Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Sistemas e Controle de Energia
Laboratório de Sistemas Modulares Robóticos

Otimização de Controladores Nebulosos de Takagi–Sugeno Utilizando Algoritmos Genéticos

Márcio André Teixeira de Sousa

Dissertação submetida à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, como parte dos requisitos exigidos para obtenção do título de Mestre em Engenharia Elétrica

Banca Examinadora:
Prof. Dr. Marconi Kolm Madrid (Orientador) - DSCE/FEEC/UNICAMP
Prof. Dr. Fernando José Von Zuben - DCA/FEEC/UNICAMP
Prof. Dr. Katsuhito Takita - DEC/FT/Universidade do Amazonas
Prof. Dr. João Maurício Rosário - FEM/UNICAMP

Campinas, 21 de agosto de 2000.

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

So85o

Sousa, Márcio André Teixeira de
Otimização de controladores nebulosos de Takagi-Sugeno utilizando algoritmos genéticos / Márcio André Teixeira de Sousa.--Campinas, SP: [s.n.], 2000.

Orientador: Marconi Kolm Madrid
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Sistemas difusos. 2. Algoritmos difusos. 3. Algoritmos genéticos. 4. Controle em tempo real. 5. Pêndulo. I. Madrid, Marconi Kolm. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Aos meus pais João e Francisca,
à minha irmã Michelle
e à minha sobrinha Priscila.

Resumo

Esta tese, propõe uma técnica que emprega algoritmos genéticos e teoria de conjuntos nebulosos integrados, visando o desenvolvimento automático de controladores de alta performance para servomecanismos tipo elo-acionado, ou módulo de junta robótica.

Nesta abordagem, a teoria de conjuntos nebuloso é utilizada no desenvolvimento de controladores não lineares com estrutura flexível e grande quantidade de graus de liberdade. Devido às características apresentadas, estes controladores possuem potencial para resolver uma enorme variedade de problemas, inclusive problemas nos quais os métodos convencionais não são aplicáveis.

Os algoritmos genéticos são métodos de busca inspirados no processo evolutivo natural que apresentam-se como uma alternativa eficiente para o ajuste automático de controladores não lineares. O algoritmo genético proposto neste trabalho é utilizado para o ajuste paramétrico de controladores nebulosos e controladores clássicos tipo PID.

Os resultados experimentais mostraram que tal técnica é muito eficiente para o controle de juntas robóticas e para uma infinidade de outros sistemas de engenharia que possuam dinâmica semelhante, podendo-se assegurar sua aplicação prática com êxito, conseguindo-se uma excelente relação de custo/benefício.

Abstract

This thesis proposes a technique that uses genetic algorithms and fuzzy sets theory in a integrated way, seeking the automatic development of high performance controller for servomechanisms like driven-links, or robotic joints modules. The fuzzy sets theory is used for developing nonlinear controllers with flexible structure and great amount of degrees of freedom. Due to the presented characteristics, these controllers possess potential to solve an enormous variety of problems, including problems which the conventional methods are not suitable.

Genetic algorithms are search methods inspired by natural evolutionary process that come as an efficient alternative for automatic tuning of nonlinear controllers. The genetic algorithm proposed here is used for the parametric adjustment of fuzzy controllers and classic proportional+integral+derivative controllers. The experimental results showed that such technique is very efficient for the control of robotic joints and for an infinity of other engineering systems that possess similar dynamics. It can be assured its practical application with success and an excellent cost/benefit relation.

Agradecimentos

Agradeço a todas as pessoas que de alguma forma contribuíram para a realização deste trabalho. Em especial, agradeço:

- Ao professor Madrid pela excelente orientação, amizade e companheirismo nesse período de mestrado, sempre estando disponível para esclarecer dúvidas, dar conselhos e ajudar no que fosse preciso.
- Aos meus orientadores do período de graduação, Prof. Takita e Profa. Antonieta, que me deram a oportunidade de fazer parte da equipe do LAI-UFPA, sem dúvida um dos melhores laboratórios na área de automação que já houve neste país. Tenho certeza que os ensinamentos e experiência que adquiri neste período foram fundamentais para minha formação profissional.
- Ao Prof. Von Zuben por ter me introduzido ao mundo fantástico da inteligência artificial e pelas profícuas discussões sobre algoritmos genéticos.
- Ao amigo Dionne que teve papel fundamental no meu ingresso à UNICAMP e que também me acolheu em sua casa durante minha fase inicial em Campinas.
- Aos amigos quase irmãos Vitor e André por todos os momentos ebrifestivos de mpb, violão, reggae, forró, poesia e divagações.
- A Bonani pela amizade e excelente trabalho técnico na construção do sistema mecânico utilizado nesta tese.
- Ao amigo Franciraldo pelas proveitosas discussões nas áreas de controle, cinema, filosofia, Shakespere, mpb e sistemas de Takagi-Sugeno.
- A Reinaldo pelas preciosas dicas sobre LaTeX.
- Aos amigos de republica Daniela, Verena, Rafael e César; pela amizade e paciência durante esses dois anos que moramos juntos.
- A todos amigos da comunidade recifense, em particular: Márcia, Daniel, Joselan, Divanilson, Gustavo e Fabrício.
- Ao Convênio Capes pelo apoio financeiro.

Conteúdo

Lista de Figuras

Capítulo 1

Motivação, Objetivos e Organização

1.1 Introdução

Vários teoremas denominados de teoremas da impossibilidade, em muitos campos científicos, provam que sempre existirão fatos que não podem ser provados como verdadeiros nem como falsos via qualquer abordagem matemática. Neste sentido, sem alguma suposição estrutural para resolver um problema de otimização, nenhum outro algoritmo pode ter melhor performance média do que um que use o método de busca cega (Macready & Wolpert 1997). Embora para muitos destes tais teoremas da impossibilidade as respectivas provas sejam longas, difíceis, e freqüentemente não intuitivas, pode-se ter facilmente o sentimento do que eles sustentam considerando o problema/provérbio de que "algo seja ainda mais difícil do que encontrar uma agulha no palheiro". Claramente para a solução de um problema desta natureza, nenhum algoritmo terá melhor chance de encontrar o ótimo do que um procedimento de busca cega (Ho 1999).

