Universidade Federal de Minas Gerais

"Desenvolvimento de um Sistema Parametrizado de Projeto Assistido por Computador"

Mestrado em Engenharia Elétrica
Túlio Pinheiro Pessoa de Mendonça

AGRADECIMENTOS

Ao Prof. Renato C. Mesquita, pela sua disposição, seu incentivo, acompanhamento e amizade durante estes dois anos e meio do Mestrado;

Aos Profs. Rodney R. Saldanha e João A. Vasconcelos, por sua amizade e incentivo durante todo o curso de Mestrado;

Aos demais professores e funcionários do CPDEE, que direta ou indiretamente me ajudaram a completar esta missão.

RESUMO

Neste trabalho é apresentado um sistema parametrizado para o projeto assistido por computador que compõe, junto a outros módulos, o pré-processador de um sistema computacional de cálculo de campos eletromagnéticos.

Este sistema possibilita maior facilidade e automatização no projeto de dispositivos eletromagnéticos porque parametriza pontos da geometria, criando expressões matemáticas para os mesmos. Além disso, este sistema proporciona ferramentas que podem ser utilizadas na otimização destes projetos.

O sistema parametrizado é composto de três subsistemas: um avaliador de expressões, responsável pelo processamento e pelo armazenamento interno em tempo de execução das expressões e parâmetros do sistema parametrizado; o subsistema de interface, responsável pela interface com o usuário e o subsistema de armazenamento, responsável pelo armazenamento dos dados inseridos pelo usuário.

O sistema parametrizado, em conjunto com os sistemas de processamento, baseado no Método de Elementos Finitos, e otimização constitui um sistema computacional completo para o projeto de dispositivos eletromagnéticos.

ABSTRACT

This work presents a parametric system to be used in a computer aided design process.

This system provides facilities to the design of electromagnetic devices since points of geometry can be parametrized with mathematic expressions. Moreover, it provides tools that can be used to optimise the design.

The parametric system is composed of three subsystems. The expression evaluator subsystem is responsible for the evaluation and internal runtime storement of the system's parameters. The interface subsystem makes a graphical interface with the designer. Finally, the storement subsystem records the data inserted by the designer.

The parametric system together with a processor system based in the Finite Element Method and an optimisation system constitute a complete computational system for the design of electromagnetic devices.

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO

Métodos de parametrização surgiram em resposta à crescente demanda industrial por projeto e manufatura, particularmente a necessidade de redução do tempo de desenvolvimento de produtos [Shah, 1995].

A introdução de ferramentas de controle computacional, cerca de 40 anos atrás, criou a necessidade de uma representação computacional das informações de projeto e manufatura empregados pelas indústrias. A introdução de gerações posteriores de tecnologias de produção avançadas como sistemas flexíveis de manufatura, robôs e plantas automatizadas, aumentaram a necessidade por informações completas e precisas sobre os produtos [Shah, 1995].

Ao mesmo tempo, mudanças sociais e econômicas em nossas sociedades e no comércio internacional mudaram significativamente o modo com que as tecnologias de produção estão sendo usadas. Ao invés das até então dominantes fábricas organizadas funcionalmente, operando em isolamento quase total em relação à situação do mercado, os sistemas de produção se tornaram mais orientados ao produto, visando o decréscimo do tempo de resposta ao mercado, a minimização do trabalho em cada processo, fluxo de material *just-in-time*, e alta eficiência e flexibilidade na utilização da capacidade de manufatura. O termo *agile manufacturing* abrange todas estas características [Shah, 1995].

Para implementar a *agile manufacturing*, as funções de projeto de produtos e engenharia de produção devem se tornar o mais integradas possível com a manufatura, e todos os gargalos no fluxo de produto e engenharia de informação dessas funções para a função de manufatura devem ser eliminados. A proximidade de integração entre as funções de projeto, planejamento e manufatura requer que informações suficientemente completas e precisas sobre todos os aspectos dos produtos, processos de produção e

operações estejam disponíveis. Consequentemente, futuros sistemas de projeto e de planejamento serão integrados com tecnologias de manufatura e futuros sistemas de manufatura precisarão de informações mais completas e precisas sobre os produtos do que é possível hoje. Idealmente, esta integração levará à *concurrent life-cycle engineering* [Shah, 1995], onde todas as questões de projeto, de manufatura e de manutenção e de eventual desmantelamento e de reuso dos produtos poderiam ser considerados simultaneamente durante seu projeto.

A necessidade de integração vai além das fronteiras de uma única companhia; há um acréscimo no número de companhias trabalhando em cooperação ou parceria com outras companhias, desenvolvedores e subcontratados, criando o que se convencionou chamar de *virtual enterprise*. A operação ágil destes empreendimentos virtuais se baseará na troca precisa de informações sobre processos e produtos. Junto às características da *agile manufacturing*, os *virtual enterprises* formam a base do que muitos especialistas acreditam que será o próximo paradigma da manufatura dominante a partir do ano 2000 [Shah, 1995].

Estimulado por estes fatos, o GOPAC, Grupo de Otimização e Projeto Assistido por Computador, do Centro de Pesquisa e Desenvolvimento em Engenharia Elétrica da UFMG vem desenvolvendo desde 1995 um sistema CAD destinado ao projeto de dispositivos eletromagnéticos denominado AutoPAC. Este software tem por objetivo final disponibilizar ao projetista todas as ferramentas necessárias ao projeto de um dispositivo eletromagnético, desde a construção de sua geometria até a otimização das características relevantes do dispositivo idealizado.

Este trabalho teve como meta acrescentar ao AutoPAC um sistema parametrizado. Este sistema tem importância fundamental no processo de otimização de um dispositivo eletromagnético pois permite que alguns pontos de interesse de sua geometria sejam definidos como parâmetros que

poderão ser otimizados em uma etapa posterior [Shah, 1995]. Além disso, a criação de um conjunto de geometrias semelhantes pode ser automatizada, mantendo-se o contorno geral da geometria e alterando-se os valores de determinados parâmetros.

A maior parte dos softwares comerciais disponíveis na área projeto de dispositivos eletromagnéticos não apresenta nenhuma ferramenta semelhante ao sistema de parametrização descrito neste trabalho. Estes softwares apresentam ferramentas sofisticadas de modelagem e análise em duas e três dimensões, mas não incluem nenhuma ferramenta de parametrização e otimização [Infolytica].

O capítulo 2 descreve o AutoPAC antes da inclusão do sistema parametrizado. O capítulo 3 descreve o módulo avaliador de expressões, responsável pela criação, processamento e armazenamento dos parâmetros. No capítulo 4 é feita uma descrição do comportamento do AutoPAC após a inclusão do sistema parametrizado, suas características e seu potencial. No capítulo 5 encontram-se exemplos de aplicação do sistema parametrizado.

CAPÍTULO 2 - O AUTOPAC

Este capítulo descreve o AutoPAC, modelador geométrico bidimensional para fins didáticos e de pesquisa que está sendo desenvolvido pelo GOPAC, Grupo de Otimização e Projeto Assistido por Computador, do Centro de Pesquisa em Engenharia Elétrica da UFMG.

Esta parte do trabalho descreve como o modelador bidimensional AutoPAC foi idealizado e está sendo desenvolvido [Souza, 1997].

2.1 Especificação de Requisitos e Funcionalidade

Todos os sistemas de modelagem bidimensional possuem componentes básicos, como uma estrutura de dados para armazenar os esquemas de representação, um conjunto de algoritmos básicos para a manipulação dessas estruturas e mecanismos de interface com o usuário que permitem implementar diferentes técnicas de modelagem e edição [Foley, 1990]. Baseado no estudo de requisitos e funcionalidade para modeladores bidimensionais, foi elaborada uma descrição informal que reflete a funcionalidade, os componentes e os principais requisitos desejados para o AutoPAC.

2.1.1 Descrição Informal

O AutoPAC é um modelador geométrico capaz de suportar vários tipos de primitivas bidimensionais. Permite a criação, edição e acesso à representação de primitivas. Além dos processos ativados pelo usuário durante a fase de modelagem interativa, outros processos que acessam estas representações são as rotinas de visualização, consideradas parte integrante do modelador, que permitem uma melhor observação da configuração geométrica do modelo que está sendo elaborado. Programas aplicativos externos ao modelador podem utilizar as representações geradas para a simulação do modelo, analisando-o de forma estática ou dinâmica para determinar distribuições de campos elétricos

e magnéticos, temperaturas e, a partir destas distribuições, grandezas como indutâncias, forças, torques, resistências, perdas, capacitâncias, etc.

Em uma etapa inicial, o sistema suporta apenas a representação parametrizada das primitivas linha, arco e círculo. Pretende-se futuramente estender o conjunto de primitivas existentes para incluir, por exemplo, poli-linhas, visando o aumento do potencial descritivo do modelador.

O AutoPAC é funcionalmente composto por três subsistemas principais, ilustrados na Fig. 2.1:

- o Subsistema de Interface, responsável por fornecer aos usuários as diversas maneiras de descrever objetos bidimensionais, bem como um conjunto de operações de manipulação da forma que possam ser executadas sobre estes objetos. Este subsistema fornece também recursos para a visualização dos objetos representados. As tarefas a serem realizadas pelo modelador são descritas através da interface, capturadas por procedimentos deste subsistema, que ativam procedimentos dos outros subsistemas; os resultados das tarefas são recebidos pelos procedimentos e exibidos através da interface.
- o Subsistema de Modelagem, responsável por traduzir as descrições dos objetos e operações de manipulação recebidas do Subsistema de Interface em comandos para a geração das representações internas. Este subsistema contém os procedimentos que interpretam a descrição de um novo modelo ou de uma modificação, e a seguir acionam os procedimentos do Subsistema de Representação para criar ou alterar, respectivamente, uma representação.
- O Subsistema de Representação, responsável pela manutenção, gerenciamento e acesso ao conjunto de representações internas (estruturas de dados) admitidas pelo sistema. Este subsistema inclui também funções

para armazenamento da descrição dos objetos e composições em bases de dados permanentes. Cada entidade do AutoPAC possui uma estrutura de dados e procedimentos específicos de manipulação da entidade. As informações das representações internas são lidas por procedimentos dos outros subsistemas, mas são criadas e modificadas apenas pelos procedimentos do Subsistema de Representação.

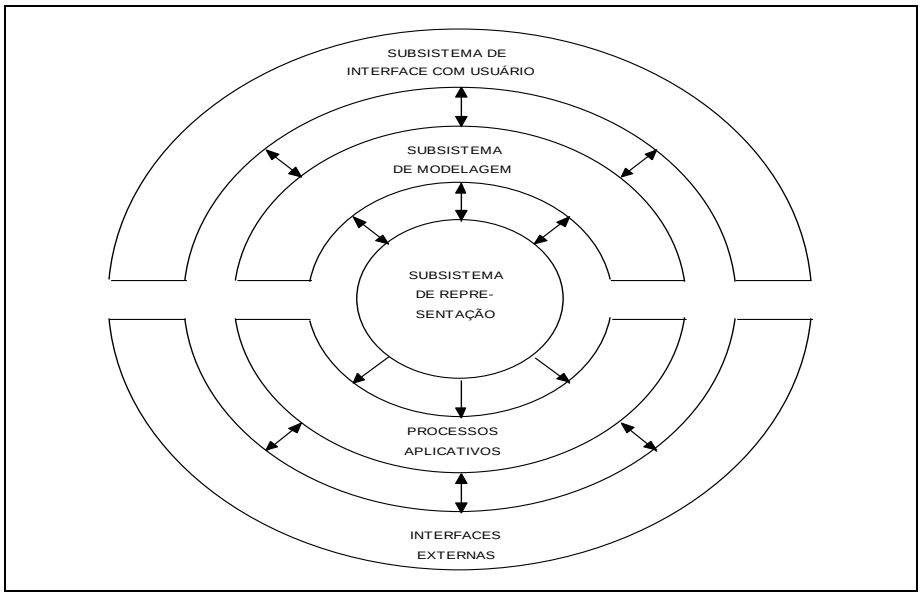

Fig. 2.1 - Estrutura Funcional do AutoPAC

Os usuários interagem com o modelador para descrever e editar objetos através de uma interface gráfica interativa que constitui o Subsistema de Interface. Os comandos recebidos pela Interface são traduzidos em solicitações para o Subsistema de Modelagem, que implementa as técnicas de modelagem disponíveis no sistema e é responsável pela execução das operações de manipulação da forma. Para tal, esse subsistema interage com o Subsistema de Representação, que acessa e altera, quando necessário, as representações de objetos mantidas pelo sistema. Uma composição de objetos gerada pode então ser armazenada sob a forma de um arquivo neutro contendo a descrição geométrica da composição, através do qual programas aplicativos e interfaces externas

poderão utilizar as informações geométricas visando um processamento específico.

Através da interface, o usuário pode incluir primitivas da maneira que lhe for mais conveniente. Inicialmente, estão disponíveis três primitivas básicas (linhas, arcos e círculos), sendo que outras poderão ser acrescentadas à medida em que o modelador evoluir.

Para a elaboração de um modelo, após a inclusão de primitivas, um segundo conjunto de operações ficará disponível para efetuar alterações na forma ou na posição ocupada pelas primitivas. Essas operações são ativadas pela Interface e executadas pelo Subsistema de Modelagem. Entre elas, estão:

- as transformações geométricas, que permitem alterar a posição, orientação e tamanho dos objetos contidos em uma composição, como a translação, a rotação, o escalonamento e o espelhamento;
- as operações locais, que realizam pequenas alterações no modelo definido, ajustando a geometria das arestas ou adicionando poucos elementos à representação interna, modificando uma região localizada da estrutura de dados de uma maneira eficiente, como o chanframento e o arredondamento;
- as operações globais, que particionam um objeto em dois ou mais novos objetos, como as operações de corte de um objeto em relação a um outro e a de quebra de um objeto.

Todas as operações são ativadas pela seleção de menus e ícones através do *mouse* e, quando conveniente, por linhas de comando textuais. Dessa maneira, o Subsistema de Interface pode incluir um analisador léxico simples para checar a sintaxe dos comandos. A Interface fornece ainda, sempre que possível, mecanismos para desfazer operações errôneas, bem como impedir a execução de operações incorretas (esse último caso requer interação com o Subsistema de Modelagem). Uma vez que o comando tenha sido

analisado e considerado correto, a interface encaminha o comando ou seleção em solicitações ao Subsistema de Modelagem.

O Subsistema de Interface fornece também procedimentos para visualizar os objetos sendo modelados, e para tal possui o módulo de visualização, que interage com o Subsistema de Representação.

O Subsistema de Modelagem é responsável pelo disparo das operações solicitadas pelo usuário através da interface. Nos casos em que a solicitação envolve comandos ou expressões, requer um interpretador (*parser*) para analisar o comando ou expressão, checar sua validade (sintática e semântica) e ativar os procedimentos de modelagem associados.

O Subsistema de Representação é responsável pelo gerenciamento e acesso direto às estruturas de dados: qualquer acesso às representações de objetos exigido pelo Subsistema de Modelagem é feito através deste gerenciador. Além disso, fica a cargo deste subsistema fornecer uma interface para comunicação com outros modeladores e aplicações externas, acionar mecanismos para armazenar descrições dos objetos em uma base de dados permanente e para efetuar a conversão entre representações

2.2 Desenvolvimento da Análise Orientada a Objetos

2.2.1 Identificação de Classes-&-Objetos

Analisando a especificação descrita no item anterior para o AutoPAC, pode-se identificar as seguintes Classes-&-Objetos [Montenegro, 1994]:

- os elementos gráficos que compõem uma interface, entre eles: botões que acionam menus, menus *pull-down* e *pop-up*, botões, linha de comando, barra de ferramentas, área de trabalho, possivelmente ainda janela de visualização e recursos para ajuda interativa;

- as estruturas de dados que compõem as representações internas, entre elas: a lista de entidades, os nós da lista de entidades e as entidades propriamente ditas, especificadas atualmente como arcos, linhas e círculos, podendo ainda incluir outros tipos de entidades, como as poli-linhas;
- as estruturas preparadas para os processos aplicativos, como por exemplo a malha gerada a partir da composição de objetos;
- as estruturas preparadas para as interfaces externas, como por exemplo o arquivo neutro.

Os processos de modelagem atuam sobre os objetos do sistema, através de métodos que criam ou transformam formas primitivas. Assim sendo, nenhuma Classe-&-Objeto foi identificada para a modelagem.

2.2.2 Identificação de Estruturas

Após a identificação das Classes-&-Objetos, foram especificadas as relações de dependência entre as Classes-&-Objetos, isto é, quais Classes-&-Objetos são usadas (referenciadas) por outras em sua definição [Montenegro, 1994]. Foi possível identificar as seguintes estruturas para o AutoPAC existente:

- uma Estrutura Todo-Parte relacionando a lista de entidades (todo) com os nós que a compõem (parte) [Montenegro, 1994]. A lista pode estar vazia ou possuir *m* elementos, enquanto que um nó da lista pode estar associado no máximo a uma lista;
- outra estrutura Todo-Parte é a interface homem máquina (todo) com os seus componentes constitutivos (parte): menus, toolbar, linha de comando, janela de visualização do desenho.
- uma Estrutura Generalização-Especialização relacionando as entidades gráficas do sistema (generalização), entre elas a linha, o arco e o círculo [Montenegro, 1994]. Futuramente poderão ser

incorporadas novas entidades ao sistema, como por exemplo as poli-linhas.

Outras Estruturas poderiam ser consideradas, como por exemplo relacionando os componentes da interface, e novas poderão aflorar à medida em que o AutoPAC for evoluindo.

