multigrid Method. *Numer. Math.*, v. 52, p. 427-458, 1988.

BARRET, R. et al. *TEMPLATES for the Solution of Linear Systems;* Building Blocks for Iterative Methods. Philadelphia: SIAM, 1994. 112 p. (e-mail: templates@cs.utk.edu).

BECKER, E. B., CAREY, G. F., ODEN, J. T. *Finite Elements;* An Introduction. Englewood Cliffs: Prentince-Hall, 1981. 258 p.

BELHACHMI, Z., BERNADI, C. Resolution of fourth-order problems by the mortar element method. *Comput. Methods in applied mechanics and engineering*, v. 116, p. 53-58, 1994.

BJORSTAD, P. E., WIDLUND, 0. B. Iterative methods for the solution of elliptic problems on regions partitioned in substructures. *SIAM J. Numer Anal.*, v. 23, n. 6, p. 1097-1120, 1986.

BRAMBLE, R. et al. Domain decomposition methods for problems with partial refinement. *SIAM J. Sci. Stat. Comput.*, v. 13, n. 1, p. 397-410, Jan. 1992.

BRAMBLE, J. H., PASIAK, J. E., SCHATZ, A. H. An Iterative Method for Elliptic Problems on Regions Partitioned into Substructures. *Math. Comput.*, v. 46, n. 174, p. 361-369, Apr. 1986.

BRAMBLE, J. H., PASIAK, J. E., SCHATZ, A. H. The construction of preconditioners for elliptic problems by substructuring. *Math. Comput.*, v. 47, n. 175, p. 103-134, July 1986.

BREBBIA, C. A., ALIABADI, M. H. (Ed.) *Adaptative Finite and Boundary Element Methods.* Southampton Boston: Computational Mechanics Publications/Elsevier Applied Science, 1993. 319 p.

BUDD, T. A. *Classic Data Structures in C++.* Reading, Massachusetts: Addison-Wesley, 1994. 543 p.

CAREY, G. F., ODEN, J. T. *Finite Elements;* A Second Course. Englewood Cliffs: Prentice-Hall, 1983. 301 p.

CAREY, G. F., ODEN, J. T. *Finite Elements;* Mathematical Aspects. Englewood Cliffs: Prentice-Hall, 1983. 195 p.

CENPES-PETROBRÁS/PUC-RIO. *MVIEW Bidimensional Mesh View-Manual do Usuario.* Grupo de tecnologia em computação gráfica-TecGraf/PUC-Rio e Grupo de Geotecnia do SEDEN/DIPREX., 1995. 36p.

CHAN, T. et al. (Ed.) *Domain decomposition methods for partial differential equations.* Philadelphia: SIAM, 1990. 491 p.

CHAN, T. F., HOU, T. Y. Eigendecomposition of domain decomposition interface operators for constant coefficient elliptic problems. *SIAM J. Sci. Stat. Comput.,* v. 12, n. 6, p. 1471-1479, Nov. 1991.

DEVLOO, P. R. B. Efficiency issues in an object oriented programming environment. In: ARTIFICIAL INTELLIGENCE AND OBJECT ORIENTED APPROACHES FOR STRUTURAL ENGINEERING, 1994, Edinburg, Scotland. *Proceedings...* Edinburg, Scotland: Civil-Comp. Ltd., 1994. p.147-157.

DEVLOO, P. R. B. On the development of a finite element program based on the object oriented programming philodophy. In: OON-SKI PROCEEDINGS OF THE FIRST ANNUAL OBJECT ORIENTED NUMERICS CONFERENCE, 1993. *Proceedings...*Rogue Wave Software - SIAM., 1993. p.183-203.

DEVLOO P. R. B., SANTANA, M. L. M. Desenvolvimento de algoritmo de sub-estruturação para elementos finitos. In: ENCIT-CONGRESSO BRASILEIRO DE ENGENHARIA E CIENCIAS TERMICAS/LATCYM-CONGRESO LATINOAMERICANO DE TRANSFERENCIA DE CALOR Y MATERIA, 6, 1996. *Proceedings...* Florianópolis, Brasil. P. 505-511.