Quando o problema em questão envolve incertezas e tem-se muito pouco conhecimento à priori dos objetos relacionados com a sua formulação, então tais buscas por soluções certamente poderão ser muito custosas e até mesmo inviáveis do ponto de vista prático, devido ao número exorbitante de passos de busca necessários. Para tais casos pode-se adotar algum tipo de suavização/relaxação da otimalidade pretendida, que conduza a soluções quasi-ótimas que sejam satisfatórias dentro do contexto. Este tipo de abordagem atualmente vem sendo muito empregada em sistemas que resolvem problemas com o uso de técnicas computacionais de inteligência artificial, porque leva a um decréscimo muito acentuado do esforço computacional exigido em virtude da eficácia do processo de busca

associado, e pode viabilizar a aplicação em sistemas reais (Abramovitch & Bushnell 1999) (Ying 2000). Os algoritmos genéticos, a lógica nebulosa, as redes neurais e os métodos de aprendizagem computacional são exemplos de abordagens que quando aplicadas com tais considerações podem produzir excelentes resultados, principalmente no desenvolvimento de controladores para sistemas de engenharia (Sousa & Madrid 2000)(Delgado *et al.* 2000)(Li & Shieh 2000).

Esta tese, propõe uma técnica que emprega algoritmos genéticos e teoria de conjuntos nebulosos integrados, visando o desenvolvimento automático de controladores de alta performance para servomecanismos tipo elo-acionado, ou módulo de junta robótica.

Nesta abordagem, a teoria de conjuntos nebuloso é utilizada no desenvolvimento de controladores não lineares com estrutura flexível e grande quantidade de graus de liberdade. Devido às características apresentadas, estes controladores possuem potencial para resolver uma enorme variedade de problemas, inclusive problemas nos quais os métodos convencionais não são eficientes ou aplicáveis. O preço pago por tal potencial está na complexidade envolvida no ajuste dos parâmetros destes controladores, e na busca por métodos eficientes para a obtenção das regras lingüísticas adequadas e suficientes, tanto do ponto de vista qualitativo quanto quantitativo, para cada caso estudado (Nobre 1997).

Vários outros tipos de controladores não lineares conhecidos e com características similares poderiam ser utilizados neste trabalho, entretanto, os controladores nebulosos possuem o atrativo ou vantagem de serem interpretáveis, caso atendam a restrições de consistência e completitude e de, em alguns casos, poderem englobar em suas estruturas o conhecimento lingüístico extraído de especialistas humanos.

Os métodos analíticos de projeto de controladores não-lineares geralmente são muito limitados e específicos, principalmente por exigirem um bom conhecimento da dinâmica e dos parâmetros físicos do sistema a ser controlado. Portanto, é praticamente inviável produzir ferramentas computacionais baseadas em métodos analíticos de projeto para a geração automática de controladores não-lineares.

Quando planeja-se controlar um sistema via uso de controladores nebulosos, por simplicidade, pode-se partir de uma formulação estrutural básica, que não necessariamente já tenha o desempenho pretendido, sendo que este desempenho pode ser melhorado se a referida estrutura for numericamente ajustada com o auxílio de processos evolutivos baseados em algoritmos genéticos.

Os algoritmos genéticos são métodos de busca inspirados no processo evolutivo

natural que apresentam-se como uma alternativa eficiente para o ajuste automático de controladores não lineares em geral. O algoritmo genético proposto neste trabalho é utilizado para o ajuste paramétrico de controladores nebulosos e controladores clássicos tipo PID (proporcional-integral-derivativo).

Tal abordagem mostrou-se muito eficiente podendo-se assegurar sua aplicação com êxito e com excelente relação de custo/benefício para o controle de juntas robóticas, e para uma infinidade de outros sistemas de engenharia que possuem dinâmica semelhante.

1.2 Controle Nebuloso

O Controle Nebuloso, introduzido em 1974 (Mamdani 1974) como uma tecnologia emergente focalizada em aplicações industriais, adicionou uma dimensão promissora ao domínio da engenharia de controle convencional. Quando não estão disponíveis modelos matemáticos precisos de sistemas físicos complexos, especialmente se a descrição do sistema requer a utilização de termos vagos, comuns e de natural emprego pelo homem, a metodologia de controle nebuloso apresenta vantagens evidentes sobre outras abordagens tradicionais.

Os algoritmos e métodos de controle nebuloso, incluindo *software* e *hardware* disponíveis atualmente no mercado, podem ser classificados como métodos inteligentes, uma vez que os mesmos podem incorporar algum tipo de conhecimento de especialistas humanos em seus componentes (conjuntos nebulosos, base de regras, etc), além de definirem as ações de controle empregando lógica nebulosa .

A utilização de conhecimento humano nas considerações de projeto não é apenas vantajosa mas muita vezes necessária. Na verdade, o conhecimento humano é incorporado até mesmo no projeto de controladores clássicos, pois o tipo de controlador, a estrutura de controle utilizada e alguns parâmetros de projeto dependem da decisão e escolha do projetista (Franklin *et al.* 1994). O controle nebuloso tende a ser uma metodologia alternativa, ao invés de substitutiva às técnicas de controle convencional ou de controle inteligente.

Comparada às abordagens tradicionais, o controle nebuloso utiliza mais informações do domínio de especialistas humanos e se atém menos às informações sobre o modelo matemático dos respectivos sistemas físicos.

Os controladores nebulosos podem ser divididos em duas classes principais: con-

troladores nebulosos de *Mamdani* (Lee 1990*a*)(Lee 1990*b*)(Driankov *et al.* 1996*a*) e controladores nebulosos de *Takagi-Sugeno* (Takagi & Sugeno 1985)(Ying 1998*b*)(Ying 1998*a*). Os controladores de *Mandani* utilizam conjuntos nebulosos como conseqüentes das regras, enquanto que os controladores de *Takagi-Sugeno* empregam funções (usualmente lineares) das variáveis de entrada como conseqüentes das regras. Os demais componentes de ambos os sistemas nebulosos são equivalentes.

Em geral os controladores de *Mamdani* são projetados a partir de uma base de regras obtida de um especialista e as funções de pertinência são ajustadas experimentalmente. Os controladores de *Takagi-Sugeno* usualmente possuem mais parâmetros ajustáveis do que os controladores de *Mamdani*. Consequentemente, o ajuste manual destes parâmetros pode ser ineficiente e até mesmo impossível se o número de parâmetros for muito grande. Além disso, a falta de intuitividade na interpretação das regras de *Takagi-Sugeno* pode dificultar ainda mais tal ajuste.

