Luís Fernando Nacif Rocha

Uma Base de Dados em Formato Neutro e uma Estrutura de Dados Orientada por Objetos para o Pré-processador de um Sistema de Cálculo de Campos Eletromagnéticos Baseado no Método de Elementos Finitos.

Dissertação submetida à banca examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da UFMG como parte dos requisitos necessários à obtenção do grau de Mestre em Engenharia Elétrica (M. E. E.)

Centro de Pesquisa e Desenvolvimento em Engenharia Elétrica

Escola de Engenharia

UFMG

Belo Horizonte - Setembro de 1995

A meus pais, Dilson e Raquel, pelo incentivo e, principalmente, pelo amor com que me cercaram toda a vida.

A Jesus Cristo, a quem eu conheci durante o Mestrado, e que me mostrou a verdadeira Vida.

“Buscar-me-eis, e me achareis, quando me buscardes de todo o vosso coração.”

Jeremias 29:13

Agradecimentos

Ao Prof. Renato C. Mesquita, pela sua disposição, desde a graduação, em me ensinar e orientar, e por seu acompanhamento e amizade durante estes dois anos e meio do Mestrado;

Ao Prof. Mauro N. Rocha (Dep. Informática/UFV), pelo incentivo e pela colaboração em todas as fases do desenvolvimento deste trabalho;

Ao Prof. Elson J. Silva, pelas colaborações e pelo entusiasmo e incentivo no acompanhamento do trabalho;

Ao Prof. Ramon P. da Silva (Dep. Eng. Estruturas/UFMG), pela sua disposição e ajuda no início do trabalho;

Ao Prof. Rodney R. Saldanha, por sua amizade e incentivo durante todo o curso de Mestrado;

A todos os alunos do curso, por terem confiado em mim enquanto representante discente, por seu apoio e sua amizade;

Aos demais professores e funcionários do CPDEE, cujo convívio, com certeza, trouxe-me crescimento profissional e pessoal.

Resumo

Apresentamos uma restruturação da forma de armazenamento dos dados para o pré-processador de um sistema computacional de cálculo de campos eletromagnéticos baseado no Método de Elementos Finitos.

A estrutura do sistema é dividida em uma parte externa, representada por uma base de dados para o armazenamento de todas as informações manipuladas pelo programa, e uma parte interna, representada pela organização da estrutura de dados utilizada pelo programa.

Na primeira parte, é apresentada a proposta de uma Base de Dados em Formato Neutro para a intercomunicação entre estes programas, buscando uma padronização para os formatos de armazenamento e transferência de dados dos diversos programas desenvolvidos pelos grupos de pesquisa de cálculo de campos eletromagnéticos brasileiros.

Na segunda parte é apresentada uma estrutura de dados para o pré-processador de tais programas, utilizando técnicas de Programação Orientada por Objetos, visando a melhora da organização da estrutura.

Procuramos, usando a Programação Orientada por Objetos, aumentar a qualidade do software de engenharia, dando ao código características desejáveis para uma maior taxa de reaproveitamento frente a mudanças na especificação.

Abstract

A data storage restructuring for an electromagnetic field computation program preprocessor based on the Finite Element Method is presented.

The system structure is divided in an external part, represented by a Data Base for the storage of all data used in the program, and an internal part, represented by the data structure organization of the preprocessor program.

In the first part, a Neutral Format Data Base for the communication between such programs is proposed, searching for a standardization of data transfering formats of the several programs developed by the Brasilian electromagnetic fields research groups.

In the second part, a data structure for the preprocessor of these programs is presented, using Object-Oriented Programming techniques, searching for a better structure organization.

Using these Object-Oriented techniques we try to improve the engineering software quality, giving code the desirable characteristics to a higher reuse when faced with any specification change.

Sumário

1) **Introdução**

1.1 - Histórico, Motivação e Divisão do Trabalho ________________________ 01
1.2 - Sistema Computacional Baseado em Elementos Finitos________________ 06

2) **Base de Dados em Formato Neutro**

2.1 - Introdução____ 09
2.2 - Conteúdo da Base de Dados______________________________________ 11
2.3 - Opções Adotadas ____ 13
2.4 - Implementação da Estrutura______________________________________ 22
2.5 - Um Exemplo da Base de Dados____________________________________ 41
2.6 - Vantagens e Desvantagens da Base de Dados em Formato Neutro______ 45

3) **Estrutura de Dados Orientada por Objetos**

3.1 - Introdução____ 47
3.2 - Programação Orientada por Objetos - Um Breve Resumo______________ 48
3.3 - Estrutura de Dados Primária______________________________________ 61
3.4 - Estrutura de Dados Orientada por Objetos__________________________ 67
3.5 - Vantagens e Desvantagens da Estrutura Orientada por Objetos ________ 92

4) **Conclusão**____ 94

Referências Bibliográficas____ 97

Apêndices

A) Geometrias das Fontes de Corrente_________________________________ 101
B) Estrutura Completa da Base de Dados em Formato Neutro______________ 103
C) Código da Estrutura de Dados Orientada por Objetos (em C++)_________ 123

Capítulo 1 - Introdução

1.1 - Histórico, Motivação e Divisão do Trabalho

O método de elementos finitos tem sido utilizado para o projeto de dispositivos e equipamentos eletromagnéticos desde o início dos anos 70. Para determinadas geometrias, pode-se utilizar o cálculo em duas dimensões, como, por exemplo, no caso de geometrias com simetria axial, ou fazendo uma aproximação da realidade tridimensional. Existem problemas, porém, onde somente o cálculo em três dimensões traz resultados confiáveis (Mesquita, 1990).

Os programas para o cálculo de campos eletromagnéticos são, normalmente, extremamente grandes e complexos. Para se ter uma idéia, o programa desenvolvido pelo Grupo de Otimização e Projeto Assistido por Computador (GOPAC) da UFMG tem, hoje, cerca de cem mil linhas de código. O mesmo pode efetuar o cálculo tridimensional de problemas eletrostáticos, magnetostáticos, correntes de Foucault, ensaios não destrutivos e aquecimento indutivo, envolvendo diversas formulações matemáticas e diversos tipos de acoplamento entre estas formulações. Este programa está dividido em três etapas: pré-processamento, processamento e pós-processamento e veremos sua estrutura com maiores detalhes nos capítulos seguintes.

Apesar de terem sido adotadas metodologias estruturadas para o projeto do sistema, a sua manutenção é bastante difícil. É extremamente complexo o gerenciamento e a coordenação do desenvolvimento de novas extensões para o programa, o que praticamente inviabiliza a existência de uma equipe coordenada de desenvolvimento de software. O que acaba acontecendo é que começam a existir várias versões do programa, cada uma desenvolvida para atender a objetivos específicos, sem que estes desenvolvimentos se incorporem ao produto final do software, devido à necessidade de revisão de várias estruturas do sistema (estruturas de dados, de arquivos, interfaces de subrotinas etc.) que acaba afetando o trabalho da equipe de desenvolvimento.

Outra limitação importante do programa é a rigidez da estrutura de dados do seu pré-processador, que fornece pouca flexibilidade à criação de estruturas geométricas novas e a aplicação de novas técnicas de geração de malha.

Estes fatores sugerem a pesquisa de novas metodologias para o desenvolvimento de software, merecendo destaque a Programação Orientada por Objetos (Stroustrup, 1988), que pode ser utilizada com consideráveis vantagens no desenvolvimento de programas de elementos finitos quando comparada com o enfoque da programação por procedimentos, devido às características modulares do método (Forde et al., 1990; Silva et al., 1994). Os principais objetivos desta nova técnica são alcançar a portabilidade, a reusabilidade, a confiabilidade e a extensibilidade de programas complexos (Meyer, 1988). Estes objetivos são alcançados através da introdução de novos conceitos como a abstração e o encapsulamento de dados, a herança e o polimorfismo (Booch, 1994). Maiores detalhes sobre estes dados serão apresentados no Capítulo 3.

O Grupo de Otimização e Projeto Assistido por Computador (GOPAC) da UFMG realizou recentemente a implementação de um programa processador tridimensional com metodologia orientada por objetos, obtendo excelentes resultados (Silva et al., 1994), sendo, aparentemente, a primeira aplicação desta metodologia no cálculo de campos eletromagnéticos. Os bons resultados obtidos serviram de motivação para a expansão desta experiência, incluindo novas formulações matemáticas e novos tipos de elementos finitos no programa processador, bem como estender a aplicação da metodologia aos pré e pós-processadores.

No programa pré-processador, com a aplicação das técnicas da Programação Orientada por Objetos, deseja-se aumentar a organização da estrutura de dados, permitindo ao programador realizar a manutenção da mesma com maior facilidade, bem como efetuar a criação de novos tipos de estruturas sem invalidar as já existentes.

As primeiras experiências com a aplicação desta metodologia na análise de equações diferenciais parciais por elementos finitos ocorreram no início desta década (Forde et al., 1990; Zimmermann et al., 1992; Pèlerin et al., 1992). Nos trabalhos de Zimmermann e Pèlerin utilizou-se a linguagem Smalltalk, que não é uma linguagem eficiente do ponto de vista numérico.

Já no trabalho de Silva, foi desenvolvido um programa de cálculo magnetostático tridimensional utilizando elementos de aresta e o potencial vetor magnético. Como os códigos de elementos finitos demandam uma linguagem que trabalhe eficientemente do ponto de vista numérico, a linguagem C++ foi escolhida, por combinar as vantagens de Programação Orientada por Objetos com a eficiência computacional do C (Ellis & Stroustrup, 1993).

Outros trabalhos aplicados a partes de um sistema de cálculo de campos têm sido desenvolvidos, como o do grupo da Universidade da Akron (EUA), que implantou um sistema de classes e métodos para o cálculo matricial (Sharafuddin et al., 1993). Neste caso, também foi utilizada a linguagem C++.

Outro ponto a ser considerado é a necessidade de definição de uma base de dados em formato neutro, generalizada, para a intercomunicação de programas de cálculo de campos eletromagnéticos, que possa ser utilizada pelos diversos grupos de pesquisa no Brasil. As vantagens que esta definição proporciona são grandes. A partir dela, cada grupo poderia trabalhar em partes específicas do cálculo e compartilhar sua experiência com os demais grupos, independentemente da implementação do programa que estivessem utilizando. Aumentando a cooperação entre os grupos, a comunidade de pesquisa, como um todo, ganharia sensivelmente em troca de informações, reduzindo o isolamento dos diversos grupos.

Esta não é uma tarefa simples, pois exige um trabalho integrado dos diversos grupos de pesquisa, cada um fornecendo sua contribuição, para que a base seja a mais genérica possível. Nosso objetivo é fornecer uma base inicial, com uma estrutura que não chega a abranger todas as possibilidades. A participação de outros grupos que não sejam específicos do estudo de Campos Eletromagnéticos, como Engenharia de Estruturas, Fluidos e Térmica, também é interessante, contribuindo ainda mais para sua generalização.

Uma boa descrição dos requisitos fundamentais a serem atendidos para a definição de bases de dados para programas de elementos finitos é feita por George (1991). Trabalhos específicos para a área eletromagnética foram desenvolvidos por Webb et al. (1985), Santana et al. (1988) e Ancelle et al. (1988). Deve-se lembrar que os primeiros esforços de normalização estão em processo, a nível internacional. A International Standards

Organization (ISO) está discutindo uma proposta de norma para bancos de dados de engenharia, incluindo dados geométricos e de elementos finitos. Esta proposta foi denominada STEP (Specification for the Transfer and Exchange of Product data). Uma linguagem de modelamento de dados para atender à STEP, denominada *Express* (ISO, 1991), foi especificada, porém, não existe nenhuma proposta de normalização para os problemas eletromagnéticos, sendo que várias características não são atendidas pela STEP (Sadi et al., 1994).

Buscamos, através deste trabalho, montar uma estrutura de armazenamento de dados para um programa de cálculo de campos eletromagnéticos, dividindo-o, basicamente, em duas partes:

- a elaboração de uma proposta para a estrutura de uma Base de Dados em Formato Neutro para a intercomunicação de dados em, e entre, programas de cálculo de campos eletromagnéticos baseados no Método de Elementos Finitos;

- a elaboração de uma estrutura de dados interna, baseada em técnicas de Programação Orientada por Objetos, para o pré-processador de um programa de cálculo de campos eletromagnéticos.

Com isto, procuramos definir a estrutura de dados que é utilizada no pré-processamento dos dados, tanto interna, quanto externamente ao programa.

Com a Base de Dados em Formato Neutro, buscamos uma padronização de um modo de armazenamento dos dados de programas de cálculo de campos eletromagnéticos, visando a intercomunicação entre programas e ambientes diferentes, procurando diminuir os esforços de uma cooperação entre grupos. Apresentamos o conteúdo necessário para que tal base fosse projetada, mostrando quais as opções adotadas a nível de projeto, para que todas as necessaidades fossem atendidas. Em seguida, apresentamos a implementação da estrutura da base, com todos os detalhes, mostrando um exemplo de sua utilização. Procuramos, também, ressaltar suas vantagens e desvantagens.

Com a estrutura de dados, visamos elaborar um pré-processador orientado por objetos, primeiro passo para um programa de cálculo de campos eletromagnéticos orientado por objetos. Apresentamos um breve resumo da teoria da Programação Orientada por Objetos, visando familiarizar o leitor com seus conceitos.

A estrutura de dados tem como origem o programa anterior, desenvolvido pelo Grupo de Otimização e Projeto Assistido por Computador (GOPAC), com a filosofia de orientação por procedimentos. Esta estrutura antiga é apresentada e, em seguida, mostramos a estrutura de dados orientada por objetos, evidenciando as vantagens e desvantagens de seu uso.

Como um bom programa de cálculo de campos eletromagnéticos deve possuir flexibilidade suficiente para implementar novos tipos de geometrias, materiais, fontes de excitação, condições de contorno etc., nossa preocupação foi a de conceder às duas estruturas um caráter flexível, onde a generalização dos dados não prejudicasse a organização das mesmas.

1.2 - Sistema Computacional Baseado em Elementos Finitos

O objeto deste trabalho é um sistema computacional de cálculo baseado no Método de Elementos Finitos. Tal sistema é, geralmente, dividido em três blocos funcionais distintos, descritos a seguir:

a) Pré-processador

Módulo de entrada e preparação dos dados, onde toda a informação necessária para a criação de um problema é fornecida. Pode-se atribuir três funções ao pré-processador:

- definição da geometria do problema;
- definição de regiões e contornos;
- geração da malha de elementos finitos.

b) Processador

A função do processador é efetuar a montagem e resolução do sistema de equações lineares ou não-lineares resultante da formulação do problema. Este módulo tem como entrada de dados a discretização do domínio em elementos finitos, as suas propriedades físicas e as condições de contorno.

c) Pós-processador

Terminada a resolução do sistema, as variáveis de estado são computadas em cada nó da malha, mas nem sempre são diretamente utilizáveis porque:

- o número de valores numéricos é grande (geralmente milhares);
- as variáveis de estado nem sempre têm um significado físico claro, por exemplo, o vetor potencial magnético A).

O pós-processador permite a visualização de variáveis escalares e vetoriais em secções, partes da geometria, ou ao longo de linhas. Permite, também, o cálculo de variáveis globais, como força e temperatura em regiões definidas.

A Fig. 1.1 ilustra a disposição destes três blocos:

Fig. 1.1 - Divisão de um sistema de cálculo usando o M.E.F.

O esforço deste trabalho está concentrado na etapa do pré-processamento dos dados.

O pré-processador pode ser subdividido em duas partes: uma que trata do desenho geométrico, onde é estabelecida a geometria do problema e são feitas as atribuições físicas (materiais, fontes de excitação, condições de contorno), e outra que é o gerador de malha que, a partir dos dados recolhidos na parte anterior, monta a malha de elementos finitos do problema. A Fig. 1.2 ilustra esta subdivisão:

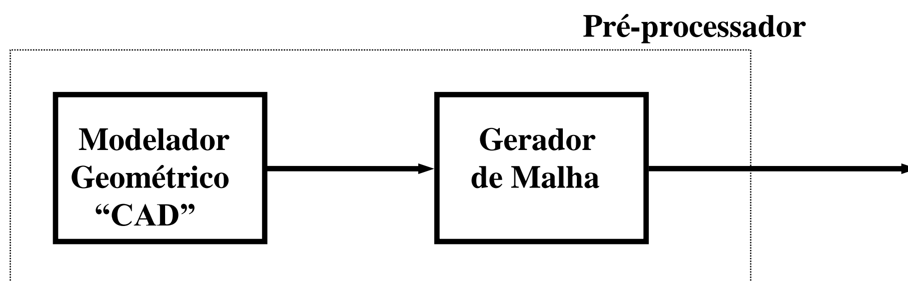


Fig. 1.2 - Subdivisão interna de um pré-processador genérico.

As divisões e subdivisões aqui apresentadas não são rígidas. Um sistema de cálculo baseado no Método de Elementos Finitos, tanto pode ser composto de 3 ou 4 módulos de programas, quanto por apenas um programa que executa todas as etapas.

Uma outra prática comum é dividir o sistema em dois programas: um contendo o processamento do cálculo e outro unindo pré e pós-processamento, devido ao fato de ambos trabalharem com a visualização gráfica da geometria.

A transferência de dados entre os módulos é feita via arquivo, onde cada um armazena os dados que processa em disco, para que ele mesmo ou qualquer outro módulo possa recuperá-los.

Uma base de dados em formato neutro pode ser utilizada para efetuar esta transferência de dados. Como vemos na Fig. 1.3, a base faria a ligação entre as partes do programa e, por possuir um formato neutro, permitiria a intercomunicação entre programas diferentes. Isto significa que diferentes programas poderiam ser utilizados para efetuar o processamento, por exemplo, desde que estes programas possuíssem um módulo que efetuasse a leitura da base de dados no formato neutro e gravasse os seus resultados em formato também compatível com a mesma.

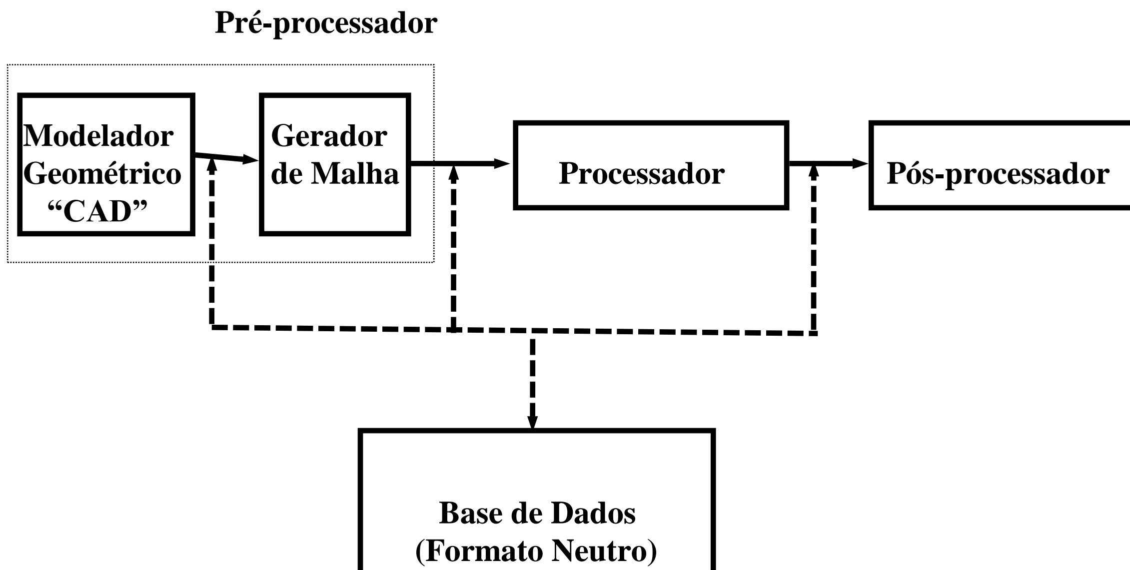


Fig. 1.3 - detalhamento de um sistema de cálculo usando o M.E.F.

Capítulo 2 - Base de Dados em Formato Neutro

2.1 - Introdução

Como pudemos verificar na Fig. 1.3, a Base de Dados em Formato Neutro tem como função interligar as várias partes de um programa de cálculo baseado em elementos finitos, não ficando seu raio de ação restrito ao Método de Elementos Finitos, mas também ao Método de Diferenças Finitas e o Método de Elementos de Contorno. Ele pode ser usado, tanto para a comunicação entre módulos (por exemplo, entre o processador e o pós-processador), quanto para a comunicação de dados dentro de um mesmo módulo (como vemos na Fig. 1.3, entre o modelador geométrico e o gerador de malha).

A Base de Dados serve como uma base de intercomunicação de dados dentro de um mesmo programa ou entre programas diferentes, trabalhando também como uma base de armazenamento permanente dos dados. Assim, funciona apenas externamente ao programa, podendo este assumir a estrutura interna que desejar.

Esta base pode ser considerada uma memória secundária do programa e, por isso, um considerável cuidado deve ser tomado para que sua estrutura seja organizada de modo a dar rapidez e eficiência à transferência dos dados, não prejudicando o desempenho do programa que a utiliza.

Se a preocupação com a velocidade da transferência dos dados for excessiva, a base pode sofrer uma perda na clareza de sua estruturação, tornando-a, muitas vezes, compacta e de difícil entendimento. Por outro lado, se a preocupação com a organização da base for excessiva, pode-se perder em velocidade de acesso aos dados (Webb,1985).

Para um perfeito funcionamento da Base de Dados em Formato Neutro, esta deve ser completa quanto aos dados necessários ao processamento dos cálculos. Qualquer dado que não conste no arquivo deve ter condições de ser derivado de outros dados existentes, ou o programa funcionará com limitações.

O formato desta base deve ser tal que facilite ao usuário programar uma rotina para lê-la, independentemente da linguagem de programação utilizada. Com isto, garante-se uma independência do arquivo em relação à implementação do programa que irá utilizá-la.

Em contrapartida, a base deve possuir uma estrutura de fácil entendimento, quando de uma inspeção feita pelo usuário, ou seja, analisando uma listagem da base, o usuário deve ser capaz de identificar os dados nela contidos. Para tal, deve-se atentar para a organização e identificação de todos os campos de dados nela existentes.

Uma característica de suma importância no projeto de uma Base de Dados em Formato Neutro é a previsão da modificação e/ou extensão dos dados. Deve haver uma generalidade tal na base que a mesma suporte modificações ou extensões em sua estrutura de dados sem invalidar os dados já existentes (Lowther,1988).

Podemos citar um exemplo onde temos uma base de dados projetada para suportar apenas problemas magnéticos estáticos envolvendo materiais não-lineares via Método de Elementos Finitos; então será difícil expandir o software para incluir outros tipos de materiais, problemas, condições de contorno ou mesmo elementos de ordem superior.

Uma dificuldade de expansão da base de dados devido a um erro de projeto da mesma pode causar um ônus ao usuário final, se o mesmo tiver que alterar seus dados já existentes para atender a uma nova versão da base.

Devemos ressaltar que esta base não possui necessariamente o formato de armazenamento normal do programa, ou seja, é possível que haja, junto com este formato, um formato de gravação binário, por exemplo, particular a cada programa, permanecendo esta base apenas para transferências de dados. Os arquivos binários, por terem uma maior velocidade de acesso, são melhores para serem usados como principal meio de armazenamento.

Uma base de dados constituída de vários arquivos provê um grau maior de flexibilidade, permitindo uma futura expansão da base com prejuízos minimizados. Esta forma distribuída, também chamada modularização, permite que alterações em uma parte da base afetem apenas o arquivo relativo aos dados modificados. Por exemplo, uma mudança na definição de uma propriedade de um material novo apenas modifica o arquivo referente aos dados de materiais, deixando intacto o resto da base.

O maior problema que uma organização em vários arquivos produz é a necessidade de cuidado na sua manipulação e armazenamento, podendo a perda de um dos arquivos acarretar a falência de toda a base.

2.2 - Conteúdo da Base de Dados

Considerando uma base de dados para a utilização em um programa de cálculo baseado no Método de Elementos Finitos, podemos elaborar o conteúdo da base fazendo a seguinte divisão de dados:

- Definições Geométricas - são os dados que dizem respeito exclusivamente à geometria do problema, como as dimensões e divisões dos objetos ou das figuras, definidas pelo usuário através de um programa de modelamento geométrico (CAD). Qualquer outra informação que não constitui a geometria do problema deve estar contida em, ou formar, uma outra parte;

- Condições de Contorno - são as informações relacionadas às condições de contorno impostas ao problema. Estas informações têm relação direta com os dados da geometria, pois a definição das regiões onde são aplicadas tais condições têm suporte nos dados geométricos;

- Excitação - temos aqui quaisquer fontes de excitação que porventura venham a existir, como fontes de corrente, tensão, cargas, etc. Juntamente com as condições de contorno e materiais, formam o conjunto de condições físicas impostas ao problema;

- Materiais - Aqui são associados a cada região do problema algum material que a preenche. As propriedades físicas destes materiais são definidas separadamente;

- Detalhes de Cálculo - dados relativos ao processamento do cálculo, baseado no Método de Elementos Finitos, como ordem de integração e tipo de potencial imposto para o cálculo em um elemento;

- Propriedades dos Materiais - são os dados das propriedades físicas dos materiais utilizados. Como, geralmente, os materiais utilizados em um problema podem ser utilizados em um outro problema, é comum que se faça um banco de dados de materiais, independente do problema em questão, que pode ser compartilhado por todos;

- Dados da Malha - são os dados provenientes do gerador da malha, que processa os dados geométricos e físicos existentes, formando a malha de elementos finitos, para o módulo processador;

- Resultados do Processamento - são os dados provenientes do módulo processador. Estes dados passam por um pós-processamento, para o cálculo de outras grandezas, e para a apresentação ao usuário, geralmente na forma de desenhos, gráficos e tabelas;

Para que tenhamos uma base de dados completa, é necessário que sejam alocados todos os dados relacionados acima. A seguir, veremos como foi feita a distribuição dos dados na base.

2.3 - Opções Adotadas

Baseados nas características desejadas em uma Base de Dados em Formato Neutro, e fazendo uma análise dos dados relacionados no item anterior, buscamos organizar uma estrutura que melhor atendesse às necessidades de um programa voltado para o cálculo de campos eletromagnéticos. Estruturas de bases já desenvolvidas por grupos de pesquisa brasileiros, especialmente na área de estruturas (Grupo de Eng. de Estruturas/UFMG, 1992; Computer Graphics Technology Group - TecGraf, 1992), e por grupos de pesquisa estrangeiros (George, 1991; Santana et al, 1988; Webb et al, 1985) serviram como ponto de partida para esta.

Divisão dos arquivos

Analisando o volume e as características dos dados que compõem a base, procuramos separá-los em mais de um arquivo, buscando organizar a base, dando-a maior versatilidade.

Inicialmente, a base foi subdividida em três arquivos: Geometria, Malha e Materiais (Rocha & Mesquita,1994). Em uma etapa posterior, quando a base começou a ser utilizada pelo Grupo de Otimização e Projeto Assistido por Computador, a necessidade de implementação de uma malha superficial fez com que se cogitasse a separação do arquivo Malha em duas partes, uma com a malha volumétrica e outra com a malha superficial, e também transformar o arquivo Materiais em um arquivo que contivesse todas as definições dos atributos físicos do(s) problema(s).

Esta divisão, com o uso, mostrou-se inadequada, pois, em primeiro lugar, a separação de Malha em dois arquivos sacrificou a característica genérica da Base de Dados e, em segundo lugar, haviam certos atributos físicos que eram específicos do problema, enquanto os materiais formavam um grupo de dados independente.

Com isso, retornou-se à estrutura original, onde o arquivo Malha foi acrescido dos dados da malha superficial, tornando-se o arquivo Malhas, devido à sua generalidadee, como veremos adiante.

A seguir, apresentaremos os dados que compõem cada um dos arquivos da base, enquanto que em uma seção posterior veremos os detalhes de toda a estrutura.

Geometria

Este arquivo contém os dados da descrição geométrica do problema estudado. Para a representação desta geometria, podemos considerar duas maneiras principais de se descrever uma geometria qualquer (Kroszynski et al, 1989): a Brep (Boundary Representation, ou Representação por Contorno), e a CSG (Constructive Solid Geometry, ou Geometria Sólida Construtiva).

Na CSG, objetos são definidos através de operações booleanas (união, subtração ou interseção) com figuras primárias, como blocos, cilindros, cones, esferas etc. A estrutura de dados é freqüentemente uma árvore binária, onde os nós representam combinações de outros nós ou folhas, que são as estruturas primárias (Fig. 2.1).

Fig. 2.1 - Composição de geometria na representação CSG

As vantagens desta forma de representação são:

- quando as estruturas primárias são bem adaptadas aos tipos de objetos a serem modelados, os dados resultantes são particularmente concisos;
- aproxima bastante do processo intelectual de formação da geometria;
- tem uma melhor adaptação a eventuais mudanças na geometria.

As desvantagens são:

- há a necessidade de algoritmos freqüentemente complexos para uma representação geométrica dos objetos, ou a conservação em paralelo de uma segunda base de dados, mais simples de ser desenhada;
- para a geração da malha de elementos finitos é também necessária a passagem por uma segunda estrutura de dados, como a B-rep.

Na Brep, os objetos são representados dividindo-se sua superfície de contorno em um número finito de superfícies. Cada uma das superfícies é dividida em uma lista de linhas (retas ou curvas) e cada linha formada de pontos (Fig. 2.2).

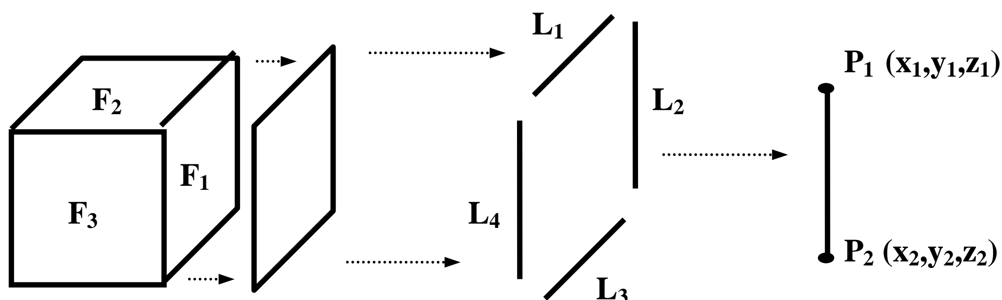


Fig. 2.2 - Composição de geometria na representação Brep

As vantagens desta representação são:

- maior simplicidade na representação gráfica;
- a representação é usada diretamente na geração da malha de elemetos finitos;
- separa a informação métrica (coordenadas de pontos) da informação topológica (arestas, contornos, faces etc.).

E as desvantagens são:

- necessidade de armazenamento de grande quantidade de informação;
- a entrada de dados é bem menos interativa com o usuário;
- o armazenamento de faces curvas usando esta estrutura não é trivial, devendo ser, normalmente, feita usando superfícies paramétricas (Mortenson, 1985; Foley et al., 1990).

Estas duas representações geométricas, por possuírem características diferentes, atuando de forma diferente na geometria, podem ser usadas em conjunto, de modo a se tornarem complementares. Uma maneira de fazer isso é utilizar a representação CSG para a construção da geometria, por sua facilidade de representação dos objetos de um modo mais real, automaticamente transformando-a em Brep para a geração da malha de elementos finitos, o que é bem mais fácil de ser feito.

Escolhemos como modo de representação da nossa Base a estrutura Brep, devido à sua facilidade de uso, tanto a nível de operação, quanto para a geração da malha de elementos finitos, que julgamos ser de vital importância.

A construção dos objetos da geometria começa com a definição dos pontos que formam as arestas da mesma, através de suas coordenadas. Definidos os pontos, inicia-se um processo de montagens de listas de dados. Inicialmente, uma linha é definida como uma lista de pontos (considerando a existência de poli-linhas). Em seguida, o contorno é formado por uma lista de linhas e, por sua vez, a face é uma lista de contornos. Através

deste procedimento, também temos o envelope (listas de faces), o volume (lista de envelopes) e o objeto (lista de volumes), de acordo com a seguinte figura (Santana,1988; Wilson, 1985):

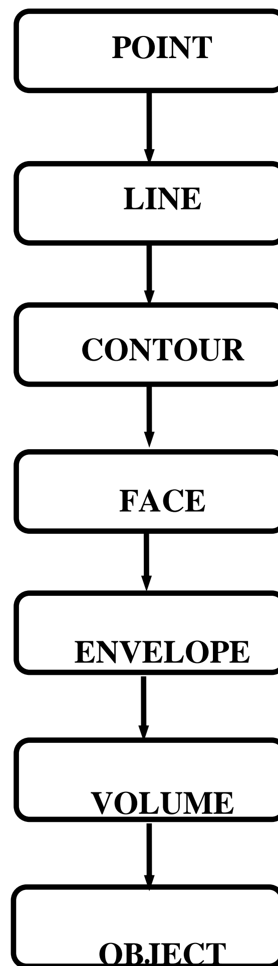


Fig. 2.3 - Hierarquia da estrutura de entidades geométricas

Há alguns pontos a ressaltar nesta estrutura:

- o fato de uma face ser composta por uma lista de contornos permite que sejam criadas faces com falhas, ou "buracos", em seu interior. O primeiro contorno da lista define o contorno externo, enquanto os outros definem as falhas (Fig. 2.4);

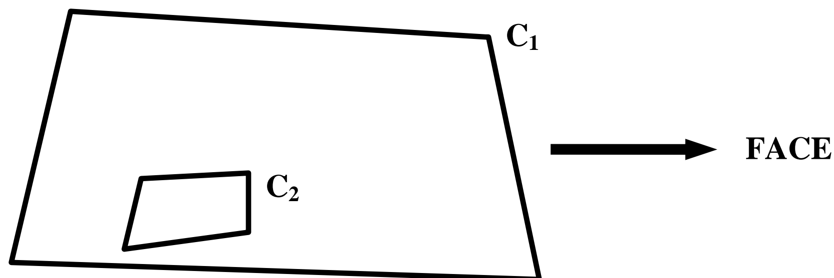


Fig. 2.4 - Face composta por dois contornos

- o mesmo ocorre com envelopes e volumes, onde um volume pode ter envelopes internos, representando "bolhas", sendo o primeiro envelope o mais externo;

- a estrutura de objetos funciona como agrupador de volumes, que formam ou representam um único sólido, por exemplo, o estator e o rotor de uma máquina elétrica. O objetivo desta composição é dar uma maior organização aos dados, ressaltando o sentido físico dos volumes;

- um label, opcional, composto de uma cadeia de 20 caracteres, pode ser associado a cada um dos componentes da geometria, de ponto a objeto, tanto para facilitar a identificação de quaisquer de suas partes, quanto para servir como referência na atribuição de características físicas;

- a estrutura na forma apresentada não inclui o modelamento de superfícies curvas, porém, a sua extensão para a inclusão destas superfícies já está prevista pela possibilidade de criação de subtipos derivados do tipo FACE.