2.2.3 Identificação de Assuntos

Seguindo a estratégia descrita, para a identificação de assuntos deve-se transformar a Classe superior de cada Estrutura em Assunto, e depois transformar cada Classe-&-Objeto não pertencente a uma Estrutura em um Assunto [Coad, 1992]. Desta forma, temos:

- Assunto Lista: envolve a Estrutura Todo-Parte para a lista de entidades;
- Assunto Entidades: envolve a Estrutura Generalização-Especialização para as entidades gráficas;
- Assunto Interface: envolve todas as Classes-&-Objetos pertencentes à interface, como os botões que acionam menus, menus *pull-down* e *pop-up*, botões, linha de comando, barra de ferramentas e área de trabalho.
- Assunto Aplicativos: envolve todas as estruturas preparadas para os processos aplicativos, como por exemplo a malha gerada a partir da composição de objetos;
- Assunto Interface Externa: envolve todas as estruturas preparadas para as interfaces externas, como por exemplo o arquivo neutro.

2.2.4 Identificação de Atributos

Os Atributos identificados para as Classes-&-Objetos da estrutura orientada a objetos atual do AutoPAC foram:

- Lista de Entidades: apontador para o início da lista (*header) e o número de elementos da lista (nElements) [Ziviani, 1993]

- Nó da Lista de Entidades: como conteúdo do nó, um apontador para uma Entidade, e como membro da lista, um apontador para o próximo nó [Ziviani, 1993];
- Entidade: atributos padrão para as entidades, limites que contêm as entidades, estado atual da entidade;
- Entidade Linha: parâmetros para definição da primitiva Linha, como pontos inicial e final [Foley, 1990];
- Entidade Arco: parâmetros para definição da primitiva Arco, como raio, centro, ângulos inicial e final [Foley, 1990];
- Entidade Círculo: parâmetros para definição da primitiva Círculo, como centro e raio [Foley, 1990].

2.2.5 Identificação de Conexões de Ocorrência

Na estrutura orientada a objetos atual do AutoPAC foi possível identificar a Conexão de Ocorrência entre as Classes-&-Objetos Nó da Lista de Entidades e Entidade, pois o conteúdo do nó da lista de entidades é um apontador para uma entidade [Montenegro, 1994]. Assim, para cada ocorrência de um nó na lista de entidades, deve existir uma entidade correspondente.

2.2.6 Identificação de Serviços

Os Serviços identificados para as Classes-&-Objetos da estrutura orientada a objetos atual do AutoPAC foram [Montenegro, 1994]:

- Lista de Entidades: construtores da lista de entidades, cada um utilizando um conjunto diferente de argumentos e o destrutor da lista;
- Nó da Lista de Entidades: construtores do nó da lista de entidades, cada um utilizando um conjunto diferente de argumentos, o destrutor do nó, pesquisa pelo número de elementos da lista, inclusão e exclusão de elementos na lista, acesso a um elemento da lista, percorrimento da lista para desenho de seus

elementos, para procura por elemento selecionado ou para outras operações de interesse, recuperação e armazenamento da lista em bases de dados permanentes;

- Entidade: construtores da entidade, cada um utilizando um conjunto diferente de argumentos, o destrutor da entidade, retorno de características referentes à entidade, como sua Classe, seu estado, os limites que a contêm, seus parâmetros, seus atributos, desenho e remoção da entidade, seleção e de-seleção da entidade, verificação da seleção da entidade, definição de valores para seus atributos e características (cor, espessura da linha, parâmetros), operações de edição, como transformações geométricas sobre as entidades (mover, rotacionar, escalar, espelhar) e transformações globais sobre as entidades (cortar em relação a outra entidade, particionar);
- Entidade Linha: construtores da entidade Linha, cada um utilizando um conjunto diferente de argumentos, o destrutor de uma Linha, identificação da Classe, definição e obtenção de valores para os parâmetros, verificação de seleção da entidade, operações de edição;
- Entidade Arco: construtores da entidade Arco, cada um utilizando um conjunto diferente de argumentos, o destrutor de um Arco, identificação da Classe, definição e obtenção de valores para os parâmetros, verificação de seleção da entidade, operações de edição;
- Entidade Círculo: construtores da entidade Círculo, cada um utilizando um conjunto diferente de argumentos, o destrutor de um Círculo, identificação da Classe, definição e obtenção de valores para os parâmetros, verificação de seleção da entidade, operações de edição;

- Relacionados à Interface: inicialização da interface, criação e modificação de botões, manipulação (ativação e desativação) de menus e itens de menu, identificação de opções de menu selecionadas e recebimento de comandos fornecidos via teclado, identificação de coordenadas e desenho de pontos fornecidos através do mouse, manipulação de caixas de diálogo, apresentação de seleção, apresentação de avisos, desenho de entidades na tela;
- Relacionados à Modelagem: execução de comandos de modelagem selecionados via menu, interpretação e verificação de comandos entrados via linha de comandos, conversão entre sistemas de coordenadas, realização de operações de modelagem, como identificação de interseções e verificação de pertinências, cálculos diversos;
- Relacionados à Processos Aplicativos: identificação de contornos fechados, geração de malhas, geração de linhas equipotenciais;
- Relacionados à Interfaces Externas: manipulação de arquivos neutros.

2.2.7 Identificação de Conexões de Mensagem

A quantidade de fluxo de mensagem entre os objetos é bastante grande, ocorrendo diversas Conexões de Mensagens entre eles, principalmente para a realização da modelagem [Montenegro, 1994]. Estas Conexões de Mensagens não serão graficamente representadas aqui devido ao fato de que somente a estrutura de dados (representação interna) do sistema foi originalmente desenvolvida seguindo a metodologia da orientação a objetos; as outras partes do sistema foram adaptadas posteriormente.

2.2.8 Visão Geral do Modelo Orientado a Objetos Existente para o AutoPAC

A Fig. 2.2 a seguir apresenta uma visão geral da estrutura de dados básica definida para o AutoPAC, seguindo os passos da metodologia orientada a objetos apresentada. A representação gráfica referente aos Assuntos Interface, Interface Externa e Aplicativos não foi apresentada devido ao fato desta parte do sistema não ter sido desenvolvida de maneira orientada a objetos. A parte inferior de cada Classe-&-Objeto, correspondendo aos serviços por eles realizados, não foi preenchida devido à grande quantidade de serviços a serem incluídos, o que congestionaria a figura.

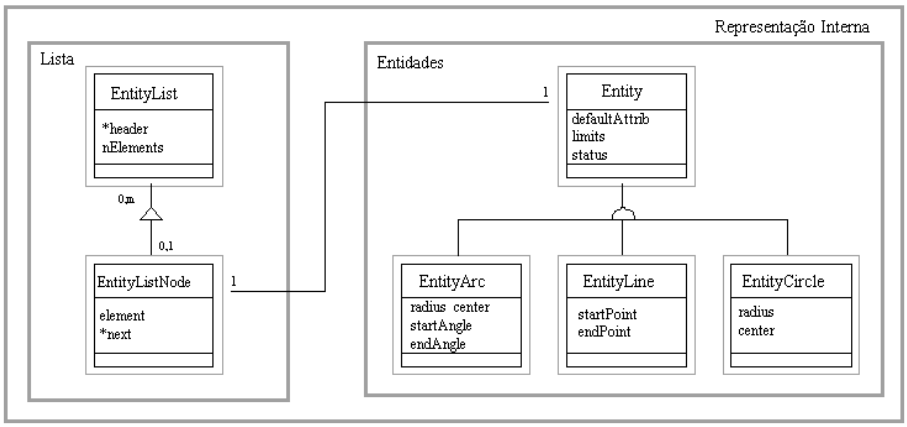

Fig. 2.2 - Modelo orientado a objetos do AUTOPAC

2.3 Visão Geral do Sistema

O AutoPAC possui 3 módulos principais: Interface, Modelagem e Representação. Paralelamente a estes módulos, está sendo desenvolvido um módulo independente, relacionado à aplicação pretendida para o modelador. A interação entre os módulos do sistema é ilustrada na Fig. 2.3.

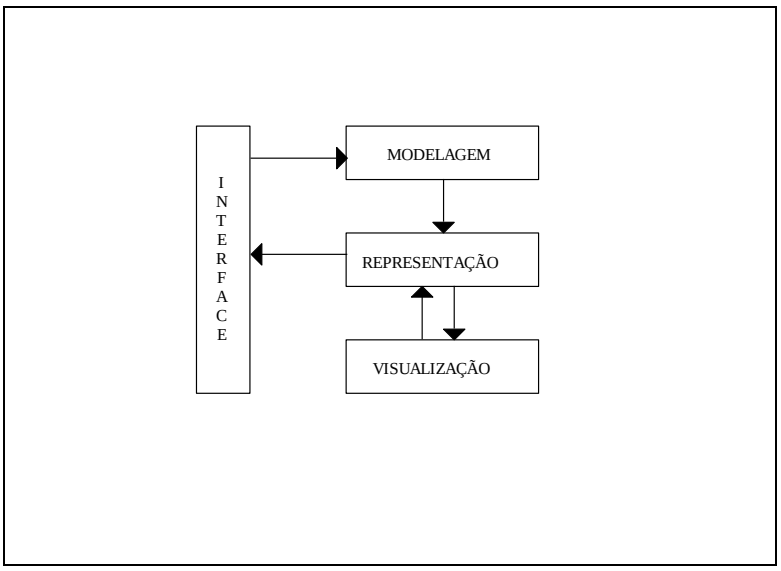

Fig. 2.3 - Módulos do AUTOPAC e a interação entre eles

O módulo de Representação, ou Núcleo, é responsável pela manutenção, gerenciamento e acesso às representações internas. Este módulo contém os procedimentos que criam e modificam as representações dos objetos. Cada entidade do AutoPAC possui uma estrutura de dados e procedimentos específicos para manipulação desta estrutura. As informações das representações são lidas por procedimentos de outros módulos, mas são criadas e modificadas apenas pelos procedimentos do núcleo.

O módulo de Modelagem é responsável pelas operações que atuam sobre a representação interna, gerando e editando objetos. Este módulo contém os procedimentos que interpretam as descrições de um novo objeto ou de uma modificação (através de técnicas de modelagem disponíveis no modelador), e a seguir acionam os procedimentos do módulo de representação para criar ou alterar, respectivamente, uma representação.

O módulo de comunicação, ou Interface, cuida da comunicação, visualização e interação com o usuário. Este módulo contém os procedimentos que manipulam a interface gráfica, através da qual o usuário interage com o AutoPAC.

As tarefas a serem realizadas pelo modelador são descritas e capturadas pelos procedimentos da interface, que ativam procedimentos de outros módulos, responsáveis por suas execuções. Os resultados obtidos são recebidos e exibidos pelos procedimentos da interface.

O módulo Aplicação contém procedimentos relacionados à utilização pretendida para o modelador, entre elas a identificação de regiões fechadas , a geração de malha e de superfícies equipotenciais (Apêndice).

CAPÍTULO 3 - O AVALIADOR DE EXPRESSÕES

Avaliar expressões é uma operação comum em linguagens de programação, tão comum que a maioria dos programadores desconhece a complexidade da implementação de um avaliador de expressões - até o dia em que eles precisam implementar um. Precedência de operadores aritméticos e booleanos, dados variáveis, e notação infixada (notação em que o operador aritmético é colocado entre os dois operandos; exemplo: 10-5) são complicadores que surgem no processo de um avaliador de expressões [Rogers, 1996]. Normalmente tenta-se simplificar a implementação, reduzindo ao mínimo o número de operadores ou desabilitando parênteses e variáveis. Sempre é possível driblar os problemas de precedência de operadores obrigando os usuários a escreverem expressões em notação pósfixada ou préfixada. Mas é provável que poucos usuários aceitarão mudar sua forma de pensar apenas para se adequar aos requisitos de uma aplicação.

[c1] Comentário: Notação em que o operador aritmético é colocado entre os dois operandos. Exemplo: 10-5

Neste capítulo é apresentado um conjunto de objetos de expressão que implementam um avaliador de expressões flexível e expansível. Este avaliador de expressões também demonstra uma aplicação em que herança e polimorfismo são usados naturalmente.

3.1 Descrição do problema

O problema é: como escrever um programa que recebe como entrada uma string contendo uma expressão numérica como (10-5)*3, e calcular a resposta apropriada? A análise de expressões é muito simples, e, de certa forma, é ainda mais simples que outras tarefas de programação, porque trabalha de acordo com regras rígidas da álgebra.

Os operadores aritméticos podem ser combinados em expressões de acordo com as regras da álgebra. Por exemplo:

```
10-8
(100-5)*14/6
a+b-c
```

```
a=10-b
```

Existem regras de precedência para cada operador. Os operadores e funções são organizados na seguinte ordem, da maior precedência para a menor: funções, *, /; +, -; =. Operadores de igual precedência são avaliados da esquerda para a direita [Rogers, 1996].

Existem algumas maneiras de se escrever uma rotina que avalie expressões do tipo exemplificado acima, como os analisadores dirigidos por tabelas, onde um outro programa deve gerar tabelas que serão empregadas pelo analisador na resolução da expressão, ou os analisadores recursivos descendentes, que partem da expressão completa e avaliam seu valor dissecando-a recursivamente até chegar à solução [Schildt, 1991]. O analisador recursivo descendente foi o escolhido porque sua métodologia é semelhante que o ser humano usa para analisar uma expressão aritmética.

3.2 Dissecando uma Expressão

Um avaliador recursivo descendente deve ser capaz de dividir uma expressão em seus componentes. Por exemplo, a expressão

```
a * pt - (W+10)
```

contém os componentes a, *, pt, -, (, W, +, 10 e). Cada componente representa uma unidade indivisível da expressão. É necessário que o analisador tenha uma rotina que devolva cada item de uma expressão individualmente, sendo capaz de ignorar espaços em branco e tabulações e detectar o final da expressão. É necessário também que a rotina seja capaz de verificar a sintaxe da expressão para garantir que esta seja corretamente avaliada; ou seja, é necessário que a sintaxe da expressão seja bem definida [Schildt, 1991].

A cada componente lido de uma expressão é associado um objeto, dependendo do tipo do componente. Todos os operadores, funções e mesmo os delimitadores da expressão são objetos. Cada um destes objetos tem um atributo denominado prioridade, que é responsável pela correta

aplicação da precedência dos diversos operadores e funções na avaliação de uma expressão. É utilizada uma estrutura do tipo pilha [Ziviani, 1993] para escalonar a ordem em que os componentes serão avaliados, e esta ordem é definida pelo atributo prioridade de cada objeto.

Um analisador recursivo descendente avalia as expressões como sendo estruturas de dados recursivas, isto é, expressões que são definidas em termos delas mesmas [Schildt, 1991]. Se, por um momento, restringirmos as expressões a usar apenas os operadores +, -, *, / e parênteses, todas as expressões podem ser definidas com as seguintes regras:

```
expressão -> termo[+termo][-termo]
termo -> fator[*fator][/fator]
fator -> variável, número ou (expressão)
```

Os colchetes designam um elemento opcional e -> significa "produz". As regras são normalmente chamadas de regras de produção da expressão. Assim, a definição de termo poderia ser a seguinte: "Termo produz fator vezes fator ou fator dividido por fator". A precedência dos operadores está implícita na maneira como uma expressão é definida.

A expressão

```
10 + 5 * B
```

tem dois termos: 10 e 5*B. Tem três fatores: 10, 5 e B. Esses fatores consistem em dois números e uma variável.

Por outro lado, a expressão

```
14 * (7-C)
```

tem dois termos: 14 e (7-C). Os termos consistem em um número e uma expressão entre parênteses. A expressão entre parênteses é avaliada como um número e uma variável.

Esse processo forma a base de um analisador recursivo descendente, que é basicamente um conjunto de funções mutuamente recursivas que opera de forma encadeada [Schildt, 1991]. A cada passo, o analisador executa as operações especificadas na seqüência algebricamente correta.

3.3 Especificações de Projeto

É necessário definir um pequeno conjunto de restrições antes de iniciar o desenvolvimento de um avaliador de expressões [Schildt, 1991]. Especificamente, o avaliador deverá:

1.avaliar expressões em notação algébrica infixada
2.prover o seguinte conjunto de funcionalidades:
- operadores aritméticos (+, -, *, /)
- operadores booleanos (AND, OR, NOT)
- operadores relacionais (>, <, >=, <=, ==, !=)
- manipular dados variáveis e constantes mantidos por uma tabela de símbolos

3.ser projetado de forma expansível

Apesar das especificações acima não satisfazerem todos os requisitos necessários a um avaliador de expressões, elas fornecem a maioria das ferramentas normalmente utilizadas.

3.4 Projetando uma árvore de expressões

No início do projeto, é necessário determinar como uma expressão pode ser decomposta em objetos. Cada elemento ou componente de uma expressão será chamado objeto de expressão. Como mostrado na Fig. 3.1, uma expressão como 1+2*3/(4-2), pode ser representada como uma estrutura de árvore (conhecida como árvore de expressões) que quebra uma expressão em seus componentes básicos [Rogers, 1996].