O modelo de sistema nebuloso escolhido para o desenvolvimento dos controladores propostos nesta tese é o modelo de *Takagi-Sugeno*. As motivações listadas a seguir comprovam a adequação do modelo adotado para os propósitos almejados neste trabalho.

- O modelo de *Takagi-Sugeno* permite a incorporação de conhecimento de especialista humano, como definição de regras via conseqüentes baseados nos controladores clássicos tipo *PID*, gerando assim uma estrutura não linear com características de controladores *PID* não lineares. Os controladores clássicos *PID* são empregados com sucesso numa infinidade de problemas de controle, consequentemente, versões não-lineares destes controladores podem superar o desempenho das versões clássicas (lineares).

- As regras de *Takagi-Sugeno* funcionam como múltiplos controladores lineares locais. Pode-se afirmar, então, que cada regra (ou bloco de regras) possui um controlador linear local associado que contribui para a resposta do controlador nebuloso. Cada controlador linear local é acionado quando o estado das variáveis de entrada pertence ao seu domínio.

- Do ponto de vista de otimização de sistemas nebulosos, o sistema de *Takagi-Sugeno* apresenta a vantagem de utilizar apenas o processo de otimização paramétrica, ou seja, ajuste das funções de pertinência e dos parâmetros das funções dos conseqüentes das regras. Enquanto isso, os controladores nebulosos de *Mamdani* necessitam de dois processos: otimização paramétrica (ajuste das funções de pertinência) e otimização da base de regras lingüísticas.

1.3 Algoritmos Genéticos

O princípio da evolução é o conceito primordial da biologia que relaciona todos os organismos existentes numa cadeia histórica de eventos. Cada criatura da cadeia é produto de uma série de "acidentes" sucedidos de forma não-determinística, sendo que a pressão seletiva do ambiente se encarrega de escolher "cuidadosamente" os indivíduos cujos genes irão perdurar, dando uma direção para a evolução ao longo das gerações. Após várias gerações, a variabilidade criada a partir de eventos aleatórios e a seleção natural modelam o comportamento e a estrutura dos indivíduos de forma a se adaptarem às condições do ambiente. Esta adaptação pode ser fantástica e extremamente interessante, como mostra a figura 1.1, uma clara indicação que a evolução é criativa. Apesar da evolução aparentemente não possuir nenhum propósito intrínseco, sendo apenas mero efeito de leis da natureza atuando nas populações e espécies, ela é capaz de gerar soluções engenhosas para os problemas de sobrevivência, as quais são únicas para cada circunstância (Fogel 2000).

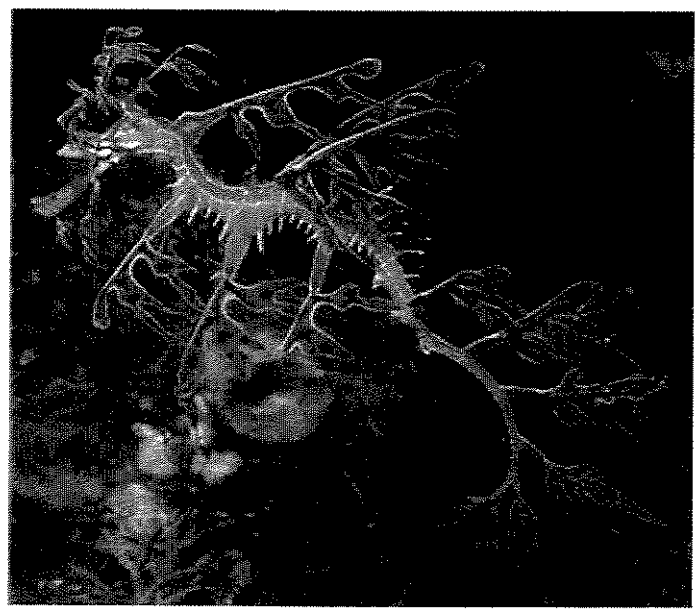


Figura 1.1: Dragão marinho - A evolução moldou o corpo do dragão marinho de forma a torná-lo parecido com a vegetação natural de seu *habitat*, uma espécie de camuflagem que o protege dos predadores.

Historicamente, já os hebreus, os gregos, e várias escolas medievais, propuseram teorias com intenção de explicar fenômenos genéticos. *Gregor Mendel*, um monge austríaco, considerado o "pai" da genética, desenvolveu enormemente esta área de pesquisa formulan-

do uma série de princípios fundamentais da hereditariedade, baseando-se em informações obtidas experimentalmente com organismos vivos. Acredita-se que seu trabalho científico, gerador dos fundamentos da genética, não tenha tido o devido reconhecimento por ter sido publicado em meios de pouca divulgação na mídia da época, 1865. Suas contribuições principais foram o princípio da segregação e da distribuição independente, bem como as definições de dominância e recessividade genética, que podem ser de enorme valia para o entendimento do que ocorre durante os processos evolutivos, e empregáveis em formulações algorítmicas para computação. Nesta mesma época, *Charles Darwin* formulou suas teorias sobre a evolução das espécies, e seu primo *Francis Galton*, cerca de três décadas após, concentrando-se em gêmeos, fez uma extensa série de estudos sobre a influência da hereditariedade nas características humanas, sem aparentar ciência das descobertas de *Mendel*. A genética, como conhecida na atualidade, é inegavelmente resultado das pesquisas científicas realizadas neste século, sendo que o termo *"gene"*, usado como referência à unidade básica da hereditariedade, foi empregado cientificamente pela primeira vez em 1909 (Jorde *et al.* 1995).

Baseados no conhecimento de que a evolução natural é um processo lento, os mais céticos poderiam afirmar que a evolução artificial dificilmente pode ser inteligente. Porém, enquanto alguns exemplos de evolução natural que produziram seres considerados inteligentes (seres humanos) são extremamente lentos, outros, ainda que não considerados inteligentes, são extremamente velozes (bactérias, virus, etc.).

A evolução pode ser vista resumidamente como um processo iterativo de dois passos, consistindo de reprodução com variação aleatória, seguida de seleção "natural". O processo natural avalia os indivíduos através de certos critérios, normalmente vinculados ao nível de adaptação ao meio, que ditam se haverá sobrevivência individual e/ou manutenção da espécie em questão. Aqueles indivíduos/espécies que possuem boas qualidades relacionadas ao meio a que pertencem, têm grandes chances de continuarem existindo como tal, ou passarem suas características adiante através de reprodução.