O formato Macro_Struct foi criado para representar formas geométricas pré-definidas, como cilindro, paralelepípedos, planos, cascas, arcos etc. O tipo da estrutura é identificado por um label, e o número de parâmetros que há para cada tipo depende de sua topologia.

A atribuição de características físicas é feita associando-se o label dado a qualquer entidade a identificadores que as referenciam aos outros dados da base. Estas características podem ser, tanto o tipo de material, potencial para cálculo, condição de contorno ou valores de corrente e tensão impostas, quanto a cor da entidade para a sua representação geométrica. Pode-se, então, entender o label como um ponteiro para uma tabela de ponteiros para as características físicas e visuais da geometria.

Malhas

Este arquivo é composto, tanto dos dados relativos às malhas superficial e volumétrica do problema, quanto de outras especificações de características físicas particulares do problema.

A estrutura começa com a definição das coordenadas dos nós que compõem a malha, seguida da definição de arestas para o caso de uma malha de elementos de aresta. Os elementos são definidos pelos seus nós, possuindo, também, um label para posterior referência. Estes elementos podem ser lineares, superficiais e volumétricos, tanto nodais, quanto de aresta. Na relação dos elementos, eles são separados por tipo (triângulo, tetraedro, hexaedro etc.) e procurou-se abranger os tipos comumente utilizados no caso particular do cálculo de campos eletromagnéticos nesta versão da base, como veremos oportunamente.

Permite-se, ainda neste arquivo, definir uma macro-estrutura (vide arquivo Geometria) como sendo um enrolamento, associando-se a ele uma corrente ou tensão.

Para que fosse possível organizar casos diferentes sobre um mesmo problema, onde valores de grandezas como corrente e tensão pudessem ser alterados sem a necessidade de

criação de outro(s) arquivo(s), implementou-se o que aqui chamamos de dados de carregamento. Estes valores redefinem os valores já estipulados no caso inicial.

Os dados referentes ao detalhamento do cálculo do programa também são encontrados neste arquivo. Estes dados se resumem em: as ordens de integração ligadas aos elementos, os tipos de potencial aplicados aos elementos, valores de corrente, tensão e o número de espiras em enrolamentos.

As definições de condições de contorno são feitas aqui por tipo de condição, onde cada um agrupa seu conjunto de dados, sendo associado às estruturas através de seu identificador.

Como no arquivo Geometria, atributos físicos ou geométricos são relacionados com as entidades do arquivo Malhas, através de seus labels. Pode-se associar a um elemento, tanto atributos como o tipo de potencial para cálculo ou o tipo de material, quanto relacioná-lo com as regiões geométricas definidas em Geometria. Assim, pode-se saber diretamente em qual volume ou superfície um determinado elemento está inserido.

Finalizando a relação de dados do arquivo Malhas, temos os dados relativos aos resultados do processamento do cálculo. Estes dados se resumem em valores de potencial e campo nos nós e/ou arestas especificados.

Materiais

Este arquivo encerra a definição das características físicas relacionadas aos materiais utilizados pelo(s) problema(s). Procurou-se fazer uma estrutura a mais genérica possível, para que vários problemas pudessem acessar um mesmo arquivo de Materiais, tornando-o praticamente uma base de dados independente.

A organização dos dados de materiais é feita da seguinte forma: definido o número de materiais existentes e, opcionalmente, seus labels, agrupam-se os dados por propriedade física, ou seja, em vez de cada tipo de material listar suas próprias características, uma certa propriedade física indica, através de seu identificador, se ela está presente em um certo material, fornecendo o(s) valor(es) de sua grandeza.

Para que o programa saiba quais os valores das características de um certo material, o mesmo deve fazer uma varredura no arquivo, identificando os blocos de dados que fazem referência ao material desejado.

Esta organização facilita a implementação de novas características em um tipo de material, por serem elas independentes umas das outras. Assim, um programa que não está preparado para trabalhar com uma certa característica de um material, deve apenas ignorá-la no ato da leitura.

Um novo material também pode ser criado, utilizando as características já existentes, ou implementando outras, se necessário.

2.4 - Implementação da Estrutura da Base de Dados

Apresentamos, a seguir, o formato físico dos arquivos de que é composta a Base de Dados.

Os arquivos da Base são gerados em modo ASCII. Com isso, buscamos três vantagens básicas: qualquer linguagem de programação é capaz de fazer a leitura de seus dados com facilidade; é possível ao usuário examinar o arquivo com um editor de textos; e é facilitado o transporte do arquivo para diferentes plataformas computacionais.

Buscou-se eliminar qualquer tipo de alinhamento de colunas, preservando uma orientação linear dos arquivos. Isto facilita bastante, tanto a geração, quanto a leitura da Base, eliminando a preocupação com o posicionamento dos dados em uma linha do arquivo.

A Base é seccionada em blocos de dados, cada um deles começando com um rótulo (*label*) precedido por um asterisco (*), que permite ao programa identificar ou selecionar o bloco a ser lido, desprezando os blocos que não são necessários. Para uma melhor padronização, todos os rótulos são escritos na língua inglesa e com letras maiúsculas (caixa alta).

Para facilitar o entendimento do formato, foi adotada, para os tipos de variáveis, a seguinte convenção:

- [v] - variável inteira;
- <v> - variável real;
- ‘v(n)’ - string de, no máximo, **n** caracteres.

A seguir, são relacionados todos os dados existentes nesta versão da Base, fornecendo, em detalhes, cada um dos blocos, seus dados e seus rótulos de identificação.

Como foi mencionado anteriormente, a Base de Dados é composta por três arquivos: Geometria, Malhas e Materiais. Com exceção dos Dados Gerais, presentes em todos, veremos separadamente cada um deles.

Dados Gerais (todos os arquivos)

Consiste em informações que identificam a Base de Dados, como título, autor, data, versão, tipos de unidade e a dimensão do espaço em estudo (bi ou tridimensional).

*HEADER.FILE
Nome do arquivo

*HEADER.AUTHOR
Nome do autor ou do programa que gerou o arquivo.

*HEADER.DATE
Data da criação do arquivo.

*HEADER.VERSION
Versão do arquivo de formato neutro

*HEADER.TITLE
'File_title(80)'

*HEADER.ANALYSIS
'Analysis_type(20)'

*HEADER.UNITS
'length_label(5)' 'mass_label(5)' 'time_label(5)' 'angle_label(5)' 'temperature_label(5)'

*HEADER.DIM
[dimension_type]

Os labels de unidades propostos nesta versão do programa são (o primeiro de cada lista é o default):

- length (comprimento): M (metro), CM (centímetro), MM (milímetro), MC (micrômetro), INCH (polegada), FEET (pé);
- mass (massa): KG (quilograma), G (grama), LB (libra);
- time (tempo): SEC (segundo), MIN (minuto), HR (hora);
- angle (ângulo): RAD (radianos), DEG (graus), CYC (ciclos);

- temperature (temperatura): DEGC (graus Celsius), DEGF (graus Fahrenheit), DEGK (Kelvin).

Dois outros blocos de uso geral são o de comentários (REMARK), cuja informação é ignorada, servindo apenas para introduzir explicações ao longo da Base, e o de fim de arquivo (END), que determina o final físico do arquivo:

*REMARK
O label REMARK pode aparecer em qualquer ponto do arquivo (entre seções) e seu conteúdo é ignorado. É usado para documentar o arquivo.

*END
Este label determina o fim do arquivo. Toda informação que vem após o label é ignorada.

Dados da Geometria (arquivo Geometria)

Como foi descrito anteriormente, a representação geométrica é baseada em uma estrutura de listas. A definição dos pontos é feita como abaixo (rótulo POINT). Dado o número de pontos, são definidas as suas coordenadas (COORD) e seus labels (LABEL), que são opcionais.

```
*POINT
[#_of_points]

*POINT.COORD
[point_id]  <x>  <y>  <z>
...

*POINT.LABEL
[#_of_point_labels]
[point_id]  ‘point_label(20)’
...
```

De Linha (LINE) a Objeto (OBJECT) os dados são organizados em listas. Para tal, dado o número de entidades, é fornecido o conteúdo de cada lista (CONTENTS) através do número de itens, seguido dos identificadores das entidades que formam a lista.

Para o caso especial das linhas, há uma subdivisão, onde podem ser representadas por segmentos, formados por uma ou mais retas, ou por arcos. No primeiro caso, os

identificadores dos pontos representam os vértices da(s) reta(s). No segundo caso, representam os vértices do arco e um ponto qualquer no seu interior.

*LINE
[#_of_lines]

*LINE.LABEL
[#_of_line_labels]
[line_id] ‘line_label(20)’
...

*LINE.SEGMENT.CONTENTS
[#_of_segments]
[line_id] [#_of_contents] [point_1_id] [point_2_id] ... [point_#_of_contents_id]
...

*LINE.ARC.CONTENTS
[#_of_arcs]
[line_id] [point_1_id] [point_med_id] [point_2_id]
...

*CONTOUR
[#_of_contours]

*CONTOUR.LABEL
[#_of_contour_labels]
[contour_id] ‘contour_label(20)’
...

*CONTOUR.CONTENTS
[contour_id] [#_of_contents] [line_1_id] [line_2_id] ... [line_#_of_contents_id]
...

*FACE
[#_of_faces]

*FACE.LABEL
[#_of_face_labels]
[face_id] ‘face_label(20)’
...

*FACE.PLANE.CONTENTS
[face_id] [#_of_contents] [contour_1_id] [contour_2_id] ... [contour_#_of_contents_id]
...

*ENVELOPE
[#_of_envelopes]

*ENVELOPE.LABEL
[#_of_envelope_labels]
[envelope_id] 'envelope_label(20)'
...

*ENVELOPE.CONTENTS
[envelope_id] [#_of_contents] [face_1_id] [face_2_id] ... [face_#_of_contents_id]
...

*VOLUME
[#_of_volumes]

*VOLUME.LABEL
[#_of_volume_labels]
[volume_id] 'volume_label(20)'
...

*VOLUME.CONTENTS
[volume_id] [#_of_contents] [envel_1_id] [envel_2_id] ... [envel_#_of_contents_id]
...

*OBJECT
[#_of_objects]

*OBJECT.LABEL
[#_of_object_labels]
[object_id] 'object_label(20)'
...

*OBJECT.CONTENTS
[object_id] [#_of_contents] [vol_1_id] [volume_2_id] ... [volume_#_of_contents_id]
...

A macro-estrutura (MACRO_STRUCT) não é uma lista de entidades, mas uma figura geométrica representada por um conjunto de parâmetros que varia em número, dependendo do tipo da geometria. Para a identificação desta macro-estrutura, há um label que define qual o seu tipo (TYPE), e, a partir do mesmo, sabe-se quais os parâmetros que serão lidos (PARAMETERS).

*MACRO_STRUCT
[#_of_macro_structs]

*MACRO_STRUCT.LABEL
[#_of_macro_struct_labels]
[macro_struct_id] 'macro_struct_label(20)'
...

*MACRO_STRUCT.TYPE
[macro_struct_id] 'macro_struct_type(20)'
...

*MACRO_STRUCT.PARAMETERS
[macro_struct_id] [#_of_params] <param_1> ... <param_#>
...

Dados dos Atributos Físicos (arquivo Geometria)

Estes dados relacionam qualquer uma das entidades da geometria, identificados através de seus labels, após o rótulo ATTRIBUTES, a propriedades físicas, como o tipo de seu material (MATERIAL), uma condição de contorno imposta (BOUNDARY), um tipo de potencial para o cálculo (POTENTIAL), um valor imposto de corrente (CURRENT), um valor imposto de tensão (VOLTAGE), ou um certo número de espiras para o caso de uma macro-estrutura representar um enrolamento (WINDING). A informação da cor da entidade, para a representação gráfica da geometria, também é fornecida como um atributo (COLOR).

*ATTRIBUTES
'label_of_entity(20)'

*ATTRIBUTES.MATERIAL
[material_id]

*ATTRIBUTES.BOUNDARY
[BC_id]

*ATTRIBUTES.POTENTIAL
[potential_id]

*ATTRIBUTES.CURRENT
[current_id]

*ATTRIBUTES.VOLTAGE
[voltage_id]

*ATTRIBUTES.WINDING
[winding_id]

*ATTRIBUTES.COLOR
[color_id]

Dados dos Nós (arquivo Malhas)

De modo semelhante aos pontos da estrutura de geometria, os nós da malha de elementos finitos são definidos através de suas coordenadas no espaço.

*NODE
[#_of_nodes]

*NODE.LABEL
[#_of_node_labels]
[node_id] ‘node_label(20)’
...

*NODE.COORD
[node_id] <x> <y> <z>
...

Dados das Arestas (arquivo Malhas)

A definição das arestas é feita relacionando-se os nós que formam cada aresta, através de seus identificadores. Nesta versão da Base de dados, está programada a existência de dois tipos de aresta: com dois e com três nós.

*EDGE
[#_of_edges]

```
*EDGE.LABEL
[#_of_edges_labels]
[edge_id]  'edge_label(20)'
...

*EDGE.NODE2
[#_of_edge2]
[edge_id]  [node_1_id]  [node_2_id]
...


*EDGE.NODE3
[#_of_edge3]
[edge_id]  [node_1_id]  [node_2_id]  [node_3_id]
...
```

Dados dos Elementos (arquivo Malhas)

Os elementos também são definidos através de uma lista dos identificadores dos nós que os formam. Entre os elementos nodais há os lineares (LINE), os superficiais (TRIANGLE e QUADRANGLE) e os volumétricos (BRICK e TETRAHEDRON). O número de nós na definição dos elementos pode variar, e isso é indicado pelo número existente após sua identificação. Por exemplo, *ELEMENT.TETRAHEDRON4 indica um tetraedro de 4 nós, enquanto *ELEMENT.TETRAHEDRON10 indica um tetraedro de 10 nós. Entre os elementos de aresta, identificados pelo rótulo com a palavra EDGE, há os superficiais (TRIANGLE e QUADRANGLE) e os volumétricos (BRICK e TETRAHEDRON).

Cada elemento traz um label, que permite que se faça uma associação do mesmo aos dados pertinentes ao cálculo. Por ser uma informação obrigatória, esta informação vem logo em seguida à lista de nós, pois para todo elemento há atributos físicos a serem associados.

Abaixo, vemos cada uma dessas definições. O número total dos elementos pode ser fornecido, mas é uma informação opcional, já que cada definição de tipo de elemento traz seu próprio número de elementos.

```
*ELEMENT
[#_of_elements]
```

*ELEMENT.LINE2
[#_of_line2_elements]
[element_id] [node_1_id] [node_2_id] 'element_label(20)'
...

*ELEMENT.LINE3
[#_of_line3_elements]
[element_id] [node_1_id] [node_2_id] [node_3_id] 'element_label(20)'
...

*ELEMENT.TRIANGLE3
[#_of_triangle3_elements]
[element_id] [node_1_id] [node_2_id] [node_3_id] 'element_label(20)'
...

*ELEMENT.TRIANGLE6
[#_of_triangle6_elements]
[element_id] [node_1_id] [node_2_id] ... [node_6_id] 'element_label(20)'
...

*ELEMENT.QUADRANGLE4
[#_of_quadrangle4_elements]
[element_id] [node_1_id] [node_2_id] [node_3_id] [node_4_id] 'element_label(20)'
...

*ELEMENT.QUADRANGLE8
[#_of_quadrangle8_elements]
[element_id] [node_1_id] [node_2_id] ... [node_8_id] 'element_label(20)'
...

*ELEMENT.BRICK8
[#_of_brick8_elements]
[element_id] [node_1_id] [node_2_id] ... [node_7_id] [node_8_id] 'element_label(20)'
...

*ELEMENT.BRICK20
[#_of_brick20_elements]
[element_id] [node_1_id] [node_2_id] ... [node_19_id] [node_20_id] 'element_label(20)'
...

*ELEMENT.TETRAHEDRON4
[#_of_tetrahedron4_elements]
[element_id] [node_1_id] [node_2_id] [node_3_id] [node_4_id] 'element_label(20)'
...

*ELEMENT.TETRAHEDRON10
[#_of_tetrahedron10_elements]
[element_id] [node_1_id] ... [node_10_id] ‘element_label(20)’
...

*ELEMENT.EDGE.TRIANGLE
[#_of_triangle_edge_elements]
[element_id] [edge_1_id] [edge_2_id] [edge_3_id] ‘element_label(20)’
...

*ELEMENT.EDGE.QUADRANGLE
[#_of_quadrangle_edge_elements]
[element_id] [edge_1_id] [edge_2_id] [edge_3_id] [edge_4_id] ‘element_label(20)’
...

*ELEMENT.EDGE.TETRAHEDRON
[#_of_tetrahedron_edge_elements]
[element_id] [edge_1_id] [edge_2_id] ... [edge_6_id] ‘element_label(20)’
...

*ELEMENT.EDGE.BRICK
[#_of_brick_edge_elements]
[element_id] [edge_1_id] [edge_2_id] ... [edge_11_id] [edge_12_id] ‘element_label(20)’
...

Outros tipos de elementos podem ser acrescentados a esta lista, desde que o formato de seu rótulo e a disposição de seus dados obedeçam ao padrão aqui proposto.

Dados de Enrolamentos (arquivo Malhas)

Os enrolamentos podem ser definidos através de um conjunto de elementos finitos ou de uma topologia específica. No primeiro caso, os labels dos elementos indicarão uma referência aos atributos de corrente. No segundo caso, o tipo do enrolamento é definido por uma referência a uma macro-estrutura (arquivo Geometria) que conterá as referências necessárias a correntes, número de espiras etc.

O bloco abaixo permite que esta referência seja feita, associando-se o identificador da macro-estrutura à existência de um enrolamento, e fornecendo um label ao mesmo.

```
*WINDING.TOPOLOGY
[#_of_windings_with_topology]
[winding_id]  [macro_struct_id]  ‘winding_label(20)’
...
```

Dados de Cálculo (arquivo Malhas)

Traz a identificação de várias características e valores ligados ao cálculo do problema, como a ordem de integração, o tipo de potencial associado ao elemento e os valores de corrente, tensão e número de espiras em um enrolamento. Qualquer entidade geométrica ou elemento pode fazer referência a estes dados através de seus atributos. Algumas destas características podem ser modificadas no arquivo, se o mesmo contiver a seção de Dados de Carregamento, como veremos posteriormente.

```
*INTEGRATION.ORDER
[#_of_element_integration_orders]
[integration_id]  [order_r]  [order_s]  [order_t]
...

*POTENTIAL
[#_of_potential_types]
[potential_id]  ‘potential_label(20)’
...

*CURRENT
[#_of_current_types]
[current_id]  <J_real>  <J_imag>  <frequency>
...

*VOLTAGE
[#_of_voltage_types]
```

[voltage_id] <voltage_real> <voltage_imag> <frequency>
...

*WINDING
[#_of_winding_types]
[winding_id] [#_of_coils]
...

Para que esta Base de Dados possa ser utilizada em programas diferentes, de vários grupos de pesquisa, a padronização dos labels do tipo de potencial (POTENTIAL) é necessária. A lista de labels abaixo traz esta padronização:

‘ELECTRIC_SCALAR’, ‘MAGNETIC_SCALAR’,
‘ELECTRIC_VECTOR’, ‘MAGNETIC_VECTOR’,
‘ELECTRIC_FIELD’, ‘MAGNETIC_FIELD’,
‘ELECTRIC_INDUCTION’, ‘MAGNETIC_INDUCTION’,
‘TEMPERATURE’, ‘MAGNETIC_REDUCED’.

Condições de Contorno (arquivo Malhas)

São as informações sobre as condições de contorno impostas ao problema. As condições de contorno são separadas por tipo, onde a identificação é feita através do rótulo do bloco, que traz os seus dados específicos. Um label também pode ser associado à condição.

*BOUNDARY
[#_of_boundary_conditions]

*BOUNDARY.LABEL
[#_of_BC_labels]
[BC_id] ‘BC_label(20)’
...

*BOUNDARY.SCALAR.DIRICHLET
[#_of_Dirichlet_BC]
[BC_id] <Potential>

...

*BOUNDARY.SCALAR.NEUMANN
[#_of_Neumann_BC]
[BC_id] <Density>
...

*BOUNDARY.VECTOR.NORMAL
[#_of_Normal_BC]
[BC_id] <normal_value>
...

*BOUNDARY.SCALAR.FLOATING
[#_of_floating_BC]
[BC_id]
...

*BOUNDARY.VECTOR.TANGENT
[#_of_tangential_BC]
[BC_id] <A_1> <A_2> 'BC_vector_tangent_label(2)'
...

O label de vetor tangente define em qual plano se encontra as componentes do vetor A. As componentes desse vetor são representadas por A_1 e A_2. Os labels podem ser: 'XY', 'XZ', 'YZ', 'RZ', 'TZ', 'RT', 'RF', 'TF', onde R=Raio, T=teta e F=fi.

Ex: 1 A_x A_y 'XY'
2 A_θ A_z 'TZ'

Dados de Carregamento (arquivo Malhas)

Este bloco foi criado para que o usuário pudesse realizar cálculos sucessivos com a mesma geometria, apenas alterando valores das fontes de corrente e tensão, sem a necessidade de se criar vários problemas diferentes, cada qual para um valor desejado. Assim, pode-se fazer o que chamamos de carregamento de um caso, que consiste em redefinir valores de corrente e/ou tensão das fontes para que o processamento seja refeito, substituindo os valores originais pelos mesmos. Definido o número de casos, um label determina o início dos dados de cada caso (CASE), seguido dos dados que serão

modificados. Uma nova declaração de um label marca o início de um novo caso, encerrando-se o anterior.

*LOAD
[#_of_load_cases]

*LOAD.CASE
[current_load_case_id] ‘case_label(20)’

*LOAD.CASE.CURRENT
[#_of_currents]
[current_id] <J_real> <J_imag> <frequency>
...

*LOAD.CASE.VOLTAGE
[#_of_voltages]
[voltage_id] <voltage_real> <voltage_imag> <frequency>
...

Dados dos Atributos Físicos e de Geometria (arquivo Malhas)

Os dados apresentados abaixo relacionam os elementos, identificados pelo label, a características físicas ou geométricas do problema. A um elemento pode ser associado, tanto um tipo de material ou condição de contorno, como feito com as geometrias, quanto uma entidade geométrica (linha, face, volume etc.). Como no arquivo Geometria, após o rótulo ATTRIBUTES, seguido do label do elemento, ficam dispostas as características a serem associadas ao elemento.

Alguns desses atributos são uma repetição do arquivo de Geometria, o que é necessário, já que os elementos da malha também possuem atributos físicos como tipo de material e condição de contorno. Estes atributos são específicos de um elemento ou de um conjunto de elementos, enquanto os atributos do arquivo de Geometria se referem às geometrias como um todo.

*ATTRIBUTES
‘label_of_entity(20)’

*ATTRIBUTES.MATERIAL
[material_id]

*ATTRIBUTES.BOUNDARY
[BC_id]

*ATTRIBUTES.POTENTIAL
[potential_id]

*ATTRIBUTES.CURRENT
[current_id]

*ATTRIBUTES.VOLTAGE
[voltage_id]

*ATTRIBUTES.WINDING
[winding_id]

*ATTRIBUTES.COLOR
[color_id]

*ATTRIBUTES.INTEGRATION_ORDER
[integration_order_id]

*ATTRIBUTES.POINT
[point_id]

*ATTRIBUTES.LINE
[line_id]

*ATTRIBUTES.CONTOUR
[contour_id]

*ATTRIBUTES.FACE
[face_id]

*ATTRIBUTES.ENVELOPE
[envelope_id]

*ATTRIBUTES.VOLUME
[volume_id]

*ATTRIBUTES.OBJECT
[object_id]

*ATTRIBUTES.MACRO_STRUCT
[macro_struct_id]

Dados dos Resultados (arquivo Malhas)

Resultados do processamento (ou pós-processamento) dos dados do problema. O número de casos é, no mínimo, igual a um (caso default), acrescido do número de casos definidos em ***Dados de Carregamento***. Os dados são organizados de modo semelhante aos dados de carregamento: o identificador do caso, com seu label, precedem os resultados do caso especificado.

Os tipos de resultados são separados de acordo com o tipo de elementos, nodais (RESULTS.NODE...) ou de aresta (RESULTS.EDGE...), e podem assumir o valor em forma de potenciais (POTENTIAL) ou campos (FIELD), reais (REAL) ou complexos (COMPLEX). O número de variáveis vai depender do tipo de cálculo do elemento, e é fornecido para cada nó ou aresta.

*RESULTS
[#_of_cases]

*RESULTS.CASE
[current_case_id] 'case_label(20)'

*RESULTS.NODE.POTENTIAL.REAL
[#_of_nodes]
[node_id] [#_of_variables] <var_1> <var_2> ... <var_#>
...

*RESULTS.NODE.POTENTIAL.COMPLEX
[#_of_nodes]
[node_id] [#_of_variables] <var_1_real> <var_1_imag> ... <var_#_real> <var_#_imag>
...

*RESULTS.NODE.FIELD.REAL
[#_of_nodes] 'field_type_label(20)'
[node_id] [#_of_variables] <var_1> <var_2> ... <var_#>
...

*RESULTS.NODE.FIELD.COMPLEX
[#_of_nodes] ‘field_type_label(20)’
[node_id] [#_of_variables] <var_1_real> <var_1_imag> ... <var_#_real> <var_#_imag>
...

*RESULTS.EDGE.POTENTIAL.REAL
[#_of_edge]
[edge_id] [#_of_variables] <var_1> <var_2> ... <var_#>
...

*RESULTS.EDGE.POTENTIAL.COMPLEX
[#_of_edge]
[edge_id] [#_of_variables] <var_1_real> <var_1_imag> ... <var_#_real> <var_#_imag>
...

*RESULTS.EDGE.FIELD.REAL
[#_of_edge] ‘field_type_label(20)’
[edge_id] [#_of_variables] <var_1> <var_2> ... <var_#>
...

*RESULTS.EDGE.FIELD.COMPLEX
[#_of_edge] ‘field_type_label(20)’
[edge_id] [#_of_variables] <var_1_real> <var_1_imag> ... <var_#_real> <var_#_imag>
...

*RESULTS.NODE.TEMPERATURE
[#_of_nodes]
[node_id] <Temperature>
...

Dados dos Materiais (arquivo Materiais)

Como foi detalhado anteriormente, as características dos materiais são agrupadas por propriedade física, ou seja, em vez de haver tipos fixos de material, cada um listando suas próprias características, uma certa propriedade física indica se ela está contida em um certo material, que é identificado através de seu label, fornecendo o valor, ou valores, de seus atributos.

Para que o programa saiba quais são os valores dos atributos de um certo material, deve fazer uma varredura no arquivo, identificando os blocos de dados que fazem referência ao material desejado.

Abaixo temos as características apresentadas nesta versão da Base de Dados. Qualquer outra característica pode ser acrescentada a esta lista, sem prejuízo da estrutura, desde que siga o padrão do formato.

*MATERIAL
[#_of_materials]

*MATERIAL.LABEL
[#_of_material_labels]
[material_id] ‘material_label(20)’
...

*MATERIAL.PROPERTY.SATURATED.BH
[#_of_saturated_materials]
[material_id] [#_of_points] <m>
[point_id] <H> <B>
...
[material_id] ...
...

*MATERIAL.PROPERTY.MAGNETIC
[#_of_magnetic_materials]
[material_id] <B_x> <B_y> <B_z>
...

*MATERIAL.PROPERTY.ISOTROPIC.CONDUCTIVITY
[#_of_iso_material_conductivties]
[material_id] <s>
...

*MATERIAL.PROPERTY.ISOTROPIC.PERMEABILITY
[#_of_iso_material_permeabilities]
[material_id] <m>
...

*MATERIAL.PROPERTY.ISOTROPIC.PERMISSIVITY
[#_of_iso_material_permissivities]
[material_id] <e>

...

*MATERIAL.PROPERTY.ORTHOTROPIC.CONDUCTIVITY
[#_of_ortho_material_conductivties]
[material_id] <s_x> <s_y> <s_z>
...

*MATERIAL.PROPERTY.ORTHOTROPIC.PERMEABILITY
[#_of_ortho_material_permeabilities]
[material_id] <m_x> <m_y> <m_z>
...

*MATERIAL.PROPERTY.ORTHOTROPIC.PERMISSIVITY
[#_of_ortho_material_permissivities]
[material_id] <e_x> <e_y> <e_z>
...

*MATERIAL.PROPERTY.TERMIC.CONDUCTIVITY
[#_of_material_termic_conductivities]
[material_id] <k>
...

*MATERIAL.PROPERTY.TERMIC.HEAT
[#_of_material_termic_specific_heats]
[material_id] <C>
...

Assim, temos os três arquivos que formam a Base de Dados em Formato Neutro. No Apêndice B, ela é apresentada integralmente, sem comentários, para uma referência rápida. A seguir, mostramos um exemplo de parte da Base de Dados que seria criada para uma geometria bidimensional simples.

2.5 - Um Exemplo da Base de Dados

Apresentamos, a seguir, um exemplo de uso da Base de Dados em Formato Neutro. Montamos o arquivo de Geometria para o caso de um núcleo de aço, envolvido por uma bobina que contém uma certa densidade de corrente. A Figura 2.5 mostra este exemplo em corte:

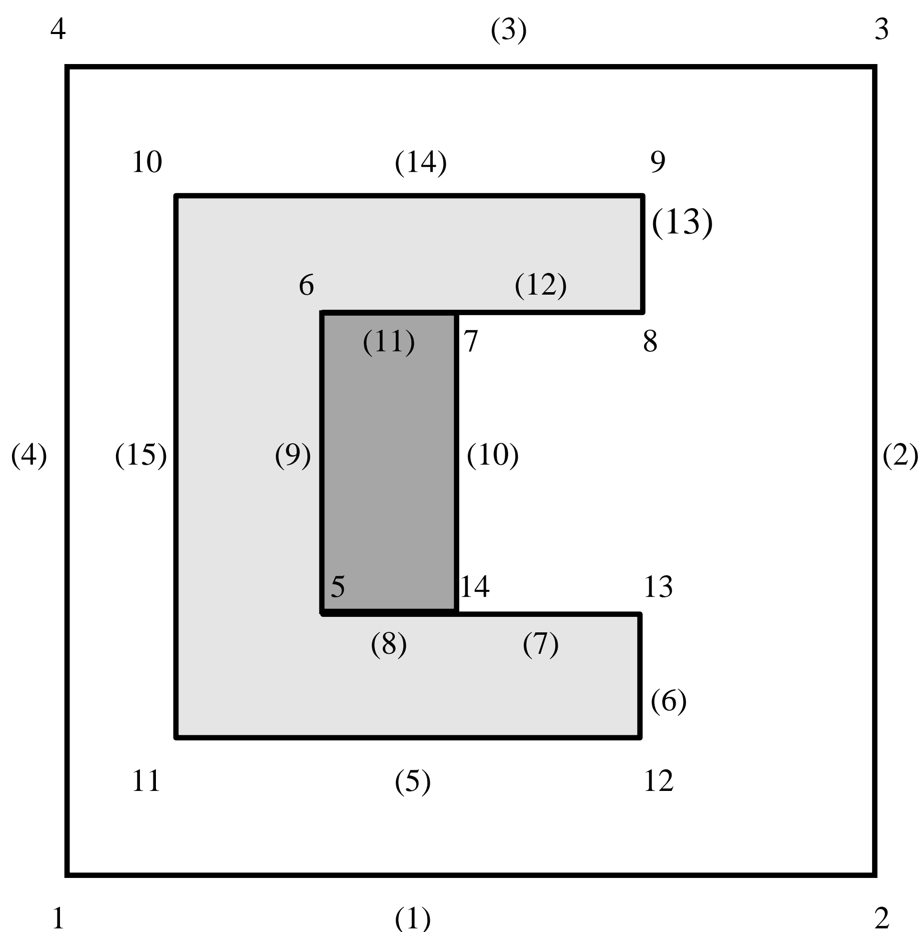


Fig. 2.5 - Exemplo de uso da Base de Dados

A numeração sem parênteses significa os pontos dos vértices da geometria, enquanto a numeração entre parênteses representa as linhas que compõem as faces. Vemos, a seguir, como a geometria deste problema é representada na Base de Dados. Todos os valores apresentados servem apenas como exemplo, não tendo nenhuma representatividade real.

```
*HEADER.TITLE
Arquivo exemplo da representação da geometria na Base de Dados

*HEADER.DIM
2

*HEADER.UNITS
CM

*POINT
14

*POINT.COORD
1  0.0  1.0
2  3.0  1.0
3  3.0  4.0
4  0.0  4.0
5  1.0  2.0
...
13  2.1  2.0
14  1.5  2.0

*LINE
15

*LINE.LABEL
4
1  Contorno
2  Contorno
3  Contorno
4  Contorno

*LINE.SEGMENT.CONTENTS
1  2  1  2
2  2  2  3
...
10  2  7  14
11  2  6  7
...
14  2  9  10
15  2  10  11

*CONTOUR
4
```

*CONTOUR.CONTENTS
1 4 1 2 3 4
2 10 5 6 7 8 9 11 12 13 14 15
3 4 8 10 11 9
4 8 5 6 7 10 12 13 14 15
*FACE
3

*FACE.LABEL
3
1 Aço
2 Bobina
3 Ar

*FACE.PLANE.CONTENTS
1 1 2
2 1 3
3 2 1 4

*ATTRIBUTES
Contorno

*ATTRIBUTES.BOUNDARY
1

*ATTRIBUTES
Aço

*ATTRIBUTES.MATERIAL
3

*ATTRIBUTES.COLOR
5

*ATTRIBUTES
Bobina

*ATTRIBUTES.MATERIAL
1

*ATTRIBUTES.CURRENT
3

*ATTRIBUTES.VOLTAGE
7

*ATTRIBUTES
Ar

*ATTRIBUTES.MATERIAL
10

2.6 - Vantagens e Desvantagens da Base de Dados em Formato Neutro

Apesar do pouco tempo de uso desta Base de Dados em Formato Neutro dentro do Grupo de Otimização e Projeto Assistido por Computador, podemos traçar alguns comentários a respeito de sua estrutura, baseados no seu projeto e na proposta que ele traz:

- com a organização a partir de blocos de dados rotulados, o programa pode "escolher" quais os blocos de dados são relevantes ao caso em processamento. Assim, a existência de dados a mais, que são relevantes a uma certa implementação, não invalida a Base de Dados para uma outra implementação;

- por ser composta de blocos de dados independentes entre si, a atualização da Base é feita com facilidade, pois novos tipos de dados podem ser implementados sem a necessidade de se alterar os dados já existentes;

- pelo mesmo motivo anterior, a criação de uma versão atualizada da Base não invalida versões anteriores, já que os novos blocos de dados são criados em cima da estrutura já existente, sem alterá-la;

- difundido o seu uso, a Base de Dados trará uma generalização dos formatos de transferência de dados para os programas de cálculo de campos eletromagnéticos dos diversos grupos de pesquisa brasileiros. Com isso, haverá uma maior interação entre tais grupos, trazendo um crescente ganho na troca de informações e experiências entre os mesmos;

- outra conseqüência da difusão do uso desta Base entre os grupos de pesquisa é a facilidade de uma posterior passagem para um formato mais global, por exemplo, da ISO, que porventura possa surgir. Com todos os grupos utilizando o mesmo formato, e com uma cooperação mínima entre si, esta transição é feita com

facilidade pela criação de um filtro de tradução único, o que não ocorre se cada grupo possuir unicamente um formato particular;

- mesmo tendo uma estrutura bem genérica, a utilização da Base de Dados implica em dispensar um esforço de programação no sentido de criar rotinas para gerar e ler os seus arquivos. Como cada grupo utiliza um formato particular de armazenamento de dados, esta Base serviria como formato secundário de armazenamento, não invalidando os formatos já existentes de cada grupo;

- o fato de o arquivo ser armazenado em formato ASCII prejudica sua leitura no que se refere à velocidade de acesso aos dados, comprometendo um pouco a sua performance.