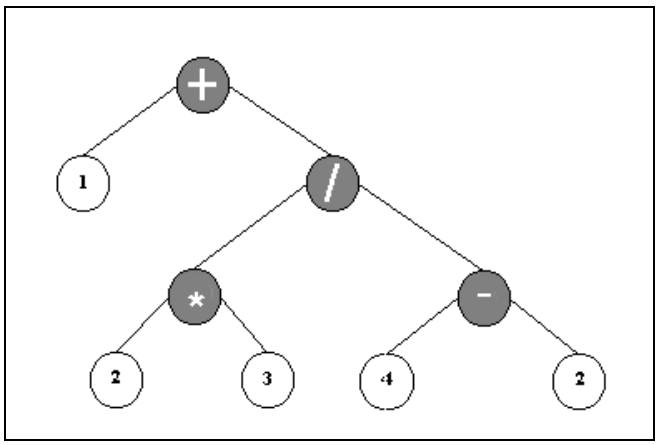

Fig. 3.1

Caminhando na árvore de expressões em uma ordem transversal a partir de seu interior é possível reconstruir a expressão infixada. Cada nó da árvore possui um "valor", que pode ser numérico (como os literais) ou ser alguma função dos valores de seus filhos (como em operadores como +, -, * e /). Avaliando o nó raiz (o + da Fig. 3.1) causará o percorrimento em ordem pós-transversal da árvore de expressões, resultando no valor da expressão como um todo [Rogers, 1996].

3.5 Classes base

Começando com a classe Exp_Obj mostrada abaixo, foi iniciada a modelagem dos nós na árvore de expressões com objetos.

```
// Base Expression Object
class Exp_Obj
{
 public:
  virtual float GetValue( Symbol_Table &sym) {return 0.0;};
  virtual int GetType( void ) {return NONE;};
  virtual int GetPriority( void ) {return 0 ;};
};
```

Esta classe não faz nada, mas define uma interface padrão para o restante dos objetos de expressão. Derivando direta ou indiretamente todos os outros objetos da classe base Exp_Obj foi possível usar polimorfismo e ligação dinâmica para facilitar a manipulação de objetos de

expressão, não importando seus tipos e funcionalidades específicos [Montenegro, 1994]. A operação GetValue retorna o valor do objeto de expressão. As classes derivadas implementam a funcionalidade do objeto pela redefinição desta operação. O parâmetro sym de GetValue proporciona acesso à informação contida na tabela de símbolos.

Foram definidas cinco categorias ou tipos de objetos de expressão: NONE, OPERAND_TYPE, BINARY_OPERATOR_TYPE, UNARY_OPERATOR_TYPE, e EXPRESSION_TYPE. A operação GetType retorna um objeto do tipo expressão.

GetPriority retorna a prioridade infixa de um objeto, que é usada para manter a precedência de um operador quando uma expressão está sendo avaliada. Na expressão 1+2*3/(4-2), os parênteses tem a maior precedência, então seu conteúdo é avaliado primeiro. Os operadores * e / têm a mesma precedência na expressão, então eles são avaliados um após o outro, da esquerda para a direita. O operador + tem a menor precedência na expressão e é avaliado por último. A Tabela 3.1 mostra a prioridade infixa dos operadores que foram implementados como objetos de expressão.

Operador	Prioridade
( )	9
NOT	8
- (unário)	7
* /	6
+ -	5
< <= > >=	4
= <>	3
AND	2
OR	1

Tabela 3.1

As classes Unary_Obj (mostrada abaixo) e Binary_Obj foram derivadas de Exp_Obj para abstrair características comuns aos dois tipos de operadores: aqueles que operam sobre um único objeto e aqueles que operam sobre dois

objetos. A classe Unary_Obj mantém um ponteiro para outro Exp_Obj. Os construtores de Unary_Obj inicializam este ponteiro para NULL ou para apontar para um Exp_Obj. SetObj permite mudar o ponteiro após sua construção e GetType sempre retorna UNARY_OPERATOR_TYPE. Classes derivadas de Unary_Obj poderão redefinir a função virtual pura GetValue para incorporar a funcionalidade de um operador unário específico [Montenegro, 1994].

```
class Unary_Obj : public Exp_Obj
{
 protected:
  Exp_Obj *obj;
 public:
  Unary_Obj( void ) { obj=NULL; }
  Unary_Obj( Exp_Obj *o ) { obj=o; }
  virtual void SetObj(Exp_Obj *o) { obj=o; }
  virtual int GetType( void ) {return UNARY_OPERATOR_TYPE;}
  virtual float GetValue( Symbol_Table &sym )=0;
};
```

A classe Binary_Obj é muito similar à classe Unary_Obj. Ela mantém ponteiros para um Exp_Obj à esquerda e um à direita. Os ponteiros são inicializados pelo construtor e a partir deste momento podem ser alterados pelos operadores SetLeft e SetRight. GetType sempre retorna BINARY_OPERATOR_TYPE e a função virtual pura GetValue é definida pelas classes derivadas para implementar a funcionalidade dos operadores binários.

```
class Binary_Obj : public Exp_Obj
{
 protected:
  Exp_Obj *left, *right;
 public:
  Binary_Obj( void ) { left=NULL; right=NULL; }
  Binary_Obj( Exp_Obj *l, Exp_Obj *r ) { left=l; right=r; }
  virtual void SetLeft(Exp_Obj *l){ left=l;}
  virtual void SetRight(Exp_Obj *r) {right=r;}
  virtual int GetType( void ) {return BINARY_OPERATOR_TYPE;}
  virtual float GetValue( Symbol_Table &sym )=0;
};
```

3.6 Operadores

Uma vez definida a estrutura básica da árvore, é necessário preenchê-la com objetos mais concretos. Operadores podem ser conectados a objetos finais (que podem ser variáveis, como A, ou literais tais como 123.4) ou a

sub-árvores. Como exemplos, são apresentadas duas classes que implementam um operador unário e um binário concretos.

A classe Unary_Minus_Obj implementa o operador menos unário (-). Ela é derivada de Unary_Obj. GetValue retorna o negativo do valor referenciado por Exp_Obj e GetPriority retorna a precedência do operador. GetType e SetObj também estão disponíveis pois elas são herdadas da classe Unary_Obj.

```
class Unary_Minus_Obj : public Unary_Obj
{
 public:
  Unary_Minus_Obj( Exp_Obj *o ):Unary_Obj( o ) {};
  Unary_Minus_Obj( void ):Unary_Obj() {};
  virtual float GetValue(Symbol_Table &sym)
  {
   return -(obj->GetValue (sym));
  }
  virtual GetPriority( void ) { return UNARY_SUB_PRIORITY; }
};
```

A classe Add_Obj implementa uma adição binária em seu operador GetValue e retorna a prioridade infixa do operador em GetPriority. Todas as classes filhas de Binary_Obj herdam SetLeft, SetRight e GetType. As operações de subtração (-), multiplicação (+) e divisão são implementadas de forma semelhante.

```
class Add_Obj : public Binary_Obj
{
 public:
  Add_Obj(Exp_Obj *l, Exp_Obj *r): Binary_Obj(l, r) {};
  Add_Obj( void ):Binary_Obj() {};
  virtual int GetValue( Symbol_Table &sym )
  {
   return left ->GetValue(sym) + right->GetValue(sym);
  }
  virtual int GetPriority( void ) { return ADD_SUB_PRIORITY; }
};
```

3.7 Objetos finais

Variáveis e valores literais são chamados de finais; eles ocupam as folhas da árvore de expressões. Os valores literais foram modelados com a classe Literal_Obj, que define uma variável para armazená-los. O valor do literal é inicializado pelo construtor da classe, GetValue retorna o valor literal armazenado no objeto. Para simplificar a interface para todos os objetos, GetValue sempre faz

referência à tabela de símbolos, mesmo quando isto não é necessário.

```
class Literal_Obj : public Exp_Obj
{
 private:
  float value;
 public:
  Literal_Obj( float v) { value=v;}
  virtual float GetValue( Symbol_Table &sym ) { return value; }
  virtual int GetType( void ) { return OPERAND_TYPE; }
};
```

O último componente da árvore de expressões é o Variable_Obj, que não armazena os dados da variável, mas o seu nome. Os dados das variáveis ficam armazenados em uma tabela de símbolos. A tabela de símbolos mantém o nome e o valor de cada variável e proporciona acesso aos dados de cada variável. A classe Symbol_Table provê três operações: Get, Set e Is_In. A operação Get recupera os dados da variável a partir de seu nome, Set atualiza os dados da variável também a partir de seu nome (ou cria a variável se ela não está na tabela), e Is_In retorna um ponteiro para um registro na tabela de símbolos se a variável existir. Na classe Variable_Obj, a operação GetValue simplesmente busca o valor da variável na tabela de símbolos.

```
class Variable_Obj : public Exp_Obj
{
 private:
  char name[25];
 public:
  Variable_Obj( char *var_name ) { strcpy(name, var_name); }
  virtual float GetValue( Symbol_Table &sym )
  {
   return sym.Get(name);
  }
  virtual int GetType( void ) { return OPERAND_TYPE; }
};

class Symbol_Table
{
 private:
  struct entry { char name[25]; float value; entry *next; };
  entry *head;
 public:
  Symbol_Table( void ) { head=NULL; };
  ~Symbol_Table( void )
  {
   entry *temp;
   while (head!=NULL)
   {
    temp=head; head=head->next;
    delete temp;
   };
  }
```

```
entry *Is_In( char *name )
{
 entry *temp=head;
 while (temp!=NULL)
  if (strcmp(name, temp->name)==0)
   return temp;
  else
 temp=temp->next;
 return NULL;
}
void Set( char *name, float value )
{
 entry *temp=Is_In( name );
 if (temp==NULL)
 {
  entry *new_head=new entry;
  strcpy(new_head->name, name);
  new_head->next=head;
  head=new_head;
 }
 else
  temp->value=value;
}
float Get( char *name )
{
 entry *temp=Is_In(name);
 if (temp!=NULL)
  return temp->value;
 else
  return 0;
};
```

3.8 Unindo os nós da árvore de expressões

O avaliador de expressões precisa encontrar os objetos de expressão arranjados de acordo com sua árvore de expressões associada [Rogers, 1996]. O processo de ligar os objetos de expressão para formar a árvore de expressões é a parte mais complicada do avaliador de expressões. O código abaixo mostra parte de uma nova classe, Base_Expression, derivada de Exp_Obj para construir uma árvore de expressões a partir de uma string contendo uma expressão.

```
class Base_Expression : Exp_Obj
{
 protected:
  Exp_Obj *exp;
  ExpStack exp_stack, op_stack;
  // mais elementos protegidos ...
 public:
  Base_Expression( void ) { exp=NULL; }
  ~Base_Expression( void ) { delete exp; }
  virtual void GetToken(char *str, int &pos, char *buffer,
                        int &type, int &error)=0;
  virtual int GetType( void ) { return EXPRESSION_TYPE; }
  virtual float GetValue(void)
  virtual void Push_Tree(Exp_Obj *obj);
  virtual void Build( Exp_Obj *obj );
  virtual Exp_Obj *Build_Expression( char *str );
  // mais funções membro...
```

```
};
```

Em Base_Expression o ponteiro exp da classe Exp_Obj aponta para a raiz da árvore de expressões. GetType retorna EXPRESSION_TYPE e GetExp retorna a string da expressão original. Base_Expression provê duas operações GetValue, e ambas retornam o valor do nó raiz da árvore de expressões. A primeira função é equivalente à GetValue em qualquer outro objeto de expressão. A segunda função cria uma tabela de símbolos temporária vazia de forma que o usuário não tem que criar uma tabela de símbolos quando está avaliando expressões que não precisam de uma.

A construção de uma árvore de expressões envolve quatro operações: Get_Token, Build_Expression, Build e Push_Tree. A operação Get_Token encontra o próximo *token* (elemento) na string da expressão juntamente com seu tipo e informa quando ocorrem erros neste processo (Fig. 3.2). Get_Token é uma função virtual pura, que deve ser definida em uma classe filha. Então a sintaxe (como os operadores, identificadores, etc., são combinados) é abstraída dos objetos básicos [Montenegro, 1994]. Isso dá ao usuário um meio muito flexível para implementar qualquer número de regras de sintaxe.

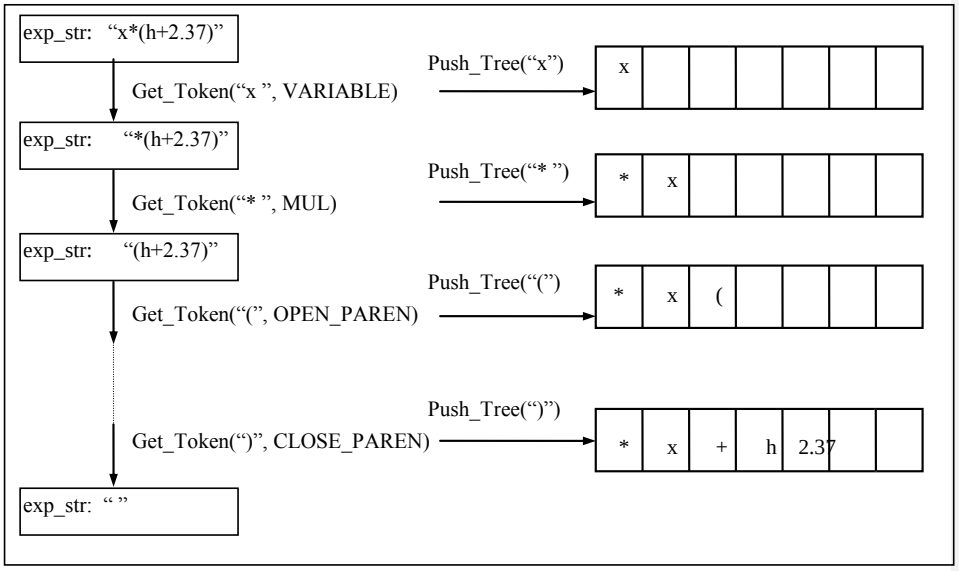

Fig. 3.2

A operação Build_Expression cria os objetos de expressão apropriados de acordo com a informação que ele recebe de Get_Token e usa a operação Build para adicionar objetos à árvore de expressões.

O algoritmo Build requer o uso de duas pilhas: uma para a expressão que está sendo construída e uma outra para armazenar temporariamente os operadores [Rogers, 1996]. A classe ExpStack mostrada abaixo implementa uma estrutura de dados do tipo pilha para armazenar objetos do tipo Exp_Obj e provê três operações: Push, Pop e Empty. Como todos os objetos de expressão são derivados direta ou indiretamente de Exp_Obj, o polimorfismo permite que qualquer objeto de expressão seja armazenado na pilha, não importando seu tipo específico [Montenegro, 1994].

```
class ExpStack
{
 private:
  struct node
  {
   Exp_Obj *data;
   node *next;
  };
  node *head;
 public:
  ExpStack( void ) { head=NULL; };
  ~ExpStack( void )
  {
   node *temp;
   while (head!=NULL)
   {
    temp=head; head=head->next;
    delete temp;
   }
  }
  void Push(Exp_Obj *obj)
  {
   node *temp=new node;
   temp->data=obj; temp->next=head;
   head=temp;
  }
  Exp_Obj *Pop( void )
  {
   if (head==NULL)
   {
    cout << "Stack Underflow...\n";
    exit(1);
   }
   node *temp=head;
   head=head->next;
   Exp_Obj *tempData=temp->data;
   delete temp;
```

```
        return tempData;
      }
      int Empty( void )
      {
       if (head==NULL)
        return 1;
       else
        return 0;
      }
    };
```

Foram criados quatro objetos de controle para ajudar no processo de construção da árvore: Open_Paren_Obj, Close_Paren_Obj, Start_Obj e Stop_Obj (abaixo). Open_Paren_Obj e Close_Paren_Obj fornecem uma prioridade e um tipo, podendo ser usados na operação Build para corrigir superposição de precedência de operadores quando necessário (Fig. 3.2). Os objetos Start_Obj e Stop_Obj identificam o primeiro e o último *token*, respectivamente.

```
class Start_Obj : public Exp_Obj
{
 public:
  virtual int GetPriority( void ) { return 0; }
  virtual Int GetType( void ) {return START_TYPE; }
};
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Stop_Obj : public Exp_Obj
{
 public:
  virtual int GetType( void ) {return STOP_TYPE; }
};
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Open_Paren_Obj : public Exp_Obj
{
 public:
  virtual int GetPriority( void ) { return OPEN_PAREN_PRIORITY;
}
  virtual Int GetType( void ) {return OPEN_PAREN-TYPE; }
};
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Close_Paren_Obj : public Exp_Obj
{
 public:
  virtual int GetPriority( void ) {return CLOSE_PAREN_PRIORITY;}
  virtual Int GetType( void ) {return CLOSE_PAREN_TYPE;}
};
```

À medida que os objetos vão sendo colocados (push) na pilha, a operação Push_Tree converte a pilha em uma árvore de expressões enquanto ela vai sendo construída (Fig. 3.2). Para Push_Tree adicionar (push) um operador unário à árvore, ela deverá primeiro retirá-lo da pilha (pop), direcionar o ponteiro de objetos para este operador e colocá-lo na árvore (push) [Ziviani, 1993]. Se um operador

binário é adicionado à árvore, Push_Tree vai retirar dois objetos da pilha, assinalar os ponteiros esquerdo e direito do operador binário para os dois objetos e finalmente colocar o objeto operador binário na árvore. O único objeto deixado na pilha depois de feito o *parser* (interpretação) da string da expressão será a raiz da árvore de expressões.

Sendo Get_Token uma função virtual pura em Base_Expression, uma classe filha foi derivada para implementá-la. Foi criada a classe Pascal_Like_Expression que implementa uma sintaxe semelhante à da linguagem Pascal. A listagem abaixo mostra um exemplo utilizando as classes de expressão para avaliar uma expressão escrita na sintaxe da linguagem Pascal.