Os algoritmos de otimização baseados em processos evolutivos são chamados de Algoritmos Evolutivos, tendo os "Algoritmos Genéticos"(*AG*) (Holland 1975) como uma de suas principais instâncias. Nos algoritmos genéticos, as soluções dos problemas são codificadas em cadeias de dados denominadas cromossomos, normalmente formadas por números reais ou binários. Cada indivíduo da população do algoritmo genético possui um cromossomo e um valor de adaptação (*fitness*). O algoritmo genético é iniciado a partir de uma população de indivíduos cujos cromossomos são gerados aleatoriamente ou a partir de informações a priori sobre as soluções do problema abordado. Os indivíduos da população

inicial geram descendentes através do processo de reprodução, que envolve operadores de recombinação e mutação. Os indivíduos descendentes são avaliados através de uma função objetivo que mede seus graus de adaptação. A seleção consiste num processo de escolha realizado com base na medida de adaptação de cada indivíduo. Em média, os indivíduos mais adaptados tendem a passar para a próxima geração e os indivíduos menos adaptados tendem a ser eliminados.

O algoritmo genético proposto neste trabalho para o ajuste paramétrico dos controladores nebulosos, utiliza codificação em ponto flutuante e operadores de reprodução especiais. Os parâmetros do controlador nebuloso selecionados para ajuste são codificados numa cadeia cromossômica. Logo, cada indivíduo tratado no algoritmo possui um cromossomo e representa um controlador nebuloso.

O grau de adaptação de cada indivíduo é avaliado com base na performance do controlador nebuloso associado, o qual é testado através da execução de uma tarefa de controle padrão que depende das especificações exigidas para o controlador. No cálculo da função objetivo podem ser utilizados parâmetros de medida de desempenho como sobresinal, erro de regime, tempo de estabilização, integral do erro, etc.

1.4 Algoritmos Genéticos e Controle Nebuloso

A área de pesquisa de controladores nebulosos para o controle de sistemas não lineares teve um progresso bastante significativo nos últimos anos, sendo que tem-se feito muito progresso a partir das extensões permitidas pelo uso da computação evolutiva. Mas apesar desses avanços, a criação e a aplicação prática destes tipos de controladores ainda está bastante defasada da grande parte do conhecimento teórico já produzido para emprego no controle de sistemas reais tanto de grande, média como pequena complexidades, sendo que na atualidade há uma grande corrida tecnológica neste sentido, principalmente de parte da comunidade científica e dos setores industriais que empregam alta tecnologia no desenvolvimento de novos equipamentos e produtos.

Na atualidade, o projeto prático destes tipos de controladores têm intrinsecamente incluídas características restritivas como grande complexidade computacional, operações com quantidades observáveis e não observáveis, e implementabilidade em tempo real, que tendem a reduzir a velocidade do avanço tecnológico pretendido.

Os primeiros trabalhos empregando algoritmos genéticos para o ajuste/geração

de controladores nebulosos foram desenvolvidos por Linkens & Nyongesa (1995*a*). Na primeira parte do trabalho (Linkens & Nyongesa 1995*a*), é proposto um algoritmo genético para o ajuste dos fatores de escala e funções de pertinência do sistema nebuloso, mantendo uma base de regras fixa. Na segunda parte (Linkens & Nyongesa 1995*b*), os autores empregam um sistema classificador para implementação de controladores nebulosos adaptativos. Em ambos os casos, os resultados apresentados limitam-se a simulações, sem tratar da viabilidade de sua implementação prática.

Em 1996, Li & Yi (1996) desenvolveram um método de seleção de regras nebulosas utilizando algoritmos genéticos. A abordagem proposta é aplicada no projeto de controladores nebulosos do tipo múltiplas entradas e múltiplas saídas. Os resultados apresentados envolvem o controle de um pêndulo invertido duplo, sendo um dos primeiros trabalhos com resultados práticos.

Outros trabalhos de simulação na área de algoritmos genéticos aplicados a controle nebuloso podem ser encontrados em (Qi & Chin 1997, Gurocak 1999, Herrera *et al.* 1998, Sousa & Madrid 1999).

1.5 Organização da Tese

O **Capítulo 2** apresenta uma introdução sobre os controladores nebulosos, tratando resumidamente sua formulação matemática. São apresentados os dois principais tipos de controladores nebulosos e também a versão nebulosa do controlador clássico tipo *PID*. O capítulo é finalizado com uma breve argumentação sobre a estabilidade de controladores nebuloso.

O **Capítulo 3** inicia-se com a apresentação do algoritmo genético clássico. Em seguida, é realizada uma explicação detalhada sobre o algoritmo genético, com codificação em ponto flutuante, proposto para o problema de otimização de controladores nebulosos. Finalmente, é apresentada a função objetivo utilizada par . avaliação dos controladores nebulosos.

O **Capítulo 4** apresenta os principais resultados práticos obtidos na otimização de controladores nebulosos aplicados no controle de um pêndulo acionado. No total, são mostrados 4 experimentos que envolvem controladores nebulosos tipo *PD* e *PID*, e um controlador nebuloso tipo *PD* com compensação de gravidade.

O **Capítulo 5** argumenta as contribuições do trabalho e apresenta uma lista de sugestões e perspectivas para trabalhos futuros.

O **Anexo A** trata dos detalhes técnicos de implementação do hardware e software de controle em tempo real, apresentando também o sistema prático desenvolvido para os experimentos. O **Anexo B** descreve as bibliotecas desenvolvidas em C^{++} para implementação dos algoritmos de controle nebuloso e algoritmos genéticos. O **Anexo C** contém uma listagem completa do código fonte das bibliotecas e funções desenvolvidas.

Capítulo 2

Controladores Nebulosos

2.1 Introdução

A teoria de conjuntos nebulosos, introduzida por *Zadeh* em 1965 (Zadeh 1965), apresenta-se como uma extensão da teoria de conjuntos ordinários, na qual um conjunto pode ter um significado lingüístico, tornando possível definir matematicamente alguns conceitos comuns ao ser humano, como: alto, baixo, quente, frio, etc.

Como a maior parte do conhecimento humano é representada em termos lingüísticos, é importante que exista uma maneira sistemática e eficiente de incorporar este conhecimento em sistemas de engenharia. Os sistemas baseados em regras nebulosas são sistemas que permitem incorporar o conhecimento de especialistas humanos em sua estrutura, além de possibilitar a interpretação do conjunto de regras resultante.