Capítulo 3 - Estrutura de Dados Orientada por Objetos

3.1 - Introdução

Como vimos no Capítulo 1, a Programação Orientada por Objetos vem sendo cada vez mais utilizada no desenvolvimento de software em todas as áreas, inclusive a engenharia. Hoje em dia é impossível falarmos de software de qualidade sem mencionarmos este novo paradigma de análise e projeto de grandes sistemas. Como os sistemas computacionais de engenharia têm-se tornado cada vez maiores e mais complexos, nada mais natural que nos valermos destas novas técnicas para que estes conceitos de qualidade possam valer também neste meio.

Baseado na estrutura de um programa de cálculo de campos eletromagnéticos desenvolvido pelo Grupo de Otimização e Projeto Assistido por Computador da UFMG, apresentamos uma estrutura de dados de um Pré-processador (Fig. 1.2) organizado com a utilização de técnicas de Programação Orientada por Objetos.

De início, apresentamos, de maneira resumida, as principais características deste novo conceito de programação, tentando familiarizar o leitor com as técnicas aqui apresentadas que formam a base deste novo paradigma.

Em seguida, apresentamos a estrutura do referido programa que deu origem ao desenvolvimento deste trabalho, suas divisões e organização, procurando, assim, dar um melhor entendimento da nova estrutura.

Introduzidas estas informações, apresentamos a estrutura de dados do pré-processador baseada em objetos.

3.2 - Programação Orientada Por Objetos - Um Breve Resumo

Por ser um conceito relativamente novo, e por estar sendo aplicado há bem pouco tempo em sistemas computacionais para engenharia, a Programação Orientada por Objetos - tratada aqui como OOP, do inglês Object-Oriented Programming - ainda não se tornou comum à grande maioria dos engenheiros e programadores. Para nos familiarizarmos com este novo paradigma, apresentamos, em um breve resumo, a explicação de seus principais conceitos.

Para facilitar a ilustração e considerando que todo o trabalho aqui apresentado foi desenvolvido em C++, utilizamos esta linguagem para exemplificar alguns conceitos.

Não pretendemos dar uma explicação completa e detalhada da OOP, nem das características do C++, mas apenas esclarecer alguns pontos, deixando as referências para uma consulta mais profunda (Weimer & Pinson, 1988; Meyer, 1988; Stevens, 1991; Booch, 1994).

O Que é Programação Orientada por Objetos?

Mesmo sendo um termo muito utilizado atualmente, há uma clara falta de concenso quando tentamos definir a Programação Orientada por Objetos. Enquanto alguns a definem imprecisamente como um novo modelamento de software baseado em objetos de nosso mundo real, outros usam termos como Tipo Abstrato de Dados, afirmando que a OOP envolve o uso de tais tipos. Utilizando um caminho de menor rigor, descrevemos a OOP como um novo modo de organizar os programas, sendo completamente independente da linguagem de programação, ainda que a linguagem que suporta OOP torne mais fácil a implementação de suas técnicas.

Programação Orientada por Objetos é apenas um método para projeto e implementação de software. O uso da OOP, por si só, não confere nada ao produro final que o usuário possa ver. Contudo, a pessoa que desenvolve o software é beneficiada, especialmente em grandes projetos, comuns em engenharia. Como a OOP permite que trabalhemos com conceitos e modelos mais próximos dos problemas de nosso mundo real,

podemos trabalhar melhor com a complexidade do problema que quando somos forçados a mapear o problema para encaixá-lo nas características de uma linguagem. Pode-se, também, tirar vantagem da modularidade dos objetos e implementar o programa em unidades relativamente independentes, fazendo uma manutenção separada de cada uma.

Programação Convencional Orientada por Procedimentos

Antes de conhecermos com mais detalhes a Programação Orientada por Objetos, tentaremos mostrar brevemente a programação convencional, como vem sendo feita ao longo dos anos. Na falta de outro termo, usamos Programação Orientada por Procedimentos para descrever estas técnicas de programação convencionais.

Em uma aproximação com orientação por procedimentos, vemos nosso problema como uma seqüência de coisas a fazer - escrevemos um certo número de funções (procedimentos) que nos fazem completar a seqüência de tarefas. Os dados podem ser organizados em estruturas, mas o foco principal é sempre nas funções. Como na ilustração que se segue, a função transforma os dados de alguma maneira. Por exemplo, podemos ter uma função para adicionar um conjunto de números, outra para computar uma raiz quadrada, e outra para mostrar uma string na tela do vídeo. Não é necessário ir muito longe para encontrar esta estrutura - quase todas as bibliotecas de rotinas são implementadas deste modo. Cada função de tal biblioteca realiza uma operação muito bem definida em seus argumentos de entrada, e devolve o dado transformado como valor de retorno, através de um ponteiro para uma área de armazenamento ou diretamente a um dispositivo como a tela de vídeo. A Fig. 3.1 ilustra esta transformação de dados, onde a função *sqrt* recebe um valor *x* e retorna sua raiz quadrada.

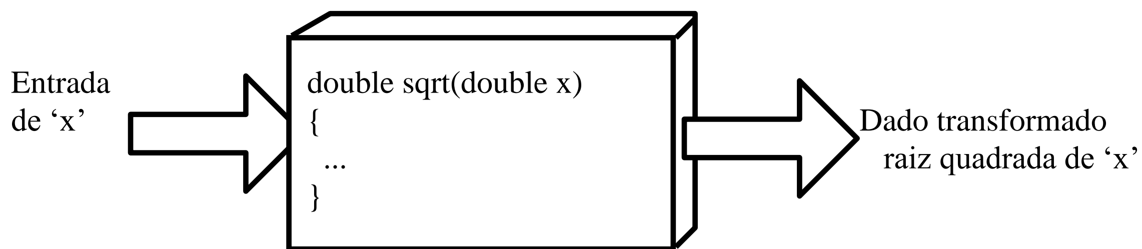


Fig. 3.1: Transformação de dados por função

Não seria correto afirmar que com técnicas orientadas por procedimento não há preocupação com a organização dos dados. De fato, tem-se grande atenção nos dados utilizados em uma aplicação e é costume organizar partes relacionadas de dados em unidades, como as *struct* da linguagem C. Mas, feito isso, escrevem-se as funções que operam com esta estrutura de dados. O fato é que não há nenhuma ligação forte entre dados e funções. A diferença entre a programação convencional e a orientada por objetos é a maneira com que os dados e funções são agrupados.

As Bases da Programação Orientada por Objetos

Diferentemente da programação orientada por procedimentos, a OOP trata dados como primários e funções como secundários. Pode-se dizer que ao invés de dados serem o que as funções usam, funções são o que os dados fazem. Em vez de permanecerem isoladas, funções são fortemente associadas aos dados. Se a linguagem de programação suporta OOP, há certamente uma sintaxe específica que pode ser usada para indicar esta relação, por exemplo, a construção *class* em C++.

Embora especialistas não concordem em uma definição exata da OOP, uma descrição geral pode ser sintetizada em três idéias básicas: *abstração de dados* (com encapsulamento), *herança* e *polimorfismo*. À medida que cada uma for descrita, poderemos perceber que são características utilizadas e procuradas por bons programadores, mesmo na programação convencional. Entretanto, mesmo que vários conceitos da OOP sejam antigos e familiares, sua integração sob este rótulo é, de certo modo, novo em engenharia. Por esta razão, este tópico merece um exame mais detalhado.

Abstração de Dados e Encapsulamento - Como exemplo de um tipo abstrato de dados, podemos estudar um conjunto de rotinas de entrada e saída de dados em arquivo, como é feito na biblioteca de rotinas da linguagem C. Estas rotinas enxergam o arquivo como um fluxo de bytes, e permitem a realização de várias operações neste fluxo. Por exemplo, pode-se abrir um arquivo (*fopen*), fechá-lo (*fclose*), ler um caractere do arquivo (*getc*), escrever

um caractere no arquivo (*putc*), e assim por diante. Este modelo abstrato de arquivo, no caso da linguagem C, é implementado pela definição de um novo tipo de dados, FILE. Todavia, para usar o tipo FILE, não é necessário ver ou se preocupar com a estrutura de dados do C que a define. Na verdade, a estrutura de dados de FILE pode variar de um sistema para outro. Contudo, as rotinas de entrada/saída em um arquivo trabalham da mesma maneira em todos os sistemas. Este conceito é conhecido como encapsulamento de dados, ou seja, não é necessário ao usuário saber como o tipo de dados foi implementado, e sim como usá-lo, quais as subrotinas que o formam etc. Há, na abstração de dados, uma separação entre dados privados e dados públicos, ou seja, dados que são acessíveis ao usuário e dados que ele não pode acessar. Com este encapsulamento o programador pode restringir o acesso dos dados, forçando o usuário a utilizar apenas as rotinas que manipulam os dados (métodos).

Abstração de dados é a combinação da definição de um tipo de dados e do seu encapsulamento. O exemplo do tipo de dados FILE da linguagem C é ilustrado na Fig. 3.2:

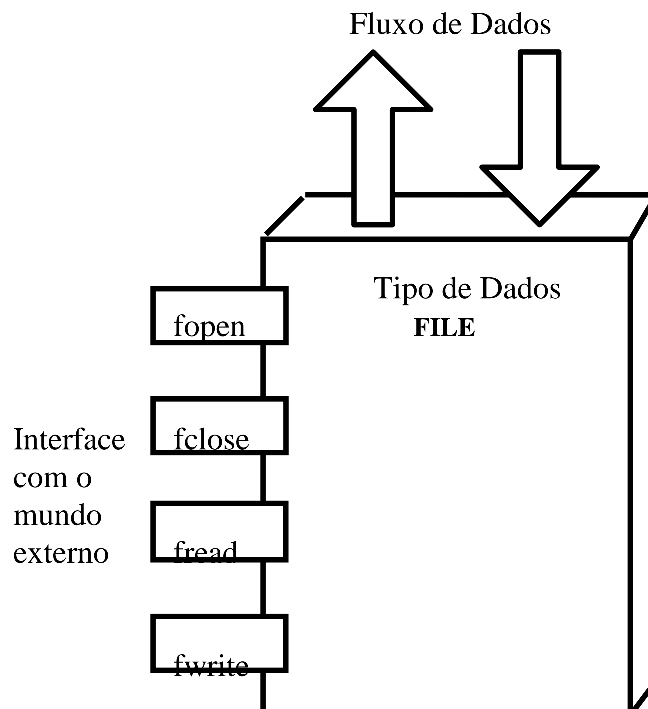


Fig. 3.2: Exemplo de um tipo abstrato de dados.

Objetos e Abstração de Dados - Usa-se a idéia de abstração de dados para criar um objeto, definindo um bloco de dados junto com as funções necessárias para operar estes dados. Os dados representam a informação contida em um objeto e as funções modelam o comportamento do objeto - elas definem as operações que podem ser realizadas sobre o objeto. Os dados não são acessíveis ao mundo externo. A única maneira de se utilizar o objeto é chamando uma das funções que implementam seu comportamento.

Assim, o tipo de dados FILE, juntamente com as rotinas de entrada/saída de arquivo, pode ser considerado uma definição de um objeto. Cria-se um objeto FILE chamando a função *fopen*, que retorna um ponteiro para uma estrutura FILE. O acesso ao objeto FILE é feito através deste ponteiro. Toda vez que *fopen* for chamada, um novo objeto FILE é criado. Pode-se imaginar o tipo FILE como um modelo (*template*) que é usado para gerar novas ocorrências de FILEs quando *fopen* é chamado. O termo *objeto* refere-se a esta ocorrência.

Classes e Métodos - Na terminologia da OOP, o modelo que define o tipo de dados de um objeto é usualmente chamado de classe, termo que pode variar de uma linguagem com OOP para outra. Desse modo, cada ocorrência de uma classe é um objeto.

As funções que operam em um objeto recebem um nome especial - são chamadas de métodos, por ser este o nome usado na linguagem Smalltalk, uma das primeiras a oferecer recursos para a OOP. São os métodos, chamados em C++ de funções membro da classe, que definem o comportamento de um objeto. Assim como os dados, as funções membro pertencem a uma classe.

O Smalltalk forneceu outro termo à OOP - o de enviar uma mensagem a um objeto. Refere-se ao ato de instruir um objeto a realizar uma operação através da chamada de um de seus métodos. Em C++ isto é feito chamando-se uma função membro apropriada do objeto.

Herança - Um objeto do mundo real é quase sempre uma extensão de um objeto existente. Por exemplo, costuma-se descrever coisas usando sentenças como: Y é como X, com exceção de que Y também faz isto e aquilo. Com isto, define-se um novo objeto apontando como suas características diferem das de um objeto anterior.

OOP permite implementar esta noção de definir um objeto em termos de um anterior. O termo *herança* é utilizado para este conceito, pois pode-se pensar em um objeto herdando as propriedades de outro ou, mais precisamente, uma classe herdando o comportamento de outra classe. A herança impõe um relacionamento hierárquico ancestral-descendente entre as classes onde o descendente herda de seu ancestral. Em C++ costuma-se utilizar o termo classe base para o ancestral e classe derivada para seu descendente.

Outra visão possível da herança é a de uma classe mais específica sendo criada a partir de uma classe mais genérica. Como vemos na Fig. 3.3, temos um exemplo onde criamos, a partir de uma classe genérica *vegetal*, três especializações: *árvores*, *arbustos* e *fungos*. Já a classe *árvores* pode ainda dar origem a duas classes ainda mais específicas: *caducifólios* e *perenifólios* (quanto à permanência de suas folhas nas estações frias).

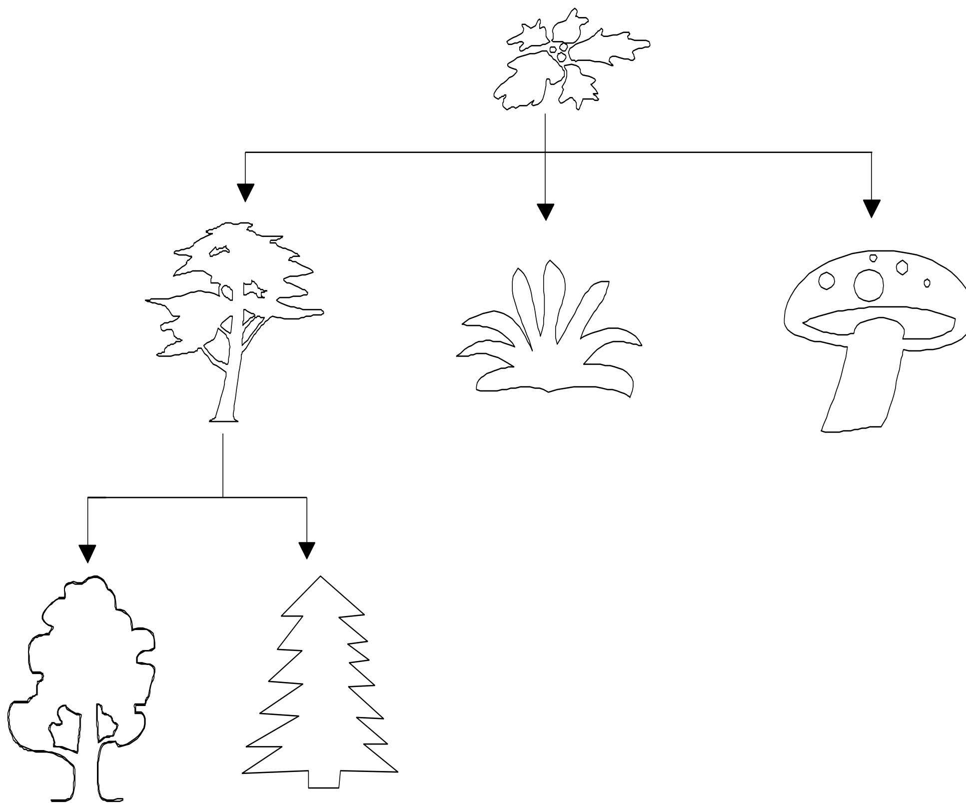


Fig. 3.3: Exemplo de classes com herança

Assim, a classe base contém apenas informações mais gerais, comuns a todas as espécies de vegetais, deixando às suas derivadas, cada vez mais específicas, o refinamento das características.

Há também a possibilidade de um descendente possuir mais de um ancestral, o que é chamado de herança múltipla, onde o descendente herda as características de cada um de seus ancestrais.

Polimorfismo - Polimorfismo é a propriedade que permite que uma operação tenha diferentes comportamentos em diferentes objetos. Em outras palavras, objetos diferentes reagem de modo diferente à mesma mensagem. Por exemplo, consideremos a operação de adição. Para dois números, a adição gera um resultado numérico que é a soma de seus valores. Mas, e se considerarmos os objetos da adição como sendo strings? É desejável, então, que a adição resulte na concatenação de uma string na outra.

De modo semelhante, suponhamos um certo número de formas geométricas que compreendam a mensagem *desenha*. Cada objeto reage a esta mensagem se mostrando na tela do vídeo. Obviamente, um objeto retângulo se desenhará diferentemente de um objeto círculo, apesar de a mesma mensagem *desenha* ter sido enviada a ambos.

Polimorfismo tem um papel importante em ajudar-nos a simplificar a sintaxe de executar a mesma operação a uma coleção de objetos. Por exemplo, polimorfismo permite que todas as formas geométricas sejam desenhadas com um loop semelhante a:

para cada forma no array de formas
envie mensagem *desenha* à forma

Isto é possível porque, a despeito da geometria exata de cada forma, cada objeto entende a mensagem desenha e reage de maneira apropriada às suas características.

Programação Orientada por Objetos e C++

Como foi mencionado, a linguagem utilizada na implementação deste trabalho é o C++. Apresentaremos nesta seção as características principais da OOP que o C++ oferece, sendo uma linguagem projetada para suportá-la, mostrando como os três pontos citados (abstração de dados, herança e polimorfismo) são implementados na linguagem.

Abstração de dados, classes e herança - Em C++, objetos são definidos usando a construção *class*, que é similar ao *struct* em C. Como exemplo, implementamos duas classes de objetos gráficos, círculo e retângulo, usando técnicas de OOP. A Fig. 3.4 ilustra o fato de que cada uma dessas classes será herdeira de uma classe que chamamos Objeto Gráfico Genérico.

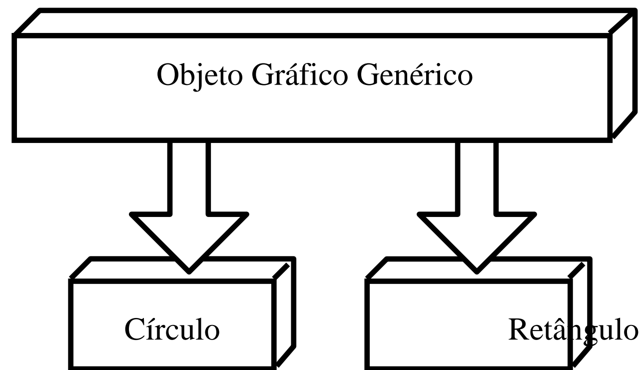


Fig. 3.4: Hierarquia de classes do exemplo.

Podemos definir o objeto *Circulo* como abaixo:

```
class Circulo : public Objeto_Grafico
{
      int Raio;
  public:
      Circulo(int x, int y, int raio);
      void Desenha(int cor, int x, int y);
};
```

No código entre chaves, o valor inteiro `Raio` representa os dados da classe `Circulo`. Imediatamente em seguida aos dados estão os protótipos das funções precedidas pela palavra-chave `public`. Estas são as funções membro da classe e a palavra-chave `public` determina que são visíveis às outras partes do programa. Uma ocorrência da classe é

declarada e suas funções membro são acessadas como nas estruturas em C. Por exemplo, para chamar uma função `Desenha` de um `Circulo` de nome `c1`, escreve-se:

```
Circulo c1;
...
c1.Desenha(7,100,100);
```

A variável `Raio`, que aparece antes da palavra-chave `public`, é, por *default*, *private*, ou privada, e não é acessível a nenhuma função que não for definida dentro da classe.

Quando uma *class* é definida, um novo, e provavelmente complexo, tipo de dados é definido. Usa-se a palavra-chave `private` para esconder os detalhes internos desta estrutura de dados do mundo externo. O único meio deste ambiente manipular os dados é através das funções membro públicas. Deste modo, o mecanismo *class* permite a implementação da abstração de dados e a promoção do encapsulamento (modularidade).

Quando declaramos uma classe, indicamos, também, se ela é herdeira de outra classe. Na primeira linha da declaração *class*, põe-se um sinal de dois pontos (:) seguido da lista de classes base das quais esta classe é herdeira. Como `Circulo` é herdeira de `Objeto_Grafico`, a primeira linha da declaração de class indica esta relação:

```
class Circulo : public Objeto_Grafico
{
...
};
```

Neste caso, a classe `Objeto_Grafico` é a classe base e `Circulo` é a classe derivada. A palavra-chave `public`, que precede `Objeto_Grafico` significa que qualquer membro público (dado ou função) de `Objeto_Grafico` será considerado também público em `Circulo`.

Implementando os objetos gráficos em C++ - A definição das classes que implementam os objetos gráficos em C++ é a primeira tarefa a ser feita. Na listagem a seguir, temos as definições das classes `Objeto_Grafico`, `Circulo` e `Retangulo`.

Listagem 1

```
class Objeto_Grafico
{
      int x;
      int y;
  public:
      void Move(int x, int y);
      virtual void Desenha(int cor, int x, int y);
};


class Circulo : public Objeto_Grafico
{
      int Raio;
  public:
      Circulo(int x, int y, int raio);
      void Desenha(int cor, int x, int y);
};

class Retangulo : public Objeto_Grafico
{
      int Largura;
      int Altura;
  public:
      Retangulo(int x, int y, int largura, int altura);
      void Desenha(int cor, int x, int y);
};
```

O arquivo de cabeçalho encerra apenas os protótipos das funções membro. As implementações são, geralmente, feitas em arquivos separados.

Construtora e Destrutora - Podemos observar a existência em cada uma das classes - círculo e retângulo - de uma função membro com o mesmo nome da classe. Essas rotinas são conhecidas como construtoras.

A construtora, se definida, é chamada toda vez que um objeto da classe for criado. Ela permite que seja feita alocação de memória extra (na criação de um array, por exemplo), se necessário, e inicialização dos dados do objeto. A construtora sempre tem o mesmo nome da classe e sua chamada é implícita à criação do objeto.

Pode-se, também, definir uma destrutora, se há alguma necessidade de se fazer uma limpeza de memória antes da destruição de um objeto (por exemplo, se é necessário liberar memória alocada pela construtora). A destrutora também tem o mesmo nome da classe, com exceção de um til (~) prefixado. Assim, a destrutora de `Circulo` seria `~Circulo`.

A Listagem 2 mostra a implementação das classes classes `Objeto_Grafico`, `Circulo` e `Retangulo`. Este exemplo é pequeno o suficiente para que todo o código seja mantido reunido em um único arquivo, mas a separação aqui serve para ilustrar a disposição para o caso de um projeto mais complexo, utilizando técnicas de OOP.

Listagem 2

```
void Objeto_Grafico::Desenha(int cor, int x, int y)
{
}

void Objeto_Grafico::Move(int novox, int novoy)
{
 Desenha(BGCOLOR, x, y);            // BGCOLOR = cor de fundo
 Desenha(FGCOLOR, novox, novoy);    // FGCOLOR = cor de escrita
 x = novox;
 y = novoy;
}

Circulo::Circulo(int x, int y, int r)
{
  Raio=r;
  Desenha(FGCOLOR, x, y);
}

Circulo::Desenha(int cor, int x, int y)
{
  setcolor(cor);
  circle(x,y,Raio);         // função do sistema gráfico do C++
}

Retangulo::Retangulo(int x, int y, int l, int a)
{
  Largura = l;
  Altura = a;
  Desenha(FGCOLOR, x, y);
}

Retangulo::Desenha(int cor, int x, int y)
{
  setcolor(cor);
  rectangle(x, y, x+Largura, y+Altura);     // função do sistema
}                                          //   gráfico do C++
```

Como foi feito neste exemplo, primeiro define-se as classes de objetos do problema, tarefa que pode não ser tão fácil, dependendo da complexidade do problema, devido à dificuldade para se identificar as classes e sua hierarquia. Uma vez definidas as classes, decide-se sobre a interface com o mundo externo - as funções membro. Então, pode-se fazer a implementação das funções membro de cada classe.

A definição das funções membro de cada classe é semelhante à definição de uma função comum em C, com exceção do operador de escopo (::) que identifica em que classe está declarada a função, como no exemplo abaixo, para a função *Move* da classe `Objeto_Grafico`:

```
void Objeto_Grafico::Move(int novox, int novoy)
{
...
}
```

Após a implementação das classes, resta-nos escrever o código para testar o exemplo de objetos gráficos. Como foi feito nas Listagens 1 e 2, mostraremos apenas a parte principal, omitindo os detalhes da implementação, como o uso das bibliotecas e manipulação das rotinas gráficas. A Listagem 3 traz este programa teste:

Listagem 3

```
void main()
{
  int i, j, x, y;
  Objeto_Grafico *Objetos[2];

  // Inicializa o sistema gráfico

  Objetos[0] = new Circulo(100,100,50);
  Objetos[1] = new Retangulo(150,150,20,20);

  x = 100;
  y = 100;

  for(j = 0; j < 100; j++)
   {
    for (i = 0; i < 2; i++)
      Objetos[i] -> Move(x,y);
    x += 2;
   }
```

```
  delete Objetos[0];
  delete Objetos[1];

  //  Finaliza sistema gráfico

}
```

Vale a pena ressaltar aqui uma importante característica da OOP - o polimorfismo. Na declaração da classe `Objeto_Grafico`, na Listagem 1, vemos a função `Desenha` declarada do seguinte modo:

```
virtual void Desenha(int cor, int x, int y) {}
```

A palavra-chave `virtual` precede a declaração da função e esta é definida como vazia.

A função virtual é a chave para o polimorfismo que, como definido anteriormente, permite que diferentes objetos reajam diferentemente à mesma operação (ou chamada de função, em C++). No exemplo, cada classe `Circulo` e `Retangulo` define `Desenha` de modo a atender suas próprias necessidades.

Na Listagem 3, foi criado um círculo e um retângulo e armazenados seus ponteiros em um array de objetos gráficos. No loop `for` que segue, ambos objetos são movidos, pela chamada da função `Desenha`, que está em `Move`, através do ponteiro para o objeto. Porque `Desenha` na classe base foi declarada como virtual, a função `Desenha` chamada em tempo de execução é definida na classe derivada, caracterizando a ação polimórfica da função.

Apresentamos aqui, de forma reduzida, as características da OOP e como estas técnicas são implementadas em C++. Acreditamos ser o suficiente para a introdução do leitor no assunto, deixando aos mais interessados um estudo mais aprofundado através de algumas referências (Weimer & Pinson, 1988; Meyer, 1988; Stevens, 1991; Booch, 1994).

3.3 - Estrutura de Dados Primária

O pré-processador utilizado como base para a montagem da estrutura de dados orientada por objetos é o programa MALHA, desenvolvido pelo Grupo de Otimização e Projeto Assistido por Computador (GOPAC) da UFMG, que faz parte de um programa completo, juntamente com o processador e o pós-processador, de cálculo de campos eletromagnéticos tridimensionais utilizando o Método de Elementos Finitos.

A estrutura de dados do programa, desenvolvido na linguagem FORTRAN, é a representação de um domínio estratificado em camadas de elementos hexaédricos, como mostra a Fig. 3.5. A partir destas camadas está definida a malha de elementos finitos do problema, onde podem ser definidas regiões (blocos) de materiais diferentes ao do meio, fontes de corrente e as condições de contorno.

As informações necessárias para a construção de um problema foram divididas conforme relação abaixo:

- CAMADAS - define o número de camadas de elementos em cada uma das direções do espaço;

- MALHA REGULAR - define as dimensões reais das camadas de elementos;

- CONDIÇÕES DE CONTORNO - define a região e o tipo de condições de contorno impostas;

- BLOCOS - define as regiões do domínio preenchidas com outros materiais ou com tipo de potencial diferente;

- FONTES DE CORRENTE - define que tipo e qual a localização das fontes de corrente do problema;

- MATERIAIS - banco de dados com as características dos materiais utilizados.

Há ainda no programa a informação sobre a existência ou não de periodicidade dos nós para a geração da malha, mas não entraremos em detalhes sobre seu conteúdo, sendo que seus dados não foram utilizados na montagem da estrutura de dados orientada por objetos.

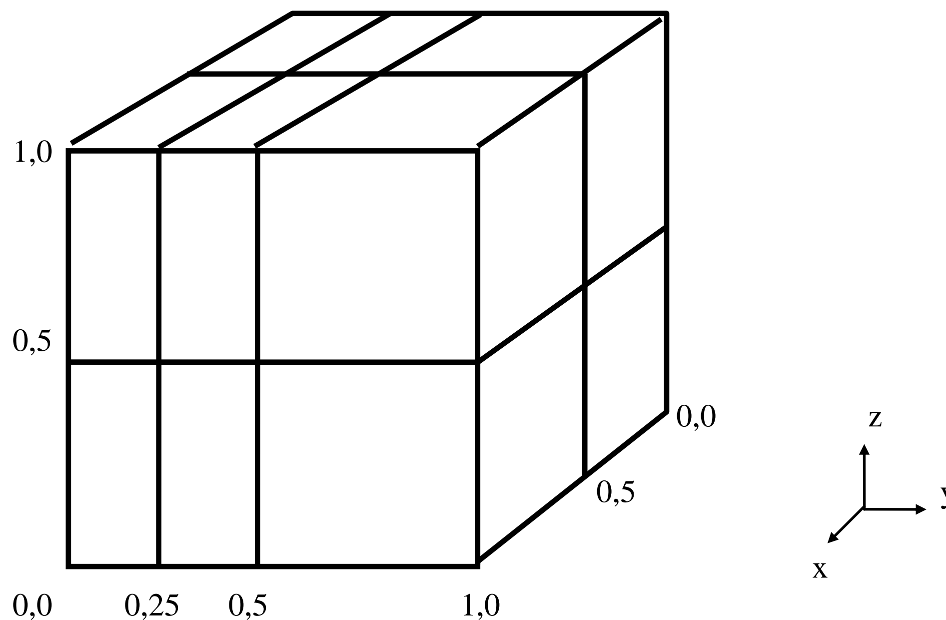


Fig. 3.5 - Exemplo de uma definição de domínio

Baseado no exemplo da Fig. 3.5 explicamos a seguir, com mais detalhes, cada um dos blocos de dados da estrutura primária:

Camadas

Como estamos trabalhando com problemas tridimensionais, o primeiro passo para a definição do domínio do problema é a escolha do número de camadas de elementos em cada uma das direções do espaço (x, y e z). No exemplo dado as camadas estão distribuídas de acordo com a Tabela 1:

	Direção X	Direção Y	Direção Z
Núm. de Camadas	2	3	2

Tabela 1: Número de camadas de elementos

Malha Regular

Completa-se a definição do domínio, especificando a dimensão de cada uma das camadas de elementos. Juntamente com Camadas, este conjunto de dados permite que uma região do domínio receba uma malha de elementos finitos mais refinada que outra. No exemplo dado, podemos ver isso claramente onde, à esquerda da figura, os elementos são divididos em elementos menores. Na Tabela 2, disposta de tal maneira a reproduzir a entrada de dados do programa, vemos que, entre um dado da dimensão e seu subseqüente, estão dispostas **n** camadas, ou seja, para o caso do eixo Y, há duas camadas entre 0,0 (coordenada inicial) e 0,5, e uma camada entre 0,5 e 1,0 (coordenada final). Os valores das dimensões são dados em P.U., e o valor nulo na coluna de camadas indica o fim da entrada de dados naquele eixo.