Inicialmente é criado o objeto p2, instância de Pascal_Like_Expression. O construtor da classe atribui a expressão "Temperature*v1/v2+v3" ao atributo protegido exp. Na tabela de símbolos são criadas e armazenadas as variáveis Temperature, v1, v2 e v3. O método GetExp retorna a string da expressão. O método GetValue dispara o processo de avaliação da expressão, que disseca a expressão, busca os valores de Temperature, v1, v2 e v3 na tabela de símbolos e calcula e retorna o valor numérico da expressão.

```
Pascal_Like_Expression p2("Temperature*v1/v2+v3");
Symbol_Table sym;
sym.Set("v1", 5) ;
sym.Set("v2", 9);
sym.Set("v3", 32);
sym.Set("Temperature", 50);
cout << p2.GetExp() << " => " << p2.GetValue()<< endl << endl;
Resultado: Temperature*v1/v2+v3 => 59.7778
```

3.9 Conclusão

As características de herança e polimorfismo do C++ são especialmente eficientes na modelagem da semântica de expressões algébricas infixadas. O avaliador de expressões orientado a objetos apresentado neste capítulo provê ao programador um conjunto de classes que podem explorar as características de uma linguagem orientada a objetos, como o C++, para criar objetos que são fáceis de entender, flexíveis e expansíveis. Usando a hierarquia apresentada

aqui como um ponto de partida, é possível criar código para manipular expressões mais elaboradas e interessantes, como veremos no próximo capítulo.

CAPÍTULO 4 - O SISTEMA PARAMETRIZADO

Sistemas CAD são muito úteis para a representação de objetos. Em aplicações práticas, há a necessidade de facilidades para o desenho conceitual, uma vez que o usuário nem sempre tem certeza sobre a melhor configuração de um objeto. Faltam aos modeladores geométricos tradicionais ferramentas que facilitem ou até mesmo automatizem os procedimentos de construção da geometria de objetos; estes procedimentos geralmente são imprevisíveis tornando o trabalho difícil e tedioso.

Estes sistemas geralmente não permitem o uso de desenhos anteriores para a criação de novos desenhos. Também é comum a necessidade de se definir com exatidão as dimensões e a localização de cada elemento gráfico.

Deve-se ressaltar que um sistema CAD não é responsável apenas pela representação gráfica do problema. O sistema deve auxiliar o usuário na confecção do projeto, proporcionando meios de automatizar e otimizar o trabalho [Lowther, 1996]. Uma característica de grande importância em sistemas deste tipo é a possibilidade de se trabalhar com parâmetros. Sistemas CAD paramétricos podem desenhar objetos protótipo que representam famílias de diferentes objetos que podem ter suas especificações (parâmetros) ajustadas por uma expressão aritmética de modo a otimizar a performance do produto final (Fig. 4.1).

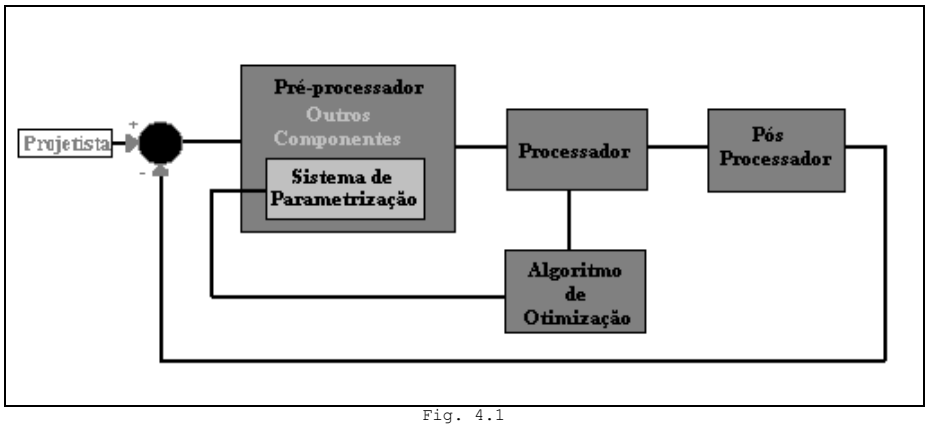

Fig. 4.1

Dado um conjunto de parâmetros específicos, componentes particulares da famílias podem ser obtidos automaticamente. O desenho paramétrico aumenta a flexibilidade do projeto, definindo a geometria e as restrições geométricas sem especificar o conjunto de dimensões concretas do objeto. Com o uso de restrições, é possível descrever dependências entre componentes dos objetos e entre objetos [Lowther, 1996].

4.1 Introdução

Está sendo desenvolvido pelo Grupo de Otimização e Projeto Assistido por Computador do Programa de Pós-graduação em Engenharia Elétrica da UFMG um sistema CAD bidimensional. Este sistema tem por objetivo automatizar e otimizar projetos de engenharia que envolvam o cálculo de campos eletromagnéticos. Este sistema, quando pronto, poderá ser dividido em três grandes blocos funcionais: o pré-processador AutoPAC, descrito no capítulo 2, possibilitará ao projetista o desenvolvimento de um desenho correspondente ao problema real e também a introdução de propriedades e restrições físicas necessárias à resolução do mesmo; o processador, que realizará todos os cálculos necessários utilizando métodos computacionais conhecidos e fornecerá as soluções do problema ao terceiro bloco, o pós-processador, que apresentará graficamente os resultados permitindo a análise e também uma eventual otimização destes. A otimização, na verdade, é distribuída ao longo de toda a cadeia: envolve o pré-processador (parametrização), o processador (cálculo) e o pós-processador (análise de sensibilidade e determinação de novos parâmetros).

Cada um dos blocos funcionais pode ser composto de um ou mais sub-blocos. Um dos componentes do bloco pré-processador é o sistema parametrizado, objetivo deste trabalho.

O sistema parametrizado é constituído de três subsistemas. O subsistema principal é o avaliador de

expressões, responsável pelo processamento e pelo armazenamento interno das expressões e parâmetros do sistema parametrizado. Os outros dois subsistemas são responsáveis pela interface e pelo armazenamento persistente dos dados inseridos pelo usuário.

4.2 Características do Sistema Parametrizado

O AutoPAC cria três tipos de entidades básicas: círculo, arco e reta [Souza, 1997]. Na construção destas entidades são criados alguns pontos base; estes pontos caracterizam as entidades e as diferenciam. A reta tem como pontos básicos os pontos extremos. Para o círculo, o ponto base é o ponto central. Para o arco, são importantes o centro e os pontos inicial e final.

Devido a estas características do modelador, foi estabelecido que os pontos básicos das três entidades seriam alvos da parametrização. Além destes pontos, foi estabelecido que valores reais ou inteiros poderiam também existir como parâmetros. Desta forma seria possível parametrizar toda a construção das entidades básicas através da criação de variáveis correspondentes aos seus pontos básicos. Para estes pontos também seria possível criar expressões utilizando-se valores reais ou inteiros (e até mesmo outros pontos) no enunciado destas expressões. Posteriormente, poder-se-ia alterar parcialmente ou totalmente a geometria através da manipulação dos parâmetros previamente criados.

Foi estabelecida também a possibilidade de se construir objetos parametrizados que serviriam como modelo para a criação de novos objetos. Para que isso fosse possível, foi necessário a definição de novas regras para a construção do arquivo neutro de geometria para que os parâmetros criados pudessem ser armazenados e posteriormente reconhecidos na leitura deste.

4.3 Adaptando o avaliador de expressões

O avaliador de expressões descrito no capítulo 3 foi desenvolvido de forma flexível, visando permitir uma futura expansão de sua funcionalidade. No desenvolvimento do sistema parametrizado foi necessário expandir o número de operações fornecidas pelo avaliador de expressões assim como os tipos de dados suportados por ele.

Um modelador geométrico bidimensional como o AutoPAC trabalha com os tipos de dados padrão e também com tipos definidos internamente. Um destes tipos exerce um papel fundamental no sistema parametrizado: o tipo realPoint.

```
struct realPoint
{
 float x, y;
};
```

Esta estrutura implementa um ponto bidimensional com abcissa x e ordenada y. O avaliador de expressões, que até então só reconhecia variáveis int e float foi alterado para tratar variáveis do tipo realPoint. Para tanto, foi necessário expandir cada um dos objetos, sobrecarregando alguns dos métodos existentes e, em alguns casos, criando novos métodos específicos para o tratamento de variáveis deste tipo.

Na classe básica Exp_Obj foi criado um método chamado GetPoint. Este método é análogo ao método GetValue, responsável por retornar o valor numérico da árvore de expressões. A diferença é que a função GetPoint retorna o resultado de uma expressão cujas variáveis e valores são coordenadas de pontos e não simples valores reais ou inteiros.

```
// Base Expression Object
class Exp_Obj
{
 public:
  virtual float GetValue( Symbol_Table &sym );
  virtual realPoint GetPoint( Symbol_Table &sym );
  virtual int GetType( void );
  virtual int GetPriority( void );
  virtual int GetFlag( Symbol_Table &sym );
};
```

Foi definido também o método GetFlag que retorna um valor que identifica o tipo de dado que o objeto manipula: VALUE (int ou float), POINT (realPoint) e POLAR (um ponto representado em coordenadas polares). Como estes dois novos métodos, GetPoint e GetFlag, foram definidos na classe básica Exp_Obj, eles são herdados pelos métodos derivados desta classe.

Foram definidas as classes Sin_Obj, Cos_Obj e Tan_Obj que implementam as funções trigonométricas seno, coseno e tangente, não fornecidas originalmente pelo avaliador. O método GetPoint destas classes retorna o resultado da função trigonométrica aplicada sobre o ângulo formado entre o ponto, argumento da função, e a origem do sistema de coordenadas. Por outro lado, o método GetValue aplica a função trigonométrica sobre um valor simples (coordenada unidimensional), que deve ser o valor de um ângulo em radianos.

```
float Cos_Obj :: GetValue( Symbol_Table &sym )
{
 return cos((obj->GetValue(sym)*PI)/180);
}
realPoint Cos_Obj :: GetPoint( Symbol_Table &sym )
{
 realPoint p = obj->GetPoint(sym);
 p.x =  p.x/sqrt(p.x*p.x + p.y*p.y);
 p.y = p.x;
 return p;
}
```

As classes Asin_Obj, Acos_Obj e Atan_Obj foram definidas para implementar as funções trigonométricas inversas. Para estas classes o método GetPoint não foi definido porque um argumento do tipo realPoint não faz sentido nestes casos.

Também foram implementadas funções matemáticas através das seguintes classes:

- Log_Obj (logaritmo natural),
- Log10_Obj (logaritmo na base 10),
- Expn_Obj (função exponencial),
- Sqr_Obj (quadrado de um número ou ponto),
- Sqrt_Obj (raiz quadrada),

- Trunc_Obj (parte inteira de um número real)

A classe Dot_Obj faz o produto escalar entre dois pontos (o operador de produto escalar é o @). Neste caso, cada ponto é tratado como um vetor. O método GetValue não é implementado para esta classe porque o produto escalar não é definido para valores simples.

```
realPoint Dot_Obj :: GetPoint( Symbol_Table &sym )
{
 // O produto escalar deve ter pontos como operandos
 if  ((  left->GetFlag(sym)==POINT  ||  left->GetFlag(sym)==POLAR
)&&
    (  right->GetFlag(sym)==POINT  ||  right->GetFlag(sym)==POLAR)
)
 {
  realPoint r = right->GetPoint(sym), l = left->GetPoint(sym);
  r.x = (l.x * r.x) + (l.y * r.y);
  r.y = r.x;
  return r;
 }
 else
 {
  cout << "\n<<< Runtime Error - Data type error >>>\n";
  return screenToReal(SRGP_defPoint(0, 0));
 }
}
int Dot_Obj :: GetFlag( Symbol_Table &sym )
{
 return POINT;
}
int Dot_Obj :: GetPriority( void )
{
 return MUL_DIV_PRIORITY;
}
int Dot_Obj :: GetType( void )
{
 return BINARY_OPERATOR_TYPE;
}
```

Depois da definição deste conjunto de operações, a Tabela 4.1 mostra como ficou a prioridade infixa dos operadores definidos como objetos de expressão no avaliador de expressões.

Operador	Prioridade
( )	9
NOT	8
- sin cos tan asin acos atan ln log exp sqr sqrt trunc	7
* / @	6
+ -	5
< <= > >=	4
= <>	3

AND	2
OR	1

Tabela 4.1

A classe Symbol_Table teve o método Set sobrecarregado para permitir a inclusão de dados do tipo realPoint na tabela de símbolos. Os métodos Get_Point e Get_Flag são análogos àqueles definidos na classe Exp_Obj.

Foram definidos ainda os métodos Get e numElements. Ambos são usados nas funções da classe Geometria (arquivo arq_neut.cc), que implementam a manipulação de arquivos do AutoPAC. O método Get é usado para pesquisar a tabela de símbolos em busca de uma determinada variável. O método numElements retorna o número de entradas preenchidas da tabela de símbolos.

4.4 O funcionamento interno do sistema parametrizado

O sistema parametrizado deve ser integrado ao AutoPAC e deve prover ao usuário uma interface de comunicação. Basicamente esta interface deve prover meios de o usuário digitar uma expressão e os valores das variáveis presentes nesta expressão e retornar o seu valor numérico.

A função expression é o ponto central do sistema parametrizado. Ela constrói uma caixa de diálogo onde o usuário pode digitar uma expressão. Baseado nesta expressão, ela cria um objeto da classe Pascal_Like_Expression, exp. O método GetToken, de exp, é chamado em um loop.

```
int expression(void)
{
 // declaração de variáveis
 ....
 // Criação da caixa de diálogo
 ....

 // Recebe a expressão digitada pelo usuário
 for(;;)
 {
  whois = SRGP_waitEvent(-1);
  if ( whois == KEYBOARD )
  {
   SRGP_getKeyboard(buffer, 40);
   strcat(buffer, " ");
   break;
```

```
 }
}

Pascal_Like_Expression expr(buffer,FALSE);

// Impressao dos resultados na tela
SRGP_copyPixel(dialogo, aux, dest_corner);
pos = SRGP_defPoint(dest_corner.x+10, dest_corner.y+130);
SRGP_inquireTextExtent("RESULTS ",&width,&height,&descent);
SRGP_text(pos,"RESULTS ");
pos.x += 10;
pos.y -= (3*height);
SRGP_inquireTextExtent("Expression Value =>
                       ",&width,&height,&descent);
SRGP_text(pos,"Expression Value => ");
pos.x +=(width+10);

// Print the expression value on the screeen
{
 char *out;
 if (expr.GetError()==NULL)
 {
  out=new char[20];
  realPoint pt;
  switch (expr.GetFlag())
  {
   case VALUE:
              {
               float evalue=expr.GetValue(sym);
               ftoa(evalue, out);
              }
              break;
   case POINT:
              {
               pt=expr.GetPoint(sym);
               char *out_aux=new char[20];
               strcpy(out, "["); ftoa(pt.x, out_aux);
               strcat(out, out_aux); strcat(out, ", ");
               strcpy(out_aux, ""); ftoa(pt.y, out_aux);
               strcat(out, out_aux); strcat(out, "]");
               delete out_aux;
              }
              break;
   case POLAR:
              {
               pt=expr.GetPoint(sym);
               pt=cartToPolar(pt);
               char *out_aux=new char[20];
               strcpy(out, "["); ftoa(pt.x, out_aux);
               strcat(out, out_aux); strcat(out, "< ");
               strcpy(out_aux, ""); ftoa(pt.y, out_aux);
               strcat(out, out_aux); strcat(out, "]");
               delete out_aux;
              }
              break;
  }
 }
 else
 {
  out=expr.GetError();
  pos.x=(width+20);
  pos.y-=20;
 }

 SRGP_text(pos, out);
 pos.x =(width+20);
```

```
 pos.y-=70;
 SRGP_text(pos, "Hit any key to continue");
 delete out;
}

// Escreve o resultado na caixa de diálogo
....

// Fecha a caixa de diálogo
....

return 0;
}
```

GetToken realiza um *parser* (interpretação) na expressão e passa cada token encontrado ao método Build_Expression que cria os objetos de expressão apropriados. O método Build adiciona os objetos de expressão à arvore de expressões. Build verifica a prioridade de cada um dos objetos de acordo com a Tabela 4.1, constrói a pilha temporária e depois a árvore de expressões.

No loop da função expression, o método GetToken retorna o tipo do token. Dentro do loop são feitos alguns testes que têm duas finalidades.