DRYJA, M. A finite element-capacitance matrix method for the elliptic problem. *SIAM J. Numer. Anal.,* v. 20, n. 20, p. 671-680, Aug. 1983.

ELLIS, M. A., STROUSTRUP, B. C. Manual de Referência Comentado. Rio de Janeiro: Editora CAMPUS, 1993. 546 p.

FARHAT, C.H., ROUX, F. X. An unconventional domain decomposition method for an efficient parallel of large scale finite element system. *SIAM J. Sci. Stat. Comput.*, v. 13, n. 1, p. 379-396, Jan. 1992.

GEORGE P. L., Automatic Mesh Generation-Application To Finite Element Methods, Paris: Wiley-Masson, 1991. 333 p.

GOLUB, G. H., VAN LOAN, C. F. Matrix Camputations. London: The Jhons Hopkins University Press, 1989. 640 p.

GROPP W. D., KEYES D. E. Domain decomposition methods in computational fluid dynamics. *Int. J. Numer. Methods Fluids,* v. 14, p. 147-165, 1992.

IBM CORPORATION. IBM Visualization Data Explorer User's Guide. Thomas J. Watson Research Center, 1994.

JUN, L. U., WHITE, D. W., CHEN, W. F. et al. A matrix class library in C++ for structural engineering computing. *Computers & Structures*, v. 55, n. 1, p. 95-111, 1995.

KEYES, D. E., SAAD, Y., TRUHLAR, D. G. (Ed.) Domain-based parallelism and problem decomposition methods in computational science and engineering, Philadelphia: SIAM, 1995. 323 p.

KEYES, D. E., GROPP, W. D. A Comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation, *SIAM J. Numer. Anal.* v.20, n.4, p.671-680. 1983.

LE TALLEC, P., DE ROECK, Y. H., VIDRASCU, E. M. Domain decomposition methods for large linearly elliptic three dimensional problems. *Journal of Computational and Applied Mathematics*, v. .34, p. 93-117, 1991.

LE TALLEC, P., SALTEL, E., VIDRASCU, E. M. Solving large scale structural problems on parallel computers using domain decomposition techniques, em Advances in parallel and vector processing for structural mechanics. In: ADVANCES IN PARALLEL AND VECTOR PROCESSING FOR STRUTURAL MECHANICS, 1994. Civil-Comp Ltd, Edinburg, Scotland. p.127-132.

LE TALLEC, P., SALTEL, E., VIDRASCU, E. M., Parallel domain decomposition algorithms for solving plate and shell problems. In: ADVANCES IN PARALLEL AND VECTOR PROCESSING FOR STRUTURAL MECHANICS, 1994, Edinburg, Scotland. *Proceedings...* Edinburg, Scotland: Civil-Comp. Ltd., 1994. p. 139-145.

LEA, D. *User's Guide to the GNU C++ Library.* Version 2.0. [s. l.] Free Software Foundation, 1992. 126 p. (e-mail: dl@g.oswego.edu).

MANDEL, J. Two-level domain decomposition preconditioning for the *p*-version finite element method in tree dimensions, *International Journal for Numerical Methods in Engineering*., v.29, p.1095-1108, 1990.

MANDEL, J. *Iterative solvers by substruturing for the p-version finite element method.* 1989, 12 p. (To appear in a special issue at Proceedings of an International Conference On Spectral And High Order Methods, Como, Italy, 1989)

ODEN, J. T. *Applied Functional Analysis;* A First Course for Students of Mechanics and Engineering Science. Englewwod Cliffs: Prentice-Hall, 1979. 426 p.

ODEN, J. T., REDDY, J. N. *An Introduction to the Mathematical Theory of Finite Elements.* New York: John Wiley & Sons, 1976. 429 p.

PRZEMIENIECKI, J. S. Matrix strutural analysis of structuras. *Journal of the American Institute of Aeronautics and Astronautics*, v. 1, n. 1, p. 138-147, Jan. 1963.