	X		Y		Z	
Num.	Dimensão	Camadas	Dimensão	Camadas	Dimensão	Camadas
1	0,0	2	0,0	2	0,0	2
2	1,0	0	0,5	1	1,0	0
3	---	---	1,0	0	---	---

Tabela 2: Dimensões da malha regular

Condições De Contorno

A definição de qualquer condição de contorno imposta ao problema passa por duas etapas. Primeiro, a definição da região onde é imposta, e depois, o tipo da condição de contorno e seus dados. A região é definida através dos nós da malha dimensionada em ***Camadas*** e ***Malha Regular***. Se temos, numa direção do espaço, 10 camadas, os nós são numerados de 1 a 11. Seguindo esta numeração, são fornecidos os nós inicial e final em cada direção. Após isto, escolhe-se o tipo da condição imposta, fornecendo seus dados particulares. Na Tabela 3 temos o exemplo de uma condição de contorno imposta na face superior perpendicular ao eixo z da Fig. 3.5, com a imposição de um potencial escalar de 10V.

Direção	X	Y	Z
Nó inicial	1	1	3
Nó final	3	4	3

Potencial imposto: 10 V.

Tabela 3: Condição de contorno imposta

Os tipos que estão implementados na versão atual do programa são: potencial escalar, potencial vetor, ambos (escalar+vetor), flutuante, térmica de convecção e térmica com temperatura imposta.

Blocos

Inicialmente, todo o domínio é preenchido por um mesmo material, e todos os elementos tendo o mesmo tipo de potencial para o processamento dos cálculos. Tanto o tipo de material, quanto o de potencial são valores default definidos pelo usuário, que também podem ser modificados. A estrutura de blocos foi criada para que qualquer região do domínio pudesse ter esta condição modificada.

Para se definir um bloco de material, basta fornecer, para cada uma das direções do espaço, qual a camada de elementos em que o bloco começa e em qual ele termina, junto com o tipo do material e o tipo do potencial desejado.

Como exemplo, na Tabela 4 definimos um bloco que ocupa a metade superior do domínio mostrado na Fig. 3.5.

Direção	X	Y	Z
Camada inicial	1	1	2
Camada final	2	3	2

Tabela 4: Definição de um bloco no domínio.

Fontes De Corrente

Podemos definir fontes de corrente que agem sobre o domínio de duas maneiras: através de blocos de camadas de elementos ou com topologias geométricas específicas (fios, planos, cilindros etc.).

As fontes em forma de blocos são definidas através das camadas, como em BLOCOS, onde são fornecidos também o valor e a direção da corrente. As topologias são geometrias pré definidas (por exemplo, um plano no espaço) onde são definidos o posicionamento e a dimensão em relação ao domínio, e o valor da corrente superficial. Os tipos de geometria que esta versão da estrutura suporta são: fio, arco, plano, casca, tijolo e cilindro, todos definidos num espaço 3D.

Materiais

Este é um banco de dados, praticamente independente do problema, que armazena os dados dos materiais utilizados, não só por este, mas por qualquer problema que venha a ser montado. O armazenamento e a recuperação de seus dados é independente do

armazenamento e da recuperação dos dados de um problema específico. Assim, para vários problemas diferentes, o programa pode acessar o mesmo arquivo de materiais, ou, também, pode ser criado mais de um banco de dados de materiais, cabendo ao usuário gerenciar o seu uso. Neste programa, os materiais são bem definidos e diferenciados: linear, saturável, ímã, condutor linear, condutor não linear e condutor térmico.

Estes dados formam a base para a criação da estrutura orientada por objetos. Em se tratando de lógica, pouco foi modificado. Foi feita a organização dos dados em classes e objetos, utilizando a herança e o polimorfismo, para o aproveitamento das vantagens desta técnica. Veremos adiante como foi feita a implementação da Programação Orientada por Objetos nesta estrutura primária.

3.4 - Estrutura de Dados Orientada por Objetos

Aplicando as técnicas de Programação Orientada por Objetos na elaboração de uma estrutura de dados baseada na estrutura de dados primária apresentada, procurou-se manter como objetivo principal a capacidade de reuso da mesma (Meyer, 1988), ou seja, desenvolvida a estrutura, qualquer implementação pode ser feita com um máximo reaproveitamento do código já escrito. Por exemplo, suponhamos que um novo tipo de fonte de corrente seja necessário para o cálculo de um certo problema. O ideal seria que o programador pudesse aproveitar ao máximo a estrutura de dados existente, para que fosse possível escrever o mínimo necessário para tal modificação.

O conceito de extensibilidade (Meyer, 1988) também é desejável, ou seja, qualquer mudança na especificação do programa deve refletir o mínimo possível na estrutura de dados, que deve permitir alterações parciais sem a necessidade de uma reestruturação completa.

Outras características de um bom programa orientado por objetos buscadas foram a compatibilidade e a portabilidade. A primeira, neste caso, revela uma estrutura que pode ser implementada em várias linguagens diferentes, apesar de a necessidade da existência dos conceitos de Programação Orientada por Objetos na linguagem restringir este universo. A segunda indica a possibilidade de se usar a implementação em máquinas (hardwares) diferentes sem a necessidade de alterações. Assim, a estrutura será praticamente independente do ambiente de sua implementação, podendo este ser alterado sem a necessidade de alteração de nenhuma característica da estrutura.

Se conseguirmos uma estrutura de dados com todas estas características, esta terá alcançado um grau muito bom de flexibilidade, permitindo que os dados sejam alterados e/ou acrescentados, ou seu ambiente modificado, sem alterações na sua organização.

Na estrutura de dados primária, com exceção do número de camadas de elementos, todos os dados são organizados em forma de listas. Assim, a primeira medida tomada foi criar uma classe gerenciadora de listas. Com base nesta classe montamos todas as listas necessárias. A seguir, veremos as classes que formam o gerenciador de listas e as classes

que são os nós de cada uma das listas (Blocos, Malha Regular, Condições de Contorno, Fontes de Corrente e Materiais).

Para um melhor entendimento das classes e suas heranças, usamos a representação gráfica utilizada em Forde et al. (1990), como apresentado na Fig. 3.6:

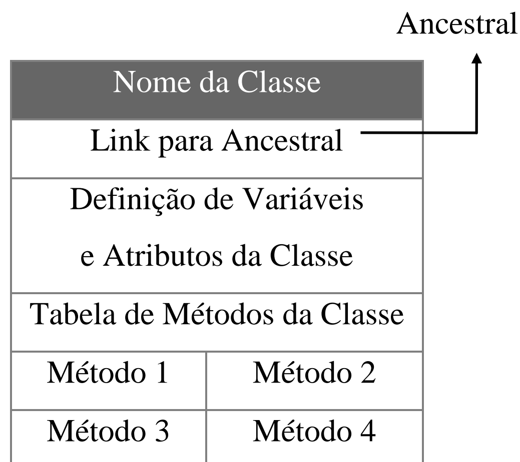


Fig. 3.6 - Representação gráfica de uma classe

A sintaxe para a definição de dados privados e públicos, bem como a de métodos virtuais, foi acrescida, onde temos a seguinte notação: dados privados são escritos com sublinhado e métodos virtuais estão em *itálico*.

Classe Base

Uma medida tomada, no que diz respeito à sua "orientabilidade", para que toda a estrutura de dados mantivesse um alto grau de orientação por objetos, foi a criação de uma classe que servisse de ancestral para qualquer outra classe, direta ou indiretamente. Esta classe, chamada **Base**, é completamente vazia, e não possui outra função, senão interligar todas as classes, fazendo com que venham de uma mesma base. Assim, considera-se que todo e qualquer dado (classe), tenha um mesmo ancestral comum, aumentando o nível de orientação por objetos de toda a estrutura. Esta decisão fica mais clara quando pensamos em

um sistema como sendo sempre composto de outros sistemas e também servindo para a composição de um sistema maior (Booch, 1994).

Gerenciador de Listas

Para o gerenciamento das listas de dados, uma classe que implementa uma lista encadeada simples (Wirth, 1989; Ziviani, 1993) foi criada. A classe **List** (Fig. 3.7) contém o ponteiro para o último nó da lista e os métodos para o gerenciamento da lista. A classe **ListNode** (Fig. 3.8) serve como base para qualquer nó da lista.

Na verdade, **List** é um gerenciador de ponteiros para objetos. As operações de inserção, remoção e busca são feitas com base em uma lista encadeada de ponteiros para a classe **ListNode**. Quando uma classe é criada, fazendo de **ListNode** sua ancestral, esta classe passa a estar habilitada para ser também encadeada na lista.

Ao declarar um objeto de tal classe, passa-se, através dos métodos de gerenciamento de **List**, o ponteiro do objeto para a lista. Recuperando os ponteiros dos objetos, também através dos métodos de **List**, o usuário pode executar qualquer dos métodos da classe listada.

Assim, vemos que, para criarmos uma lista, basta derivar uma classe de **ListNode** (ou de uma derivada) com seus dados e métodos particulares, e criar um objeto do tipo **List** (ou de uma derivada) que, com seus métodos, poderá gerenciar estes nós.

List	
LastNode	
Append	ForEach
Free	Prev
Empty	Node
Count	Remove
ForEach	FreeAll
ForEachArq	*Read*
First	*Write*
Last	Insert

Fig. 3.7 - Classe **List**

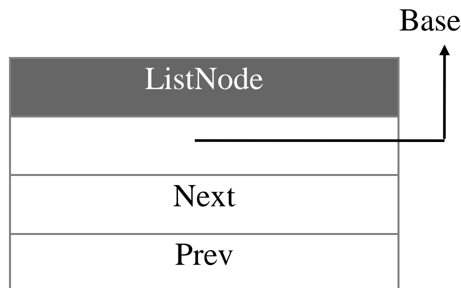


Fig. 3.8 - Classe **ListNode**

Com exceção dos métodos **Read** e **Write**, que são virtuais e vazios, indicando a necessidade de redefinição em uma derivada, todos os métodos agem sobre a lista, ou modificando-a, como **Append**, **Free** e **Insert**, ou retornando alguma informação de seu estado atual, como **Empty**, **Count**, **First** e **Next**.

Como exemplo, vejamos a declaração dos métodos **Append** e **First**. O método **Append** serve para adicionar nós ao final da lista e recebe um ponteiro para um nó a ser adicionado na lista, não retornando nada, como mostrado no seu protótipo, abaixo:

```
void Append(ListNode *N);
```

O método **First** não recebe nenhum parâmetro e retorna um ponteiro para o primeiro nó da lista:

```
ListNode *First();
```

Os métodos **Read** e **Write** foram criados para que, mediante a criação de uma classe derivada de **List**, eles pudessem ser redefinidos, com as respectivas funções de gravação e leitura em arquivo, atendendo aos nós específicos da lista em questão.

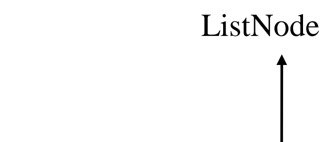
Como veremos a seguir, esta estrutura de listas é a base do armazenamento dos dados. Sua listagem completa é apresentada no Apêndice C.

Blocos

Para a montagem da lista de blocos, criamos duas classes: o nó da lista, contendo os dados para a definição de um bloco, e a própria lista que gerencia estes nós.

A primeira, chamada **NodeBloco** (Fig. 3.9), é derivada da classe **ListNode**, e contém o número das camadas inicial e final nas direções x, y e z do espaço, o tipo de material que preenche o bloco, e o tipo de potencial aplicado no cálculo.

A segunda, chamada **ListBloco** (Fig. 3.10), é derivada de **List** e redefine os métodos **Node** (retorna o ponteiro para um nó **n** da lista), **Read** e **Write** (leitura e escrita em disco).



NodeBloco	
Mxi Myi Mzi Mxf Myf Mzf Material Potencial Label	
GetMXi	GetMXf
GetMYi	GetMYf
GetMZi	GetMZf
SetLabel	GetLabel
GetMaterial	GetPotencial
SetBloco	

Fig. 3.9 - Classe **NodeBloco**

List

ListBloco	
Read	*Write*
Node	

Fig. 3.10 - Classe **ListBloco**

Assim, declarando um objeto do tipo **ListBloco**, temos todos os métodos necessários para o gerenciamento dos nós **NodeBloco**, podendo adicionar, remover e

manipular os dados de qualquer bloco. Posteriormente, veremos que a própria estrutura já faz a declaração deste objeto em uma classe específica.

Procurou-se, ao longo da definição da estrutura, seguir uma norma para o armazenamento e recuperação dos dados nos objetos:

- para os métodos que armazenam os dados das variáveis do objeto, usou-se o prefixo **Set**, seguido da descrição do(s) dado(s) ou da variável. Como exemplo, temos o método **SetBloco**(...), que armazena os valores passados como parâmetros nas variáveis de **NodeBloco**;

- para os métodos que recuperam os dados dos objetos, usou-se o prefixo **Get**, também seguido da descrição do(s) dado(s) ou da variável. Como exemplo, temos o método **GetMaterial**(), que retorna um valor inteiro: o identificador do material que preenche o bloco.

Malha Regular

A Malha Regular não é composta de uma, mas de três listas, cada uma contendo as dimensões da malha em uma das direções do espaço (eixos x, y e z). Para os nós destas listas foi definida a classe **NodeMalhaRegular** (Fig. 3.11) com os respectivos dados e os métodos de acesso aos mesmos, também derivada de **ListNode**. Para a declaração das três listas, criou-se uma classe **MalhaRegular** (Fig. 3.12) com a declaração de três ponteiros para **List**.

A organização é feita de acordo com o apresentado no item 3.3 (Malha Regular) da estrutura primária. Cada nó da lista contém os dados da dimensão e do número de divisões

para o dimensionamento da malha em cada direção. Unindo-se as divisões nas três direções, tem-se a malha regular do domínio.

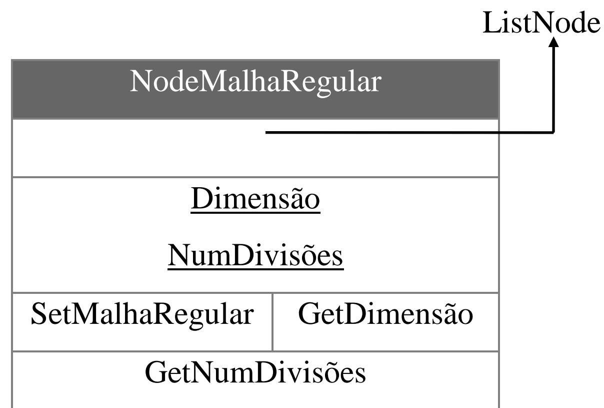


Fig. 3.11 - Classe **NodeMalhaRegular**

<table>
<tr><th colspan="2">MalhaRegular</th></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2"><u>DirX</u> <u>DirY</u> <u>DirZ</u></td></tr>
<tr><td>Append</td><td>Count</td></tr>
<tr><td>SetNode</td><td>GetPtrEixo</td></tr>
<tr><td>GetDimensãoNode</td><td>Free</td></tr>
<tr><td>Read</td><td>Write</td></tr>
<tr><td colspan="2">GetNumDivisõesNode</td></tr>
</table>

Fig. 3.12 - Classe **MalhaRegular**

As variáveis **DirX**, **DirY** e **DirZ** são ponteiros para listas e o acesso a cada uma delas é feito pelo método **GetPtrEixo**(n), que retorna o ponteiro para a lista especificada (n=1 para a direção x, n=2 para y e n=3 para z), fazendo a ligação da classe **MalhaRegular** com os métodos de **List**. Alguns desses métodos foram declarados na própria **MalhaRegular**, onde a direção é passada como parâmetro. Por exemplo,

```
Append(n_eixo,dimensao,num_divisoes);
```

tem o mesmo efeito de

```
GetPtrEixo(n_eixo)->Append(new
                    NodeMalhaRegular(dimensao,num_divisoes));
```

Assim, os métodos **Append** e **Count** de **List**, e **GetDimensão** e **GetNumDivisões** de **NodeMalhaRegular**, são redeclarados em **MalhaRegular**, onde a direção do eixo também é fornecida como parâmetro.

Condições De Contorno

Um programa de cálculo de campos eletromagnéticos deve suportar a definição de vários tipos de condições de contorno diferentes. Desse modo, a lista de condições de contorno impostas a um problema deve ter a capacidade de gerenciar seus diversos tipos.

Aproveitando a herança e o polimorfismo oferecidos pela Programação Orientada por Objetos, uma estrutura de nós foi criada para que uma lista pudesse gerenciar seus vários tipos. Um nó base contendo os dados básicos para qualquer tipo de condição de contorno foi criado, para que dele fossem derivados os diversos nós. Esta classe, chamada **NodeCondContorno**, derivada de **ListNode**, contém a informação da região onde é imposta a condição de contorno (Fig. 3.13).

Também de acordo com a estrutura primária, **NodeCondContorno** armazena os nós da malha, inicial e final, nas três direções do espaço (vide tabela da seção anterior), guardando também o identificador do tipo da condição de contorno, para o caso das classes derivadas de **NodeCondContorno**.

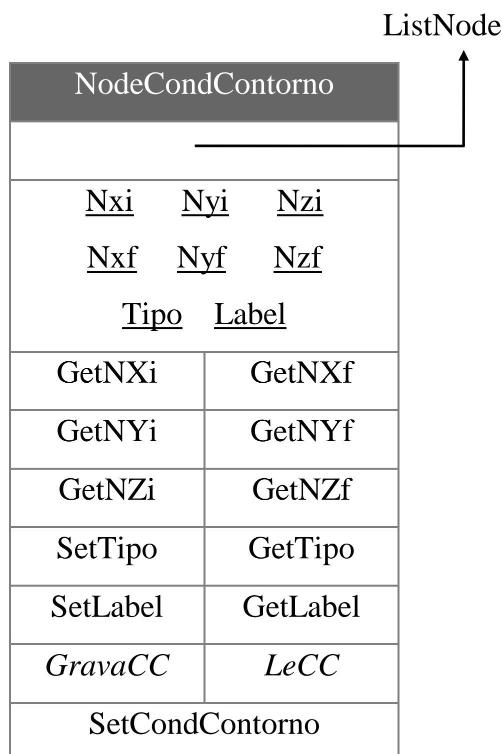


Fig. 3.13 - Classe **NodeCondContorno**

A partir desta classe qualquer nó da lista pode ser criado, bastando ao programador criar uma classe que seja derivada de **NodeCondContorno** e contenha os dados específicos do tipo da condição de contorno desejada.

Os tipos utilizados nesta versão do programa foram mencionados anteriormente e são implementadas conforme vemos a seguir (Fig. 3.14 a Fig. 3.20).

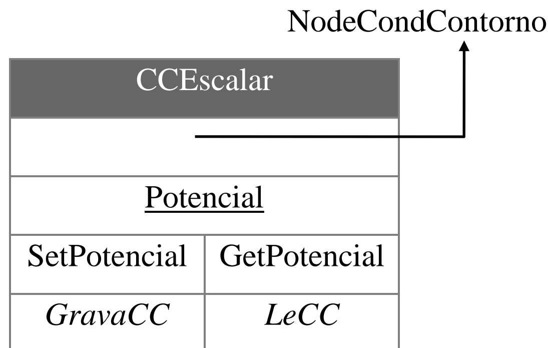


Fig. 3.14 - Classe **CCEscalar** (condição de contorno onde é imposto um potencial escalar).

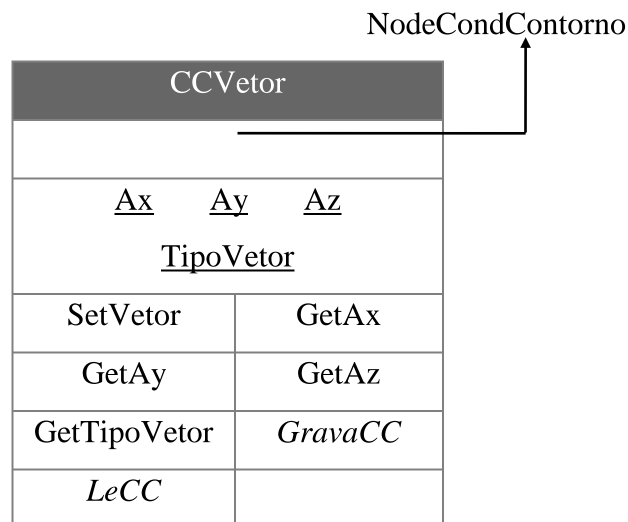


Fig. 3.15 - Classe **CCVetor** (condição de contorno onde é imposto um potencial vetor).

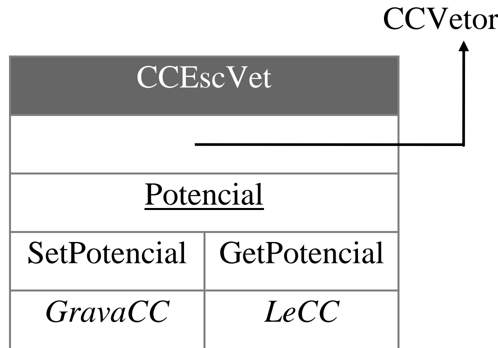


Fig. 3.16 - Classe **CCEscVet** (condição de contorno onde são impostos potencial escalar + potencial vetor).

À primeira inspeção, caberia aqui a utilização da herança múltipla, ou seja, a classe **CCEscVet** herdaria dados das classes **CCEscalar** e **CCVetor**, declarando as mesmas como virtuais (Stevens, 1991). Mas, a herança múltipla, além de não ser disponível em todas as linguagens mais comumente utilizadas (por exemplo, Object Pascal) , é uma solução que deve ser evitada quando possível, pois, envolvendo várias classes e suas ancestrais, aumenta a possibilidade de erro quando o projeto não é executado corretamente. Apesar disto, quando necessária, torna-se uma ferramenta poderosa (Meyer, 1988).

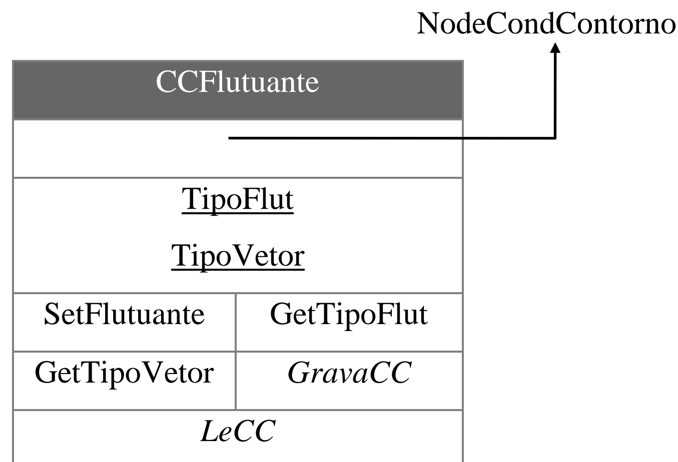


Fig. 3.17 - Classe **CCFlutuante** (condição de contorno onde é imposto um potencial flutuante).

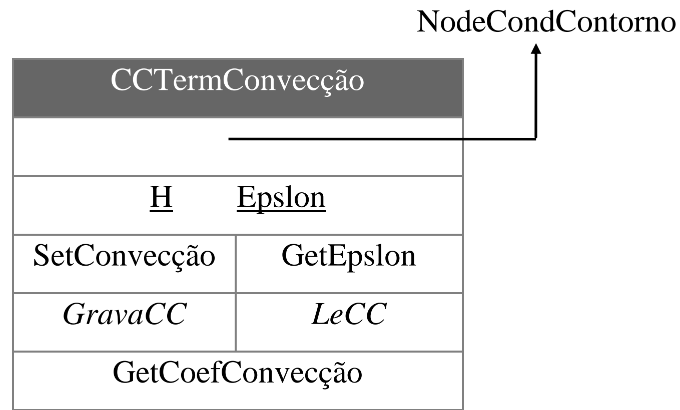


Fig. 3.18 - Classe **CCTermConvecção** (condição de contorno do tipo térmica por convecção).

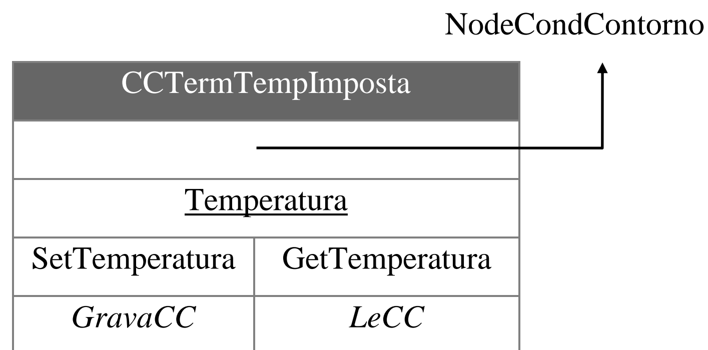


Fig. 3.19 - Classe **CCTermTempImposta** (condição de contorno térmica com temperatura imposta).

A relação desta seqüência de heranças, bem como o processo de criação de um nó a partir de outro, pode ser melhor visualizada no esquema a seguir (Fig. 3.20):

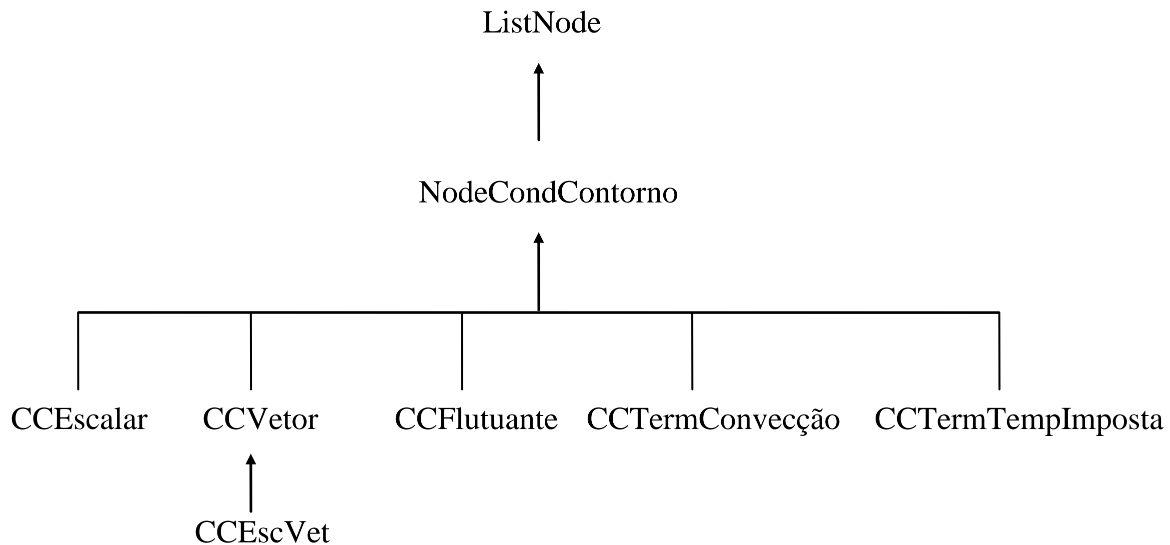


Fig. 3.20 - Esquema hierárquico dos nós das condições de contorno.

Pode-se notar que a classe **CCEscVet** não é derivada diretamente de **NodeCondContorno**, mas de **CCVetor**. Vemos que a derivação direta de **NodeCondContorno** não é necessária, podendo outros nós serem derivados de nós já existentes, desde que tenham a classe **NodeCondContorno** como uma base primária.

Pudemos ver que cada classe possui uma rotina para gravação (**GravaCC**) e outra para leitura (**LeCC**) em arquivo. Para gerenciar esta gravação/leitura, derivamos uma classe de **List**, chamada **ListCondContorno** (Fig. 3.21), e redefinimos seus métodos **Read** e **Write**.

Aqui, nos beneficiamos do polimorfismo da OOP, pois os métodos **Read** e **Write** enviam as mesmas mensagens, **LeCC** e **GravaCC**, respectivamente, para todos os nós e estes se comportam cada um conforme a sua própria característica.

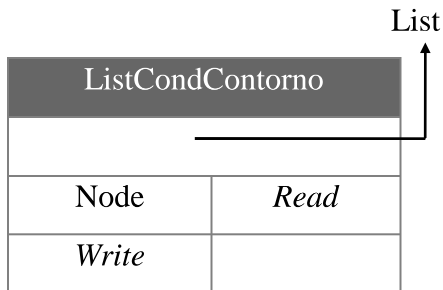


Fig. 3.21 - Classe **ListCondContorno**

Fontes De Corrente

Quando consideramos as fontes de corrente quanto à sua geometria, abrimos um leque de opções onde inúmeros tipos de fontes podem ser concebidos. Mesmo sendo tipos de fontes diferentes, cada uma possuindo a sua forma específica, todas estão incluídas em um mesmo grupo de fontes de corrente. Assim, caímos no mesmo caso anterior, onde a lista de fontes de corrente deve armazenar nós diferentes, beneficiando-se, também, do seu tratamento polimórfico. Como em Condições de Contorno, criou-se um nó básico **NodeCorrente** (Fig. 3.22), de onde são derivadas todas as geometrias das fontes de corrente.

Qualquer nó da lista pode ser criado, bastando ao programador criar uma classe que contenha os dados específicos de sua fonte de corrente, e derivá-la de **NodeCorrente**, que fornece o valor da corrente imposta, o identificador do tipo da fonte e um label que dá um nome à fonte.

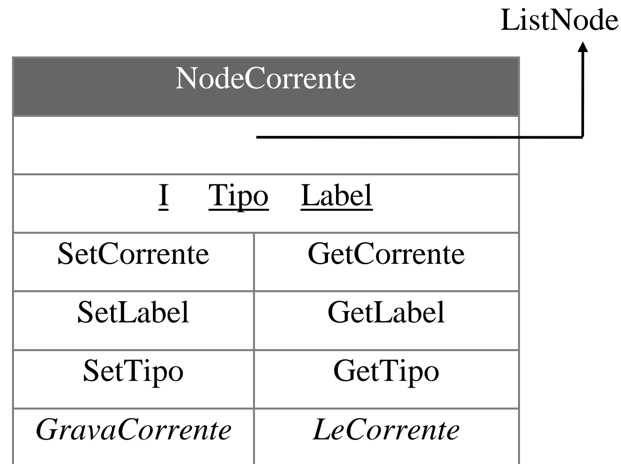


Fig. 3.22 - Classe **NodeCorrente**

Na montagem das geometrias, as duas representações descritas no item 3.3 (Fontes de Corrente) foram aqui montadas da seguinte maneira. Uma classe **I_Bloco**, derivada de **NodeCorrente**, contém todos os dados necessários para a definição de uma fonte de corrente em bloco de camadas de elementos: o número das camadas inicial e final em cada direção do espaço e o identificador da direção da corrente (x+, x-, y+, y-, z+ ou z-). Outra classe, **Fio**, também derivada de **NodeCorrente**, define um fio reto (fonte linear) no espaço e, a partir desta classe, constroem-se as classes **Plano** (fonte superficial) e **Tijolo** (fonte volumétrica). Cada classe contém os dados necessários para a definição de sua estrutura, valendo a pena ressaltar que, quando uma classe é derivada de outra, herda todos os seus dados. Acrescentando os dados do centro de uma geometria curva, damos origem às classes **Arco**, **Casca** e **Cilindro**, derivadas, respectivamente, de **Fio**, **Plano** e **Tijolo**. Nas figuras seguintes (Fig. 3.23 a Fig. 3.29) vemos cada estrutura das classes e na Fig. 3.30, o esquema completo.

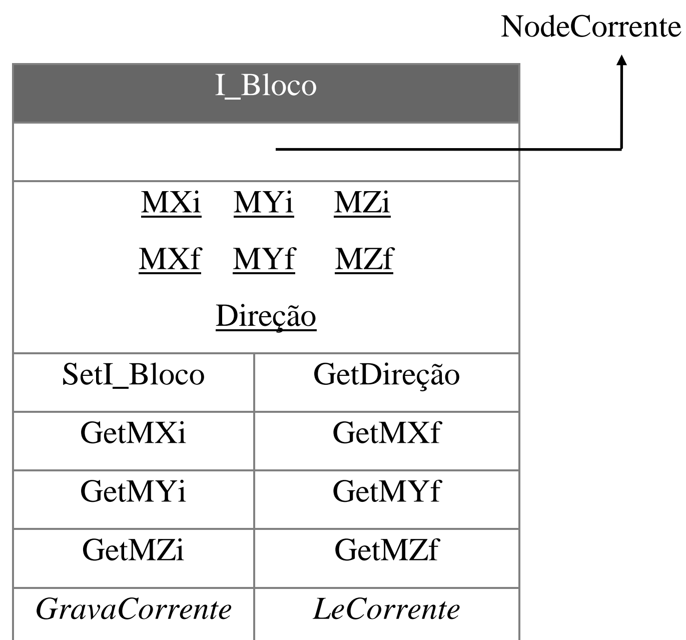


Fig. 3.23 - Classe **I_Bloco**

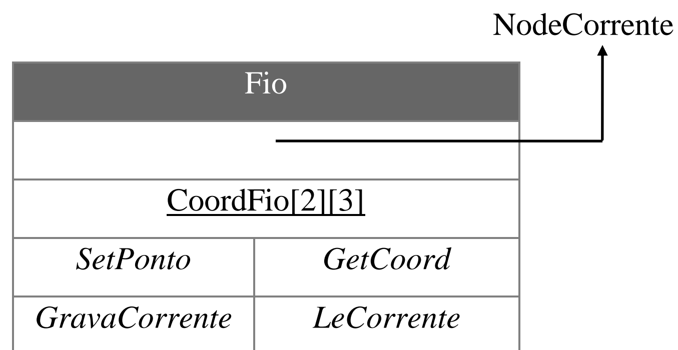


Fig. 3.24 - Classe **Fio**

Plano	
<u>CoordPlano[2][3]</u>	
SetPonto	*GetCoord*
PontosOK	*CalculaPontos*
GravaCorrente	*LeCorrente*

Fig. 3.25 - Classe **Plano**

Tijolo	
<u>CoordTijolo[4][3]</u>	
SetPonto	*GetCoord*
PontosOK	*CalculaPontos*
GravaCorrente	*LeCorrente*

Fig. 3.26 - Classe **Tijolo**

Arco	
<u>Centro[3]</u>	
GetCentro	SetCentro
GravaCorrente	*LeCorrente*
PontosOK	

Fig. 3.27 - Classe **Arco**

Casca	
Centro[3]	
GetCentro	SetCentro
GravaCorrente	*LeCorrente*
PontosOK	

Fig. 3.28 - Classe **Casca**

Tijolo

Cilindro	
Centro[3]	
GetCentro	SetCentro
GravaCorrente	*LeCorrente*
PontosOK	*CalculaPontos*

Fig. 3.29 - Classe **Cilindro**

Em algumas classes, temos os métodos **PontosOK** e **CalculaPontos**. O primeiro serve para verificar se os pontos fornecidos são coerentes com a figura que representa a fonte de corrente.