Se o tipo do token for VARIABLE, é chamada a função Get_Variable, que cria uma caixa de diálogo onde o usuário deve digitar o valor da variável. Se a variável representa um valor simples, o usuário deverá apenas digitar o valor numérico da variável. Se a variável representa coordenadas de um ponto, o usuário deve seguir a seguinte sintaxe. Em coordenadas cartesianas, a abcissa e a ordenada devem ser separadas por vírgula e envolvidas por colchetes; por exemplo, [3.12, 4.5]. Em coordenadas polares, deve ser usado o símbolo < como separador entre o valor da distância à origem do sistema de coordenadas e o ângulo em graus; por exemplo [2.25 < 45]. Get_Variable usa então a função Set, de Symbol_Table, para armazenar o valor da variável na tabela de símbolos.

```
int Base_Expression :: Get_Variable(char *var)
{
 rectangle aux;
 point dest_corner, pos;
 locator_measure measure;
 attribute_group attrib;
```

```
canvasID dialogo;
inputDevice whois;
int width, height, descent;
char buffer[40]={""};
realPoint pt;

char *coordx = new char[16];
char *coordy = new char[16];

if ((coordx == NULL) || (coordy == NULL))
 return 0;

strcpy(buffer, var);
// Se a variavel j  existe em Symbol_Table
if (sym.Is_In(buffer))
 switch(sym.Get_Flag(buffer))
 {
  case VALUE: pt.x = sym.Get_Value(buffer);
              ftoa(pt.x, coordx);
              break;
  case POINT: pt = sym.Get_Point(buffer);
            ftoa(pt.x, coordx);
              ftoa(pt.y, coordy);
              break;
  case POLAR: pt = sym.Get_Point(buffer);
              pt = cartToPolar(pt);
              ftoa(pt.x, coordx);
              ftoa(pt.y, coordy);
              break;
 }

//Constroi a caixa de dialogo
...
whois = SRGP_waitEvent(-1);
if ( whois == KEYBOARD )
{
 char b[40]={""};
 SRGP_getKeyboard(b, 40);

 // Se o usuario nao digitar nenhum valor para a variavel
 if (!strcmp(b, ""))
 { //ela nao vai ser adicionada a tabela de simbolos
  delete coordy;
  delete coordx;
  return sym.Get_Flag(var);
 }

 sym.Set(var, VALUE);
 char cd[40]={""};
 int position=0, type=START, error=VALID, error_flag=VALID;
 int kk=0;
 Pascal_Like_Expression expr(b, FALSE);
 expr.GetToken(b, position, cd, type, error);
 if (type==OPEN_COLCH) // variavel ponto
 {
  while (type!=CLOSE_COLCH && error_flag==VALID)
  {
   expr.GetToken(b, position, cd, type, error);
   if (error!=VALID)
   {
    error_flag=error;
    break;
   }
   if (type==LITERAL)
    switch(kk)
```

```
      {
       case 0: strcpy(coordx, cd);
               kk++;
               break;
       case 1: strcpy(coordy, cd);
               kk++;
               break;
       default: break;
      }
     if (type==VARIABLE)
     {
      if(Get_Variable(cd))
       switch(kk)
       {
        case 0: if (sym.Is_In(cd))
                {
                 ftoa(sym.Get_Value(cd),coordx);
                 kk++;
                }
                break;
        case 1: if (sym.Is_In(cd))
                {
                 ftoa(sym.Get_Value(cd),coordy);
                 kk++;
                }
                break;
        default: break;
       }
     }
     if (type==SEPARATOR)
      sym.Set(var, POINT);
     if (type==ANG)
      sym.Set(var, POLAR);
    }

    if ( sym.Get_Flag(var) == POINT )
    {
     pt.x = atof(coordx);
     pt.y = atof(coordy);
     sym.Set(var, pt);
    }
    if ( sym.Get_Flag(var) == POLAR )
    {
     pt.x = atof(coordx);
     pt.y = atof(coordy);
     pt = polarToCart(pt);
     sym.Set(var, pt);
    }
   }
   else // avaliar o valor da expressao e associar `a variavel
    if (type == LITERAL)
    {
     sym.Set(var, VALUE);
     sym.Set(var, (float)atof(cd));
    }
  }

  delete coordx;
  delete coordy;
  return sym.Get_Flag(var);
 }
```

Se o tipo do token retornado não for VARIABLE, os outros testes têm o objetivo de determinar, baseados nos

tipos dos tokens que compõem a expressão, qual será o tipo de retorno: int, float ou realPoint (representação cartesiana ou polar).

Quando um objeto Stop_Obj é retornado por GetToken, o loop é terminado. Neste ponto, a árvore de expressões está construída e a tabela de símbolos preenchida. Então, expression chama o método GetValue ou GetPoint de exp, dependendo dos resultados dos testes realizados dentro do loop e mostra ao usuário o valor da expressão na caixa de diálogo.

O sistema parametrizado permite que uma expressão seja associada a um ponto da geometria. Neste caso, o valor das coordenadas do ponto corresponde ao valor numérico da expressão. Se uma variável da expressão tem seu valor alterado, o valor da expressão deve ser atualizado e, consequentemente, as coordenadas do ponto associado também.

Para implementar este comportamento, foi criada uma lista (No_List) que armazena todas as expressões que foram analisadas pelo avaliador de expressões juntamente com seu valor numérico. Quando alguma variável da tabela de símbolos tem seu valor alterado pela função Get_Variable, a função update_coords é chamada. Ela percorre lista, atualizando os valores numéricos de cada uma das expressões existentes.

```
void update_coords(void)
{
 int n = no.numElements();
 for (int i=1; i<=n; i++)
 {
  // Se existir uma expressao para o ponto, as coordenadas do
  // ponto sao atualizadas
  if (strcmp(no[i].retorna_exp(), "none"))
  {
   Pascal_Like_Expression exp(no[i].retorna_exp(), FALSE);
   switch(no[i].retorna_tipo())
   {
    case VALUE: exp.GetValue(sym);
                break;
    case POINT: exp.GetPoint(sym);
                break;
   } // switch
  } // if strcmp
 } // for
}

class  No_Node:public ListNode
```

```
{
 public:
  int tipo;             // Tipo do dado (POINT ou VALUE)
  realPoint pontoNo;    // Coordenada no No (POINT) ou valor da
                        //  expressao (VALUE)
  char lside[20],       // Lado esquerdo da equação
  rside[40];            // Lado direito da equação

  // Construtores
  No_Node();
  No_Node(realPoint pt);
  // Destrutor
  ~No_Node();

  // Atribui tipo
  void atribui_tipo(int);
  int retorna_tipo(void);

  // Atribui labels `as expressoes de atribuicao
  void atribui_rside(char *);
  void atribui_lside(char *);
  char *retorna_rside(void);
  char *retorna_lside(void);
};

class No_List: public  ListaSimples
{
 public:
  No_Node& operator[] (unsigned int n); // operador de retorno
  // Construtor
  No_List();
  // Destrutor
  ~No_List();

  // Retorna a posicao do ponto pt, caso encontre
  unsigned int acha_ponto(realPoint pt);
  // Retorna a posicao da expressao caso encontre
  unsigned int acha_expressao(char *str);
  // Trata uma expressao com atribuicao de valor
  void atrib(char *, char *, float &);
  void atrib(char *, char *, realPoint &);
};
```

4.5 O subsistema de interface

O subsistema de interface é responsável pela iteração com o usuário. Ele cria uma janela de comunicação que tem a função de recolher dados relativos à parametrização. O usuário pode interagir com o sistema parametrizado de três maneiras.

4.5.1 Criando parâmetros

Neste caso, o usuário pode criar um parâmetro sem relacioná-lo a um ponto da geometria. O tipo deste parâmetro deve se enquadrar em um dos tipos permitidos, ou seja, real, inteiro ou ponto. A sintaxe

de números reais e inteiros dispensa apresentações. O tipo ponto, porém, segue uma sintaxe especial: ele deve ser limitado por colchetes e seus componentes separados por vírgula. Por exemplo, [2, 3.12] é um ponto cujas coordenadas são o número inteiro 2 e o número real 3.12.

Para que o usuário possa criar um parâmetro não relacionado a nenhum ponto da geometria ele deve digitar "expression" na linha de comando ou selecionar "Expression" no menu EDIT. A figura 4.2 mostra a janela de comunicação que se abrirá para o usuário.

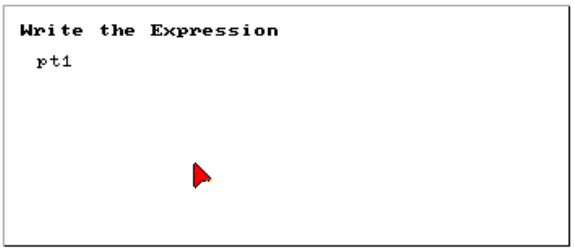

Fig. 4.2

Nesta janela, o usuário deverá digitar o nome de uma variável ou uma expressão composta de uma ou mais variáveis. No exemplo, o usuário digitou o nome de uma variável. Posteriormente, outra janela se abrirá pedindo o valor da variável. A Fig. 4.3 mostra a janela onde o usuário digitou o valor da variável pt1, do tipo ponto. Resumindo: inicialmente o usuário criou uma variável chamada pt1. Posteriormente, informou ao sistema o tipo desta variável (tipo ponto) e o seu valor.

O nome escolhido para uma variável não tem nenhuma relação com o seu tipo. O sistema é capaz de reconhecer o tipo da variável criada através do valor dado a ela pelo usuário. Daí a importância do uso da sintaxe correta.

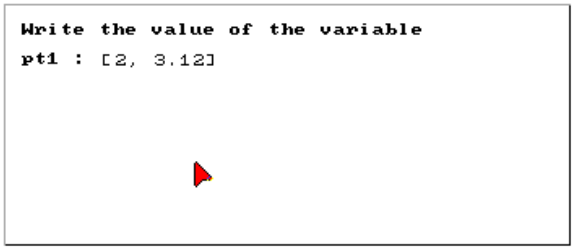

Fig. 4.3

Toda vez que o usuário digitar o nome de uma variável já existente, o sistema assume que o usuário quer alterar o valor desta variável e então mostra o nome da variável seguido de seu valor corrente entre parênteses na janela de atribuição de valor. O usuário tem a opção de digitar um novo valor do mesmo tipo do valor corrente da variável ou concordar em manter este valor, teclando <ENTER> (Fig. 4.4).

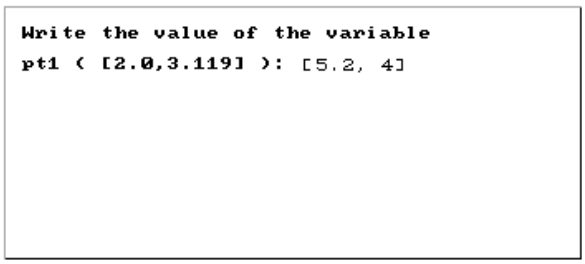

Fig. 4.4

4.5.2 Criando parâmetros relacionados a pontos da geometria

O usuário pode também criar parâmetros ou expressões relacionados a pontos da geometria. Neste caso, o valor deste parâmetro ou expressão deverá ser do tipo ponto.

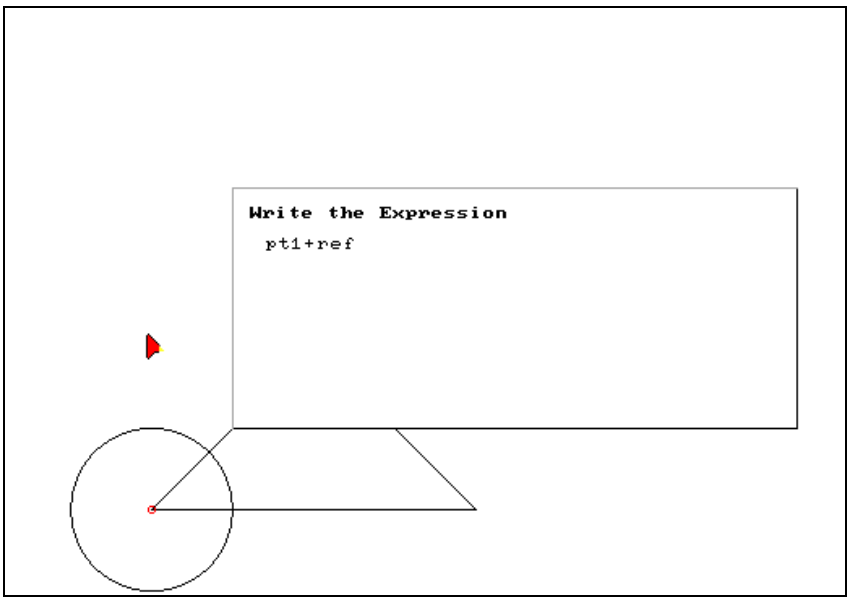

Fig. 4.5

O procedimento é um pouco diferente do caso anterior. A partir de uma geometria existente (ou em construção), o usuário deve escolher um ponto e *clicar* sobre ele com o botão esquerdo do *mouse* pressionando ao mesmo tempo a tecla <SHIFT>. Uma janela se abrirá para que o usuário digite uma expressão para o ponto escolhido (Fig. 4.5).

A Fig. 4.5 mostra a janela onde o usuário digitou uma expressão contendo duas variáveis: pt1 e ref. A primeira já havia sido criada anteriormente. A segunda variável, ref, será criada pelo sistema e receberá o valor digitado pelo usuário na janela correspondente. Depois que todas as variáveis que compõem a expressão tiverem sido criadas, o sistema calculará o valor da expressão. O valor desta expressão, neste caso, será associado ao ponto (Fig. 4.6).

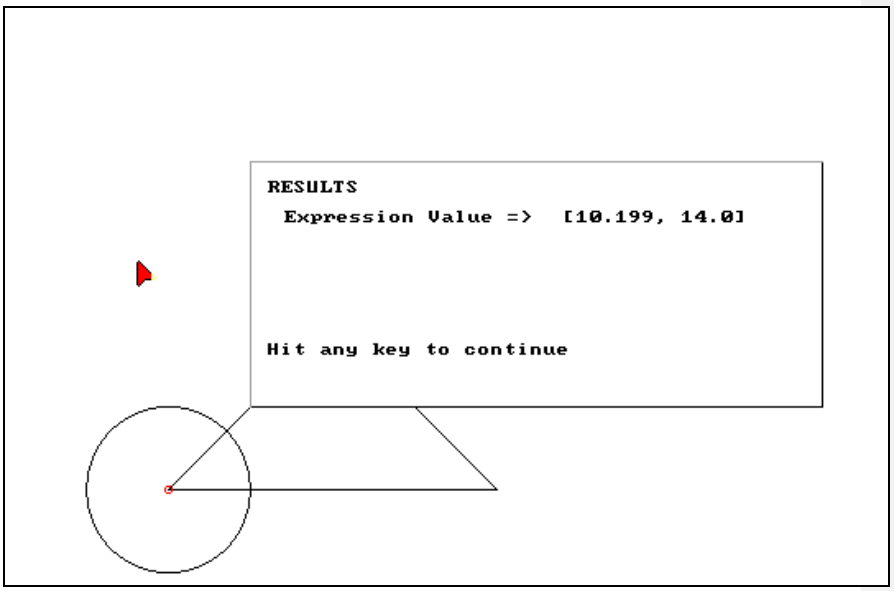

Fig. 4.6

O ponto será então deslocado para a posição correspondente a este valor e levará com ele as entidades que o compartilham. Pela Fig. 4.7 podemos ver que foram deslocados o círculo e as duas retas que compartilham o ponto parametrizado pela expressão (pt1 + ref).

Deve-se ressaltar que todas as expressões construídas ficam armazenadas internamente em uma estrutura de dados tipo lista encadeada [Ziviani, 1992]. Toda vez que uma variável é alterada, esta lista é percorrida e as expressões que dependem desta variável são recalculadas.

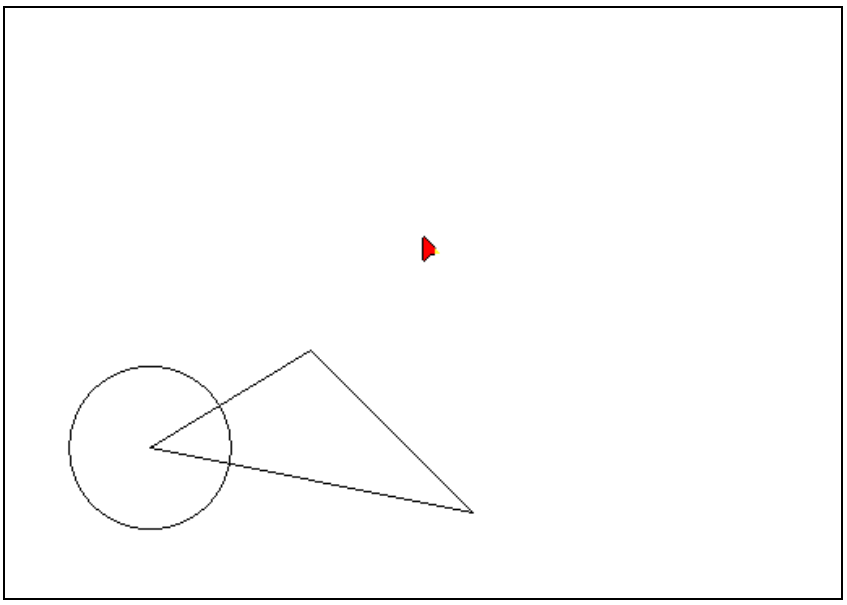

Fig. 4.7

4.5.2 Criando uma família de geometrias

Com o objetivo de automatizar o trabalho do projetista, foi incorporada uma função que cria uma família de geometrias a partir de uma geometria base. Neste caso é necessário que pelo menos um de seus pontos tenha sido parametrizado.

A geometria base será um modelo para as outras geometrias que terão sua construção baseada na alteração do valor de um ou mais parâmetros desta geometria base.

O procedimento é o seguinte. A partir da geometria base já existente, o usuário deve digitar "adg" na linha de comando ou selecionar "ADG" no menu EDIT (Fig. 4.8). Uma janela se abrirá para que o usuário digite o nome da variável cuja alteração de valor servirá de base para a criação das outras geometrias.