RANK, E. Adaptivity and accuracy estimation for FEM and BIEM. In: BREBBIA, C. A., ALIABADI, M. H. (Ed.) *Adaptative Finite and Boundary Element Methods.* Southampton Boston: Computational Mechanics Publications/Elsevier Applied Science, 1993. p.1-43.

RIBEIRO, F. L. B .*Introdução à computação gráfica.* COPPE/UFRJ, 1996. 136 p.

RUBINSTEIN, M. F., Matrix Computer Analysis Of Structures. New Jersey: Prentince Hall, Inc. 1966. 402p.

SANTANA, M. L. M., DEVLOO, P. R. B. Object-oriented matrix classes. In: JOINT CONFERENCE OF ITALIAN GROUP OF COMPUTATIONAL MECHANICS AND IBERO-LATIN AMERICAN ASSOCIATION OF COMPUTATIONAL METHODS IN ENGINEERING, 1, 1996, Pádua, Itália. *Proceedings...* Pádua, Itália. p. 325-328.

SIMULOG. MUDULEF users guide nº 1. 180 p. 1995. (modulef@simulog.fr).

SMITH, B. F. An optimal domain decomposition preconditioner for the finite element solution of linear elaticity problems. *SIAM J. Sci. Stat. Comput.*, v. 13, n. 1, p. 364-378, Jan. 1992.

SMITH, B. F. A domain decomposition algorithm for elleptic problems in three dimensions. *Numer. Math.*, v. 60, p. 219-234, 1991.

SMITH, B. F. A parallel implementation of an iterative substructuring algorithm for problems in three dimensions. *SIAM J. Sci. Comput.*, v. 14, n. 2, p. 406-423, Mar. 1992.

SMITH, B. F., WIDLUND, O. B. A Domain Decomposition Algorithm Using a Hierarchical Basis. *SIAM J. Sci. Stat. Comput.*, v. 11, p. 1212-1220, 1990.

SPENGERMANN, F., THIERAUF, G. A domain decomposition for large scale strutural optimization. In: ADVANCES IN PARALLEL AND VECTOR PROCESSING FOR STRUTURAL MECHANICS, 1994, Edinburg, Scotland. *Proceedings...* Edinburg, Scotland: Civil-Comp. Ltd., Edinburg, Scotland, 1994. p.159-164.

SZABÓ, B., BABUSKA, I. *Finite Element Analysis.* New York: John Wiley & Sons, 1991. 368 p.

YSERENTANT, H. Hierarchical Bases Give Conjugate Gradient Type Methods a Multigrid Speed of Convergence. *Applied Math. and Comput.*, New York, v. 19, p. 347-358, 1986.

YSERENTANT, H. On the Multi-Level Splitting of Finite Element Spaces. *Numer. Math.*, v. 49, p. 379-412, 1986.

ZEGLINSKI, G. W., HAN, R. P. S. Object oriented matrix classes for use in a finite element code using C++. *JNME*, v. 37, p. 3921-3937, 1994.

ZIENKIEWICZ, O. C. *The Finite Element Method in Engineering Science.* London: McGraw-Hill, 1971. 521 p.

ZIENKIEWICZ, O. C., MORGAN, K. *Finite Elements and Aproximation.* New York: John Wiley & Sons, 1983. 328 p.

12. ANEXO: CÓDIGO EXEMPLO

Neste anexo é explicado com mais detalhes como se tem acesso os objetos do ambiente PZ. Para isto será explicada a função ReadMaterialBC que pertence ao programa exemplo do capitulo 4. Este exemplo aproxima um problema elástico para um material isotrópico. A função ReadMaterialBC tem as seguintes partes:

- Leitura do número de dados (da linha 2 a linha 6);
- Leitura dos Coeficientes da equação diferencial (da linha 7 a linha 14);
- Inicialização do número de variáveis da equação diferencial(da linha 16 a linha 20);
- Leitura das condições de contorno (da linha 22 a linha 39);
- Criação dos elementos computacionais (da linha 40 a linha 50);
- Criação de um objeto **TAnalysis** (linha 51).