Para especificar uma figura, nem sempre é necessário fornecer todos os pontos da mesma. Assim, com apenas alguns pontos, o método **CalculaPontos** determina quais os valores dos pontos restantes. Uma relação dos pontos das figuras aqui representadas se encontra no Apêndice A.

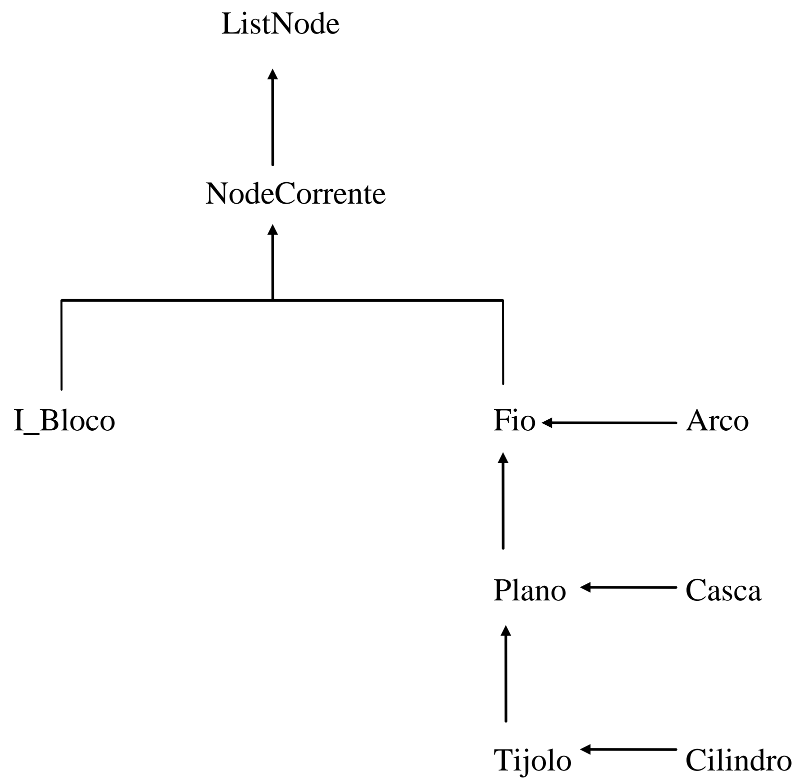


Fig. 3.30 - Esquema hierárquico dos nós das fontes de corrente

De modo semelhante à lista de condições de contorno, foi necessário redefinir os métodos **Read** e **Write** de **List**, criando uma classe derivada **ListCorrente** (Fig. 3.31). A chamada dos métodos **LeCorrente** e **GravaCorrente**, dos nós da lista, através de **Read** e **Write**, também funciona de modo idêntico à lista de condições de contorno.

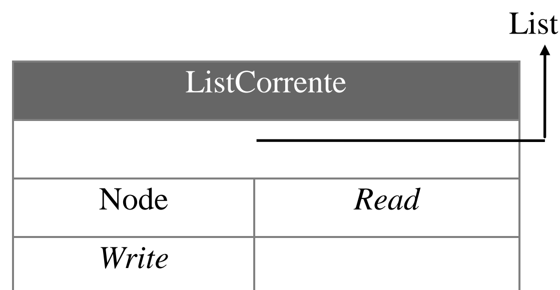


Fig. 3.31 - Classe **ListCorrente**

Materiais

Para a criação da lista de materiais, escolheu-se fazer um nó contendo todas as características físicas utilizadas em materiais nesta versão do programa, formando um material genérico. O usuário, ao fornecer os dados do material que está criando, deixa vazios os campos das grandezas físicas que não são relevantes. O próprio programa deve interpretar a existência ou não de tal característica.

Esta classe, chamada **NodeMaterial**, está representada na Fig. 3.32, onde podemos ver os dados existentes e os seus métodos de manipulação. Na Tabela 5, temos o significado físico de cada uma das variáveis.

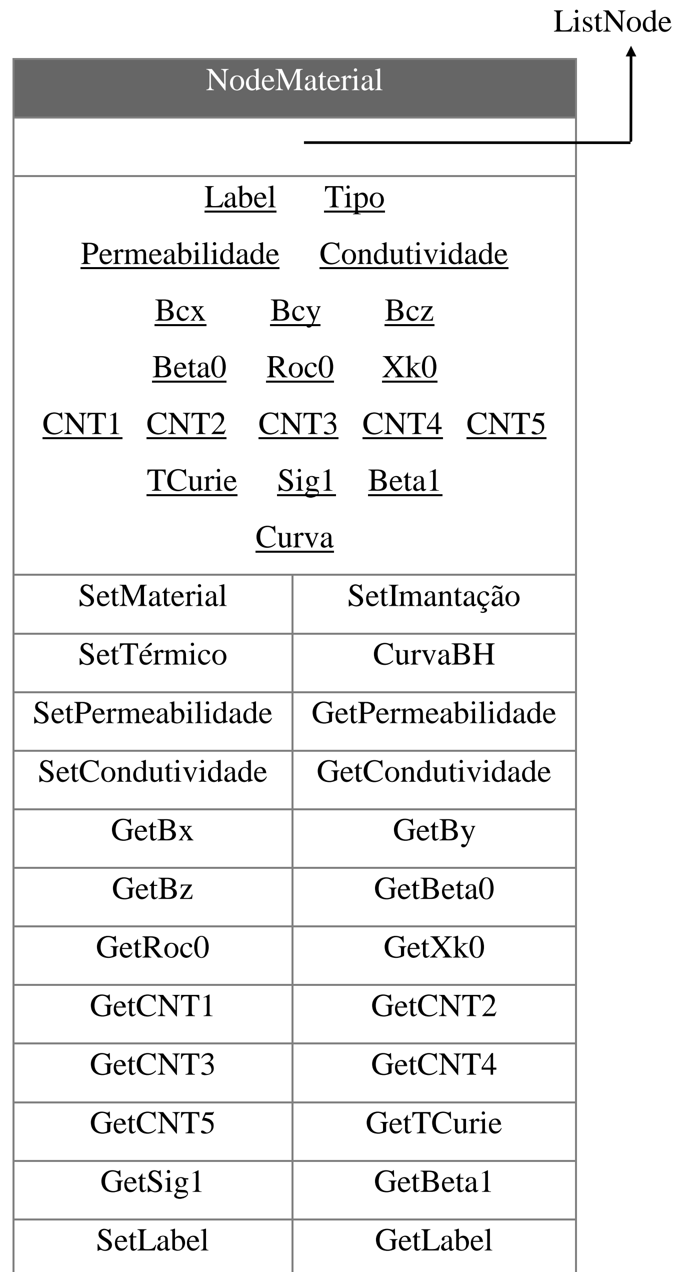


Fig. 3.32 - Classe **NodeMaterial**

Variável	Descrição
Permeabilidade	Permeabilidade elétrica
Condutividade	Condutividade elétrica
Bcx, Bcy e Bcz	Imantação (nas 3 direções do espaço)
Beta0	Fator de variação da condutividade com a temperatura
Roc0	Capacidade térmica
Xk0	Condutividade térmica
TCurie	Temperatura de Curie
Sig1	Condutividade elétrica após Curie
Beta1	Fator de variação da condutividade elétrica após Curie
CNT1	Denominador Alfa
CNT2	Mult. exp. em Roc0
CNT3	Denominador em Roc0
CNT4	Mult. exp. em k
CNT5	Denominador em k
Curva	Lista de pontos da curva BxH

Tabela 5: Significado das variáveis de **NodeMaterial**.

Para o armazenamento dos pontos das curvas BxH, foi criada uma classe de lista para o armazenamento de nós contendo os dados de um ponto da curva. A variável **Curva**, existente em **NodeMaterial**, é um ponteiro para esta lista. Se um certo material possuir uma curva BxH como característica própria, os dados dos pontos da curva são adicionados nesta lista, do mesmo modo que é feito em qualquer outra lista, através da classe **NodeCurvaBH** (Fig. 3.33). Temos, assim, uma estrutura de lista, **ListCurvaBH** (Fig. 3.34), declarada dentro de um nó de lista de materiais (Fig. 3.32). A definição tanto da lista quanto de seus nós, através da classe **NodeCurvaBH**, é apresentada abaixo:

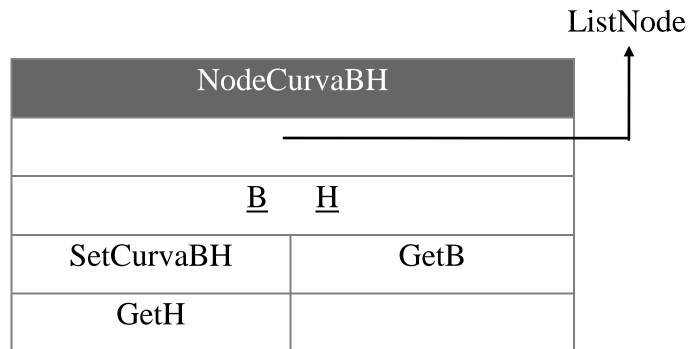


Fig. 3.33 - Classe **NodeCurvaBH**

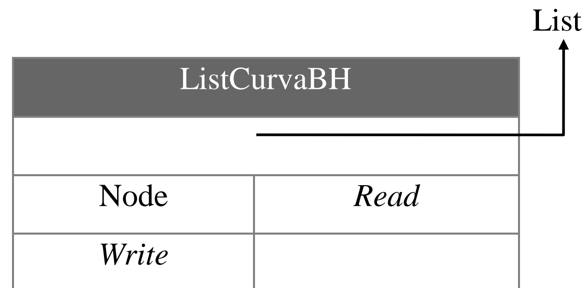


Fig. 3.34 - Classe **ListCurvaBH**

A redefinição dos métodos de gravação e leitura em arquivo também foi feita para a lista de materiais (**ListMaterial**), com a única diferença de que, enquanto os outros dados do programa são armazenados em um formato particular, os materiais são armazenados no formato neutro apresentado no Capítulo 2. Na Fig. 3.35, temos a definição da classe **ListMaterial**.

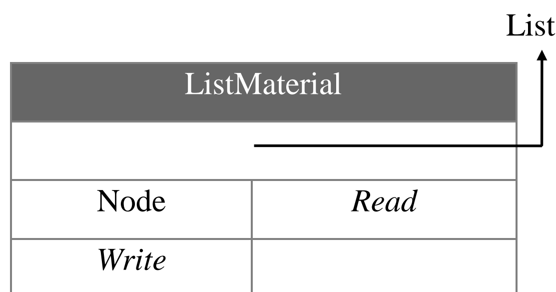


Fig. 3.35 - Classe **ListMaterial**

Base Da Estrutura De Dados

Cada uma das classes de listas apresentadas deve ter um objeto declarado para que sejam usadas durante o processamento. Uma classe especial, que funciona como base para toda a estrutura, contém a declaração de todas as listas, juntamente com outros dados gerais do problema, como o número de camadas de elementos, o material e o potencial default que preenchem o domínio. Nesta classe, chamada **Preprocessador** (Fig. 3.36), estão declarados os métodos de gravação e leitura gerais, ou seja, que gerenciam os métodos de gravação e leitura de cada uma das listas de dados (**LeDados** e **GravaDados**). Os dados de materiais são gravados/recuperados através dos métodos **GeraArqMat** e **LeArqMat**, que, conforme mencionado, utiliza o formato neutro como padrão, chamando os métodos de **ListMaterial**.

Há também o método **GeraArquivoNeutro**, para gerar os arquivos da Base de Dados em Formato Neutro para a intercomunicação dos dados entre programas, apresentado no Capítulo 2. Este método transforma os dados da estrutura apresentada, gerando a estrutura de dados para a formação dos arquivos da Base de Dados.

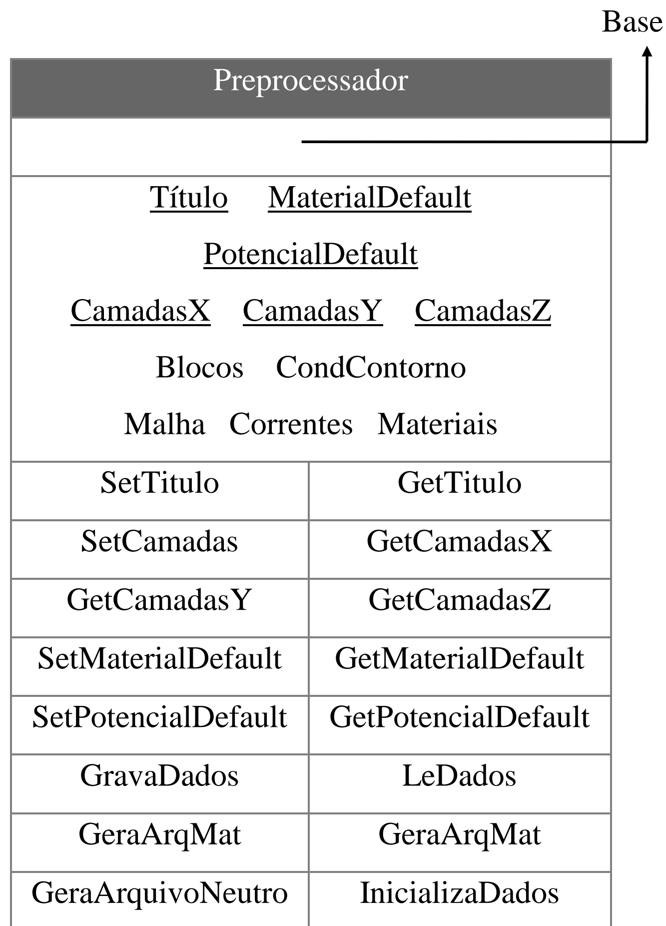


Fig. 3.36 - Classe **Preprocessador**

Com esta classe, a inicialização de toda a estrutura de dados fica restrita à declaração de um objeto do tipo **Preprocessador**. Feito isto, todas as listas e os demais dados estarão prontos para serem utilizados.

No Apêndice C, encontramos toda a implementação desta estrutura, incluíndo o exemplo de um programa teste, onde pode-se ver como ela é utilizada na prática.

3.5 - Vantagens e Desvantagens da Estrutura Orientada por Objetos

Comparando a estrutura primária, onde foi utilizada a programação convencional orientada por procedimentos, e a estrutura orientada por objetos, podemos fazer algumas observações:

- com a criação dos tipos abstratos de dados (classes), houve um sensível ganho na organização de toda a estrutura de dados, onde a relação mais rígida entre os dados e as funções que os manipulam (métodos) trouxe uma maior clareza ao programa;

- o encapsulamento dos dados traz uma maior proteção e confiabilidade ao programa, pois obriga o programador a acessar os dados através de seus métodos, dificultando a ocorrência de enganos durante o processo de programação;

- a existência de herança de classes fez com que código de programação fosse economizado, pois, construíndo-se uma classe básica genérica, como em Condições de Contorno e Fontes de Corrente, com apenas o acréscimo de dados particulares, forma-se uma nova classe/nó;

- o tratamento polimórfico das classes torna-se um ponto chave nesta estrutura, pois faz com que diferentes objetos reajam diferentemente à mesma mensagem, facilitando a criação e execução de métodos que são comuns a vários objetos de um mesmo tipo, como, por exemplo, o método de gravação em arquivo dos dados das diversas fontes de corrente;

- devido a um maior grau de modularização, a manutenção da estrutura proposta é feita com mais facilidade, pois os tipos diferentes de dados não estão fortemente relacionados entre si;

- a atualização da estrutura proposta é feita de maneira natural, onde novas classes podem ser criadas, aproveitando-se as classes já existentes. Qualquer tipo de condição de contorno ou fonte de corrente pode ser acrescentado à estrutura, bastando para isso que a classe herde as características da classe base do seu tipo (**NodeCondContorno** ou **NodeCorrente**) ou uma de suas derivadas;

- pelo fato de a Programação Orientada por Objetos não ser ainda muito utilizada no meio de engenharia, o universo de pessoas capacitadas a trabalhar com esta estrutura sem necessitar de um estudo prévio das técnicas da OOP é menor. Em contrapartida, devido à sua melhor organização dos dados, se o usuário já possuir conhecimentos de OOP, o tempo de familiarização com a estrutura se reduzirá bastante, em comparação com a estrutura orientada por procedimentos.

À luz destas observações, vemos como a Programação Orientada por Objetos traz uma melhora significativa na qualidade do software de engenharia, dando a ele uma maior organização, e, o que é mais importante, reduzindo sobremaneira o esforço de programação, visto que o reaproveitamento do código gerado inicialmente é efetuado de modo eficaz.

Capítulo 4 - Conclusão

Vimos, no presente trabalho, uma reestruturação da forma de armazenamento de dados do pré-processador de um sistema computacional de cálculo de campos eletromagnéticos baseado no Método de Elementos Finitos.

A estrutura do sistema foi dividida em uma parte externa, representada por uma base de dados para o armazenamento de todas as informações manipuladas pelo programa, e outra parte interna, representada pela organização da estrutura de dados utilizada pelo programa.

Na primeira parte, foi apresentada a proposta de uma Base de Dados em Formato Neutro para a intercomunicação entre estes programas, buscando uma padronização para os diversos programas desenvolvidos pelos grupos de pesquisa brasileiros.

Buscamos montar uma estrutura flexível, que abrangesse o maior número de casos possível, tanto em geometrias bidimensionais, quanto nas tridimensionais, cientes de que, para uma completa padronização, é necessária uma participação de toda a comunidade interessada. Esta padronização é impossível de ser alcançada sem a contribuição dos grupos e, por isso, não consideramos encerrado o trabalho mas, pelo contrário, apenas abertas as portas para uma discussão que a cada dia se torna mais necessária.

Com a popularização do uso da *Internet*, a troca de informações tem-se tornado cada dia mais intensa, onde barreiras de distância e línguas têm sido rompidas. Espera-se, com isso, um aumento das colaborações entre grupos internacionais e, com formatos de padronização a níveis locais, a globalização de uma padronização é facilitada. Acreditamos ser esta Base de Dados em Formato Neutro um primeiro passo para tal padronização a nível nacional.

Na segunda parte, apresentamos uma estrutura de dados para o pré-processador de tais programas, utilizando técnicas de Programação Orientada por Objetos, visando a melhora da organização da estrutura.

A Programação Orientada por Objetos demonstrou ser de grande utilidade no desenvolvimento do software de engenharia, fornecendo características imprescindíveis a um programa de qualidade. Além da melhora na organização dos dados, dois fatores

contribuíram muito para o bom conceito desta nova teoria. Um é a reutilização do código, ou seja, a partir de um código já existente, cria-se outro com uma excelente taxa de aproveitamento, fato que não tem acontecido no software de engenharia orientado por procedimentos. Quando se quer modificar um programa, geralmente ele é refeito, em vez de ter seu código acrescentado. Outro é a facilidade de se dar manutenção no código, devido ao alto grau de organização dos dados.

Na estrutura apresentada neste trabalho, há um ponto importante a ser ressaltado: por ela ter sido baseada em uma estrutura de um programa orientado por procedimentos, uma certa rigidez foi introduzida, pois tentou-se reaproveitar um projeto já existente. Uma estrutura baseada na estrutura da Base de Dados em Formato Neutro seria, talvez, um bom começo.

Apresentamos a seguir algumas propostas de continuação deste trabalho, sendo que algumas já se encontram em andamento:

Para a Base de Dados em Formato Neutro:

- migração da representação da geometria, atualmente Brep plana, para uma Brep curva, o que possibilitaria uma maior gama de tipos de geometrias sendo representadas;

- estudo e aplicação de conceitos de parametrização de geometria para o uso da Base de Dados com técnicas de otimização;

- acréscimo da representação CSG de geometrias, dando uma maior flexibilidade às construções geométricas representadas pela Base.

Para Sistema Computacional Orientado por Objetos:

- extensão da representação geométrica para uma forma de representação mais flexível;

- aplicação de outros métodos de geração da malha de elementos finitos;

- extensão da representação geométrica para a CSG, possibilitando a construção de uma interface mais "amigável" com o usuário;

- integração com um Processador e um Pós-processador orientados por objetos.

Referências Bibliográficas

- Ancelle, B.; Lecomte, T.; Manuel, P. & Vérité, J. C. (1988) “A Proposed Standard of Exchange File for Electromagnetic Field Softwares”, IEEE Transactions on Magnectics, vol. 24, no. 1, pp. 346-349.

- Booch, G. (1994) *Object-Oriented Analysis and Design with Applications*, 2nd. Edition, The Benjamin/Cummings Publishing Company Inc.

- Computer Graphics Technology Group - TecGraf (1992) “Neutral File Format”, Publicação Interna do ICAD/PUC-RJ, Versão: December 11, 1992.

- Ellis, M. A. & Stroustrup, B. (1993) *C++, Manual de Referência Comentado*, Editora Campus, Rio de Janeiro, p. 546.

- Foley, J. D.; Van Dam, A.; Feiner, S. K. & Hughes, J. F. (1990) *Computer Graphics - Principles and Practice*, Addison - Wesley Publishing Company, 2nd Edition, pp. 471-562.

- Forde, W. R.; Foschi, R. O. & Steimer, S. F. (1990) “Object-Oriented Finite Element Analysis”, Computers & Structures, Vol. 34, No. 3, pp. 355-374.

- George, P. L. (1991) *Automatic Mesh Generation: Application to Finite Element Methods*, John Wiley & Sons.

- Grupo de Eng. de Estruturas/UFMG (1992) “Manual de Entrada de Dados do Procedimento Preiso Utilizando Elementos Iso-paramétricos Triangulares para o Estudo Plano de Tensões e Deformações de 3 a 6 Nós Incluindo Carregamento de Domínio do Tipo Centrífugo”, Publicação Interna do DEES/UFMG, Março.

- ISO (1991) *Express Language Reference Manual*, ISO TC184//SC4/WG5-n14, Abril.

- Kroszynski, U. I.; Palstroem, B. & Trostmann, E. (1989) “Geometric Data Transfer Between CAD Systems: Solid Models”, IEEE Computer Graphics and Applications, September, pp. 57-71.

- Lowther, D.A. (1988) “Languages, Databases and Software Structures for Eletromagnetic CAD Systems”, IEEE Transactions on Magnetics, vol. 24, no. 1, January, pp. 338-341.

- Mesquita, R. C. (1990) *Cálculo de Campos Eletromagnéticos Tridimensionais Utilizando Elementos Finitos: Magnetostática, Quase-Estática e Aquecimento Indutivo*, Univ. Federal de Santa Catarina, Dezembro.

- Meyer, B. (1988) *Object-Oriented Software Construction*, Prentice Hall, London.

- Mortenson, M. E. (1985) *Geometric Modeling*, John Wiley & Sons, New York, pp. 372-485.

- Pèlerin, Y. D.; Zimmermann, T. & Bomme, P. (1992) “Object-Oriented Finite Element Programming: II A Prototype Program in Smaltalk”, Comput. Methods in Appl. and Mech. Engrg., vol. 98, no. 3, pp. 361-397.

- Rocha, L. F. N. & Mesquita, R. C. (1994) “Uma Base de Dados de Formato Neutro para Intercomunicação de Dados em Programas de Cálculo de Campos Eletromagnéticos”, XV Congresso Íbero-Latino Americano de Métodos Computacionais em Engenharia (CILAMCE), vol II, Novembro, pp. 986-995.

- Sadi, R. A.; Colyer, B., Emson, C. R. I., Simkin, J. & Maanen, J. V. (1994) “A 2D and Axisymmetric Finite Element Environment Based upon STEP-type database”, IEEE Transactions on Magnetics, vol. 30, n. 5, September, pp. 3622-3624.

- Santana, O.; Lafrate, J. P. & Coulomb, J. L. (1988), “A Data Structure for a Finite Element Package”, IEEE Transactions on Magnetics, vol. 24, no. 1, January, pp. 350-353.

- Sharafuddin, A. J., Ida, N. & Grover, J. E. (1993) Object Oriented Design and Matrix Computation, 9th Conference on the Computation of Electromagnetic Fields (COMPUMAG), Miami, USA, pp. 234-235.

- Silva, E. J.; Mesquita, R. C.; Saldanha, R. R. & Palmeira, P. F. M. (1994) "An Object-Oriented Finite Element Program for Electromagnetic Field Computation", IEEE Transactions on Magnetics, Vol. 30, No. 5, September, pp. 3618-3621.

- Stevens, A. (1991) *C++, Aprenda Você Mesmo*, LTC - Livros Técnicos e Científicos Editora, Rio de Janeiro, p. 209.

- Webb, J. P.; Lowther, D. A. & Silvester, P. P. (1985) "A Finite Element Database for Eletromagnetic Field Computation", IEEE Transactions on Magnetics, vol. Mag-21, no. 6, November, pp. 2503-2506.

- Weimer, R. S.; & Pinson, L. J.; (1988) *An Introduction to Object-Oriented Programing and C++*, Addison-Wesley, New York.

- Wilson, P. R.; Faux, I. D.; Ostrowski, M. C. & Pasquill, K. G. (1985) "Interfaces for Data Transfer Between Solid Modeling Systems", IEEE Computer Graphics and Applications, January, pp. 41-51.

- Wirth, N. (1989) *Algoritmos e Estruturas de Dados*, Prentice-Hall, Rio de Janeiro, p. 255.

- Zimmermann, T.; Pèlerin, Y. D. & Bomme, P. (1992) "Object-Oriented Finite Element Programming: I Governing Principles", Comput. Methods in Appl. and Mech. Engrg., vol. 98, no. 2, pp. 391-303.

- Ziviani, N. (1993) *Projetos de Algoritmos com Implementações em Pascal e C*, Segunda Edição, Editora Pioneira, São Paulo, p. 267.

APÊNDICES

Apêndice A

Geometrias das Fontes de Corrente

Apresentamos, neste apêndice, as figuras geométricas que foram utilizadas na criação das fontes de corrente nesta versão do programa, representadas pelas classes das figuras 3.24 a 3.29.

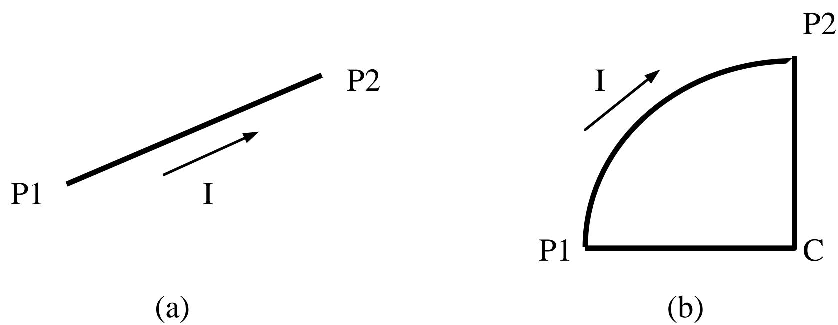


Fig. A.1 - Figuras unidimensionais: (a) Fio e (b) Arco.

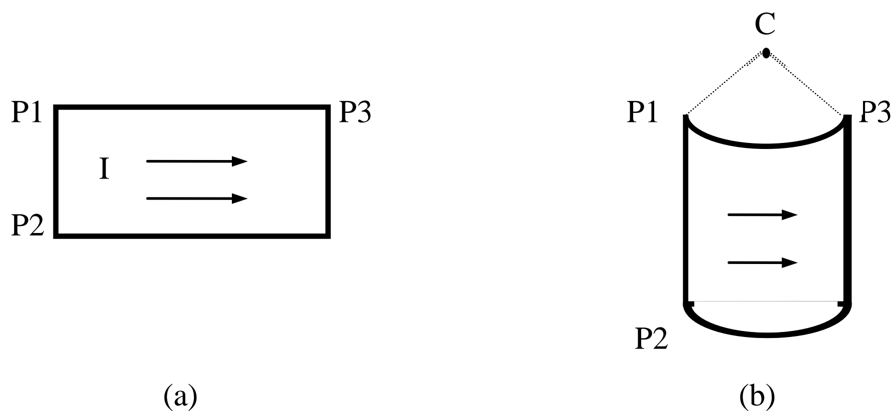
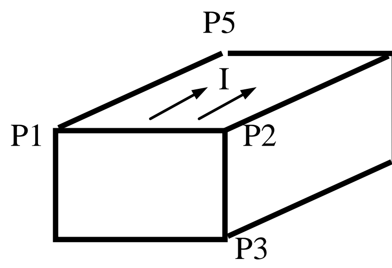
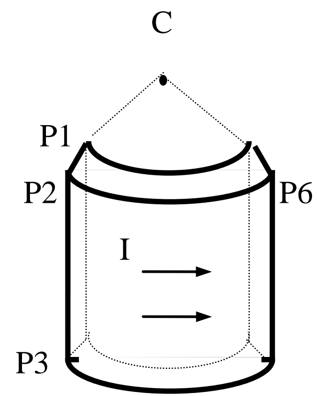


Fig. A.2 - Figuras bidimensionais: (a) Plano e (b) Casca.



(a)



(b)

Fig. A.3 - Figuras Tridimensionais: (a) Tijolo e (b) Cilindro.

Apêndice B

Estrutura Completa da Base de Dados em Formato Neutro

A seguir, apresentamos a definição completa dos arquivos que compõem a Base de Dados em Formato Neutro, em sua versão 3.0, para uma referência rápida.

CENTRO DE PESQUISA E DESENVOLVIMENTO EM ENGENHARIA ELÉTRICA

GOPAC
GRUPO DE OTIMIZAÇÃO E PROJETO ASSISTIDO POR COMPUTADOR

DEFINIÇÃO DO ARQUIVO NEUTRO PARA INTERCOMUNICAÇÃO DE DADOS EM PROGRAMAS DE CÁLCULO DE CAMPOS ELETROMAGNÉTICOS

VERSÃO 3.0

ARQUIVO NEUTRO - INFORMAÇÕES GERAIS

1) Objetivo

O arquivo neutro para a intercomunicação de dados em (e entre) programas de cálculo de campos eletromagnéticos visa padronizar os formatos de transferência de dados, melhorando a compatibilidade e a portabilidade de tais dados/arquivos, e organizar tais dados de modo a facilitar ao usuário sua geração e manipulação.

2) Características gerais

a) a base de dados é dividida em três arquivos: **Geometria**, que contém os dados relativos à geometria do problema; **Malhas**, que contém os dados da geração das malhas de elementos finitos (volumétrica ou superficial), detalhes do cálculo e condições de contorno; e **Materiais**, que contém o banco de dados das propriedades físicas dos materiais dos problemas.

b) a base de dados deve conter todos os dados necessários do pré ao pós processamento, qualquer que seja o programa que a esteja utilizando. A falta de qualquer dado imprescindível ao cálculo deve ocasionar a interrupção do mesmo;

c) o formato deve ser de fácil leitura às linguagens de programação existentes no mercado, como o C++ e Fortran. Deve também ser legível ao usuário, permitindo-o reconhecer os campos através das palavras chaves, sem a necessidade de um manual;

d) a estrutura dos arquivos deve ser de fácil entendimento, permitindo ao usuário familiarizado com a Análise de Elementos Finitos analisar os dados com a simples inspeção dos arquivos.

3) Características dos arquivos

- os arquivos estão em formato ASCII, de modo seqüencial;

- não é necessário nenhum alinhamento de colunas, pois os arquivos são orientados de modo linear;

- a base de dados é seccionada em blocos de dados, cada uma delas começando com um cabeçalho (label) precedido por um asterisco (*) que permite ao programa selecionar o bloco que importa ler, desprezando os blocos que não são necessários;

- a convenção de tipos de variáveis adotada é:

[v]	variável inteira;
<v>	variável real;
‘v(n)’	string de, no máximo, **n** caracteres.

GEOMETRIA

DADOS GERAIS

*REMARK
O label REMARK pode aparecer em qualquer ponto do arquivo (entre seções) e seu conteúdo é ignorado. É usado para documentar o arquivo.

*HEADER.FILE
Nome do arquivo

*HEADER.AUTHOR
Nome do autor ou do programa que gerou o arquivo.

*HEADER.DATE
Data da criação do arquivo.

*HEADER.VERSION
Versão do arquivo de formato neutro

*HEADER.TITLE
'File_title(80)'

*HEADER.ANALYSIS
'Analysis_type(20)'

*HEADER.UNITS
'length_label(5)' 'mass_label(5)' 'time_label(5)' 'angle_label(5)' 'temperature_label(5)'

OBS: Os labels de unidades utilizados são (o primeiro de cada lista é o default):
comprimento (length): M, CM, MM, MC, INCH, FEET.
massa (mass): KG, G, LB.
tempo (time): SEC, MIN, HR.
ângulo (angle): RAD, DEG, CYC.
temperatura (temperature): DEGC, DEGF, DEGK.

*HEADER.DIM
[dimension_type]

*END
Este label determina o fim do arquivo. Toda informação que vem após o label é ignorada.