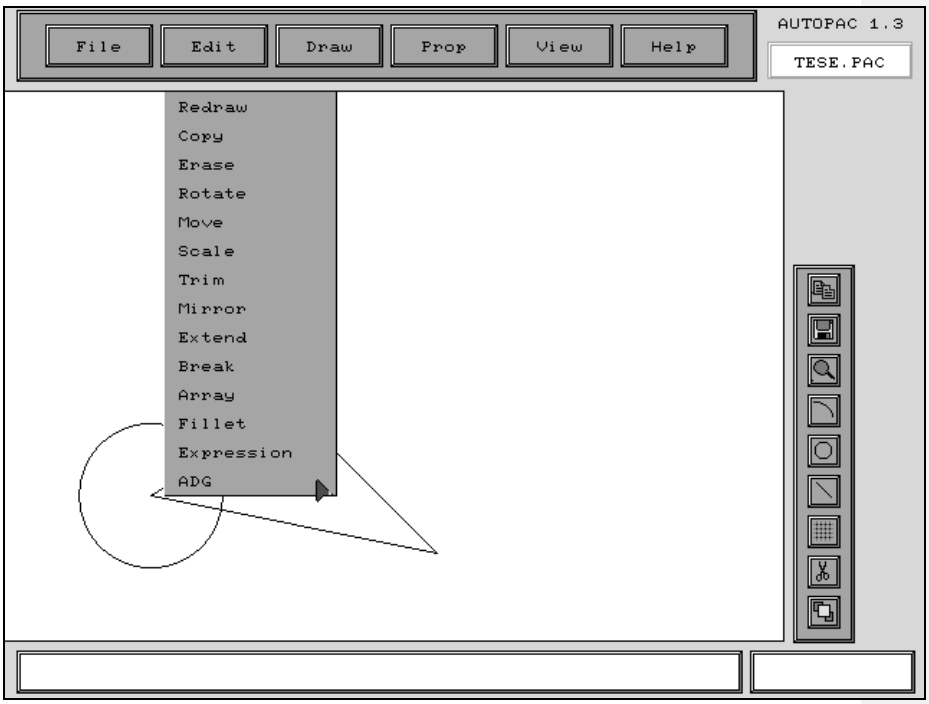

Fig. 4.8

Depois de receber o nome da variável, o usuário será questionado quanto aos valores incrementais que serão somados às coordenadas 0x e 0y da variável na criação de cada uma das geometrias derivadas. Finalmente, o usuário deverá digitar o número de geometrias que ele deseja criar. Feito isso, o sistema criará as geometrias derivadas, gravando cada uma em um arquivo independente, cujo nome também será derivado do nome do arquivo da geometria base. O valor da variável usada não é alterado no final desta operação.

4.6 O subsistema de armazenamento

Este subsistema foi desenvolvido para possibilitar o armazenamento persistente dos dados criados em um projeto desenvolvido no AutoPAC.

O subsistema de armazenamento é parte integrante do modelador geométrico do AutoPAC e trabalha com arquivos do formato arquivo neutro e PAC.

4.6.1 O arquivo neutro

O arquivo neutro é utilizado para a intercomunicação de dados em (e entre) programas de cálculo de campos eletromagnéticos. Seu formato visa padronizar os formatos de transferência de dados, melhorando a compatibilidade e a portabilidade de tais dados/arquivos, e organiza-los de modo a facilitar ao usuário sua geração e manipulação [Rocha, 1995].

A base de dados do AutoPAC por enquanto é dividida em dois arquivos: Geometria, que contém os dados relativos à geometria do problema e Malhas, que contém os dados da geração das malhas de elementos finitos, detalhes do cálculo e condições de contorno. Um terceiro arquivo, Materiais, será gerado quando forem implementados serviços que proporcionem um tratamento relativo à propriedades físicas de materiais associados a regiões da geometria.

Estes arquivos estão em formato ASCII, de modo seqüencial. A base de dados é secionada em blocos de dados, cada uma delas começando com um cabeçalho (label) precedido por um asterisco (*) que permite ao programa selecionar o bloco cuja leitura é importante, desprezando os blocos que não são necessários [Rocha, 1995].

4.6.2 O arquivo PAC

Este arquivo é utilizado quando o usuário quer armazenar apenas dados da geometria do problema.

O formato do arquivo PAC é muito parecido com o formato do arquivo neutro. A diferença está na informação que é armazenada nele. São informações

relativas à geometria do problema e também às variáveis e expressões criadas pelo usuário.

Existem *labels* para arco, círculo e reta, as entidades suportadas pelo AutoPAC.

```
%GLOBAL
0.0000000000
4.0000000000
54.0000000000
42.0000000000
%ATTRIBUTES
0
0 0 639 479
1
0 1
10 0
1 0
-1
0 0 0
0 0 0
%VARIABLES
   2
FLOAT centro 5.00000000
POINT raio 10.00000000 10.00000000
%EXPRESSIONS
   1
POINT raio 10.00000000 10.00000000
%CIRCLE
10.0000000000 10.0000000000
13.0000000000
4 1 0
none
```

Existe um *label* %VARIABLES sob o qual é gravado o número de variáveis e logo abaixo são gravadas os dados das variáveis. Cada variável tem seus dados gravados em uma linha que tem até quatro colunas, sendo colocada na primeira o tipo da variável, na segunda o seu nome e na terceira e quarta o valor da variável.

Analogamente, existe um *label* %EXPRESSIONS sob o qual é gravado o número de expressões e logo abaixo são gravadas os dados das expressões. Cada linha tem quatro colunas. Na primeira é colocado o tipo da expressão. Na segunda coluna é colocada a expressão e na terceira e quarta o valor atual da mesma.

4.7 Conclusão

O sistema parametrizado foi desenvolvido tendo como base o avaliador de expressões descrito no capítulo 3. Várias classes foram adicionadas ao avaliador, uma interface foi criada e o subsistema de armazenamento foi adaptado para se adequar às necessidades do sistema parametrizado. O uso do paradigma da orientação a objetos permitiu que esse sistema fosse construído de forma modular, facilitando sua compreensão e futura extensão.

CAPÍTULO 5 - EXEMPLOS DE APLICAÇÃO

Neste capítulo são mostrados exemplos de aplicação do sistema CAD de parametrização.

Exemplo 1

Neste exemplo foi criada uma topologia que representa uma máquina elétrica síncrona de pólos salientes. Esta geometria é constituída de dois arcos e seis retas, como mostra a Fig. 5.4. Deseja-se parametrizar a distância dos pólos do rotor ao estator da máquina.

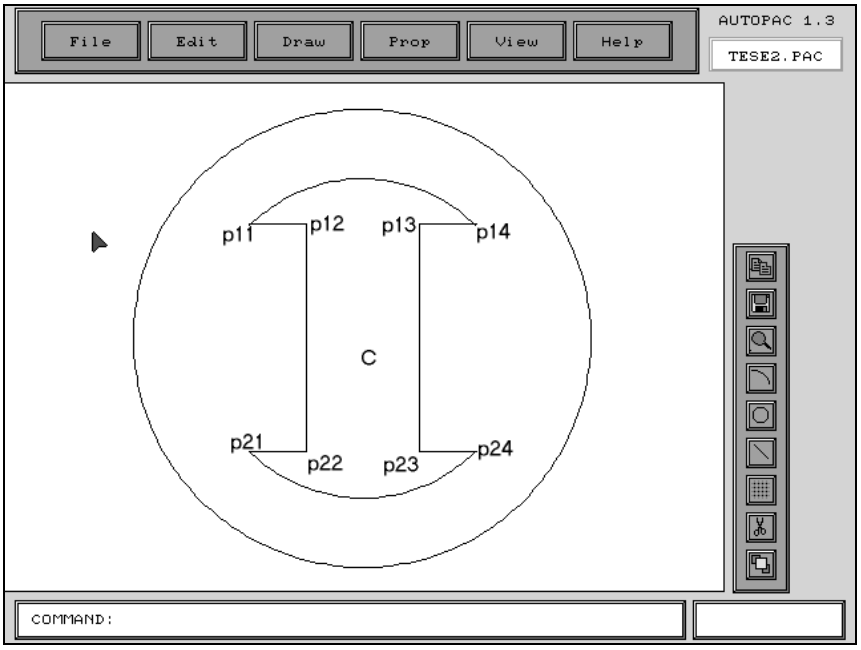

Fig. 5.4

A geometria deve ser parametrizada de modo que as entidades geométricas que compõem o rotor tenham seus parâmetros (pontos) recalculados para qualquer valor escolhido para a distância entre um polo do rotor e um ponto do estator. Deve-se garantir que as formas do rotor e do estator da máquina não sejam deformados pela parametrização.

Antes de se efetuar a parametrização é necessário escolher os pontos que serão parametrizados e a expressão adequada para estes pontos a fim de se obter o efeito desejado.

Os dois pólos do rotor devem sempre manter a mesma distância d em relação ao estator. Sendo assim, quanto menor for esta distância, mais afastados ficarão os pólos entre si e vice-versa. Por esta razão, todos os pontos da geometria do rotor devem ser parametrizados.

Na Fig. 5.4 são mostrados todos os pontos que foram parametrizados. Os pontos p_{11} e p_{14} são os pontos inicial e final do arco do polo superior e os pontos p_{12} e p_{13} são pontos pertencentes às duas retas que compõem o polo. A posição relativa destes quatro pontos deve ser mantida para qualquer valor da distância d. Consequentemente, as expressões matemáticas relacionadas a cada um destes pontos deve ser função desta distância.

Seja R_e o raio do rotor e R_r o raio do estator. A distância d pode ser representada em função destas duas grandezas, como mostra a expressão abaixo:

$$d = R_e - R_r \qquad [5.1]$$

A medida que o valor de d aumentar, R_r diminuirá, e vice-versa. O raio dos arcos que compõem o rotor é igual ao raio do rotor, e sua expressão matemática pode ser derivada de [5.1].

$$Rr = R_e - d \qquad [5.2]$$

Se dividirmos a representação mostrada na Fig. 5.4 em quatro quadrantes e analisarmos o primeiro deles, podemos deduzir as expressões matemáticas para as coodenadas do ponto p_{14}:

$$x_{14} = x_c + (R_e - d)\text{sen } 45 \qquad [5.3]$$

$$y_{14} = y_{c13} + (R_e - d)\cos 45 \qquad [5.4]$$

onde x_c e y_c são as coordenadas do ponto central C do arco que constitui o rotor.

Podemos definir uma constante k que representa a metade da distância entre a abscissa do ponto p_{14} e a abscissa do ponto central. Esta constante é usada na composição da expressão do ponto p_{13}, como mostrado abaixo.

$$x_{13} = x_{14} - k \qquad [5.5]$$

$$y_{13} = y_{14} \qquad [5.6]$$

As mesmas observações são válidas para os pontos p_{21}, p_{22}, p_{23}, p_{24}. Contudo, o polo inferior deverá ser deslocado para cima quanto maior for o valor da distância. Portanto, as expressões para estes pontos são ligeiramente diferentes das expressões mostradas deduzidas para os pontos p_{14} e p_{13}.

No arquivo PAC correspondente listado abaixo são mostradas as variáveis e as expressões construídas com estas variáveis.

```
%GLOBAL
-28.1578947368
-5.0000000000
128.1578947368
105.0000000000
%ATTRIBUTES
0
0 0 640 480
1
0 1
4 0
1 0
-1
0 0 0
0 0 0
%VARIABLES
  22
FLOAT y24 20.30599976
FLOAT x24 79.69400024
FLOAT y21 20.30599976
FLOAT x21 20.30599976
FLOAT y14 79.69400024
FLOAT x14 79.69400024
FLOAT y11 79.69400024
FLOAT x11 20.30599976
FLOAT d 8.00000000
FLOAT Re 50.00000000
FLOAT yc 50.00000000
FLOAT xc 50.00000000
FLOAT k 14.84700012
POINT p24 79.69400024 20.30599976
POINT p21 20.30599976 20.30599976
POINT p14 79.69400024 79.69400024
POINT p11 20.30599976 79.69400024
```

```
POINT C 50.00000000 50.00000000
POINT p12 35.15299988 79.69400024
POINT p13 64.84700012 79.69400024
POINT p22 35.15299988 20.30599976
POINT p23 64.84700012 20.30599976
%EXPRESSIONS
 272
FLOAT x11=xc-(Re-d)*0.707 20.30599976 0.00000000
FLOAT y11=yc+(Re-d)*0.707 79.69400024 0.00000000
FLOAT x14=xc+(Re-d)*0.707 79.69400024 0.00000000
FLOAT y14=y11 79.69400024 0.00000000
FLOAT x21=x11 20.30599976 0.00000000
FLOAT y21=yc-(Re-d)*0.707 20.30599976 0.00000000
FLOAT x24=x14 79.69400024 0.00000000
FLOAT y24=y21 20.30599976 0.00000000
FLOAT k=0.5*(Re-d)*0.707 14.84700012 0.00000000
POINT C=[xc,yc] 50.00000000 50.00000000
POINT p11=[x11,y11] 20.30599976 79.69400024
POINT p14=[x14,y14] 79.69400024 79.69400024
POINT p21=[x21,y21] 20.30599976 20.30599976
POINT p24=[x24,y24] 79.69400024 20.30599976
POINT none 83.57894737 64.18421053
POINT none 42.03947368 87.24561404
POINT none 45.80263158 87.24561404
POINT none 42.03947368 82.22807018
POINT none 45.80263158 82.22807018
POINT none 33.06578947 94.19298246
POINT none 36.82894737 94.19298246
POINT none 33.06578947 89.17543860
POINT none 36.82894737 89.17543860
POINT none 78.36842105 79.81578947
POINT none 30.75000000 89.56140351
POINT none 34.51315789 89.56140351
POINT none 30.75000000 84.54385965
POINT none 34.51315789 84.54385965
POINT none 81.81500244 81.81500244
POINT none 18.18499947 81.81500244
POINT none 46.09210526 97.08771930
POINT none 49.85526316 97.08771930
POINT none 46.09210526 92.07017544
POINT none 49.85526316 92.07017544
POINT none 72.57894737 81.84210526
POINT none 27.42896120 82.41317152
POINT none 26.11842105 88.11403509
POINT none 29.88157895 88.11403509
POINT none 26.11842105 83.09649123
POINT none 29.88157895 83.09649123
POINT p12=p11+[k,0] 35.15299988 79.69400024
POINT p13=p14-[k,0] 64.84700012 79.69400024
POINT p22=p21+[k,0] 35.15299988 20.30599976
POINT p23=p24-[k,0] 64.84700012 20.30599976
POINT none 32.77631579 73.06140351
POINT none 36.53947368 73.06140351
POINT none 32.77631579 68.04385965
POINT none 36.53947368 68.04385965
POINT none 64.03947368 60.61403509
POINT none 67.80263158 60.61403509
POINT none 64.03947368 55.59649123
POINT none 67.80263158 55.59649123
POINT none 20.61842105 87.82456140
POINT none 24.38157895 87.82456140
POINT none 20.61842105 82.80701754
POINT none 24.38157895 82.80701754
POINT none 52.02631579 58.97368421
POINT none 65.19736842 72.48245614
POINT none 68.96052632 72.48245614
```

```
POINT none 65.19736842 67.46491228
POINT none 68.96052632 67.46491228
POINT none 76.77631579 86.66666667
POINT none 80.53947368 86.66666667
POINT none 76.77631579 81.64912281
POINT none 80.53947368 81.64912281
POINT none 22.06578947 25.00877193
POINT none 25.82894737 25.00877193
POINT none 22.06578947 19.99122807
POINT none 25.82894737 19.99122807
POINT none 74.46052632 25.00877193
POINT none 78.22368421 25.00877193
POINT none 74.46052632 19.99122807
POINT none 78.22368421 19.99122807
POINT none 28.72368421 84.06140351
POINT none 32.48684211 84.06140351
POINT none 28.72368421 79.04385965
POINT none 32.48684211 79.04385965
POINT none 31.61842105 20.37719298
POINT none 35.38157895 20.37719298
POINT none 31.61842105 15.35964912
POINT none 35.38157895 15.35964912
POINT none 66.64473684 20.66666667
POINT none 70.40789474 20.66666667
POINT none 66.64473684 15.64912281
POINT none 70.40789474 15.64912281
POINT none 68.67105263 84.06140351
POINT none 72.43421053 84.06140351
POINT none 68.67105263 79.04385965
POINT none 72.43421053 79.04385965
POINT none 100.00000000 50.00000000
POINT none 0.00000000 50.00000000
POINT none 23.44759241 2020.30599976
POINT none 23.44759241 2079.69400024
POINT none 38.29459253 2020.30599976
POINT none 38.29459253 2079.69400024
POINT none 67.98859278 2020.30599976
POINT none 67.98859278 2079.69400024
POINT none 82.83559290 2020.30599976
POINT none 82.83559290 2079.69400024
POINT none 14.64466094 14.64466094
POINT none 27.38576592 28.07323371
POINT none 0.61558297 42.17827675
POINT none 2.44717419 34.54915028
POINT none 5.44967379 27.30047501
POINT none 9.54915028 20.61073739
POINT none 35.15299988 37.27399990
POINT none 35.15299988 45.75799997
POINT none 35.15299988 54.24200003
POINT none 35.15299988 62.72600010
POINT none 35.15299988 71.21000017
POINT none 26.66957371 84.91645043
POINT none 41.80744376 91.18676151
POINT none 58.19255624 91.18676151
POINT none 66.07027713 88.79708100
POINT none 73.33042629 84.91645043
POINT none 64.84700012 71.21000017
POINT none 64.84700012 62.72600010
POINT none 64.84700012 54.24200003
POINT none 64.84700012 45.75799997
POINT none 64.84700012 37.27399990
POINT none 64.84700012 28.78999983
POINT none 90.45084972 20.61073739
POINT none 94.55032621 27.30047501
POINT none 97.55282581 34.54915028
POINT none 99.38441703 42.17827675
```

```
POINT none 99.38441703 57.82172325
POINT none 97.55282581 65.45084972
POINT none 94.55032621 72.69952499
POINT none 90.45084972 79.38926261
POINT none 85.35533906 85.35533906
POINT none 79.38926261 90.45084972
POINT none 72.69952499 94.55032621
POINT none 65.45084972 97.55282581
POINT none 57.82172325 99.38441703
POINT none 50.00000000 100.00000000
POINT none 42.17827675 99.38441703
POINT none 34.54915028 97.55282581
POINT none 27.30047501 94.55032621
POINT none 20.61073739 90.45084972
POINT none 14.64466094 85.35533906
POINT none 9.54915028 79.38926261
POINT none 5.44967379 72.69952499
POINT none 2.44717419 65.45084972
POINT none 0.61558297 57.82172325
POINT none 27.68380774 72.22480810
POINT none 35.15299988 28.78999983
POINT none 72.31619226 72.22480810
POINT none 72.61423408 28.07323371
POINT none 28.80328616 34.58073618
POINT none 71.19671384 34.58073618
POINT none 28.74748145 65.61592172
POINT none 71.25251855 65.61592172
POINT none 28.77341829 41.49779563
POINT none 71.22658171 41.49779563
POINT none 28.68483487 58.54820317
POINT none 71.31516513 58.54820317
POINT none 86.70710523 27.50585698
POINT none 78.64589946 35.25935568
POINT none 89.97405579 34.34731349
POINT none 92.30017910 41.03267441
POINT none 93.87257871 47.25681338
POINT none 85.14939835 41.74662795
POINT none 78.27331889 44.02083602
POINT none 88.35878767 46.64899723
POINT none 82.99412393 47.92053546
POINT none 93.90713143 53.61641532
POINT none 89.38563864 51.29833722
POINT none 92.71808485 60.18134428
POINT none 88.24615424 56.41366220
POINT none 83.80090911 53.87803058
POINT none 90.38250531 66.71750656
POINT none 86.60611394 61.83172961
POINT none 83.02179029 58.49575092
POINT none 87.17111583 72.91633608
POINT none 82.99856724 78.49869531
POINT none 78.11869858 83.38029543
POINT none 72.57526102 87.39474700
POINT none 66.47001275 90.46568258
POINT none 59.94524051 92.53578714
POINT none 53.16764679 93.56198722
POINT none 46.31085529 93.51933998
POINT none 39.54647557 92.40688697
POINT none 33.04091935 90.25041076
POINT none 26.95345617 87.10240601
POINT none 21.43340810 83.04043714
POINT none 16.61657628 78.16475861
POINT none 12.62164585 72.59558665
POINT none 9.54707170 66.47009457
POINT none 7.46859754 59.93909041
POINT none 6.43740462 53.16336014
POINT none 6.47887549 46.30973134
```

```
POINT none 7.59198184 39.54696283
POINT none 9.74931257 33.04158026
POINT none 13.38309753 26.93117757
POINT none 21.30051346 35.02754222
POINT none 14.36813639 40.74411443
POINT none 21.19020750 44.20461633
POINT none 84.04513625 67.28949479
POINT none 81.21125092 62.87636031
POINT none 80.69258035 72.47909807
POINT none 76.52360453 77.12068653
POINT none 77.08941678 66.79115244
POINT none 76.36471323 61.46815396
POINT none 79.03871461 58.35637122
POINT none 80.12948800 54.95063255
POINT none 79.69525994 51.21168642
POINT none 75.49306642 56.33861980
POINT none 76.05555371 52.26897508
POINT none 74.87744102 48.28145374
POINT none 18.90248980 72.74556333
POINT none 22.72782463 66.74478885
POINT none 23.43480227 61.24653536
POINT none 14.96572454 48.97485590
POINT none 13.43965031 56.82323770
POINT none 13.94481495 62.60771571
POINT none 15.68213276 67.66567693
POINT none 18.86632218 62.52979051
POINT none 17.76238777 58.37544155
POINT none 19.14467066 53.57941611
POINT none 23.59870642 55.83313389
POINT none 25.61590052 50.50304474
POINT none 66.07027713 11.20291900
POINT none 58.84313483 15.84902454
POINT none 33.92972287 11.20291900
POINT none 41.80744376 8.81323849
POINT none 50.00000000 8.00634213
POINT none 58.19255624 8.81323849
POINT none 41.15686517 84.15097546
POINT none 58.84313483 84.15097546
POINT none 41.15686517 15.84902454
POINT none 52.71912888 85.74591272
POINT none 52.71912888 14.25408728
POINT none 47.11321423 14.08742831
POINT none 47.11321423 85.91257169
POINT none 42.01478629 23.27701201
POINT none 46.96674981 19.68983734
POINT none 54.39987568 20.35510063
POINT none 59.40096186 23.08725395
POINT none 57.98521371 76.72298799
POINT none 53.06481555 80.19345691
POINT none 48.71100052 81.60553178
POINT none 44.60218841 80.89499537
POINT none 40.79062650 77.98026966
POINT none 57.49964054 66.96800014
POINT none 57.49964054 58.48400007
POINT none 57.49964054 50.00000000
POINT none 57.49964054 41.51599993
POINT none 57.49964054 33.03199986
POINT none 52.22734394 26.47269584
POINT none 45.86640254 27.38062592
POINT none 41.50370700 30.55403200
POINT none 49.83316375 32.43539773
POINT none 44.93429237 34.07876131
POINT none 51.87230306 38.23843375
POINT none 46.94685662 38.34083591
POINT none 42.07015963 40.72399799
POINT none 51.84751622 44.75063405
```

```
POINT none 45.29082841 45.35836202
POINT none 41.44022145 51.80225951
POINT none 41.54063175 58.61499730
POINT none 52.51950039 74.19703265
POINT none 48.31481435 77.05936468
POINT none 44.17874772 75.21583872
POINT none 41.41990025 71.44745675
POINT none 40.16884816 66.38060603
POINT none 47.84332135 72.55566292
POINT none 44.91611792 70.51124449
POINT none 50.47874062 67.93395016
POINT none 51.75139782 61.95646391
POINT none 45.82649276 64.10154439
POINT none 47.67235398 57.68582327
POINT none 48.81058422 50.78037952
%ARC
50.0000000000 50.0000000000
41.9936578664
45.0000000000 135.0000000000
4 1 0
none
%ARC
50.0000000000 50.0000000000
41.9936578664
225.0000000000 315.0000000000
4 1 0
none
%LINE
20.3059997559 79.6940002441
35.1529998779 79.6940002441
4 1 0
none
%LINE
79.6940002441 79.6940002441
64.8470001221 79.6940002441
4 1 0
none
%LINE
20.3059997559 20.3059997559
35.1529998779 20.3059997559
4 1 0
none
%LINE
79.6940002441 20.3059997559
64.8470001221 20.3059997559
4 1 0
none
%LINE
35.1529998779 79.6940002441
35.1529998779 20.3059997559
4 1 0
none
%LINE
64.8470001221 20.3059997559
64.8470001221 79.6940002441
4 1 0
none
%ARC
50.0000000000 50.0000000000
50.0000000000
0.0000000000 180.0000000000
1 1 0
none
%ARC
50.0000000000 50.0000000000
50.0000000000
```

```
180.0000000000 0.0000000000
1 1 0
none
```