Os primeiros trabalhos utilizando teoria de conjuntos nebulosos para controle de processos foram desenvolvidos por *Mamdani* (Mamdani 1974, Mamdani & Assilian 1975, Mamdani & Baaklini 1974), dando origem aos controladores nebulosos.

Nas últimas duas décadas, a utilização de sistemas neb .losos em equipamentos eletrônicos e sistemas industriais cresceu significativamente. Alguns exemplos são: controle de qualidade de água (Yagshita *et al.* 1985), controle de elevadores (Fujitech 1988), sistemas de operação de trens (Yasunobu & Hasegawa 1985), controle de manipuladores robóticos (Tanscheit & Scharf 1988, Lim & Hiyama 1991, Silva 1995, Benerjee & Woo 1993, Yong *et al.* 1992, Kumbla & Jamshidi 1993, Sousa & Madrid 1999), transmissão de automóveis

(Kasai & Morimoto 1988), máquinas de lavar, condicionadores de ar, câmeras de foco automático, etc.

Inicialmente, os controladores nebulosos foram escolhidos como objeto de estudo desta dissertação por duas razões: os controladores nebulosos são essencialmente não-lineares, conseqüentemente, têm potencial para controlar sistemas mais complexos, superando o desempenho dos controladores convencionais; os controladores nebulosos apresentam a facilidade de incorporação do conhecimento de especialistas através de regras lingüísticas.

Após um estudo sobre as técnicas de projeto de controladores nebulosos, chegou-se à conclusão de que existe uma grande deficiência de metodologias consistentes para o projeto destes controladores. Ainda que a base de regras do controlador nebuloso possa ser obtida facilmente de um especialista, existem vários outros parâmetros de projeto, como os parâmetros das funções de pertinência, que acabam sendo ajustados por métodos de tentativa e erro.

Existem vários trabalhos que utilizam métodos de otimização para fazer o ajuste paramétrico de controladores nebulosos (Herrera *et al.* 1998, Tanscheit & Scharf 1988, Wang 1999). Entretanto, os resultados obtidos nesta área ainda são bastante limitados, pois os métodos existentes geralmente dependem da aplicação e estabelecem restrições aos sistemas nebulosos, como por exemplo, a utilização de um tipo específico de função de pertinência devido a esta possuir propriedades derivativas especiais.

Como tentativa de superar as dificuldades apresentadas, é proposto um algoritmo genético para o ajuste paramétrico de controladores nebulosos.

Neste capítulo, é apresentada uma descrição sucinta sobre os conceitos fundamentais dos controladores nebulosos que serão utilizados neste trabalho. Na seção 2.2, é feita uma descrição sobre a configuração básica de um controlador nebuloso. A seção 2.3 apresenta o modelo de controlador sugerido por *Takagi e Sugeno*. A seção 2.4 aborda a classe de controladores nebulosos tipo *PID* . Finalmente, na seção 2.5, são comentadas algumas questões sobre a estabilidade de controladores nebulosos. Mais detalhes sobre teoria de conjuntos nebulosos e controle nebuloso podem ser encontrados em (Lee 1990*a*, Lee 1990*b*, Driankov *et al.* 1996*a*, Pedrycz & Gomide 1998).

2.2 Estrutura Básica de um Controlador Nebuloso

A estrutura básica de um controlador nebuloso, conforme ilustrada na figura 2.1, consiste dos seguintes componentes: Interface de Fuzzificação [1], Base de Conhecimento, Mecanismo de Inferência e Interface de Defuzzificação [2].

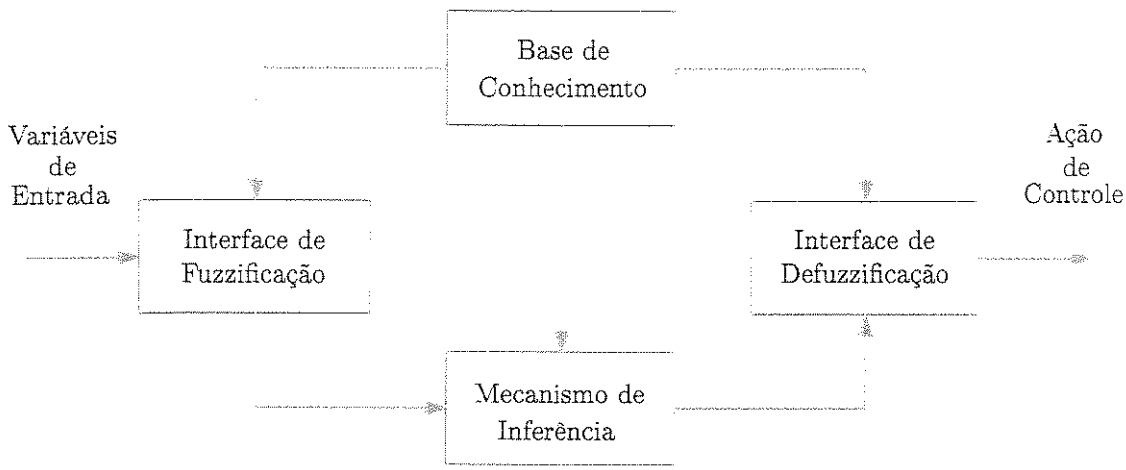


Figura 2.1: Estrutura básica de um controlador nebuloso.

2.2.1 Interface de Fuzzificação

A interface de fuzzificação possui as seguintes funções:

- Executar uma transformação de escala que mapeia os valores físicos das variáveis de entrada do processo em um valor normalizado do universo de discurso de entrada. Esta função só é aplicada quando é utilizado um universo de discurso normalizado.
- Converte os valores numéricos das variáveis de entrada em conjuntos nebulosos, tornando-as compatíveis com a representação existente nas premissas das regras.

O operador de fuzzificação utilizado neste trabalho foi o fuzzificador[3] *singleton*, que transforma o valor numérico x_i^*, de uma variável x_i, em um conjunto nebuloso unitário e normalizado, cuja função de pertinência é dada por:

[1] Tradução adotada para *fuzzification*.
[2] Tradução adotada para *defuzzification*.
[3] Tradução utilizada para *fuzzifier*

$$\mu(x_i) = \begin{cases} 1 & \text{se } x_i = x_i^* \\ 0 & \forall\, x_i \neq x_i^* \end{cases} \tag{2.1}$$

2.2.2 Base de Conhecimento

A base de conhecimento é formada por uma base de dados e uma base de regras.