Primeiro será mostrado o código e depois a sua explicação segundo os itens mencionados acima.

12.1. CÓDIGO EXEMPLO

```
1. void ReadMaterialBC(char *fname, TCompGrid &c, TAnalysis * &an){
2. //Leitura do numero de dados
3.      fstream arq(fname);
4.      int nummat,numbc;
5.
6.      arq  >> nummat >> numbc;
7. //Leitura dos coeficientes da equacao diferencial
8.      for(int im=0; im<nummat; im++) {
9.          int matnum;
10.         double E,nu,fx,fy;
11.         arq >> matnum >> E >> nu >> fx >> fy;
12.         TElasticityMaterial *mp = new TElasticityMaterial(matnum,E,nu,fx,fy);
```

```
13.     (c.MaterialMap())[matnum] = mp;
14.     }
15.
16.// Inicializacao do numero de variaveis
17.     Pix i = c.MaterialMap().first();
18.     int numvariables=0;  TMaterial *mp = 0;
19.     if(i) mp = (TMaterial *) c.MaterialMap().contents(i);
20.     if(mp) numvariables = mp->NumVariables();
21.
22.//Leitura das condicoes de contorno
23.
24.     TFMatrix val1(numvariables,numvariables), val2(numvariables,1);
25.     for (short bc = 0; bc < numbc; bc++) {
26.             int bcnumber,bctype, matnumber;
27.             arq >> bcnumber >> matnumber >> bctype;
28.             int k;
29.             for(k=0;k<numvariables;++k)
30.                     for(int l=0;l<numvariables;++l)
31.                             arq >> val1(k,l);
32.             for(k=0;k<numvariables;++k)
33.                     arq >> val2(k,0);
34.             TBndCond* bndcd;
35.             bndcd = mp->CreateBc( bcnumber, bctype, val1, val2);
36.             c.BndCondMap()[bcnumber] = bndcd;
37.     }
38.
39.// creacao de elementos computacionais
40.     TGeoGrid &g = *c.ReferenceGrid();
41.     i = g.ElementMap().first();
42.             TGeoEl *gel =0;
43.             TCompEl *cel = 0;
```

```
44.    while(i) {
45.            gel = (TGeoEl *) g.ElementMap().contents(i);
46.            g.ElementMap().next(i);
47.            cel = gel->CreateCompEl();
48.            long id = cel->Id();
49.            c.ElementMap()[id] = cel;
50.    }
51.  an = new TAnalysis(&c);
52.
53.}
```

12.2. EXPLICAÇÃO DO CÓDIGO EXEMPLO

12.2.1. LEITURA DO NÚMERO DE DADOS

Segundo este código os dados são lidos do arquivo fname segundo a entrada arq. Estes dados são: número de materiais (nummat) e número de condições de contorno (numbc).

12.2.2. LEITURA DOS COEFICIENTES DA EQUAÇÃO DIFERENCIAL

Nesta parte é executada nummat ciclos para leitura dos coeficientes da equação diferencial. Para cada um destes ciclos tem-se:

- Linha 11: Leitura de E (modulo de Young), nu (coeficiente de Poisson), fx (força de volume na direção x) e fy (força de volume na direção y).
- Linha 12: Criação dinâmica do Objeto mp do tipo TElasticityMaterial. Este objeto é criado segundo o E, nu, fx, e fy.
- Linha 13: O objeto mp é colocado com o identificador matnum dentro da árvore de materiais c.MaterialMap() que pertence a malha computacional c.

12.2.3. INICIALIZAÇÃO DO NÚMERO DE VARIÁVEIS

Como mencionado anteriormente o ambiente Pz armazena seus dados em árvores binárias [ref]. O acesso aos nós destas árvores é dado por pseudo-índices chamados de Pix. Um objeto do tipo Pix tem comportamento de ponteiro e de índice.