As Figs. 5.5a e 5.5b mostram a topologia calculada para valores de d iguais a 15 e 5 unidades, respectivamente.

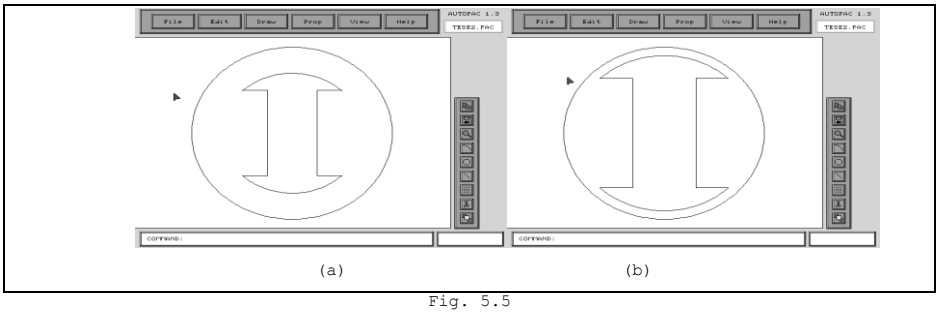

Fig. 5.5

A Fig. 5.6 mostra a malha gerada para d igual a 8 unidades.

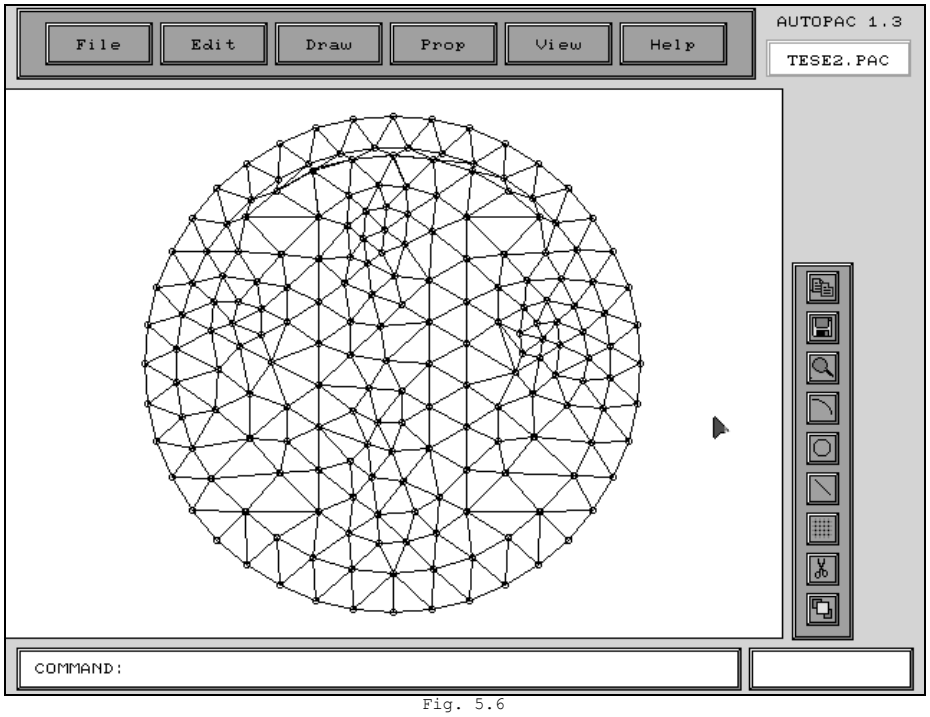

Fig. 5.6

Exemplo 2

Um outro exemplo de geometria paramétrica é mostrado na Fig. 5.7. Foi criado um modelo cujas coordenadas dos

centros dos dois círculos foram parametrizadas [Mesquita, 1997]. Na seqüência mostrada na figura, a coordenada y foi mantida constante e a coordenada x foi alterada. As Fig. 5.7(a), (b), (c) e (d) mostram a geometria avaliada para x=30, 35, 40 e 45, respectivamente.

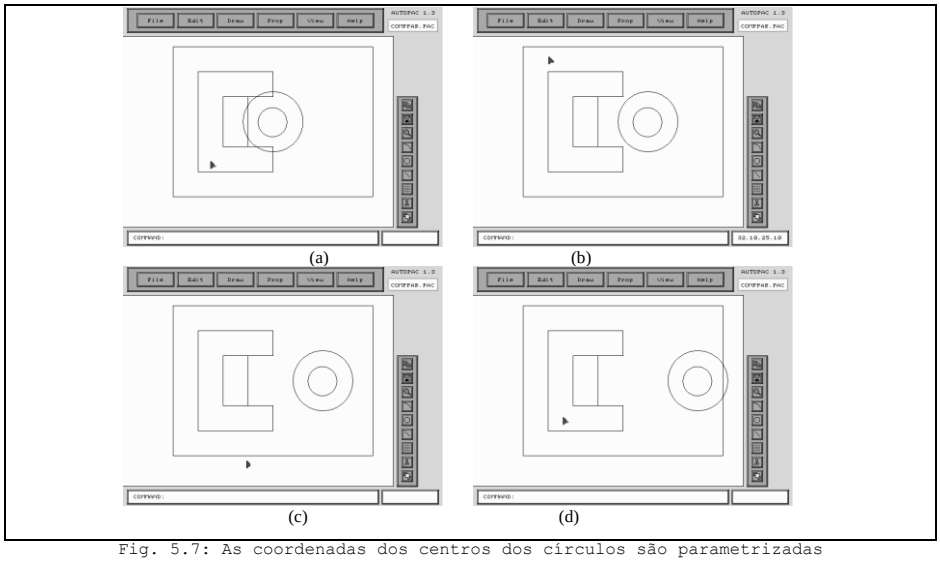

Fig. 5.7: As coordenadas dos centros dos círculos são parametrizadas

A Fig. 5.8 mostra a malha gerada para x=35 e x=45. Pode-se perceber facilmente que a alteração da topologia do modelo não provoca problemas para o gerador de malhas porque a topologia é recalculada toda vez que a geometria é alterada.

A malha de elementos finitos é usada pelo bloco processador no cálculo de algumas grandezas de interesse [Mesquita, 1997]. É possível otimizar a geometria usando o sistema parametrizado. Neste caso será necessário ter o sistema CAD de parametrização atuando em conjunto com um algoritmo de otimização (Fig. 4.1), conforme discutido no capítulo 4. O sistema CAD de parametrização é responsável pela criação, armazenamento e alteração das variáveis de interesse. Os valores destas variáveis serão usados como dados de entrada para o algoritmo de otimização em cada uma de suas iterações.

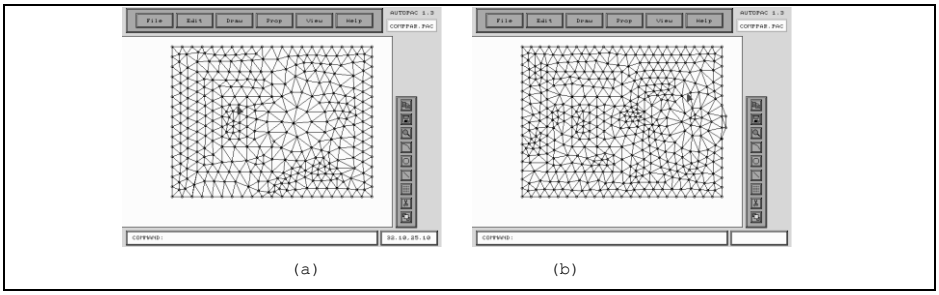

Fig. 5.8: Malha de elementos finitos para o modelo paramétrico

O algoritmo de otimização deverá usar os valores fornecidos pelo sistema parametrizado como dados de entrada para seus cálculos em cada iteração e fornecer como saída valores calculados que serão repassados ao bloco processador (Fig. 4.1). Baseado nestes valores, o processador calculará a(s) grandeza(s) de interesse (que neste exemplo poderia ser o campo elétrico). Os valores calculados pelo algoritmo de otimização deverão ser também enviados ao sistema parametrizado que atualizará os valores das variáveis e das expressões que delas dependem, redesenhando a geometria. Se o valor encontrado pelo processador não for o valor desejado, o ciclo se repetirá até que o valor desejado seja obtido. Neste momento, os valores atuais das variáveis serão os valores otimizados.

CONCLUSÃO

O sistema parametrizado apresentado neste trabalho tem por objetivo facilitar o projeto de dispositivos eletromagnéticos fornecendo ferramentas que podem elevar o grau de automatização deste processo.

Hoje o sistema paramétrico de parametrização está integrado a um pré-processador derivado de um modelador geométrico bidimensional. O objetivo final, no entanto, é adicionar outros blocos funcionais aos já existentes e construir um sistema CAD parametrizado completo para ser usado no projeto de dispositivos eletromagnéticos. O esquema mostrado na Fig. 4.1 descreve mais claramente esta idéia.

Para que este objetivo seja atingido, faltam ser adicionados os módulos processador e pós-processador. O módulo processador é um algoritmo que implementa, por exemplo, o Método de Elementos Finitos para o cálculo das grandezas eletromagnéticas de interesse. O outro módulo a ser adicionado, o pós-processador, deve apresentar graficamente ao projetista os resultados fornecidos pelo módulo processador, possibilitando uma intervenção deste na geometria do problema em busca dos resultados desejados. A inclusão destes dois módulos é essencial para que o sistema se torne completo.

É importante ressaltar que um sistema CAD paramétrico como este poderia ser aplicado ao projeto de dispositivos diversos, não apenas a dispositivos eletromagnéticos. Para isso, o método de cálculo utilizado no módulo processador seria outro, assim como as grandezas de interesse analisadas no módulo pós-processador. Portanto, estes dois últimos módulos são específicos, responsáveis diretos pela aplicação a que se destina o sistema. O sistema poderia ser geral se fornecesse ao projetista a opção por diferentes módulos de processamento em tempo de execução.

Outro módulo interessante que poderia ser futuramente adicionado a este sistema CAD paramétrico é um algoritmo de otimização. Este módulo, em conjunto com o sistema parametrizado, possibilitaria uma maior automatização do projeto além de prover resultados mais precisos, mais próximos do ideal.

Outra sugestão de aperfeiçoamento seria a extensão do sistema de parametrização, juntamente com o modelador geométrico, para suportar geometrias tridimensionais. Além disso, poderia-se também incluir novas primitivas geométricas que possibilitariam uma maior facilidade na construção de geometrias complexas entre outros benefícios.

O sistema parametrizado é fundamental para as pretensões futuras do GOPAC. Até agora, o sistema CAD estava restrito a funcionalidades bem conhecidas em softwares desta área. A inclusão do sistema de parametrização proporciona um novo leque de aplicações para o sistema CAD. A inclusão de alguns dos blocos funcionais propostos acima só será possível devido ao suporte fornecido pelo sistema paramétrico.

Como se vê, muito trabalho ainda precisa ser feito e a cada dia surgem novas técnicas que poderão ser utilizadas. Para isso é necessário ter uma base que esperamos ter ajudado a construir com este trabalho.