DADOS DA GEOMETRIA

Seguem-se as informações necessárias para compor a geometria, bem como algumas características relacionadas a esta. A geometria é baseada em listas de listas, ou seja, a partir dos pontos existentes, a linha é uma lista destes pontos, o contorno é uma lista de linhas, e assim por diante, até o objeto. Cada estrutura é definida por um identificador e pode haver um label associado a qualquer uma delas. Atributos físicos são associados às estruturas através de labels, como veremos posteriormente. Uma macro-estrutura representa uma estrutura que não segue esta organização e serve para introduzir geometrias do tipo cilindro, arco de cilindro, paralelepípedos etc.

*POINT
[#_of_points]

*POINT.COORD
[point_id] <x> <y> ou <x> <y> <z>
...

*POINT.LABEL
[#_of_point_labels]
[point_id] ‘point_label(20)’
...

*LINE
[#_of_lines]

*LINE.LABEL
[#_of_line_labels]
[line_id] ‘line_label(20)’
...

*LINE.SEGMENT.CONTENTS
[line_id] [#_of_contents] [point_1_id] [point_2_id] ... [point_#_of_contents_id]
...

*LINE.ARC.CONTENTS
[line_id] [point_1_id] [point_med_id] [point_2_id]
...

*CONTOUR
[#_of_contours]

*CONTOUR.LABEL
[#_of_contour_labels]
[contour_id] ‘contour_label(20)’

...

*CONTOUR.CONTENTS
[contour_id] [#_of_contents] [line_1_id] [line_2_id] ... [line_#_of_contents_id]
...

*FACE
[#_of_faces]

*FACE.LABEL
[#_of_face_labels]
[face_id] 'face_label(20)'
...

*FACE.PLANE.CONTENTS
[face_id] [#_of_contents] [contour_1_id] [contour_2_id] ... [contour_#_of_contents_id]
...

*ENVELOPE
[#_of_envelopes]

*ENVELOPE.LABEL
[#_of_envelope_labels]
[envelope_id] 'envelope_label(20)'
...

*ENVELOPE.CONTENTS
[envelope_id] [#_of_contents] [face_1_id] [face_2_id] ... [face_#_of_contents_id]
...

*VOLUME
[#_of_volumes]

*VOLUME.LABEL
[#_of_volume_labels]
[volume_id] 'volume_label(20)'
...

*VOLUME.CONTENTS
[volume_id] [#_of_contents] [envel_1_id] [envel_2_id] ... [envel_#_of_contents_id]
...

*OBJECT
[#_of_objects]

*OBJECT.LABEL
[#_of_object_labels]
[object_id] ‘object_label(20)’
...

*OBJECT.CONTENTS
[object_id] [#_of_contents] [vol_1_id] [volume_2_id] ... [volume_#_of_contents_id]
...

*MACRO_STRUCT
[#_of_macro_structs]

*MACRO_STRUCT.LABEL
[#_of_macro_struct_labels]
[macro_struct_id] ‘macro_struct_label(20)’
...

*MACRO_STRUCT.TYPE
[macro_struct_id] ‘macro_struct_type(20)’
...

*MACRO_STRUCT.PARAMETERS
[macro_struct_id] [#_of_params] <param_1> ... <param_#>
...

DADOS DOS ATRIBUTOS FÍSICOS E DE GEOMETRIA

Associação de atributos físicos à geometria. Através do label da geometria, faz-se referência a dados dos arquivos Malhas e Materiais.

*ATTRIBUTES
‘label_of_entity(20)’

*ATTRIBUTES.MATERIAL
[material_id]

*ATTRIBUTES.COLOR
[color_id]

*ATTRIBUTES.BOUNDARY
[BC_id]

*ATTRIBUTES.POTENTIAL
[potential_id]

*ATTRIBUTES.CURRENT
[current_id]

*ATTRIBUTES.VOLTAGE
[voltage_id]

*ATTRIBUTES.WINDING
[winding_id]

MALHAS

DADOS GERAIS

Semelhante ao arquivo de geometria.

*REMARK
O label REMARK pode aparecer em qualquer ponto do arquivo (entre seções) e seu conteúdo é ignorado. É usado para documentar o arquivo.

*HEADER.FILE
Nome do arquivo

*HEADER.AUTHOR
Nome do autor ou do programa que gerou o arquivo.

*HEADER.DATE
Data da criação do arquivo.

*HEADER.VERSION
Versão do arquivo de formato neutro

*HEADER.TITLE
'File_title(80)'

*HEADER.ANALYSIS
'Analysis_type(20)'

*HEADER.UNITS
'length_label(5)' 'mass_label(5)' 'time_label(5)' 'angle_label(5)' 'temperature_label(5)'

OBS: Os labels de unidades utilizados são (o primeiro de cada lista é o default):
comprimento (length): M, CM, MM, MC, INCH, FEET.
massa (mass): KG, G, LB.
tempo (time): SEC, MIN, HR.
ângulo (angle): RAD, DEG, CYC.
temperatura (temperature): DEGC, DEGF, DEGK.

*HEADER.DIM
[dimension_type]

*END
Este label determina o fim do arquivo. Toda informação que vem após o label é ignorada.

DADOS DOS NÓS

Os nós são definidos por um identificador e podem ter um label associado.

*NODE
[#_of_nodes]

*NODE.LABEL
[#_of_node_labels]
[node_id] 'node_label(20)'
...

*NODE.COORD
[node_id] <x> <y> <z>
...

DADOS DE ARESTAS

As arestas são definidas através de um identificador com os números dos nós, são separadas por tipo e podem ter um label associado.

*EDGE
[#_of_edges]

*EDGE.LABEL
[#_of_edges_labels]
[edge_id] 'edge_label(20)'
...

*EDGE.NODE2
[#_of_edge2]
[edge_id] [node_1_id] [node_2_id]
...

*EDGE.NODE3
[#_of_edge3]
[edge_id] [node_1_id] [node_2_id] [node_3_id]
...

DADOS DOS ELEMENTOS

Os elementos são definidos por um identificador e separados por tipo, e podem ter um label associado. Qualquer outro tipo de elemento pode ser implementado nesta seção.

*ELEMENT
[#_of_elements]

*ELEMENT.LINE2
[#_of_line2_elements]
[element_id] [node_1_id] [node_2_id] ‘element_label(20)’
...

*ELEMENT.LINE3
[#_of_line3_elements]
[element_id] [node_1_id] [node_2_id] [node_3_id] ‘element_label(20)’
...

*ELEMENT.TRIANGLE3
[#_of_triangle3_elements]
[element_id] [node_1_id] [node_2_id] [node_3_id] ‘element_label(20)’
...

*ELEMENT.TRIANGLE6
[#_of_triangle6_elements]
[element_id] [node_1_id] [node_2_id] ... [node_6_id] ‘element_label(20)’
...

*ELEMENT.QUADRANGLE4
[#_of_quadrangle4_elements]
[element_id] [node_1_id] [node_2_id] [node_3_id] [node_4_id] ‘element_label(20)’
...

*ELEMENT.QUADRANGLE8
[#_of_quadrangle8_elements]
[element_id] [node_1_id] [node_2_id] ... [node_8_id] ‘element_label(20)’
...

*ELEMENT.BRICK8
[#_of_brick8_elements]
[element_id] [node_1_id] [node_2_id] ... [node_7_id] [node_8_id] ‘element_label(20)’
...

*ELEMENT.BRICK20
[#_of_brick20_elements]
[element_id] [node_1_id] [node_2_id] ... [node_19_id] [node_20_id] ‘element_label(20)’
...

*ELEMENT.TETRAHEDRON4

[#_of_tetrahedron4_elements]
[element_id] [node_1_id] [node_2_id] [node_3_id] [node_4_id] ‘element_label(20)’
...

*ELEMENT.TETRAHEDRON10
[#_of_tetrahedron10_elements]
[element_id] [node_1_id] ... [node_10_id] ‘element_label(20)’
...

*ELEMENT.EDGE.TRIANGLE
[#_of_triangle_edge_elements]
[element_id] [edge_1_id] [edge_2_id] [edge_3_id] ‘element_label(20)’
...

*ELEMENT.EDGE.QUADRANGLE
[#_of_quadrangle_edge_elements]
[element_id] [edge_1_id] [edge_2_id] [edge_3_id] [edge_4_id] ‘element_label(20)’
...

*ELEMENT.EDGE.TETRAHEDRON
[#_of_tetrahedron_edge_elements]
[element_id] [edge_1_id] [edge_2_id] ... [edge_6_id] ‘element_label(20)’
...

*ELEMENT.EDGE.BRICK
[#_of_brick_edge_elements]
[element_id] [edge_1_id] [edge_2_id] ... [edge_11_id] [edge_12_id] ‘element_label(20)’
...

DADOS DE ENROLAMENTOS

Os enrolamentos podem ser definidos através de um conjunto de elementos finitos ou de uma topologia específica. No primeiro caso, os labels dos elementos indicarão uma referência aos atributos de corrente. No segundo caso, o tipo do enrolamento é definido por uma referência a uma macro-estrutura (arquivo de Geometria) que conterá as referências necessárias a correntes, número de espiras etc.

*WINDING.TOPOLOGY
[#_of_windings_with_topology]
[winding_id] [macro_struct_id] ‘winding_label(20)’
...

DADOS DE CÁLCULO

Identificação de várias características e valores ligados ao cálculo do problema, como a ordem de integração, o tipo de potencial associado ao elemento e os valores de corrente, tensão e número de espiras em um enrolamento. Algumas destas características podem ser modificadas no arquivo, se o mesmo contiver a seção de Dados de Carregamento.

*INTEGRATION.ORDER
[#_of_element_integration_orders]
[integration_id] [order_r] [order_s] [order_t]
...

*POTENTIAL
[#_of_potential_types]
[potential_id] ‘potential_label(20)’
...

OBS: Como padronização dos labels utilizados para os potenciais, temos:
‘ELECTRIC_SCALAR’, ‘MAGNETIC_SCALAR’,
‘ELECTRIC_VECTOR’, ‘MAGNETIC_VECTOR’,
‘ELECTRIC_FIELD’, ‘MAGNETIC_FIELD’,
‘ELECTRIC_INDUCTION’, ‘MAGNETIC_INDUCTION’,
‘TEMPERATURE’. ‘MAGNETIC_REDUCED’

*CURRENT
[#_of_current_types]
[current_id] <J_real> <J_imag> <frequency>
...

*VOLTAGE
[#_of_voltage_types]
[voltage_id] <voltage_real> <voltage_imag> <frequency>
...

*WINDING
[#_of_winding_types]
[winding_id] [#_of_coils]
...

CONDIÇÕES DE CONTORNO

Informações sobre as condições de contorno iniciais impostas ao problema. Estas condições podem ser modificadas posteriormente no arquivo, se o mesmo contiver a seção Dados de Carregamento. A cada condição é associado um identificador e, se necessário, um label.

*BOUNDARY
[#_of_boundary_conditions]

*BOUNDARY.LABEL
[#_of_BC_labels]
[BC_id] ‘BC_label(20)’
...

*BOUNDARY.SCALAR.DIRICHLET
[#_of_Dirichlet_BC]
[BC_id] <Potential>
...

*BOUNDARY.SCALAR.NEUMANN
[#_of_Neumann_BC]
[BC_id] <Density>
...

*BOUNDARY.VECTOR.NORMAL
[#_of_Normal_BC]
[BC_id] <normal_value>
...

*BOUNDARY.VECTOR.TANGENT
[#_of_tangential_BC]
[BC_id] <A_1> <A_2>   ‘BC_vector_tangent_label(2)’
...

OBS: o label de vetor tangente define em qual plano se encontra as componentes do vetor A. As componentes desse vetor são representadas por A_1 e A_2. Os labels podem ser: ‘XY’, ‘XZ’, ‘YZ’, ‘RZ’, ‘TZ’, ‘RT’, ‘RF’, ‘TF’, onde T=teta e F=fi.
Ex: 1 A_x A_y ‘XY’
     2 A_θ A_z ‘TZ’

*BOUNDARY.SCALAR.FLOATING
[#_of_floating_BC]
[BC_id]
...

DADOS DE CARREGAMENTO

Estes dados são organizados em casos. Cada caso tem sua própria combinação de dados, que inclui a especificação dos valores de corrente e tensão em enrolamentos, e das modificações nas condições de contorno originais.

*LOAD
[#_of_load_cases]

*LOAD.CASE
[current_load_case_id] ‘case_label(20)’

*LOAD.CASE.CURRENT
[#_of_currents]
[current_id] <J_real> <J_imag> <frequency>
...

*LOAD.CASE.VOLTAGE
[#_of_voltages]
[voltage_id] <voltage_real> <voltage_imag> <frequency>
...

DADOS DOS ATRIBUTOS FÍSICOS E DE GEOMETRIA

Associação de atributos físicos aos elementos. Através do label da geometria, faz-se referência entre um elemento e a geometria a qual ele pertence.

*ATTRIBUTES
‘label_of_entity(20)’

*ATTRIBUTES.MATERIAL
[material_id]

*ATTRIBUTES.COLOR
[color_id]

*ATTRIBUTES.BOUNDARY
[BC_id]

*ATTRIBUTES.POTENTIAL
[potential_id]

*ATTRIBUTES.CURRENT
[current_id]

*ATTRIBUTES.VOLTAGE
[voltage_id]

*ATTRIBUTES.WINDING
[winding_id]

*ATTRIBUTES.INTEGRATION_ORDER

[integration_order_id]

*ATTRIBUTES.POINT
[point_id]

*ATTRIBUTES.LINE
[line_id]

*ATTRIBUTES.CONTOUR
[contour_id]

*ATTRIBUTES.FACE
[face_id]

*ATTRIBUTES.ENVELOPE
[envelope_id]

*ATTRIBUTES.VOLUME
[volume_id]

*ATTRIBUTES.OBJECT
[object_id]

*ATTRIBUTES.MACRO_STRUCT
[macro_struct_id]

DADOS DOS RESULTADOS

Dados do processamento (ou pós-processamento) dos dados do problema. O número de casos é, no mínimo, um (caso default), acrescido do número de casos definidos nos Dados de Carregamento.

*RESULTS
[#_of_cases]

*RESULTS.CASE
[current_case_id] ‘case_label(20)’

*RESULTS.NODE.POTENTIAL.REAL
[#_of_nodes]
[node_id] [#_of_variables] <var_1> <var_2> ... <var_#>
...

*RESULTS.NODE.POTENTIAL.COMPLEX

[#_of_nodes]
[node_id] [#_of_variables] <var_1_real> <var_1_imag> ... <var_#_real> <var_#_imag>
...

*RESULTS.NODE.FIELD.REAL
[#_of_nodes] ‘field_type_label(20)’
[node_id] [#_of_variables] <var_1> <var_2> ... <var_#>
...

*RESULTS.NODE.FIELD.COMPLEX
[#_of_nodes] ‘field_type_label(20)’
[node_id] [#_of_variables] <var_1_real> <var_1_imag> ... <var_#_real> <var_#_imag>
...

*RESULTS.EDGE.POTENTIAL.REAL
[#_of_edge]
[edge_id] [#_of_variables] <var_1> <var_2> ... <var_#>
...

*RESULTS.EDGE.POTENTIAL.COMPLEX
[#_of_edge]
[edge_id] [#_of_variables] <var_1_real> <var_1_imag> ... <var_#_real> <var_#_imag>
...

*RESULTS.EDGE.FIELD.REAL
[#_of_edge] ‘field_type_label(20)’
[edge_id] [#_of_variables] <var_1> <var_2> ... <var_#>
...

*RESULTS.EDGE.FIELD.COMPLEX
[#_of_edge] ‘field_type_label(20)’
[edge_id] [#_of_variables] <var_1_real> <var_1_imag> ... <var_#_real> <var_#_imag>
...

*RESULTS.NODE.TEMPERATURE
[#_of_nodes]
[node_id] <Temperature>
...

MATERIAIS

DADOS GERAIS

Semelhante ao arquivo de geometria.

*REMARK
O label REMARK pode aparecer em qualquer ponto do arquivo (entre seções) e seu conteúdo é ignorado. É usado para documentar o arquivo.

*HEADER.FILE
Nome do arquivo

*HEADER.AUTHOR
Nome do autor ou do programa que gerou o arquivo.

*HEADER.DATE
Data da criação do arquivo.

*HEADER.VERSION
Versão do arquivo de formato neutro

*HEADER.TITLE
'File_title(80)'

*HEADER.UNITS
'length_label(5)' 'mass_label(5)' 'time_label(5)' 'angle_label(5)' 'temperature_label(5)'

OBS: Os labels de unidades utilizados são (o primeiro de cada lista é o default):
comprimento (length): M, CM, MM, MC, INCH, FEET.
massa (mass): KG, G, LB.
tempo (time): SEC, MIN, HR.
ângulo (angle): RAD, DEG, CYC.
temperatura (temperature): DEGC, DEGF, DEGK.

*END
Este label determina o fim do arquivo. Toda informação que vem após o label é ignorada.

DADOS DOS MATERIAIS

Os materiais são definidos através de um identificador e podem ter um label associado. Os dados dos materiais são dispostos independentemente do tipo do material. Ao ler o arquivo de materiais, o programa deve localizar, através do identificador do material, qual(is) a(s) característica(s) existente(s) e armazená-la(s).

*MATERIAL
[#_of_materials]

*MATERIAL.LABEL
[#_of_material_labels]
[material_id] ‘material_label(20)’
...

*MATERIAL.PROPERTY.SATURATED.BH
[#_of_saturated_materials]
[material_id] [#_of_points] <m>
[point_id] <H> <B>
...
[material_id] ...
...

*MATERIAL.PROPERTY.MAGNETIC
[#_of_magnetic_materials]
[material_id] <B_x> <B_y> <B_z>
...

*MATERIAL.PROPERTY.ISOTROPIC.CONDUCTIVITY
[#_of_iso_material_conductivties]
[material_id] <s>
...

*MATERIAL.PROPERTY.ISOTROPIC.PERMEABILITY
[#_of_iso_material_permeabilities]
[material_id] <m>
...

*MATERIAL.PROPERTY.ISOTROPIC.PERMISSIVITY
[#_of_iso_material_permissivities]
[material_id] <e>
...

*MATERIAL.PROPERTY.ORTHOTROPIC.CONDUCTIVITY
[#_of_ortho_material_conductivties]
[material_id] <s_x> <s_y> <s_z>
...

*MATERIAL.PROPERTY.ORTHOTROPIC.PERMEABILITY
[#_of_ortho_material_permeabilities]

[material_id] <m_x> <m_y> <m_z>
...

*MATERIAL.PROPERTY.ORTHOTROPIC.PERMISSIVITY
[#_of_ortho_material_permissivities]
[material_id] <e_x> <e_y> <e_z>
...

*MATERIAL.PROPERTY.TERMIC.CONDUCTIVITY
[#_of_material_termic_conductivities]
[material_id] <k>
...

*MATERIAL.PROPERTY.TERMIC.HEAT
[#_of_material_termic_specific_heats]
[material_id] <C>
...

Apêndice C

Código da Estrutura de Dados Orientada por Objetos (em C++)

Apresentamos, a seguir, o código da implementação (em C++) da Estrutura de Dados Orientada por Objetos desenvolvida. O código foi desenvolvido paralelamente em compiladores C++ tanto em micros da linha PC486, quanto em ambiente Unix, visando eliminar qualquer particularidade na implementação.

Arquivos:

C.1 - Listas.h: Interface das classes do gerenciador de listas;

C.2 - Listas.cpp: Implementação dos métodos de Listas.h;

C.3 - Classes.h: Interface das classes da estrutura de dados do pré-processador;

C.4 - Classes.cpp: Implementação dos métodos de Classes.h;

C.5 - Exemplo de utilização de Classes.h.

C.1 - Arquivo Listas.h

```
/*********************************************************************
  ESTRUTURA DE DADOS ORIENTADA POR OBJETOS DO PREPROCESSADOR
    DO PROGRAMA DE CALCULO DE CAMPOS ELETROMAGNETICOS 3D.
                  PPGEE - UFMG
  Mestrando: Luis Fernando Nacif Rocha
  Orientador: Prof. Renato Cardoso Mesquita
*********************************************************************/

#if !defined(__LISTAS_H)
#define __LISTAS_H

#include <fstream.h>

#define false 0;
#define true 1;

//typedef unsigned word;


/////////////////////////////////////////////////////////
//  Classe abstrata Base
/////////////////////////////////////////////////////////

class Base
{
public:
   Base();
   virtual ~Base();
};


/////////////////////////////////////////////////////////
//  ListNode : ancestral de um num. de lista encadeada
/////////////////////////////////////////////////////////

class ListNode : public Base
{
public:
   ListNode *Next;
   ListNode *Prev();
};
typedef ListNode *ListNodePtr;

typedef void (*ListAction)(ListNodePtr N);

typedef void (*ListActionArq)(ListNodePtr N, fstream& Arq);


//////////////////////////////////////////////////////////////////////
//  List : Lista encadeada cujos elementos sao descendentes de ListNode
//////////////////////////////////////////////////////////////////////

class List : public Base
{
 private:
   ListNodePtr LastNode;
 public:
   List();
   virtual ~List();

   void         Append( ListNodePtr N );
```

```
   void          FreeAll();
   int           Empty();
   long          Count();
   void          ForEach( ListAction Action );
   void          ForEachArq( ListActionArq Action, fstream& Arq );
   ListNodePtr  First();
   ListNodePtr  Last();
   void          Insert( ListNodePtr N );
   ListNodePtr  Next( ListNodePtr N );
   ListNodePtr  Prev( ListNodePtr N );
   ListNodePtr  Node(int item);
   void          Remove( ListNodePtr N );
   void          Free( ListNodePtr N );
   virtual void Read(fstream& file);
   virtual void Write(fstream& file);
};

typedef List *ListPtr;


#endif  // __LISTAS_H
```

C.2 - Arquivo Listas.cpp

```
#include "listas.h"

/////////////////////////////////////////////////////////
//  Classe abstrata Base
/////////////////////////////////////////////////////////

Base::Base()
  {
  }

Base::~Base()
  {
  }


/////////////////////////////////////////////////////////
//  ListNode : ancestral de um num. de lista encadeada
/////////////////////////////////////////////////////////

ListNodePtr ListNode::Prev()
  {
   ListNodePtr P = this;
   while (P->Next != this) P = P->Next;
   return P;
  }

//////////////////////////////////////////////////////////////////////
//  List : Lista encadeada cujos elementos sao descendentes de ListNode
//////////////////////////////////////////////////////////////////////

long auxCount;

void DelListNode( ListNodePtr N )
  {
   delete N;
  }

void CountListNode( ListNodePtr N )
  {
   ++auxCount;
  }

List::List()
  {
   LastNode = NULL;
  }

List::~List()
  {
   FreeAll();
  }

void List::Append( ListNodePtr N )
  {
   Insert( N );
   LastNode = N;
  }

void List::FreeAll()
  {
   ForEach( &DelListNode );
```

```
   LastNode = NULL;
  }

int List::Empty()
  {
   return (LastNode == NULL);
  }

long List::Count()
  {
   auxCount = 0;
   ForEach( &CountListNode );
   return auxCount;
  }

void List::ForEach( ListAction Action )
  {
   ListNodePtr Q;
   ListNodePtr P = First();
   while (P != NULL)
     {
      Q = P;
      P = Next( P );
      Action( Q );
     }
  }

void List::ForEachArq( ListActionArq Action, fstream& Arq )
  {
   ListNodePtr Q;
   ListNodePtr P = First();
   while (P != NULL)
     {
      Q = P;
      P = Next( P );
      Action( Q, Arq );
     }
  }

ListNodePtr List::First()
  {
   if (LastNode == NULL)
      return NULL;
   else return LastNode->Next;
  }

ListNodePtr List::Last()
  {
   return LastNode;
  }

void List::Insert( ListNodePtr N )
  {
   if (LastNode == NULL)
      LastNode = N;
   else N->Next = LastNode->Next;
   LastNode->Next = N;
  }

ListNodePtr List::Next( ListNodePtr N )
  {
```

```
   if (N == LastNode)
      return NULL;
   else return N->Next;
  }

ListNodePtr List::Prev( ListNodePtr N )
  {
   if (N == First())
      return NULL;
   else return N->Prev();
  }

ListNodePtr List::Node(int item)
  {
   if (item>Count()) return NULL;
   ListNodePtr PP=First();
   for (int i=0; i<item-1; i++)
     PP=Next(PP);
   return PP;
  }

void List::Remove( ListNodePtr N )
  {
   if (LastNode != NULL)
     {
      ListNodePtr P = LastNode;
      while ((P->Next != N) && (P->Next != LastNode)) P = P->Next;
      if (P->Next == N)
      {
       P->Next = N->Next;
       if (LastNode == N)
         if (P == N)
           LastNode = NULL;
         else LastNode = P;
      }
     }
  }

void List::Free( ListNodePtr N )
  {
   Remove( N );
   delete N;
  }

void List::Read(fstream& file)
  {
  }

void List::Write(fstream& file)
  {
  }
```

C.3 - Arquivo Classes.h

```
/*****************************************************************
  ESTRUTURA DE DADOS ORIENTADA POR OBJETOS DO PREPROCESSADOR
    DO PROGRAMA DE CALCULO DE CAMPOS ELETROMAGNETICOS 3D.
                  PPGEE - UFMG
  Mestrando: Luis Fernando Nacif Rocha
  Orientador: Prof. Renato Cardoso Mesquita
*****************************************************************/

#if !defined(__CLASSES_H)
#define __CLASSES_H

#include "listas.h"

///////////////////////////////////////////////////////////////////
// Estrutura de classes dos blocos (volumes)
///////////////////////////////////////////////////////////////////

// ***** No' da lista de blocos

class NodeBloco : public ListNode
{
 private:
   int MXi,MXf,MYi,MYf,MZi,MZf;
   int Material,Potencial;
   char Label[21];
 public:
   NodeBloco();
   NodeBloco(int xi, int xf, int yi, int yf, int zi, int zf, int mat, int
pot);
   void SetBloco(int xi, int xf, int yi, int yf, int zi, int zf, int mat,
int pot);
   int GetMXi();
   int GetMXf();
   int GetMYi();
   int GetMYf();
   int GetMZi();
   int GetMZf();
   int GetMaterial();
   int GetPotencial();
   void SetLabel(char *label);
   char *GetLabel();
};

typedef NodeBloco *PNodeBloco;


// Lista de blocos

class ListBloco : public List
{
 public:
   ListBloco();
   PNodeBloco Node(int item);
   virtual void Read(fstream& file);
   virtual void Write(fstream& file);
};

typedef ListBloco *PListBloco;
```

```
// No' base da lista de contornos

class NodeCondContorno : public ListNode
{
 private:
   int NXi,NXf,NYi,NYf,NZi,NZf,Tipo;
   char Label[21];
 public:
   NodeCondContorno();
   NodeCondContorno(int xi, int xf, int yi, int yf, int zi, int zf);
   void SetCondContorno(int xi, int xf, int yi, int yf, int zi, int zf);
   int GetNXi();
   int GetNXf();
   int GetNYi();
   int GetNYf();
   int GetNZi();
   int GetNZf();
   void SetTipo(int tipo);
   int GetTipo();
   void SetLabel(char *label);
   char *GetLabel();
   virtual void GravaCC(fstream& arq);
   virtual void LeCC(fstream& arq);
};

typedef NodeCondContorno *PNodeCondContorno;


// No' da lista de contornos - Tipo Escalar

class CCEscalar : public NodeCondContorno
{
 private:
   double Potencial;
 public:
   CCEscalar();
   CCEscalar(int xi, int xf, int yi, int yf, int zi, int zf, double ddp);
   void SetPotencial(double ddp);
   double GetPotencial();
   virtual void GravaCC(fstream& arq);
   virtual void LeCC(fstream& arq);
};

typedef CCEscalar *PCCEscalar;


// No' da lista de contornos - Tipo Vetor

class CCVetor : public NodeCondContorno
{
 private:
   double Ax,Ay,Az;
   int TipoVetor;
 public:
   CCVetor();
   CCVetor(int xi, int xf, int yi, int yf, int zi, int zf,
         double ax, double ay, double az, int tipo);
   void SetVetor(double ax, double ay, double az, int tipo);
   double GetAx();
   double GetAy();
   double GetAz();
```

```
   int GetTipoVetor();
   virtual void GravaCC(fstream& arq);
   virtual void LeCC(fstream& arq);
};

typedef CCVetor *PCCVetor;


// No' da lista de contornos - Tipo Escalar + Vetor

class CCEscVet : public CCVetor
{
 private:
   double Potencial;
 public:
   CCEscVet();
   CCEscVet(int xi, int xf, int yi, int yf, int zi, int zf,
         double ax, double ay, double az, int tipo, double ddp);
   void SetPotencial(double ddp);
   double GetPotencial();
   virtual void GravaCC(fstream& arq);
   virtual void LeCC(fstream& arq);
};

typedef CCEscVet *PCCEscVet;


// No' da lista de contornos - Tipo Us

class CCUs : public CCEscalar
{
 public:
   CCUs();
   CCUs(int xi, int xf, int yi, int yf, int zi, int zf, double ddp);
};

typedef CCUs *PCCUs;


// No' da lista de contornos - Tipo Flutuante

class CCFlutuante : public NodeCondContorno
{
 private:
   int TipoFlut, TipoVetor;
 public:
   CCFlutuante();
   CCFlutuante(int xi, int xf, int yi, int yf, int zi, int zf, int tipo1,
int tipo2);
   void SetFlutuante(int tipo1, int tipo2);
   int GetTipoFlut();
   int GetTipoVetor();
   virtual void GravaCC(fstream& arq);
   virtual void LeCC(fstream& arq);
};

typedef CCFlutuante *PCCFlutuante;


// No' da lista de contornos - Tipo Termico de Conveccao
```

```
class CCTermConveccao : public NodeCondContorno
{
 private:
   double H,Epslon;
 public:
   CCTermConveccao();
   CCTermConveccao(int xi, int xf, int yi, int yf, int zi, int zf, double
h, double epslon);
   void SetConveccao(double h, double epslon);
   double GetCoefConveccao();
   double GetEpslon();
   virtual void GravaCC(fstream& arq);
   virtual void LeCC(fstream& arq);
};

typedef CCTermConveccao *PCCTermConveccao;


// No' da lista de contornos - Tipo Termico com temperatura imposta

class CCTermTempImposta : public NodeCondContorno
{
 private:
   double Temperatura;
 public:
   CCTermTempImposta();
   CCTermTempImposta(int xi, int xf, int yi, int yf, int zi, int zf,
double temp);
   void SetTemperatura(double temp);
   double GetTemperatura();
   virtual void GravaCC(fstream& arq);
   virtual void LeCC(fstream& arq);
};

typedef CCTermTempImposta *PCCTermTempImposta;


// Lista de condicoes de contorno

class ListCondContorno : public List
{
 public:
   ListCondContorno();
   PNodeCondContorno Node(int item);
   virtual void Read(fstream& arq);
   virtual void Write(fstream& arq);
};

typedef ListCondContorno *PListCondContorno;


class NodeMalhaRegular : public ListNode
{
 private:
   double Dimensao;
   int NumDivisoes;
 public:
   NodeMalhaRegular();
   NodeMalhaRegular(double dim, int num);
   void SetMalhaRegular(double dim, int num);
   double GetDimensao();
```

```
   int GetNumDivisoes();
};

typedef NodeMalhaRegular *PNodeMalhaRegular;


class MalhaRegular : public Base
{
 private:
   ListPtr DirX, DirY, DirZ;

 public:
   MalhaRegular();
   ~MalhaRegular();
   ListPtr GetPtrEixo(int dir);
   void  Append(int dir, double dim, int num);
   int   Count(int dir);
   void  SetNode(int dir, int item, double dim, int num);
   double GetDimensaoNode(int dir, int item);
   int   GetNumDivisoesNode(int dir, int item);
   void  Free(int dir, int node);
   virtual void Write(fstream& file);
   virtual void Read(fstream& file);
};

typedef MalhaRegular *PMalhaRegular;


////////////////////////////////////////////////////////////
//      CLASSES DAS REGIOES PORTADORAS DE CORRENTE
////////////////////////////////////////////////////////////

class NodeCorrente : public ListNode
{
 private:
   double I;
   int Tipo;
   char Label[21];
 public:
   NodeCorrente();
   NodeCorrente(double corr);
   void SetCorrente (double Is);
   void SetTipo(int tipo);
   double GetCorrente();
   int GetTipo();
   void SetLabel(char *label);
   char *GetLabel();
   virtual void GravaCorrente(fstream& arq);
   virtual void LeCorrente(fstream& arq);
};

typedef NodeCorrente *PNodeCorrente;


class Fio : public NodeCorrente
{
 private:
   double Coord_Fio[2][3];
 public:
   Fio();
   virtual void SetPonto (int ponto, double x, double y, double z);
```

```
   virtual double GetCoord(int ponto, int eixo);
   virtual void GravaCorrente(fstream& arq);
   virtual void LeCorrente(fstream& arq);
};

typedef Fio *PFio;


class Arco : public Fio
{
 private:
   double Centro[3];
 public:
   Arco();
   void SetCentro (double x, double y, double z);
   double GetCentro(int eixo);
   virtual int PontosOK();
   virtual void GravaCorrente(fstream& arq);
   virtual void LeCorrente(fstream& arq);
};

typedef Arco *PArco;


class Plano : public Fio
{
 private:
   double Coord_Plano[2][3];
 public:
   Plano();
   virtual void SetPonto (int ponto, double x, double y, double z);
   virtual double GetCoord(int ponto, int eixo);
   virtual int PontosOK();
   virtual void CalculaPontos();
   virtual void GravaCorrente(fstream& arq);
   virtual void LeCorrente(fstream& arq);
};

typedef Plano *PPlano;


class Casca : public Plano
{
 private:
   double Centro[3];
 public:
   Casca();
   void SetCentro (double x, double y, double z);
   double GetCentro(int eixo);
   virtual int PontosOK();
   virtual void GravaCorrente(fstream& arq);
   virtual void LeCorrente(fstream& arq);
};

typedef Casca *PCasca;


class Tijolo : public Plano
{
 private:
   double Coord_Tijolo[4][3];
```

```
 protected:
   void CalculaPonto4();
 public:
   Tijolo();
   virtual void SetPonto (int ponto, double x, double y, double z);
   virtual double GetCoord(int ponto, int eixo);
   virtual int PontosOK();
   virtual void CalculaPontos();
   virtual void GravaCorrente(fstream& arq);
   virtual void LeCorrente(fstream& arq);
};

typedef Tijolo *PTijolo;


class Cilindro : public Tijolo
{
 private:
   double Centro[3];
 public:
   Cilindro();
   void SetCentro (double x, double y, double z);
   double GetCentro(int eixo);
   virtual int PontosOK();
   virtual void CalculaPontos();
   virtual void GravaCorrente(fstream& arq);
   virtual void LeCorrente(fstream& arq);
};

typedef Cilindro *PCilindro;


class I_Bloco : public NodeCorrente
{
 private:
   int MXi,MXf,MYi,MYf,MZi,MZf;
   int Direcao;
 public:
   I_Bloco();
   I_Bloco(int xi, int xf, int yi, int yf, int zi, int zf, int dir,
double corr);
   void SetI_Bloco(int xi, int xf, int yi, int yf, int zi, int zf, int
dir);
   int GetMXi();
   int GetMXf();
   int GetMYi();
   int GetMYf();
   int GetMZi();
   int GetMZf();
   int GetDirecao();
   virtual void GravaCorrente(fstream& arq);
   virtual void LeCorrente(fstream& arq);
};

typedef I_Bloco *PI_Bloco;


class ListCorrente : public List
{
 public:
   ListCorrente();
```

```
   PNodeCorrente Node(int item);
   virtual void Read(fstream& arq);
   virtual void Write(fstream& arq);
};

typedef ListCorrente *PListCorrente;


/////////////////////////////////////////////////////
//            Classes de Materiais
/////////////////////////////////////////////////////

class NodeCurvaBH : public ListNode
{
 private:
   double B,H;
 public:
   NodeCurvaBH();
   NodeCurvaBH(double b, double h);
   void SetNodeCurvaBH(double b, double h);
   double GetB();
   double GetH();
};

typedef NodeCurvaBH *PNodeCurvaBH;


class ListCurvaBH : public List
{
 public:
   ListCurvaBH();
   void AppendBH(double b, double h);
   PNodeCurvaBH Node(int item);
   virtual void Read(fstream& arq);
   virtual void Write(fstream& arq);
};

typedef ListCurvaBH *PListCurvaBH;


class NodeMaterial : public ListNode
{
 private:
   char Label[21];
   double Permeabilidade;
   double Condutividade;
   double Bcx, Bcy, Bcz;      // Imantacao
   double Beta0,        // Variacao da condutividade com a temperatura
        Roc0,               // Capacidade Termica
        Xk0,                  // Condutividade Termica
        CNT1, CNT2, CNT3, CNT4, CNT5,  // Constantes
        TCurie,         // Temperatura de Curie
        Cond1,                // Condutividade eletrica apos Curie
        Beta1;          // Var. da condutiv. com a temp. apos Curie
   PListCurvaBH Curva;

 public:
   NodeMaterial();
   NodeMaterial(double perm, double cond, double bx, double by, double
bz,
            double bet0, double roc, double xk, double cn1, double cn2,
```

```
             double cn3, double cn4, double cn5, double tc, double sig,
             double bet1);
   virtual ~NodeMaterial();
   void SetLabel(char *label);
   void SetMaterial(double perm, double cond, double bx, double by,
                 double bz, double bet0, double roc, double xk,
                 double cn1, double cn2, double cn3, double cn4,
                 double cn5, double tc, double sig, double bet1);
   void SetPermeabilidade(double perm);
   void SetCondutividade(double cond);
   void SetImantacao(double bx, double by, double bz);
   void SetTermico(double bet0, double roc, double xk,double cn1,
                double cn2, double cn3, double cn4, double cn5,
                double tc, double sig, double bet1);
   char  *GetLabel();
   double GetPermeabilidade();
   double GetCondutividade();
   double GetBx();
   double GetBy();
   double GetBz();
   double GetBeta0();
   double GetRoc0();
   double GetXk0();
   double GetCNT1();
   double GetCNT2();
   double GetCNT3();
   double GetCNT4();
   double GetCNT5();
   double GetTCurie();
   double GetCond1();
   double GetBeta1();
   PListCurvaBH CurvaBH();
};

typedef NodeMaterial* PNodeMaterial;


class ListMaterial : public List
{
 public:
   ListMaterial();
   PNodeMaterial Node(int item);
   virtual void Write(fstream &arq);
   virtual void Read(fstream &arq);
};

typedef ListMaterial *PListMaterial;


///////////////////////////////////////////////////////////////////
//        Rotinas Gerais de Arquivo e Inicializacao de Dados
///////////////////////////////////////////////////////////////////

class Preprocessador : public Base
{
 private:
   char Titulo[31];
   int CamadasX,CamadasY,CamadasZ;
   int TipoPeriodo;
   int MaterialDefault,PotencialDefault;
```

```
 public:
   PListBloco Blocos;
   PListCondContorno CondContorno;
   PMalhaRegular Malha;
   PListCorrente Correntes;
   PListMaterial Materiais;

   Preprocessador();
   ~Preprocessador();

   void InicializaDados();
   void SetTitulo(char *titl);
   char *GetTitulo();
   void SetCamadas(int x, int y, int z);
   int  GetCamadasX();
   int  GetCamadasY();
   int  GetCamadasZ();
   void SetTipoPeriodo(int tipo);
   int  GetTipoPeriodo();
   void SetMaterialDefault(int matdef);
   int  GetMaterialDefault();
   void SetPotencialDefault(int potdef);
   int  GetPotencialDefault();
   virtual void GravaDados(char *NomeArq);
   virtual void LeDados(char *NomeArq);
   virtual void GeraArqMat(char *NomeArq);
   virtual void LeArqMat(char *NomeArq);
   void GeraArquivoNeutro(char *NomeArq);

};

typedef Preprocessador *PPreprocessador;


#endif // __CLASSES_H
```