O armazenamento em árvore binária esta implementada na biblioteca de classes Gnu++ [ref] com o nome de *AVLMaps*. Alguma de suas funções mais usadas são:

first() : Retorna o primeiro elemento da árvore. Retorna zero se esta vazia.

next(ind) : ind avança para o próximo elemento da árvore(ind é um Pix)

seek(k) : Retorna o índice Pix do elemento. O argumento k é o identificador do nó.

contents(i) : Retorna o conteúdo do nó com Pix=i.

Neste código tem-se as seguintes linhas:

- Linha 17 : É declarado um índice tipo Pix que aponta ao primeiro elemento da árvore de materiais da malha computacional c
- Linha 18 : Declaração do inteiro numvariables onde vai ser armazenado o número de variáveis da equação diferencial. Também é declarado mp como ponteiro para **TMaterial**.
- Linha 19 : Se i existe, então mp aponta para o objeto tipo TMaterial de indice i (tipo Pix). Este objeto esta dentro da árvore de materiais que pertence a malha computacional c.
- Linha 20 : Se mp existe numvariables armazena o valor retornado pela função NumVariables() que pertence a classe **TMaterial.**

12.2.4. LEITURA DAS CONDIÇÕES DE CONTORNO.

Na linha 24 são declaradas matrizes do tipo cheia para armazenar as condições de contorno. As dimensões desta matriz são concordantes com o numero de variáveis da equação diferencial que esta sendo aproximada,

•Linha 25 : Inicio dos numbc ciclos para criação das condições de contorno.

•Linha 27 : Leitura do número (*Id*) da condição de contorno (bcnumber), leitura do número (*Id*) do material (matnumber) e do tipo da condição de contorno (bctype). Estes tipos podem ser condições mistas, de Dirichlet ou Neumann.

•Linha 29-33 : Nestes ciclos são preenchidas as matrizes com os valores da condição de contorno corrente (bcnumber).

•Linha 34 : Declaração de um ponteiro bndcd do tipo **TBndCond**.

•Linha 35 : O objeto bndcd aponta para as condições de contorno criadas dentro do material mp. Estas condições de contorno tem número (*Id*) bcnumber, são do tipo bctype e tem seus valores armazenados nas matrizes val1 e val2.

•Linha 36 : No mapeamento das condições de contorno da malha computacional c é inserido a condição de contorno bndcd com o número (*Id*) bcnumber.

•Linha 37 : Fim dos numbc ciclos.

12.2.5. CRIAÇÃO DE ELEMENTOS COMPUTACIONAIS

•Linha 40 : É declarado um objeto **TGeoGrid** que está referenciada à malha geométrica retornada por ReferenceGrid() da malha computacional c.

•Linha 41 : O *Pix* i aponta para o primeiro elemento da árvore de elementos da malha geométrica g.

•Linha 42 : Declaração de um ponteiro nulo (*gel) para um elemento geométrico (**TGeoEl**).

•Linha 43 : Declaração de um ponteiro nulo (*cel) para um elemento computacional (**TCompEl**).

•Linha 44 : Enquanto exista elementos geométricos (i) no mapeamento serão executadas as linhas 45-49.

•Linha 45 : O elemento geométrico gel aponta para o elemento geométrico i da malha g.

•Linha 46 : i avança para o próximo nó da árvore de elementos da malha g.

•Limha 47 : O elemento computacional cel aponta para o elemento computacional criado tendo como base gel.

- Linha 48 : id armazena o identificador (Id) do elemento computacional cel.
- Linha 49 : O elemento cel é adicionado a árvore de elementos da malha computacional c com identificador id.

12.2.6. CRIAÇÃO DE UM OBJETO TAnalysis.

Depois de ter colocado todas as condições de contorno e criado todos os elementos computacionais, já é possivel construir um objeto do tipo **TAnalysis.** Este objeto calcula a seqüência dos nos geométricos, coletividade. Na linha 51 An é um objeto do tipo **TAnalysis** construído dinamicamente tendo como base a malha computacional c.