APÊNDICE - GERAÇÃO DE MALHAS

Neste apêndice será descrito o algoritmo de geração de malhas do AutoPAC [Souza, 1996].

No projeto de dispositivos eletromagnéticos é fundamental a criação de uma malha, construída usando o Método de Elementos Finitos, para que possam ser calculados os valores de grandezas de interesse em vários pontos importantes da geometria. O algoritmo de geração de malhas é dividido em duas partes: identificação das faces da geometria e a geração da malha propriamente dita [George, 1991].

A identificação das faces se baseia no rastreamento de todos os elementos da geometria e na identificação daqueles que compõem seus contornos e, dentre estes, identificar os que compõem uma face [George, 1991]. Feito isso, o método do avanço de front é aplicado sobre as faces previamente identificadas para gerar a malha de elementos finitos.

A.1 - Identificação das faces

Antes de identificar as faces da geometria, é necessário identificar todos os nós que a compõem.

Um nó é definido como o ponto final de dois ou mais elementos gráficos [George, 1991]. Portanto, um círculo deve ser decomposto em dois arcos para satisfazer esta definição.

Os nós são organizados em uma estrutura, denominada estrutura de nós (Fig. A-1), que contém uma lista de todos os elementos que se iniciam ou terminam em um nó. No campo PontoNo desta estrutura é armazenado o valor das coordenadas (x,y) de um dos Nós da estrutura. No campo ListaEntidades são armazenados os nomes de todas as entidades das quais PontoNo é um dos extremos. Este campo é um ponteiro para a estrutura ListaEntidades. O campo Label é usado para armazenar um label qualquer associado a um dos Nós. Os campos Teste e NóPreso são índices. Por fim, o

campo Próximo é um ponteiro para o próximo elemento da lista encadeada. Quando não existirem mais elementos, o campo Próximo será NULO.

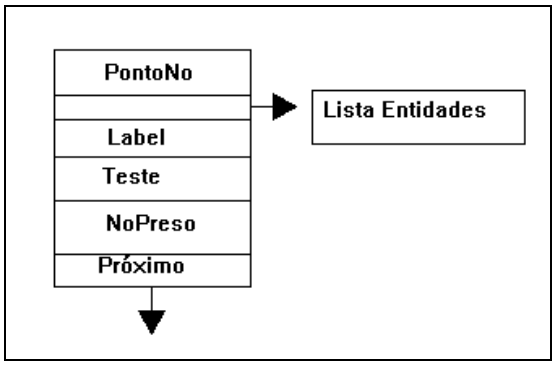

Fig. A-1 A estrutura de nós

A estrutura ListaEntidades (Fig. A-2) contém ponteiros para os dois nós do elemento gráfico referenciado e um ponteiro para o próximo nó. O campo NomeEntidade dessa estrutura armazena o nome de uma das entidades da lista. Através deste nome, é possível obter todas informações geométricas relativas a esta entidade. Como cada entidade tem um ponto inicial e um ponto final distintos, seu nome é armazenado em duas listas de Nós. Desse modo, o campo outroNo é um ponteiro para a região de memória onde se encontra o outro extremo da entidade e, da mesma maneira, o ponteiro outraEntidade é um ponteiro para a região de memória onde está localizada esta mesma entidade no outro nó. Os campos teste[2], ehExterna e ehExternaFalsa são índices. O campo Próximo é um ponteiro para o próximo elemento da lista. Da mesma maneira que na estrutura de nós, caso não haja mais elementos, próximo será NULO.

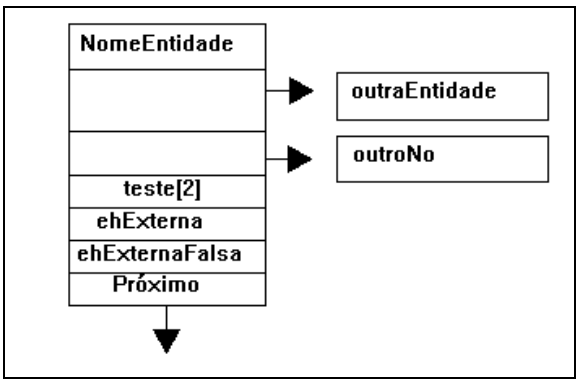

Fig. A-2 A estrutura ListaEntidades

Estas duas estruturas juntas possibilitam a determinação de todos os contornos existentes na geometria.

Para a determinação das faces, foi montada outra estrutura de dados denominada Estrutura de Áreas Fechadas, cujo diagrama está mostrado na Fig. A-3. O campo *contornoExterno* é um ponteiro para uma estrutura do tipo Lista (Fig. A-5). Este campo armazena os nomes de todas as entidades pertencentes ao contorno externo a uma face específica. O campo contornosInternos é um ponteiro para as listas de contorno internos à uma face específica da geometria. Os campos Label da Face e Label Cexterno armazenam labels específicos à face e ao contorno externo em questão.

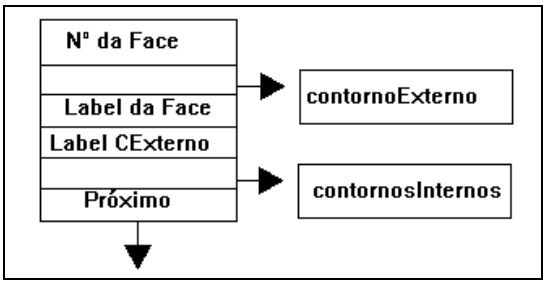

Fig. A-3 A estrutura de ÁreasFechadas

A estrutura de contornoInterno é mostrada na Fig. A-4. Ela é uma composição de listas. O campo *N° cont. Interno* armazena um índice específico para este contorno interno a uma das faces da geometria. O campo *Lista* é um ponteiro para a lista de entidades deste contorno. O campo *Label*

Cinterno armazena um label que o usuário queira dar a este contorno. Por fim o campo *Próximo* aponta para o próximo contorno externo a esta face.

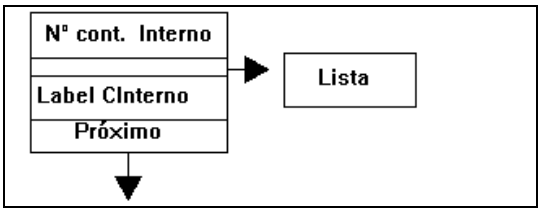

Fig. A-4 A lista de contorno interna

Na estrutura Lista (Fig. A-5), o campo NomeEntidade armazena o nome de uma das entidades do contorno. O campo Label armazena um label específico para esta entidade e o campo Próximo é um ponteiro para o próximo elemento da lista.

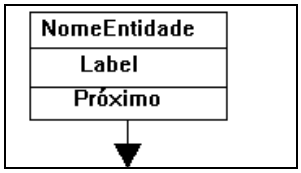

Fig. A-5 A estrutura Lista

A.2 - O algoritmo de geração de malhas

Para a geração de malhas, foi escolhido o Método de Avanço de Front devido à sua fácil adaptação a geometrias arbitrárias. O método pode ser descrito da seguinte maneira [George, 1991]:

1. Partindo de uma face identificada, é construído um contorno único que será discretizado em N segmentos. Estes segmentos originarão o front inicial. O processo de discretização será descrito mais adiante.

   Se houverem contornos internos à face, o front inicial corresponderá ao contorno externo discretizado. Em caso contrário, o contorno externo deverá ser conectado aos contornos internos através de segmentos virtuais. Estes segmentos devem ser os menores possíveis (Fig. A-6).

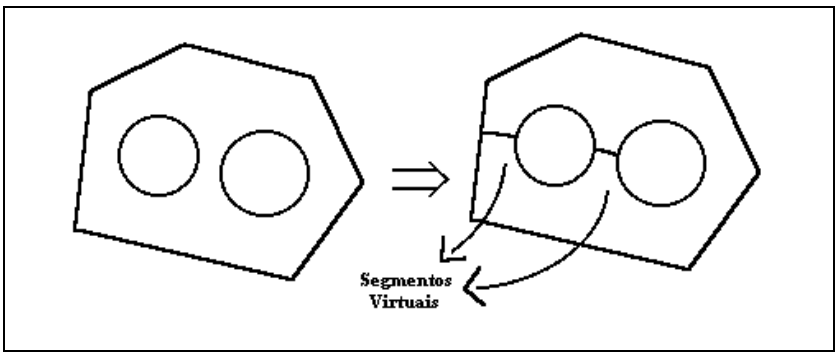

Fig. A-6 Segmentos Virtuais

2.Os segmentos criados no front inicial são analisados e usados na geração de um elemento triangular. O segmento usado para gerar um triângulo é removido do front e novos são inseridos.

3.Este processo interativo é repetido atualizando o front até ele ficar vazio.

O processo global é mostrado na Fig. A-7.

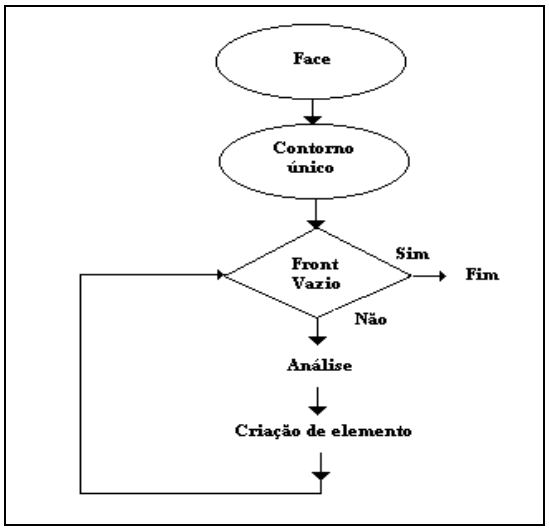

Fig. A-7 Diagrama do Método de Avanço de Front

A.3 - Processo de discretização

Para criar o front inicial, é necessário gerar um número finito de segmentos sobre a geometria. Por isso, as seguintes convenções foram adotadas de forma que o

comprimento dos segmentos e a malha obtida sejam otimizados [Souza, 1996].

1. Cada elemento será dividido em relação a um segmento base.
2. O comprimento do segmento base será o menor entre 1/8 do comprimento médio de todos os elementos presentes no desenho ou o menor comprimento dentre todos os elementos.
3. O comprimento dos segmentos próximos a um nó é associado ao peso deste nó. Uma unidade de peso significa que o comprimento do segmento próximo desse nó será igual ao do segmento base. O dobro da unidade de peso significa que o comprimento do segmento será igual a duas vezes o comprimento do segmento base.

O próximo passo é dividir o segmento do seu ponto inicial até o final de seu comprimento (Fig. A-8).

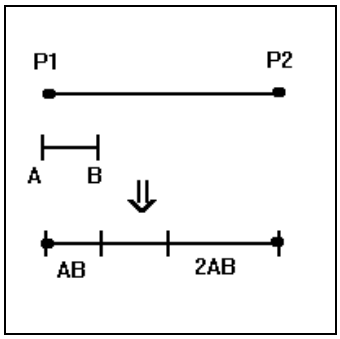

Fig. A-8 - Divisão dos elementos

Para fazer isso, a transição de um comprimento ao outro é considerada suave [George, 1991] (Fig. A-9).

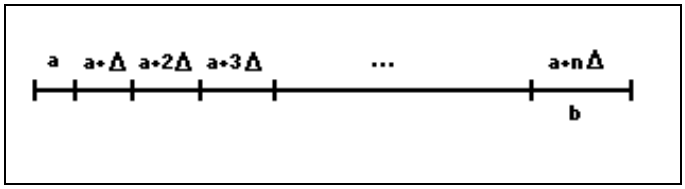

Fig. A-9 Transição de segmentos

O número de divisões (Ndiv) será o número inteiro de segmentos cujo comprimento é a média aritmética entre "a" e

"b" que é possível de se colocar no comprimento do elemento.

$$Ndiv = trunc\left(\frac{C}{\left(\frac{a+b}{2}\right)}\right) \qquad [A.1]$$

onde C é o comprimento do elemento.

Pela Fig. A-9:

$$Ndiv = n + 1 \quad \Rightarrow \quad n = Ndiv - 1 \qquad [A.2]$$

$$a + n.\Delta = b \Rightarrow \Delta = \left(\frac{b-a}{Ndiv-1}\right) \qquad [A.3]$$

O resultado exato de Ndiv foi truncado. Isso gerou um erro que será distribuído a todos os Ndiv segmentos gerados. O erro é calculado pela expressão:

$$erro = \left(\frac{C - \left(a.Ndiv + \frac{Ndiv.(Ndiv-1).\Delta}{2}\right)}{Ndiv}\right) \qquad [A.4]$$

Todos os elementos serão discretizados desta maneira [George, 1991].

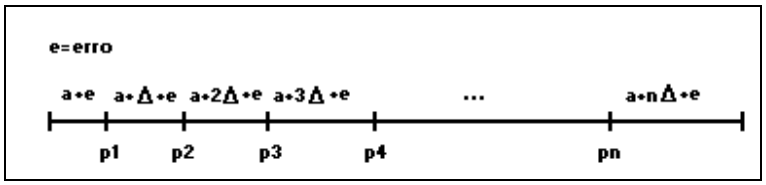

Fig. A-10 Processo de discretização final

A.4 - Análise do Front: geração de elemento

Seja α o ângulo entre dois segmentos consecutivos do Front. Este ângulo deve ser analisado para se gerar um elemento triangular. Três situações são identificadas [George, 1991].

1. $\alpha < \pi/2$

Neste caso (Fig. A-11), o triângulo $S_2S_3S_4$ é criado. Os segmentos S_2S_3 e S_3S_4 são removidos do Front e segmento S_2S_4 é inserido.

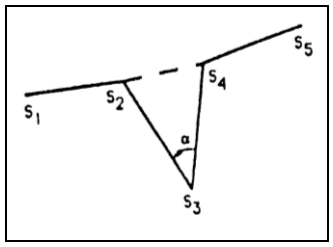

Fig. A-11 Primeira condição

2. $\pi/2 \leq \alpha \leq 2\pi/3$

Esta situação é mostrada na Fig. A-12.

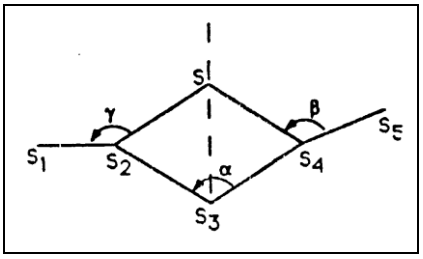

Fig. A-12 Segunda condição

Neste caso, os triângulos S_3SS_2 e S_4SS_3 são criados. O ponto S é gerado sobre a linha que bissecciona o ângulo α e sua localização é avaliada pela expressão.

$$d_{SS_3} = \frac{1}{6}\left(2d_{S_2S_3} + 2d_{S3S4} + d_{S1S_2} + d_{S4S_5}\right)$$

[A.5]

Os segmentos S_2S_3 e S_3S_4 são removidos do front e os segmentos S_2S e SS_4 são inseridos.

3. $\alpha > 2\pi/3$

Neste caso (Fig. A-13), um triângulo equilátero é criado a partir do elemento mais curto.

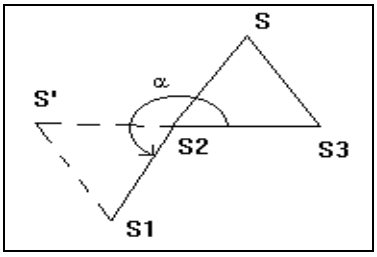

Fig. A-13 Terceira Condição

O segmento S_2S_3 é removido do front e os segmentos S_2S e SS_3 são inseridos.

REFERÊNCIAS BIBLIOGRÁFICAS

[Coad, 1992] Coad, Peter & Yourdon, Edward, *Análise Baseada em Objetos* - 2ª Edição, Editora Campus, 1992.

[Foley, 1990] J.D. Foley, A. van Dam, S.K. Feiner, J.F.Hughes, *Computer Graphics, Principles and Practice*, Addison Wesley Publishing Company, Reading, Massachusetts, 1990.

[George, 1991] P.L. George, *Automatic Mesh Generation*, John Wiley & Sons, New York, 1991.

[Infolytica, 1998]

[Lowther, 1996] David A. Lowther, "Computational Paradigms for the Design and Analysis of Eletromagnetic Devices", 1996.

[Mesquita, 1997] R.C. Mesquita, R.P. Souza, T. Pinheiro, A.L.C.C. Magalhães "An Object-Oriented Platform for Teaching Finite Element Pre-Processor Programming and Design Techniques", 1997.

[Montenegro, 1994] F. Montenegro e R. Pacheco, *Orientação a Objetos em C++,* Editora Moderna, 1994.

[Rocha,] L.F.N. 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*, UFMG, 1995.

[Rogers, 1996] J. Rogers "An Object-Oriented Expression Evaluator", *C/C++ Users Journal*, April, 1996.

[Shah, 1995] J.J. Shah and M. Mantyla, *Parametric and Feature-Based CAD/CAM*, John Wiley & Sons, New York, 1995.

[Schildt, 1991] Herbert Schildt, *C Completo e Total*, Makron Books, São Paulo, 1991.

[Souza, 1996] R.P. Souza & R.C. Mesquita, "An Advancing Front Mesh Generator Built Inside AutoCAD", 1996.

[Souza, 1997] R.P. Souza, *Manual de desenvolvimento do AutoPAC*, UFMG, 1997.

[de Vasconcelos, 1994] J.A. Vasconcelos *Optimisation de Forme des Structures Électromagnétiqués*, L'ecole Centrale de Lyon, 1994.

[Vector Fields, 1998]

[Ziviani, 1993] N. Ziviani, *Projetos de Algoritmos*, Segunda Edição, Editora Pioneira, São Paulo, 1993.