C.4 - Arquivo Classes.cpp

```
#include <stdlib.h>
#include <fstream.h>
#include <math.h>
#include <string.h>

#include "classes.h"
#include "arqneutr.h"


//////////////////////////////////////////////////////////////////////
// Estrutura de classes dos blocos (volumes)
//////////////////////////////////////////////////////////////////////

// ***** No' da lista de blocos

NodeBloco::NodeBloco()
  { MXi=MXf=MYi=MYf=MZi=MZf=Material=0; Potencial=1; strcpy(Label,""); }

NodeBloco::NodeBloco(int xi, int xf, int yi, int yf, int zi, int zf, int
mat, int pot)
  { MXi=xi; MXf=xf; MYi=yi; MYf=yf; MZi=zi; MZf=zf; Material=mat;
Potencial=pot; strcpy(Label,""); }

void NodeBloco::SetBloco(int xi, int xf, int yi, int yf, int zi, int zf,
int mat, int pot)
  { MXi=xi; MXf=xf; MYi=yi; MYf=yf; MZi=zi; MZf=zf; Material=mat;
Potencial=pot; }

int NodeBloco::GetMXi()
     { return MXi; }

int NodeBloco::GetMXf()
     { return MXf; }

int NodeBloco::GetMYi()
     { return MYi; }

int NodeBloco::GetMYf()
     { return MYf; }

int NodeBloco::GetMZi()
     { return MZi; }

int NodeBloco::GetMZf()
     { return MZf; }

void NodeBloco::SetLabel(char *label)
     { strcpy(Label,label); }

char *NodeBloco::GetLabel()
     { return Label; }

int NodeBloco::GetMaterial()
     { return Material; }

int NodeBloco::GetPotencial()
     { return Potencial; }

void GravaNodeBloco(ListNodePtr P, fstream& arq)
  {
   PNodeBloco Ptr=(PNodeBloco) P;
```

```
   arq << Ptr->GetLabel() << '\n';
   arq << Ptr->GetMXi() << ' ' << Ptr->GetMXf() << ' ' << Ptr->GetMYi()
       << ' ' << Ptr->GetMYf() << ' ' << Ptr->GetMZi() << ' ' << Ptr-
>GetMZf()
       << ' ' << Ptr->GetMaterial() << ' ' << Ptr->GetPotencial() <<
'\n';
  }

void LeNodeBloco(ListNodePtr P, fstream& arq)
  {
   int xi, xf, yi, yf, zi, zf, mat, pot;
   char str[80];
   char label[20];
   PNodeBloco Ptr=(PNodeBloco) P;
   arq.getline(label,20,'\n');
   arq >> xi >> xf >> yi >> yf >> zi >> zf >> mat >> pot;
   arq.getline(str,80,'\n');
   Ptr->SetBloco(xi, xf, yi, yf, zi, zf, mat, pot);
   Ptr->SetLabel(label);
  }


// Lista de blocos

ListBloco::ListBloco()
  {
  }

PNodeBloco ListBloco::Node(int item)
  {
   ListNodePtr p=List::Node(item);
   return (PNodeBloco) p;
  }

void ListBloco::Write(fstream& arq)
  {
   arq << Count() << '\n';
    ForEachArq( &GravaNodeBloco, arq);
  }

void ListBloco::Read(fstream& arq)
  {
   FreeAll();
   int n;
   char str[80];
   arq >> n;
   arq.getline(str,80,'\n');
   for (int i=0; i<n; i++)
     Append(new NodeBloco());
    ForEachArq( &LeNodeBloco, arq);
  }


// No' base da lista de contornos

NodeCondContorno::NodeCondContorno()
  { NXi=NXf=NYi=NYf=NZi=NZf=Tipo=0; strcpy(Label,""); }

NodeCondContorno::NodeCondContorno(int xi, int xf, int yi, int yf, int
zi, int zf)
  { NXi=xi; NXf=xf; NYi=yi; NYf=yf; NZi=zi; NZf=zf; strcpy(Label,""); }
```

```
void NodeCondContorno::SetCondContorno(int xi, int xf, int yi, int yf,
int zi, int zf)
  { NXi=xi; NXf=xf; NYi=yi; NYf=yf; NZi=zi; NZf=zf; }

int NodeCondContorno::GetNXi()
     { return NXi; }

int NodeCondContorno::GetNXf()
     { return NXf; }

int NodeCondContorno::GetNYi()
     { return NYi; }

int NodeCondContorno::GetNYf()
     { return NYf; }

int NodeCondContorno::GetNZi()
     { return NZi; }

int NodeCondContorno::GetNZf()
     { return NZf; }

void NodeCondContorno::SetTipo(int tipo)
     { Tipo=tipo; }

int NodeCondContorno::GetTipo()
     { return Tipo; }

void NodeCondContorno::SetLabel(char *label)
     { strcpy(Label,label); }

char *NodeCondContorno::GetLabel()
     { return Label; }

void NodeCondContorno::GravaCC(fstream& arq)
  {
   arq << Label << '\n';
   arq << NXi << ' ' << NXf << ' ' << NYi << ' '
       << NYf << ' ' << NZi << ' ' << NZf << '\n';
  }

void NodeCondContorno::LeCC(fstream& arq)
  {
   char str[80];
   arq.getline(Label,20,'\n');
   arq >> NXi >> NXf >> NYi >> NYf >> NZi >> NZf;
   arq.getline(str,80,'\n');
  }


// No' da lista de contornos - Tipo Escalar

CCEscalar::CCEscalar()
  { Potencial=0.0; SetTipo(1); }

CCEscalar::CCEscalar(int xi, int xf, int yi, int yf, int zi, int zf,
double ddp):
     NodeCondContorno(xi,xf,yi,yf,zi,zf) { Potencial=ddp; SetTipo(1); }

void CCEscalar::SetPotencial(double ddp)
```

```
     { Potencial=ddp; }

double CCEscalar::GetPotencial()
     { return Potencial; }

void CCEscalar::GravaCC(fstream& arq)
     {
      NodeCondContorno::GravaCC(arq);
      arq << Potencial << '\n';
     }
void CCEscalar::LeCC(fstream& arq)
     {
      char str[80];
      NodeCondContorno::LeCC(arq);
      arq >> Potencial;
      arq.getline(str,80,'\n');
     }


// No' da lista de contornos - Tipo Vetor

CCVetor::CCVetor()
  { Ax=Ay=Az=0.0; TipoVetor=1; SetTipo(2); }

CCVetor::CCVetor(int xi, int xf, int yi, int yf, int zi, int zf,
         double ax, double ay, double az, int tipo):
     NodeCondContorno(xi,xf,yi,yf,zi,zf)
     { Ax=ax; Ay=ay; Az=az; TipoVetor=tipo; SetTipo(2); }

void CCVetor::SetVetor(double ax, double ay, double az, int tipo)
     { Ax=ax; Ay=ay; Az=az; TipoVetor=tipo; }

double CCVetor::GetAx()
     { return Ax; }

double CCVetor::GetAy()
     { return Ay; }

double CCVetor::GetAz()
     { return Az; }

int CCVetor::GetTipoVetor()
     { return TipoVetor; }

void CCVetor::GravaCC(fstream& arq)
     {
      NodeCondContorno::GravaCC(arq);
      arq << TipoVetor << ' ' << Ax << ' ' << Ay << ' ' << Az << '\n';
     }

void CCVetor::LeCC(fstream& arq)
     {
      char str[80];
      NodeCondContorno::LeCC(arq);
      arq >> TipoVetor >> Ax >> Ay >> Az;
      arq.getline(str,80,'\n');
     }


// No' da lista de contornos - Tipo Escalar + Vetor
```

```
CCEscVet::CCEscVet()
  { SetTipo(3); }

CCEscVet::CCEscVet(int xi, int xf, int yi, int yf, int zi, int zf,
         double ax, double ay, double az, int tipo, double ddp):
     CCVetor(xi,xf,yi,yf,zi,zf,ax,ay,az,tipo) { Potencial=ddp;
SetTipo(3); }

void CCEscVet::SetPotencial(double ddp)
     { Potencial=ddp; }

double CCEscVet::GetPotencial()
     { return Potencial; }

void CCEscVet::GravaCC(fstream& arq)
     {
      CCVetor::GravaCC(arq);
      arq << Potencial << '\n';
     }

void CCEscVet::LeCC(fstream& arq)
     {
      char str[80];
      CCVetor::LeCC(arq);
      arq >> Potencial;
      arq.getline(str,80,'\n');
     }


// No' da lista de contornos - Tipo Us

CCUs::CCUs()
    { SetTipo(4); }

CCUs::CCUs(int xi, int xf, int yi, int yf, int zi, int zf, double ddp):
     CCEscalar(xi,xf,yi,yf,zi,zf,ddp) { SetTipo(4); }


// No' da lista de contornos - Tipo Flutuante

CCFlutuante::CCFlutuante()
  { TipoFlut=TipoVetor=1; SetTipo(5); }

CCFlutuante::CCFlutuante(int xi, int xf, int yi, int yf, int zi, int zf,
int tipo1, int tipo2):
     NodeCondContorno(xi,xf,yi,yf,zi,zf) { TipoFlut=tipo1;
TipoVetor=tipo2; SetTipo(5); }

void CCFlutuante::SetFlutuante(int tipo1, int tipo2)
     { TipoFlut=tipo1; TipoVetor=tipo2; }

int CCFlutuante::GetTipoFlut()
     { return TipoFlut; }

int CCFlutuante::GetTipoVetor()
     { return TipoVetor; }

void CCFlutuante::GravaCC(fstream& arq)
     {
      NodeCondContorno::GravaCC(arq);
      arq << TipoFlut << ' ' << TipoVetor << '\n';
```

```
     }

void CCFlutuante::LeCC(fstream& arq)
     {
      char str[80];
      NodeCondContorno::LeCC(arq);
      arq >> TipoFlut >> TipoVetor;
      arq.getline(str,80,'\n');
     }


// No' da lista de contornos - Tipo Termico de Conveccao

CCTermConveccao::CCTermConveccao()
  { H=Epslon=0.0; SetTipo(6); }

CCTermConveccao::CCTermConveccao(int xi, int xf, int yi, int yf, int zi,
int zf, double h, double epslon):
     NodeCondContorno(xi,xf,yi,yf,zi,zf) { H=h; Epslon=epslon;
SetTipo(6); }

void CCTermConveccao::SetConveccao(double h, double epslon)
     { H=h; Epslon=epslon; }

double CCTermConveccao::GetCoefConveccao()
     { return H; }

double CCTermConveccao::GetEpslon()
     { return Epslon; }

void CCTermConveccao::GravaCC(fstream& arq)
     {
      NodeCondContorno::GravaCC(arq);
      arq << Epslon << ' ' << H << '\n';
     }

void CCTermConveccao::LeCC(fstream& arq)
     {
      char str[80];
      NodeCondContorno::LeCC(arq);
      arq >> Epslon >> H;
      arq.getline(str,80,'\n');
     }


// No' da lista de contornos - Tipo Termico com temperatura imposta

CCTermTempImposta::CCTermTempImposta()
  { Temperatura=0.0; SetTipo(7); }

CCTermTempImposta::CCTermTempImposta(int xi, int xf, int yi, int yf, int
zi, int zf, double temp):
     NodeCondContorno(xi,xf,yi,yf,zi,zf) { Temperatura=temp; SetTipo(7);
}

void CCTermTempImposta::SetTemperatura(double temp)
     { Temperatura=temp; }

double CCTermTempImposta::GetTemperatura()
     { return Temperatura; }
```

```
void CCTermTempImposta::GravaCC(fstream& arq)
      {
       NodeCondContorno::GravaCC(arq);
       arq << Temperatura << '\n';
      }

void CCTermTempImposta::LeCC(fstream& arq)
      {
       char str[80];
       NodeCondContorno::LeCC(arq);
       arq >> Temperatura;
       arq.getline(str,80,'\n');
      }


// Lista de condicoes de contorno

ListCondContorno::ListCondContorno()
 {
 }

PNodeCondContorno ListCondContorno::Node(int item)
  {
   ListNodePtr p=List::Node(item);
   return (PNodeCondContorno) p;
  }

void ListCondContorno::Write(fstream& arq)
  {
   PNodeCondContorno Pl;
   ListNodePtr P = First();
   arq << Count() << '\n';
   while (P!=NULL) {
      Pl=(PNodeCondContorno) P;
      arq << Pl->GetTipo() << '\n';
      Pl->GravaCC(arq);
      P=Next(P);
    }
  }

void ListCondContorno::Read(fstream& arq)
  {
   int i, n, tipo;
   char str[80];
   FreeAll();
   PNodeCondContorno Pl;
   arq >> n;
   arq.getline(str,80,'\n');
   for(i=0; i<n; i++)
     {
      arq >> tipo;
      arq.getline(str,80,'\n');
      switch (tipo) {
      case 1: Append( new CCEscalar() );
          break;
      case 2: Append( new CCVetor() );
          break;
      case 3: Append( new CCEscVet() );
          break;
      case 4: Append( new CCUs() );
          break;
```

```
      case 5: Append( new CCFlutuante() );
          break;
      case 6: Append( new CCTermConveccao() );
          break;
      case 7: Append( new CCTermTempImposta() );
          break;
      }
      Pl=(PNodeCondContorno) Last();
      Pl->SetTipo(tipo);
      Pl->LeCC(arq);
     }
  }


// ****** No' da malha regular

NodeMalhaRegular::NodeMalhaRegular()
     { Dimensao=0.0; NumDivisoes=0; }

NodeMalhaRegular::NodeMalhaRegular(double dim, int num)
     { Dimensao=dim; NumDivisoes=num; }

void NodeMalhaRegular::SetMalhaRegular(double dim, int num)
     { Dimensao=dim; NumDivisoes=num; }

double NodeMalhaRegular::GetDimensao()
     { return Dimensao; }

int NodeMalhaRegular::GetNumDivisoes()
     { return NumDivisoes; }

void GravaNodeMalhaR(ListNodePtr P, fstream& arq)
  {
   PNodeMalhaRegular Ptr=(PNodeMalhaRegular) P;
   arq << Ptr->GetDimensao() << ' ' << Ptr->GetNumDivisoes() << '\n';
  }

void LeNodeMalhaR(ListNodePtr P, fstream& arq)
  {
   int num;
   double div;
   char str[80];
   PNodeMalhaRegular Ptr=(PNodeMalhaRegular) P;
   arq >> div >> num;
   arq.getline(str,80,'\n');
   Ptr->SetMalhaRegular(div,num);
  }


// Listas da malha regular

MalhaRegular::MalhaRegular()
     { DirX = new List(); DirY = new List(); DirZ = new List(); }

MalhaRegular::~MalhaRegular()
      {
      DirX->FreeAll(); DirY->FreeAll(); DirZ->FreeAll();
      delete DirX; delete DirY; delete DirZ;
      }

ListPtr MalhaRegular::GetPtrEixo(int dir)
```

```
  {
   if (dir==1)
      return DirX;
   else if (dir==2)
         return DirY;
      else if (dir==3)
            return DirZ;
           else return NULL;
  }

void MalhaRegular::Append(int dir, double dim, int num)
  { GetPtrEixo(dir)->Append(new NodeMalhaRegular(dim,num)); }

int MalhaRegular::Count(int dir)
  { return GetPtrEixo(dir)->Count(); }

void MalhaRegular::SetNode(int dir, int item, double dim, int num)
  { (PNodeMalhaRegular (GetPtrEixo(dir)->Node(item)))-
>SetMalhaRegular(dim,num); }

double MalhaRegular::GetDimensaoNode(int dir, int item)
  { return (PNodeMalhaRegular (GetPtrEixo(dir)->Node(item)))-
>GetDimensao(); }

int MalhaRegular::GetNumDivisoesNode(int dir, int item)
  { return (PNodeMalhaRegular (GetPtrEixo(dir)->Node(item)))-
>GetNumDivisoes(); }

void MalhaRegular::Free(int dir, int item)
  { GetPtrEixo(dir)->Free(GetPtrEixo(dir)->Node(item)); }

void MalhaRegular::Write(fstream& arq)
  {
   arq << DirX->Count() << '\n';
   DirX->ForEachArq( &GravaNodeMalhaR, arq );
   arq << DirY->Count() << '\n';
   DirY->ForEachArq( &GravaNodeMalhaR, arq );
   arq << DirZ->Count() << '\n';
   DirZ->ForEachArq( &GravaNodeMalhaR, arq );
  }

void MalhaRegular::Read(fstream& arq)
  {
   int n;
   char str[80];
   DirX->FreeAll();
   DirY->FreeAll();
   DirZ->FreeAll();
   arq >> n;
   arq.getline(str,80,'\n');
   for (int i=0; i<n; i++)
     DirX->Append(new NodeMalhaRegular() );
   DirX->ForEachArq( &LeNodeMalhaR, arq );
   arq >> n;
   arq.getline(str,80,'\n');
   for (i=0; i<n; i++)
     DirY->Append(new NodeMalhaRegular() );
   DirY->ForEachArq( &LeNodeMalhaR, arq );
   arq >> n;
   arq.getline(str,80,'\n');
   for (i=0; i<n; i++)
```

```
     DirZ->Append(new NodeMalhaRegular() );
   DirZ->ForEachArq( &LeNodeMalhaR, arq );
  }


////////////////////////////////////////////////////////////
//      CLASSES DAS REGIOES PORTADORAS DE CORRENTE
////////////////////////////////////////////////////////////


// No' base da lista de correntes

NodeCorrente::NodeCorrente()
     { I=0.0; Tipo=0; strcpy(Label,""); }

NodeCorrente::NodeCorrente(double corr)
     { I=corr; Tipo=0; strcpy(Label,""); }

void NodeCorrente::SetCorrente (double Is)
     { I=Is; }

void NodeCorrente::SetTipo(int tipo)
     { Tipo=tipo; }

double NodeCorrente::GetCorrente()
     { return I; }

int NodeCorrente::GetTipo()
     { return Tipo; }

void NodeCorrente::SetLabel(char *label)
     { strcpy(Label,label); }

char *NodeCorrente::GetLabel()
     { return Label; }

void NodeCorrente::GravaCorrente(fstream& arq)
     { arq << Label << '\n' << I << '\n'; }

void NodeCorrente::LeCorrente(fstream& arq)
     {
      char str[80];
      arq.getline(Label,20,'\n');
      arq >> I;
      arq.getline(str,80,'\n');
     }


// no' para Fio - Geometria Unidimensional

Fio::Fio()
     {
      int i,j;
      for (i=0; i<2; i++)
      for (j=0; j<3; j++)
        Coord_Fio[i][j]=0.0;
      SetTipo(1);
     }

void Fio::SetPonto (int ponto, double x, double y, double z)
     {
```

```
       Coord_Fio[ponto-1][0]=x;
       Coord_Fio[ponto-1][1]=y;
       Coord_Fio[ponto-1][2]=z;
      }

double Fio::GetCoord(int ponto, int eixo)
      {
       if ((ponto<1) || (ponto>2) || (eixo<1) || (eixo>3))
       {
        cout << "Erro nos Parametros de GetCoord(ponto,eixo) - Fio\n";
        exit(1);
       }
       return Coord_Fio[ponto-1][eixo-1];
      }

void Fio::GravaCorrente(fstream& arq)
       {
        NodeCorrente::GravaCorrente(arq);
        int i,j;
        for (i=0; i<2; i++)
       {
        for (j=0; j<3; j++)
          arq << Coord_Fio[i][j] << ' ';
        arq << '\n';
       }
       }

void Fio::LeCorrente(fstream& arq)
       {
        NodeCorrente::LeCorrente(arq);
        int i,j;
        char str[80];
        for (i=0; i<2; i++)
       {
        for (j=0; j<3; j++)
          arq >> Coord_Fio[i][j];
        arq.getline(str,80,'\n');
       }
       }


// no' para Arco - Geometria Unidimensional

Arco::Arco()
  { Centro[0]=Centro[1]=Centro[2]=0.0; SetTipo(2); }

void Arco::SetCentro (double x, double y, double z)
      { Centro[0]=x; Centro[1]=y; Centro[2]=z; }

double Arco::GetCentro(int eixo)
      {
       if ((eixo<1) || (eixo>3))
       {
        cout << "Erro no Parametro de GetCentro(eixo)\n";
        exit(1);
       }
       return Centro[eixo-1];
      }

int Arco::PontosOK()
     {
```

```
     double R1=0,R2=0;
     for (int i=1; i<=3; i++)
       {
      R1+=pow((GetCoord(1,i)-GetCentro(i)),2);
      R2+=pow((GetCoord(2,i)-GetCentro(i)),2);
       }
     if (fabs(R1-R2)/R1>0.05)
       return false
     else return true;
    }

void Arco::GravaCorrente(fstream& arq)
     { Fio::GravaCorrente(arq);
       arq << Centro[0] << ' ' << Centro[1] << ' ' << Centro[2] << '\n';
}

void Arco::LeCorrente(fstream& arq)
     { char str[80];
       Fio::LeCorrente(arq);
       arq >> Centro[0] >> Centro[1] >> Centro[2];
       arq.getline(str,80,'\n'); }


// no' para Plano - Geometria Bidimensional

Plano::Plano()
     {
      int i,j;
      for (i=0; i<2; i++)
      for (j=0; j<3; j++)
        Coord_Plano[i][j]=0.0;
      SetTipo(3);
     }

void Plano::SetPonto (int ponto, double x, double y, double z)
     {
       if (ponto<3)
       Fio::SetPonto(ponto,x,y,z);
       else { Coord_Plano[ponto-3][0]=x;
              Coord_Plano[ponto-3][1]=y;
              Coord_Plano[ponto-3][2]=z; }
     }

double Plano::GetCoord(int ponto, int eixo)
     {
       if ((ponto<1) || (ponto>4) || (eixo<1) || (eixo>3))
       {
        cout << "Erro nos Parametros de GetCoord(ponto,eixo) - Plano\n";
        exit(1);
       }
      if (ponto<3)
      return Fio::GetCoord(ponto,eixo);
      else return Coord_Plano[ponto-3][eixo-1];
     }

int Plano::PontosOK()
    {
     // Testa se P3 e' perpendicular a P1-P2
     double TVTS;
     TVTS=(GetCoord(1,1)*GetCoord(2,1))*(GetCoord(3,1)-GetCoord(1,1))+
        (GetCoord(3,2)*GetCoord(1,2))*(GetCoord(1,2)-GetCoord(2,2))+
```

```
          (GetCoord(1,3)*GetCoord(2,3))*(GetCoord(3,3)-GetCoord(1,3));
      TVTS/=sqrt(pow((GetCoord(3,1)-GetCoord(2,1))+
                 (GetCoord(3,2)-GetCoord(2,2))+
                 (GetCoord(3,3)-GetCoord(2,3)),2));
      if (fabs(TVTS)>0.05)
       return false
      else return true;
     }

void Plano::CalculaPontos()
      {
       double XNum,XDen,T2;
       XNum=(GetCoord(2,2)-GetCoord(3,2))*(GetCoord(3,1)-GetCoord(1,1))+
          (GetCoord(3,2)-GetCoord(1,2))*(GetCoord(3,1)-GetCoord(2,1));
       XDen=(GetCoord(2,2)-GetCoord(1,2))*(GetCoord(3,1)-GetCoord(1,1))+
          (GetCoord(1,2)-GetCoord(3,2))*(GetCoord(2,1)-GetCoord(1,1));
       if ((fabs(XNum)<1e-5) && (fabs(XDen)<1e-5))
       T2=1.0;
       else T2=XNum/XDen;
       Coord_Plano[1][0]=GetCoord(3,1)+(GetCoord(2,1)-GetCoord(1,1))*T2;
       Coord_Plano[1][1]=GetCoord(3,2)+(GetCoord(2,2)-GetCoord(1,2))*T2;
       Coord_Plano[1][2]=GetCoord(3,3)+(GetCoord(2,3)-GetCoord(1,3))*T2;
      }

void Plano::GravaCorrente(fstream& arq)
       {
        Fio::GravaCorrente(arq);
        int i,j;
        for (i=0; i<2; i++)
       {
        for (j=0; j<3; j++)
          arq << Coord_Plano[i][j] << ' ';
        arq << '\n';
       }
       }

void Plano::LeCorrente(fstream& arq)
       {
        Fio::LeCorrente(arq);
        int i,j;
        char str[80];
        for (i=0; i<2; i++)
        {
         for (j=0; j<3; j++)
           arq >> Coord_Plano[i][j];
         arq.getline(str,80,'\n');
        }
       }


// no' para Casca - Geometria Bidimensional

Casca::Casca()
  { Centro[0]=Centro[1]=Centro[2]=0.0; SetTipo(4); }

void Casca::SetCentro (double x, double y, double z)
      { Centro[0]=x; Centro[1]=y; Centro[2]=z; }

double Casca::GetCentro(int eixo)
      {
       if ((eixo<1) || (eixo>3))
```

```
       {
        cout << "Erro no Parametro de GetCentro(eixo)\n";
        exit(1);
       }
       return Centro[eixo-1];
      }

int Casca::PontosOK()
     {
      // Testa se confere o centro da casca
      if (!Plano::PontosOK())
        return false
      else
       {
        double R1=0,R2=0;
        for (int i=1; i<=3; i++)
          {
           R1+=pow((GetCoord(1,i)-GetCentro(i)),2);
           R2+=pow((GetCoord(3,i)-GetCentro(i)),2);
          }
        if (fabs(R1-R2)/R1>0.05)
          return false
        else return true;
       }
     }

void Casca::GravaCorrente(fstream& arq)
      { Plano::GravaCorrente(arq);
        arq << Centro[0] << ' ' << Centro[1] << ' ' << Centro[2] << '\n';
}

void Casca::LeCorrente(fstream& arq)
      { char str[80];
        Plano::LeCorrente(arq);
        arq >> Centro[0] >> Centro[1] >> Centro[2];
        arq.getline(str,80,'\n'); }


// no' para Tijolo - Geometria Tridimensional

Tijolo::Tijolo()
      {
       int i,j;
       for (i=0; i<4; i++)
       for (j=0; j<3; j++)
          Coord_Tijolo[i][j]=0.0;
       SetTipo(5);
      }

void Tijolo::SetPonto (int ponto, double x, double y, double z)
      {
        if (ponto<5)
        Plano::SetPonto(ponto,x,y,z);
        else { Coord_Tijolo[ponto-5][0]=x;
             Coord_Tijolo[ponto-5][1]=y;
             Coord_Tijolo[ponto-5][2]=z; }
      }

double Tijolo::GetCoord(int ponto, int eixo)
      {
        if ((ponto<1) || (ponto>8) || (eixo<1) || (eixo>3))
```

```
        {
         cout << "Erro nos Parametros de GetCoord(ponto,eixo) - Tijolo\n";
         exit(1);
        }
        if (ponto<5)
        return Plano::GetCoord(ponto,eixo);
        else return Coord_Tijolo[ponto-5][eixo-1];
      }

int Tijolo::PontosOK()
     {
      double Teste1=0,Teste2=0;
      if (!Plano::PontosOK())
        return false
      else
       {
        for (int i=1; i<=3; i++)
          {
           Teste1+=(GetCoord(5,i)-GetCoord(1,i))*(GetCoord(1,i)-
GetCoord(3,i));
           Teste2+=pow((GetCoord(1,i)-GetCoord(3,i)),2);
          }
        Teste1/=sqrt(Teste2);
        if (fabs(Teste1)>0.05)
          return false
        else return true;
       }
     }

void Tijolo::CalculaPonto4()
      {
       double XNum,XDen,T2,x,y,z;
       XDen=(GetCoord(2,2)-GetCoord(1,2))*(GetCoord(3,1)-GetCoord(2,1))+
          (GetCoord(2,2)-GetCoord(3,2))*(GetCoord(2,1)-GetCoord(1,1));
       XNum=(GetCoord(1,2)-GetCoord(3,2))*(GetCoord(2,1)-GetCoord(1,1))+
          (GetCoord(1,2)-GetCoord(2,2))*(GetCoord(1,1)-GetCoord(3,1));
       if (fabs(XDen)<1e-5)
       T2=1.0;
       else T2=XNum/XDen;
       x=GetCoord(1,1)+(GetCoord(3,1)-GetCoord(2,1))*T2;
       y=GetCoord(1,2)+(GetCoord(3,2)-GetCoord(2,2))*T2;
       z=GetCoord(1,3)+(GetCoord(3,3)-GetCoord(2,3))*T2;
       SetPonto(4,x,y,z);
      }

void Tijolo::CalculaPontos()
      {
       double DX,DY,DZ,x,y,z;
       CalculaPonto4();
       DX=GetCoord(5,1)-GetCoord(1,1);
       DY=GetCoord(5,2)-GetCoord(1,2);
       DZ=GetCoord(5,3)-GetCoord(1,3);
       for (int i=6; i<=8; i++)
       {
        x=GetCoord(i-4,1)+DX;
        y=GetCoord(i-4,2)+DY;
        z=GetCoord(i-4,3)+DZ;
        SetPonto(i,x,y,z);
       }
      }
```

```
void Tijolo::GravaCorrente(fstream& arq)
      {
       Plano::GravaCorrente(arq);
       int i,j;
       for (i=0; i<4; i++)
       {
        for (j=0; j<3; j++)
          arq << Coord_Tijolo[i][j] << ' ';
        arq << '\n';
       }
      }

void Tijolo::LeCorrente(fstream& arq)
      {
       Plano::LeCorrente(arq);
       int i,j;
       char str[80];
       for (i=0; i<4; i++)
       {
        for (j=0; j<3; j++)
          arq >> Coord_Tijolo[i][j];
        arq.getline(str,80,'\n');
       }
      }


// no' para Cilindro - Geometria Tridimensional

Cilindro::Cilindro()
   { Centro[0]=Centro[1]=Centro[2]=0.0; SetTipo(6); }

void Cilindro::SetCentro (double x, double y, double z)
     { Centro[0]=x; Centro[1]=y; Centro[2]=z; }

double Cilindro::GetCentro(int eixo)
     {
      if ((eixo<1) || (eixo>3))
      {
       cout << "Erro no Parametro de GetCentro(eixo)\n";
       exit(1);
      }
      return Centro[eixo-1];
     }

void Cilindro::GravaCorrente(fstream& arq)
   { Tijolo::GravaCorrente(arq);
     arq << Centro[0] << ' ' << Centro[1] << ' ' << Centro[2] << '\n'; }

void Cilindro::LeCorrente(fstream& arq)
   { char str[80];
     Tijolo::LeCorrente(arq);
     arq >> Centro[0] >> Centro[1] >> Centro[2];
     arq.getline(str,80,'\n'); }

int Cilindro::PontosOK()
   {
    if (!Tijolo::PontosOK())
      return false
    else
      {
       double R1=0,R2=0;
```

```
        for (int i=1; i<=3; i++)
        {
         R1+=pow((GetCoord(2,i)-GetCentro(i)),2);
         R2+=pow((GetCoord(6,i)-GetCentro(i)),2);
        }
        R1=sqrt(R1);
        R2=sqrt(R2);
        if (fabs(R1-R2)/R1>0.01)
        return false
        else return true;
       }
   }

void Cilindro::CalculaPontos()
    {
     double R1=0,R2=0,R3[3],XMOD1,XMOD2,DX,DY,DZ;
     double P5x,P5y,P5z,P8x,P8y,P8z,P7[3];
     CalculaPonto4();
     for (int i=1; i<=3; i++)
       {
      R3[i]=GetCoord(6,i)-GetCoord(2,i);
      R1+=pow((GetCoord(1,i)-GetCentro(i)),2);
      R2+=pow((GetCoord(6,i)-GetCentro(i)),2);
       }
     XMOD1=sqrt(R1);
     XMOD2=sqrt(R2);
     DX=R3[0]*XMOD1/XMOD2;
     P5x=GetCoord(1,1)+DX;
     P8x=GetCoord(4,1)+DX;
     DY=R3[1]*XMOD1/XMOD2;
     P5y=GetCoord(1,2)+DY;
     P8y=GetCoord(4,2)+DY;
     DZ=R3[2]*XMOD1/XMOD2;
     P5z=GetCoord(1,3)+DZ;
     P8z=GetCoord(4,3)+DZ;
     for (i=0; i<3; i++)
       P7[i]=GetCoord(3,i+1)+R3[i];
    SetPonto(5,P5x,P5y,P5z);
    SetPonto(7,P7[0],P7[1],P7[2]);
    SetPonto(8,P8x,P8y,P8z);
    }


// no' para Topologia - Geometria Tridimensional

I_Bloco::I_Bloco()
  { MXi=MXf=MYi=MYf=MZi=MZf=0; Direcao=1; SetTipo(7); }

I_Bloco::I_Bloco(int xi, int xf, int yi, int yf, int zi, int zf, int dir,
double corr):
  NodeCorrente(corr) { MXi=xi; MXf=xf; MYi=yi; MYf=yf; MZi=zi; MZf=zf;
Direcao=dir; SetTipo(7); }

void I_Bloco::SetI_Bloco(int xi, int xf, int yi, int yf, int zi, int zf,
int dir)
  { MXi=xi; MXf=xf; MYi=yi; MYf=yf; MZi=zi; MZf=zf; Direcao=dir; }

int I_Bloco::GetMXi()
     { return MXi; }

int I_Bloco::GetMXf()
```

```
     { return MXf; }

int I_Bloco::GetMYi()
     { return MYi; }

int I_Bloco::GetMYf()
     { return MYf; }

int I_Bloco::GetMZi()
     { return MZi; }

int I_Bloco::GetMZf()
     { return MZf; }

int I_Bloco::GetDirecao()
     { return Direcao; }

void I_Bloco::GravaCorrente(fstream& arq)
     {
      NodeCorrente::GravaCorrente(arq);
      arq << MXi << ' ' << MXf << ' ' << MYi << ' ' << MYf << ' '
        << MZi << ' ' << MZf << ' ' << Direcao << '\n';
     }

void I_Bloco::LeCorrente(fstream& arq)
     {
      char str[80];
      NodeCorrente::LeCorrente(arq);
      arq >> MXi >> MXf >> MYi >> MYf >> MZi >> MZf >> Direcao;
      arq.getline(str,80,'\n');
     }

// Lista de geometrias de corrente

ListCorrente::ListCorrente()
  {
  }

PNodeCorrente ListCorrente::Node(int item)
  {
   ListNodePtr p=List::Node(item);
   return (PNodeCorrente) p;
  }

void ListCorrente::Write(fstream& arq)
  {
   PNodeCorrente Pl;
   ListNodePtr P = First();
   arq << Count() << '\n';
   while (P!=NULL) {
      Pl=(PNodeCorrente) P;
      arq << Pl->GetTipo() << '\n';
      Pl->GravaCorrente(arq);
      P=Next(P);
    }
  }

void ListCorrente::Read(fstream& arq)
  {
   int n, tipo;
   char str[80];
```

```
   FreeAll();
   PNodeCorrente Pl;
   arq >> n;
   arq.getline(str,80,'\n');
   for(int i=0; i<n; i++)
     {
      arq >> tipo;
      arq.getline(str,80,'\n');
      switch (tipo) {
      case 1: Append( new Fio() );
          break;
      case 2: Append( new Arco() );
          break;
      case 3: Append( new Plano() );
          break;
      case 4: Append( new Casca() );
          break;
      case 5: Append( new Tijolo() );
          break;
      case 6: Append( new Cilindro() );
          break;
      case 7: Append( new I_Bloco() );
          break;
      }
      Pl=(PNodeCorrente) Last();
      Pl->LeCorrente(arq);
     }
  }


/////////////////////////////////////////////////////
//           Classes de Materiais
/////////////////////////////////////////////////////

// No' da lista de curva BH

NodeCurvaBH::NodeCurvaBH()
     { B=0.0; H=0.0; }

NodeCurvaBH::NodeCurvaBH(double b, double h)
     { B=b; H=h; }

void NodeCurvaBH::SetNodeCurvaBH(double b, double h)
     { B=b; H=h; }

double NodeCurvaBH::GetB()
     { return B; }

double NodeCurvaBH::GetH()
     { return H; }

void GravaNodeCurvaBH(ListNodePtr P, fstream& arq)
  {
   PNodeCurvaBH Ptr=(PNodeCurvaBH) P;
   arq << Ptr->GetB() << ' ' << Ptr->GetH() << '\n';
  }

void LeNodeCurvaBH(ListNodePtr P, fstream& arq)
  {
   double b,h;
   char str[80];
```

```
   PNodeCurvaBH Ptr=(PNodeCurvaBH) P;
   arq >> b >> h;
   arq.getline(str,80,'\n');
   Ptr->SetNodeCurvaBH(b,h);
  }


// Lista de geometrias de corrente

ListCurvaBH::ListCurvaBH()
  {
  }

void ListCurvaBH::AppendBH(double b, double h)
     { List::Append(new NodeCurvaBH(b,h)); }

PNodeCurvaBH ListCurvaBH::Node(int item)
  {
   ListNodePtr p=List::Node(item);
   return (PNodeCurvaBH) p;
  }

void ListCurvaBH::Write(fstream& arq)
  {
   arq << Count() << '\n';
   ForEachArq( &GravaNodeCurvaBH, arq );
  }

void ListCurvaBH::Read(fstream& arq)
  {
   int n;
   FreeAll();
   arq >> n;
   for (int i=0; i<n; i++)
     List::Append( new NodeCurvaBH() );
   ForEachArq( &LeNodeCurvaBH, arq );
  }


// no' base da lista de materiais

NodeMaterial::NodeMaterial()
     {
       strcpy(Label,""); Permeabilidade=0.0; Condutividade=0.0;
       Bcx=Bcy=Bcz=0.0;
       Beta0=Roc0=Xk0=CNT1=CNT2=CNT3=CNT4=CNT5=TCurie=Cond1=Beta1=0.0;
       Curva = new ListCurvaBH();
     }

NodeMaterial::NodeMaterial(double perm, double cond, double bx, double
by,
                     double bz, double bet0, double roc, double xk,
                     double cn1,    double cn2, double cn3, double cn4,
                     double cn5, double tc, double cond1, double bet1)
     {
       strcpy(Label,"");
       Permeabilidade=perm; Condutividade=cond; Bcx=bx; Bcy=by; Bcz=bz;
       Beta0=bet0; Roc0=roc; Xk0=xk; CNT1=cn1; CNT2=cn2; CNT3=cn3;
       CNT4=cn4; CNT5=cn5; TCurie=tc; Cond1=cond1; Beta1=bet1;
       Curva = new ListCurvaBH();
     }
```

```
NodeMaterial::~NodeMaterial()
     { Curva->FreeAll(); delete Curva; }

void NodeMaterial::SetLabel(char *label)
     { strcpy(Label,label); }

void NodeMaterial::SetMaterial(double perm, double cond, double bx,
double by,
                     double bz, double bet0, double roc, double xk,
                     double cn1,    double cn2, double cn3, double cn4,
                     double cn5, double tc, double cond1, double bet1)
     {
       Permeabilidade=perm; Condutividade=cond; Bcx=bx; Bcy=by; Bcz=bz;
       Beta0=bet0; Roc0=roc; Xk0=xk; CNT1=cn1; CNT2=cn2; CNT3=cn3;
       CNT4=cn4; CNT5=cn5; TCurie=tc; Cond1=cond1; Beta1=bet1;
     }

void NodeMaterial::SetPermeabilidade(double perm)
     { Permeabilidade=perm; }

void NodeMaterial::SetCondutividade(double cond)
     { Condutividade=cond; }

void NodeMaterial::SetImantacao(double bx, double by, double bz)
     { Bcx=bx; Bcy=by; Bcz=bz; }

void NodeMaterial::SetTermico(double bet0, double roc, double xk,double
cn1,
                        double cn2, double cn3, double cn4, double cn5,
                        double tc, double cond1, double bet1)
    {
     Beta0=bet0; Roc0=roc; Xk0=xk; CNT1=cn1; CNT2=cn2; CNT3=cn3;
     CNT4=cn4; CNT5=cn5; TCurie=tc; Cond1=cond1; Beta1=bet1;
    }

char *NodeMaterial::GetLabel()
     { return Label; }

double NodeMaterial::GetPermeabilidade()
     { return Permeabilidade; }

double NodeMaterial::GetCondutividade()
     { return Condutividade; }

double NodeMaterial::GetBx()
     { return Bcx; }

double NodeMaterial::GetBy()
     { return Bcy; }

double NodeMaterial::GetBz()
     { return Bcz; }

double NodeMaterial::GetBeta0()
     { return Beta0; }

double NodeMaterial::GetRoc0()
     { return Roc0; }

double NodeMaterial::GetXk0()
```

```
     { return Xk0; }

double NodeMaterial::GetCNT1()
     { return CNT1; }

double NodeMaterial::GetCNT2()
     { return CNT2; }

double NodeMaterial::GetCNT3()
     { return CNT3; }

double NodeMaterial::GetCNT4()
     { return CNT4; }

double NodeMaterial::GetCNT5()
     { return CNT5; }

double NodeMaterial::GetTCurie()
     { return TCurie; }

double NodeMaterial::GetCond1()
     { return Cond1; }

double NodeMaterial::GetBeta1()
     { return Beta1; }

PListCurvaBH NodeMaterial::CurvaBH()
   {
    return Curva;
   }


// Lista de materiais

ListMaterial::ListMaterial()
  {
  }

PNodeMaterial ListMaterial::Node(int item)
  {
   ListNodePtr p=List::Node(item);
   return (PNodeMaterial) p;
  }

void ListMaterial::Write(fstream &arq)
  {
   int i,j,NumMat=Count();

   arq << "*MATERIAL\n" << NumMat << endl << endl;

   arq << "*MATERIAL.LABEL\n";
   j=0;
   for (i=0; i<NumMat; i++)
     if (strcmp(Node(i+1)->GetLabel(),""))
       j++;
   arq << j << endl;
   for (i=0; i< NumMat; i++)
     if (strcmp(Node(i+1)->GetLabel(),""))
       arq << i+1 << "  " << Node(i+1)->GetLabel() << endl;
   arq << endl;
```

```
   arq << "*MATERIAL.PROPERTY.SATURATED.BH\n";
   j=0;
   for (i=0; i<NumMat; i++)
     if ((Node(i+1)->CurvaBH()->Count())!=0)
       j++;
   arq << j << endl;
   for (i=0; i<NumMat; i++)
     if ((Node(i+1)->CurvaBH()->Count())!=0)
       {
      arq << i+1 << "  " << Node(i+1)->CurvaBH()->Count() << "  0.0\n";
      for (j=0; j<(Node(i+1)->CurvaBH()->Count()); j++)
        arq << j+1 << "  " << Node(i+1)->CurvaBH()->Node(j+1)->GetB()
            << "  " << Node(i+1)->CurvaBH()->Node(j+1)->GetH() << '\n';
       }
   arq << endl;

   arq << "*MATERIAL.PROPERTY.ISOTROPIC.PERMEABILITY\n";
   arq << NumMat <<endl;
   for (i=0; i<NumMat; i++)
     arq << i+1 << "  " << Node(i+1)->GetPermeabilidade() << endl;
   arq << endl;

   arq << "*MATERIAL.PROPERTY.ISOTROPIC.CONDUCTIVITY\n";
   j=0;
   for (i=0; i<NumMat; i++)
     if (Node(i+1)->GetCondutividade()!=0.0)
       j++;
   arq << j << endl;
   for (i=0; i<NumMat; i++)
     if (Node(i+1)->GetCondutividade()!=0.0)
       arq << i+1 << "  " << Node(i+1)->GetCondutividade() << endl;
   arq << endl;

   arq << "*MATERIAL.PROPERTY.MAGNETIC\n";
   j=0;
   for (i=0; i<NumMat; i++)
     if ((Node(i+1)->GetBx()!=0.0) || (Node(i+1)->GetBy()!=0.0) ||
       (Node(i+1)->GetBz()!=0.0))
       j++;
   arq << j << endl;
   for (i=0; i<NumMat; i++)
     if ((Node(i+1)->GetBx()!=0.0) || (Node(i+1)->GetBy()!=0.0) ||
       (Node(i+1)->GetBz()!=0.0))
       arq << i+1 << "  " << Node(i+1)->GetBx() << "  " << Node(i+1)-
>GetBy() << "  "
         << Node(i+1)->GetBz() << endl;
   arq << endl;

   arq << "*MATERIAL.PROPERTY.TERMIC.CONDUCTIVITY\n";
   j=0;
   for (i=0; i<NumMat; i++)
     if (Node(i+1)->GetXk0()!=0.0)
       j++;
   arq << j << endl;
   for (i=0; i<NumMat; i++)
     if (Node(i+1)->GetXk0()!=0.0)
       arq << i+1 << "  " << Node(i+1)->GetXk0() << endl;
   arq << endl;

  }
```

```
void ListMaterial::Read(fstream &arq)
  {
   char str[80],label[21];
   int Num,Num2,i,j,id,id2;
   double dbl1,dbl2,dbl3;

   FreeAll();

   PosicionaCursorArq("*MATERIAL",arq);
   arq >> Num;
   arq.getline(str,80,'\n');
   for (i=0; i<Num; i++)
     Append( new NodeMaterial());

   PosicionaCursorArq("*MATERIAL.LABEL",arq);
   arq >> Num;
   arq.getline(str,80,'\n');
   for (i=0; i<Num; i++)
     {
      arq >> id;
      SaltaEspacos(arq);
      arq.getline(label,20,'\n');
      Node(id)->SetLabel(label);
     }

   PosicionaCursorArq("*MATERIAL.PROPERTY.SATURATED.BH",arq);
   arq >> Num;
   arq.getline(str,80,'\n');
   for (i=0; i<Num; i++)
     {
      arq >> id >> Num2;
      arq.getline(str,80,'\n');
      for (j=0; j<Num2; j++)
      Node(id)->CurvaBH()->Append( new NodeCurvaBH() );
      for (j=0; j<Num2; j++)
      {
       arq >> id2 >> dbl1 >> dbl2;
       arq.getline(str,80,'\n');
       Node(id)->CurvaBH()->Node(id2)->SetNodeCurvaBH(dbl1,dbl2);
      }
     }

   PosicionaCursorArq("*MATERIAL.PROPERTY.ISOTROPIC.PERMEABILITY",arq);
   arq >> Num;
   arq.getline(str,80,'\n');
   for (i=0; i<Num; i++)
     {
      arq >> id >> dbl1;
      arq.getline(str,80,'\n');
      Node(id)->SetPermeabilidade(dbl1);
     }

   PosicionaCursorArq("*MATERIAL.PROPERTY.ISOTROPIC.CONDUCTIVITY",arq);
   arq >> Num;
   arq.getline(str,80,'\n');
   for (i=0; i<Num; i++)
     {
      arq >> id >> dbl1;
      arq.getline(str,80,'\n');
      Node(id)->SetCondutividade(dbl1);
```

```
      }

   PosicionaCursorArq("*MATERIAL.PROPERTY.MAGNETIC",arq);
   arq >> Num;
   arq.getline(str,80,'\n');
   for (i=0; i<Num; i++)
      {
       arq >> id >> dbl1 >> dbl2 >> dbl3;
       arq.getline(str,80,'\n');
       Node(id)->SetImantacao(dbl1,dbl2,dbl3);
      }

   PosicionaCursorArq("*MATERIAL.PROPERTY.TERMIC.CONDUCTIVITY",arq);
   arq >> Num;
   arq.getline(str,80,'\n');
   for (i=0; i<Num; i++)
      {
       arq >> id >> dbl1;
       arq.getline(str,80,'\n');
       PNodeMaterial P=Node(id);
       P->SetTermico(P->GetBeta0(), P->GetRoc0(), dbl1, P->GetCNT1(),
                 P->GetCNT2(), P->GetCNT3(), P->GetCNT4(), P->GetCNT5(),
                 P->GetTCurie(), P->GetCond1(), P->GetBeta1());
      }

  }



//////////////////////////////////////////
// Objeto principal da estrutura de dados
//////////////////////////////////////////

Preprocessador::Preprocessador()
{
 strcpy(Titulo,"");
 CamadasX=CamadasY=CamadasZ=0;        // Camadas
 TipoPeriodo=0;     // Tipo de Periodicidade
 MaterialDefault=0;
 PotencialDefault=1;
 Blocos= new ListBloco();
 CondContorno= new ListCondContorno();
 Malha = new MalhaRegular();
 Correntes= new ListCorrente();
 Materiais= new ListMaterial();
}

Preprocessador::~Preprocessador()
{
 delete Blocos;
 delete CondContorno;
 delete Malha;
 delete Correntes;
 delete Materiais;
}

void Preprocessador::InicializaDados()
      {
      strcpy(Titulo,"");
      CamadasX=CamadasY=CamadasZ=0;
      TipoPeriodo=0;
```

```
      MaterialDefault=0;
      PotencialDefault=1;
      Blocos->FreeAll();
      CondContorno->FreeAll();
      Malha->GetPtrEixo(1)->FreeAll();
      Malha->GetPtrEixo(2)->FreeAll();
      Malha->GetPtrEixo(3)->FreeAll();
      Correntes->FreeAll();
      Materiais->FreeAll();
      }

void Preprocessador::SetTitulo(char *titl)
      { strcpy(Titulo,titl); }

char *Preprocessador::GetTitulo()
      { return Titulo; }

void Preprocessador::SetCamadas(int x, int y, int z)
     { CamadasX=x; CamadasY=y; CamadasZ=z; }

int Preprocessador::GetCamadasX()
     { return CamadasX; }

int Preprocessador::GetCamadasY()
     { return CamadasY; }

int Preprocessador::GetCamadasZ()
     { return CamadasZ; }

void Preprocessador::SetTipoPeriodo (int tipo)
     { TipoPeriodo=tipo; }

int Preprocessador::GetTipoPeriodo()
     { return TipoPeriodo; }

void Preprocessador::SetMaterialDefault(int matdef)
      { MaterialDefault=matdef; }

int Preprocessador::GetMaterialDefault()
     { return MaterialDefault; }

void Preprocessador::SetPotencialDefault(int potdef)
      { PotencialDefault=potdef; }

int Preprocessador::GetPotencialDefault()
     { return PotencialDefault; }

void Preprocessador::GravaDados(char *NomeArq)
  {
   fstream arq(NomeArq,ios::out);

   arq << "#TITULO\n" << Titulo << '\n';
   arq << "#CAMADAS\n" << CamadasX << ' ' << CamadasY << ' ' << CamadasZ
<< '\n';
   arq << "#TIPO_PERIODO\n" << TipoPeriodo << '\n';
   arq << "#MATERIAL_DEF\n" << MaterialDefault << '\n';
   arq << "#POTENCIAL_DEF\n" << PotencialDefault << '\n';
   arq << "#BLOCOS\n";
   Blocos->Write(arq);
   arq << "#COND_CONTORNO\n";
   CondContorno->Write(arq);
```

```
    arq << "#MALHA REGULAR\n";
    Malha->Write(arq);
    arq << "#CORRENTES\n";
    Correntes->Write(arq);
    arq.close();
   }

void Preprocessador::LeDados(char *NomeArq)
   {
    fstream arq(NomeArq,ios::in);
    char str[80];

    arq.getline(str,80,'\n');
    arq.getline(Titulo,31,'\n');
    arq.getline(str,80,'\n');
    arq >> CamadasX >> CamadasY >> CamadasZ;
    arq.getline(str,80,'\n');
    arq.getline(str,80,'\n');
    arq >> TipoPeriodo;
    arq.getline(str,80,'\n');
    arq.getline(str,80,'\n');
    arq >> MaterialDefault;
    arq.getline(str,80,'\n');
    arq.getline(str,80,'\n');
    arq >> PotencialDefault;
    arq.getline(str,80,'\n');
    arq.getline(str,80,'\n');
    Blocos->Read(arq);
    arq.getline(str,80,'\n');
    CondContorno->Read(arq);
    arq.getline(str,80,'\n');
    Malha->Read(arq);
    arq.getline(str,80,'\n');
    Correntes->Read(arq);
    arq.close();
   }

void Preprocessador::GeraArqMat(char *NomeArq)
   {
    fstream arq(NomeArq,ios::out);
    Materiais->Write(arq);
    arq.close();
   }

void Preprocessador::LeArqMat(char *NomeArq)
   {
    fstream arq(NomeArq,ios::in);
    Materiais->Read(arq);
    arq.close();
   }
```