A função principal da base de dados é fornecer as informações necessárias para o bom funcionamento da interface de fuzzificação, da base de regras e da interface de defuzzificação.

As informações contidas na base de dados são formadas pelas funções de pertinência dos conjuntos nebulosos, que representam os valores lingüísticos das variáveis de entrada e de saída, e pelos fatores de escala que determinam as operações de normalização e desnormalização.

A base de regras tem a função de representar de forma estruturada a política de controle de um experiente operador de processo e/ou de um engenheiro de controle, através de um conjunto de regras do tipo:

$$R^{(l)}\text{: } \underline{\textbf{SE}}\ x_1 \text{ é } A_1^l\ \underline{\textbf{E}}\ x_2 \text{ é } A_2^l\ \underline{\textbf{E}} \ldots\ \underline{\textbf{E}}\ x_n \text{ é } A_n^l\ \underline{\textbf{ENTÃO}}\ y \text{ é } B^l.$$

onde $l = 1, \ldots, \Omega$, A_i^l $(i = 1, \ldots, n)$ e B^l são conjuntos nebulosos, $x = (x_1, \ldots, x_n)^T \in U = U_1 \times \ldots \times U_n$ são as variáveis lingüísticas de entrada, $y \in V$ é a variável lingüística de saída e $R^{(l)}$ denota uma regra.

A parte "**SE** *<condição>*" de cada regra é chamada antecedente da r gra e a parte "**ENTÃO** *<ação>*" é chamada conseqüente da regra.

Cada regra **SE–ENTÃO** define uma relação nebulosa $A_1^l \times A_2^l \times \ldots \times A_n^l \rightarrow B^l$ no espaço $U \times V$. O conectivo **E** é geralmente interpretado por uma conjunção nebulosa, cujos operadores mais comuns são o *mínimo* e o *produto algébrico.*

Exemplo:

$$\begin{aligned} \mu_{A_1^l \times A_2^l \times \ldots \times A_n^l}(x) &= min\{\mu_{A_1^l}(x_1), \mu_{A_2^l}(x_2), \ldots, \mu_{A_n^l}(x_n)\} \quad \text{(mínimo)} \\ \mu_{A_1^l \times A_2^l \times \ldots \times A_n^l}(x) &= \mu_{A_1^l}(x_1) \cdot \mu_{A_2^l}(x_2) \ldots \cdot \mu_{A_n^l}(x_n) \quad \text{(produto algébrico)} \end{aligned} \tag{2.2}$$

Os parâmetros de projeto envolvidos na construção de uma base de regras são:

* Escolha das variáveis de estado de entrada e das variáveis de saída (ação de controle).
* Escolha do conteúdo dos antecedentes e conseqüentes das regras.
* Escolha dos termos lingüísticos para as variáveis de entrada e de saída.
* Derivação do conjunto de regras **SE–ENTÃO**.

2.2.3 Mecanismo de Inferência

Existem duas abordagens utilizadas no projeto do mecanismo de inferência de um controlador nebuloso: (1) inferência baseada na composição das regras e (2) inferência individual de cada regra. Neste trabalho, foi considerado apenas o segundo mecanismo de inferência, devido à sua predominante utilização em aplicações de controle nebuloso.

No mecanismo de inferência, é necessário definir a forma pela qual as regras nebulosas serão interpretadas. Em lógica nebulosa as regras são interpretadas por implicações nebulosas. Entretanto, os controladores nebulosos geralmente empregam conjunções no lugar das implicações, tais como *mínimo* (interpretação de *Mamdani*) e *produto algébrico* (interpretação *Larsen*).

A inferência individual de cada regra tem como resultado um conjunto nebuloso, cuja função de pertinência é dada por:

$$\begin{aligned} \mu_{\overline{B}^l}(u) &= min\{\mu_{A_1^l \times A_2^l \times \ldots \times A_n^l}(x^*), \mu_{B^l}(u)\} \quad \text{(interpretação de } \textit{Mamdani}\text{)} \\ \mu_{\overline{B}^l}(u) &= \mu_{A_1^l \times A_2^l \times \ldots \times A_n^l}(x^*) \cdot \mu_{B^l}(u) \quad \text{(interpretação de } \textit{Larsen}\text{).} \end{aligned} \tag{2.3}$$

O conjunto nebuloso que resulta da combinação de todas as regras é obtido através da união de todos os conjuntos nebulosos resultantes de cada inferência individual. Utilizando o operador *máximo* para realizar a união, obtém-se a seguinte função de pertinência para o conjunto nebuloso de saída:

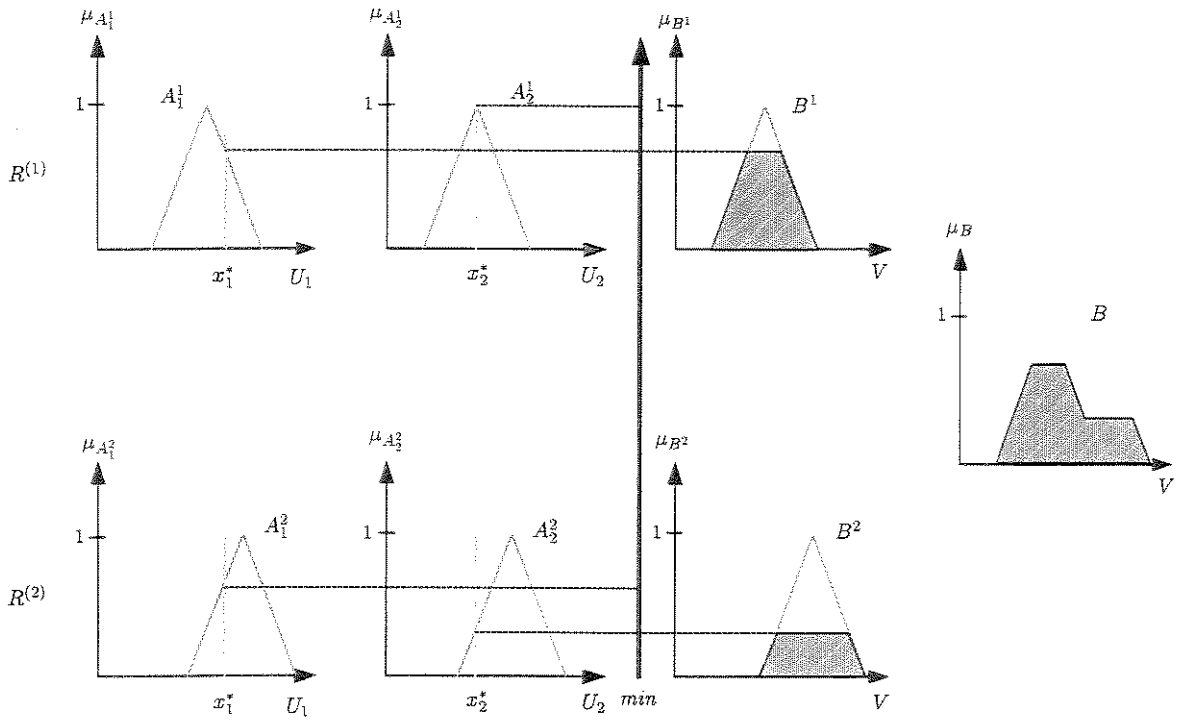


Figura 2.2: Mecanismo de inferência utilizando o operador *mínimo* como conectivo **E**. As regras são interpretadas pela conjunção *mínimo* (*Mamdani*) e a agregação das regras é realizada pelo operador *máximo*.

$$\forall u \, \forall l : \mu_B(u) = \max_l \mu_{\overline{B}^l}(u). \tag{2.4}$$

As figuras 2.2 e 2.3 ilustram o mecanismo de inferência de um controlador nebuloso utilizando duas regras e duas variáveis de entrada.

2.2.4 Interface de Defuzzificação

A interface de defuzzificação tem a função de transformar um conjunto nebuloso B em V para um valor numérico $y \in V$. Em outras palavras, a interface de defuzzificação é responsável pela transformação da ação de controle nebuloso, definida por um conjunto nebuloso, em uma ação de controle numérica. A operação de defuzzificação é utilizada porque a maioria das aplicações de controle exigem uma ação de controle exata, isto é, definida por um número específico em um dado momento.

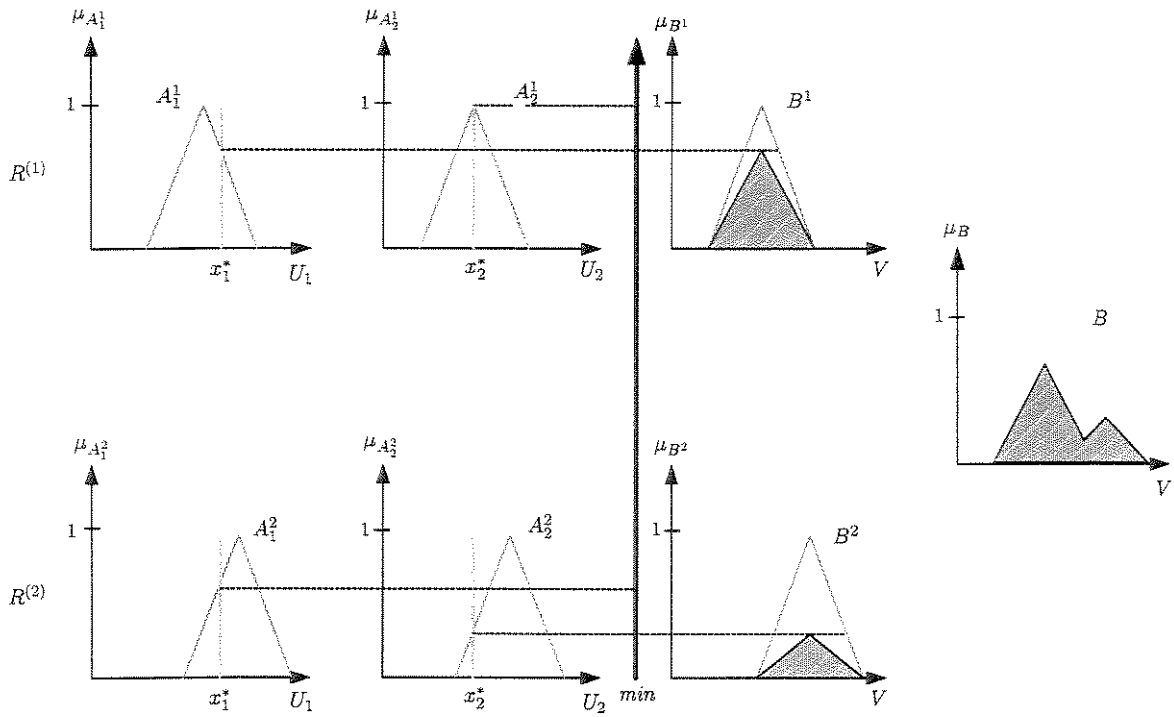


Figura 2.3: Mecanismo de inferência utilizando o operador *mínimo* como conectivo E. As regras são interpretadas pelo *produto algébrico* (*Larsen*) e a agregação das regras é realizada pelo operador *máximo*.

Dentre as principais estratégias de defuzzificação encontradas na literatura, destacam-se:

- Média dos Máximos.
- Centro de Área.
- Critério do Máximo.

Não foi objetivo deste trabalho realizar comparações entre os diversos métodos de defuzzificação. Para todos os exemplos abordados utilizou-se o método do centro de área. Informações detalhadas adicionais sobre os diversos métodos de defuzzificação existentes podem ser encontradas por exemplo em (Lee 1990*a*, Lee 1990*b*, Driankov *et al.* 1996*a*).

O método do centro de área retorna o centro de área do conjunto nebuloso (B). Considerando um universo de discurso discreto a saída produzida por este defuzzificador é:

$$y_o = \frac{\sum_{j=1}^{n} \mu_B(w_j) \cdot w_j}{\sum_{j=1}^{n} \mu_B(w_j)} \tag{2.5}$$

onde n é o número de elementos do universo de discurso de saída e $\mu_B()$ é a função de pertinência do conjunto nebuloso B.

Nos casos em que se utiliza o universo de discurso de saída normalizado, deve-se executar uma desnormalização, que mapeia o valor numérico resultante da defuzzificação para o valor de saída em seu domínio físico, por exemplo, $[-12,0\,;12,0]$ Volts.