C.5 - Arquivo Exemplo.cpp

```
#include <stdlib.h>
#include <string.h>
#include "classes.h"


void MostraNodeBloco(ListNodePtr P)
  {
    PNodeBloco Ptr=(PNodeBloco) P;
    cout << "Xi=" << Ptr->GetMXi() << "  Xf=" << Ptr->GetMXf()
       << "  Yi=" << Ptr->GetMYi() << "  Yf=" << Ptr->GetMYf()
       << "  Zi=" << Ptr->GetMZi() << "  Zf=" << Ptr->GetMZf()
       << "\nMaterial=" << Ptr->GetMaterial() << "  Potencial="
       << Ptr->GetPotencial() << "\n";
  }


void MostraCondContorno(ListNodePtr P)
  {
    PNodeCondContorno Ptr=(PNodeCondContorno) P;
    cout << "Xi=" << Ptr->GetNXi() << "  Xf=" << Ptr->GetNXf()
       << "  Yi=" << Ptr->GetNYi() << "  Yf=" << Ptr->GetNYf()
       << "  Zi=" << Ptr->GetNZi() << "  Zf=" << Ptr->GetNZf()
       << "\nTipo=" << Ptr->GetTipo() << "\n";
  }


void MostraNodeMalhaR(ListNodePtr P)
  {
   PNodeMalhaRegular Ptr=(PNodeMalhaRegular) P;
   cout << "Dim=" << Ptr->GetDimensao() << " ;  Num. Div="
    << Ptr->GetNumDivisoes() << endl;
  }


void MostraNodeCurvaBH(ListNodePtr P)
  {
   PNodeCurvaBH Ptr=(PNodeCurvaBH) P;
   cout << "B = " << Ptr->GetB() << ";  H = " << Ptr->GetH() << endl;
  }


void main()
{
 char Nome[]="dados.dat";
 char Nome1[]="exemBDFN";
 ListNodePtr PL;

 cout << "\n\n\n*************Inicio do processamento************\n\n";

// ***************** Inicializacao ***********************
 Preprocessador Mesh;
 Mesh.SetTitulo("Caso Exemplo");

// ******************** Camadas ***************************
 Mesh.SetCamadas(3,2,2);

// ********************** Blocos ************************
{
 Mesh.Blocos->Append( new NodeBloco(1,1,1,1,1,1,10,1) );
 Mesh.Blocos->Node(1)->SetLabel("Bloco 1");
 Mesh.Blocos->Append( new NodeBloco(2,2,1,1,1,1,20,2) );
```

```
 Mesh.Blocos->Append( new NodeBloco(3,3,1,1,1,1,100,10) );
 PNodeBloco PP=(PNodeBloco) Mesh.Blocos->Node(2);
 cout << "Num. de Blocos: " << Mesh.Blocos->Count() << endl;
 cout << Mesh.Blocos->Node(1)->GetMaterial() << endl;
 cout << PP->GetMXi() << "  " << PP->GetMaterial() << endl;
 Mesh.Blocos->ForEach( &MostraNodeBloco );
}

// ***************** Cond Contornos *********************
 {
 Mesh.CondContorno->Append( new CCEscalar(1,2,1,2,1,2,1.01) );
 Mesh.CondContorno->Node(1)->SetLabel("CC 1");
 Mesh.CondContorno->Append( new CCUs(2,1,2,1,2,1,2.02) );
 Mesh.CondContorno->Node(2)->SetLabel("CC 2");
 cout << "Num. de Contornos: " << Mesh.CondContorno->Count() << endl;
 Mesh.CondContorno->ForEach( &MostraCondContorno );
 PCCEscalar P;
 PL=Mesh.CondContorno->Node(1);
 P=(PCCEscalar) PL;
 cout << P->GetPotencial() << endl;
 PCCUs P1;
 PL=Mesh.CondContorno->Node(2);
 P1=(PCCUs) PL;
 cout << P1->GetPotencial() << endl;
 }

// ****************** Malha Regular **********************
 Mesh.Malha->Append(1, 0.0, 1);
 Mesh.Malha->Append(1, 0.5, 2);
 Mesh.Malha->Append(1, 1.0, 0);
 Mesh.Malha->Append(2, 0.0, 1);
 Mesh.Malha->Append(2, 0.5, 1);
 Mesh.Malha->Append(2, 1.0, 0);
 Mesh.Malha->Append(3, 0.0, 1);
 Mesh.Malha->Append(3, 0.5, 1);
 Mesh.Malha->Append(3, 1.0, 0);
 cout << "Malha Regular\n";
 cout << "Direcao X\n";
 Mesh.Malha->GetPtrEixo(1)->ForEach( &MostraNodeMalhaR );
 cout << "Direcao Y\n";
 Mesh.Malha->GetPtrEixo(2)->ForEach( &MostraNodeMalhaR );
 cout << "Direcao Z\n";
 Mesh.Malha->GetPtrEixo(3)->ForEach( &MostraNodeMalhaR );
 cout << Mesh.Malha->GetNumDivisoesNode(1,2) << endl;
 cout << Mesh.Malha->GetDimensaoNode(2,2) << endl;

// ****************** Correntes ***********************
 {
 Mesh.Correntes->Append(new Fio());
 PFio P;
 PL=Mesh.Correntes->Node(1);
 P=(PFio) PL;
 P->SetPonto(1,10.0,20.0,30.0);
 P->SetPonto(2,11.0,21.0,31.0);
 P->SetCorrente(1.10);
 P->SetLabel("Fio 1");
 cout << "\nCorrente - Fio Unidimensional\n";
 cout << "P1 = " << P->GetCoord(1,1) << ' ' << P->GetCoord(1,2) << ' ' <<
P->GetCoord(1,3)
      << "\nP2 = " << P->GetCoord(2,1) << ' ' << P->GetCoord(2,2) << ' '
<< P->GetCoord(2,3) << endl;
```

```
 cout << "I = " << P->GetCorrente() << endl;
 cout << "Label: " << P->GetLabel() << "  " << strlen(P->GetLabel()) <<
endl;
 }
 {
 Mesh.Correntes->Append(new I_Bloco());
 PI_Bloco P;
 PL=Mesh.Correntes->Last();
 P=(PI_Bloco) PL;
 P->SetI_Bloco(1,2,3,4,5,6,10);
 cout << "\nCorrente - Topologica\n";
 cout << "P = " << P->GetMXi() << ' ' << P->GetMXf() << ' ' << P-
>GetMYi()
      << ' ' << P->GetMYf() << ' ' << P->GetMZi() << ' ' << P->GetMZf()
<< endl;
 cout << "Dir = " << P->GetDirecao() << endl;
 }
 {
 Mesh.Correntes->Append(new Cilindro());
 PCilindro P;
 PL=Mesh.Correntes->Node(1);
 P=(PCilindro) PL;
 P->SetPonto(1,10.0,20.0,30.0);
 P->SetPonto(2,11.0,21.0,31.0);
 P->SetPonto(3,12.0,22.0,32.0);
 P->SetPonto(6,13.0,23.0,33.0);
 P->SetCentro(-1,-2,-3);
 P->SetCorrente(1.10);
 P->CalculaPontos();
 if (!P->PontosOK()) exit(1);
 cout << "\nCorrente - Cilindro Tridimensional\n";
 cout << "P1 = " << P->GetCoord(1,1) << ' ' << P->GetCoord(1,2) << ' ' <<
P->GetCoord(1,3)
      << "\nP2 = " << P->GetCoord(2,1) << ' ' << P->GetCoord(2,2) << ' '
<< P->GetCoord(2,3) << endl;
 cout << "P1 = " << P->GetCoord(3,1) << ' ' << P->GetCoord(3,2) << ' ' <<
P->GetCoord(3,3)
      << "\nP2 = " << P->GetCoord(6,1) << ' ' << P->GetCoord(6,2) << ' '
<< P->GetCoord(6,3) << endl;
 cout << "C = " << P->GetCentro(1) << ' ' << P->GetCentro(2) << ' ' << P-
>GetCentro(3) << endl;
 cout << "I = " << P->GetCorrente() << endl;
 }

// ***************** Materiais ************************
 Mesh.Materiais->Append(new
      NodeMaterial(1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,
                   10.0,11.1,12.2,13.3,14.4,15.5,16.6));
 Mesh.Materiais->Append(new NodeMaterial());

 cout << "\n\nNumero de Materiais = " << Mesh.Materiais->Count() << endl;
 PL=Mesh.Materiais->First();
 PNodeMaterial PT = (PNodeMaterial) PL;
 PT->SetLabel("Material Exemplo 1");
 PT->CurvaBH()->AppendBH(1.01,2.02);
 PT->CurvaBH()->AppendBH(3.03,4.04);
 PT->CurvaBH()->AppendBH(5.05,6.06);
 cout << "Descricao = " << PT->GetLabel() << "  " << strlen(PT-
>GetLabel()) << endl;
 cout << "Permeabilidade = " << PT->GetPermeabilidade() << endl;
 cout << "Bx = " << PT->GetBx() << endl;
```

```
 cout << "By = " << PT->GetBy() << endl;
 cout << "Bz = " << PT->GetBz() << endl << endl;
 cout << "Curva BxH:\n";
 PT->CurvaBH()->ForEach( &MostraNodeCurvaBH );

 PL=Mesh.Materiais->Node(2);
 PT = (PNodeMaterial) PL;
 PT->SetLabel("Material Exemplo 2");
 cout << "\n\nDescricao = " << PT->GetLabel() << "  " << strlen(PT-
>GetLabel()) << endl;
 cout << "Permeabilidade = " << PT->GetPermeabilidade() << endl;
 cout << "Bx = " << PT->GetBx() << endl;
 cout << "By = " << PT->GetBy() << endl;
 cout << "Bz = " << PT->GetBz() << endl << endl;
 cout << "Curva BxH:\n";


// ***************** Grava Dados ***********************
 cout << "Gravando Dados...\n";
 Mesh.GravaDados(Nome);

// ***************** Limpa Dados ********************
 cout << "Inicializa Dados...\n";
 Mesh.InicializaDados();

// ****************** Le Dados *******************
 cout << "Lendo Dados...\n";
 Mesh.LeDados(Nome);

// ****************** Gera BD Formato Neutro ***************
 Mesh.GeraArquivoNeutro(Nome1);

}
```