2.3 Controlador de *Takagi & Sugeno*

Takagi e *Sugeno* (Takagi & Sugeno 1985) propuseram um modelo alternativo de sistema nebuloso para aplicação em identificação de sistemas e controle. As regras definidas para este tipo de sistemas são denominadas regras nebulosas *Takagi-Sugeno* (TS), e possuem a seguinte forma:

$$R^{(l)}\text{: } \underline{\textbf{SE}}\ x_1 \text{ é } A_1^l\ \underline{\textbf{E}}\ x_2 \text{ é } A_2^l\ \underline{\textbf{E}} \ldots\ \underline{\textbf{E}}\ x_n \text{ é } A_n^l\ \underline{\textbf{ENTÃO}}\ y^l = f^l(x_1^*, x_2^*, \ldots, x_n^*).$$

onde $l = 1, \ldots, \Omega$; A_i^l $(i = 1, \ldots, n)$ são conjuntos nebulosos; $x = (x_1, \ldots, x_n)^T \in U = U_1 \times \ldots \times U_n$ são as variáveis nebulosas de entrada; y^l é a variável de saída do conseqüente da regra $R^{(l)}$ e $x^* = (x_1^*, \ldots, x_n^*)^T$ é o vetor formado pelos valores numéricos das entradas do sistema nebuloso.

Originalmente nos controladores de *Takagi–Sugeno*, os conseqüentes das regras são funções lineares das variáveis de entrada, portanto, a função f^l é dada por:

$$f^l(x_1^*, x_2^*, \ldots, x_n^*) = c_0^l + c_1^l\, x_1^* + c_2^l\, x_2^* + \ldots + c_n^l\, x_n^*. \tag{2.6}$$

onde c_i^l, $i = \{0, \ldots, n\}$, são parâmetros ajustáveis do conseqüente da regra $R^{(l)}$.

A saída de controle obtida da composição de todas as regras nebulosas *Takagi-Sugeno* é dada pela seguinte soma ponderada:

$$y = \frac{\sum_{l=1}^{\Omega} \mu_l \cdot y^l}{\sum_{l=1}^{\Omega} \mu_l} \tag{2.7}$$

onde $\mu_l = \mu_{A_1^l}(x_1^*) \wedge \mu_{A_2^l}(x_2^*) \wedge \ldots \wedge \mu_{A_n^l}(x_n^*)$, Ω é o número total de regras e $\wedge$ é uma conjunção nebulosa, usualmente interpretada pelos operadores *mínimo* ou *produto algébrico*.

A abordagem de *Takagi–Sugeno* tem sido amplamente utilizada em problemas de modelagem nebulosa, escalonamento de ganhos[4], identificação de sistemas e controle multi-variável (Buckley 1993, Wang *et al.* 1995, Driankov *et al.* 1996*b*, Begovich *et al.* 2000, Ying 2000).

Uma desvantagem destes controladores é que os conseqüentes das regras não são conjuntos nebulosos, de modo que não permitem a incorporação direta do conhecimento de especialistas.

Por outro lado, como os conseqüentes das regras TS são funções explícitas das variáveis de entrada, o sinal de controle de saída é calculado diretamente através de uma soma ponderada (2.7), dispensando a utilização dos operadores de agregação e de defuzzificação. Portanto, os controladores de *Takagi–Sugeno* resultam em algoritmos mais rápidos que os dos controladores nebulosos tradicionais, por esta razão são bastante adequados para aplicações de tempo real.

Do ponto de vista de otimização de sistemas nebulosos, a abordagem de *Takagi-Sugeno* apresenta a vantagem de utilizar apenas o processo de otimização paramétrica, ou seja, ajuste das funções de pertinência e dos parâmetros das funções dos conseqüentes das regras. Enquanto isso, os controladores nebulosos tradicionais necessitam de dois processos: otimização paramétrica (ajuste das funções de pertinência) e otimização da base de regras lingüísticas.

2.4 Controladores Nebulosos Tipo PID

Os controladores clássicos tipo PID (proporcional+integral+derivativo) e suas variações (PI e PD) têm sido amplamente utilizados em aplicações industriais, principalmente devido à sua simplicidade de implementação e sintonia. Alguns métodos heurísticos

[4]Tradução utilizada para *gain scheduling*

conhecidos para sintonia de parâmetros (Ziegler & Nichols 1942, Aström *et al.* 1992) fornecem ferramentas e/ou critérios eficientes para o projeto destes controladores.

Os controladores nebulosos que utilizam como variáveis ligüísticas o erro (e), a variação do erro (Δe) e a integral do erro (δe), são denominados de controladores nebulosos PID (Peng *et al.* 1988) ou controladores lingüísticos PID (Kwok *et al.* 1990).

O fato dos controladores nebulosos poderem possuir uma grande quantidade de parâmetros e uma estrutura não linear resulta em uma maior flexibilidade de sintonia local e global. Por exemplo, a modificação de alguns parâmetros locais com o objetivo de melhorar o tempo de subida sem comprometer o sobre-sinal (Zheng 1992).

A partir de uma configuração específica de um controlador nebuloso pode-se obter um controlador clássico PID, portanto, o controlador PID pode ser visto como uma classe particular dos controladores nebulosos.

Vários trabalhos experimentais têm procurado mostrar que os controladores nebulosos PID apresentam desempenhos superiores, ou no mínimo semelhantes, aos dos controladores clássicos PID (Peng *et al.* 1988, Li & Lau 1989, Coleman & Godbole 1994).

Existem duas abordagens utilizadas para a implementação dos controladores nebulosos PID :

(1) Utilizando regras puramente lingüísticas: permite uma maior interação com especialistas e operadores, possibilitando fazer-se uma descrição lingüística das regras de controle.

(2) Utilizando regras *Takagi–Sugeno*: implementa um escalonador de ganhos, no qual vários controladores PID são projetados para diferentes regiões de operação, e o mecanismo de inferência das regras realiza transições suaves entre estes controladores de acordo com cada ponto de operação.

Os controladores nebulosos PID e PD utilizados neste trabalho são implementados através de regras *Takagi–Sugeno*, por facilidade de análise e por serem bastante adequadas para processos de otimização.

As regras *Takagi–Sugeno* para um controlador nebuloso PID podem ser descritas da seguinte forma:

$R^{(1)}$: **SE** $e(k)$ é A_1^1 **E** $\Delta e(k)$ é A_2^1 **E** $\delta e(k)$ é A_3^1
ENTÃO $u_1 = K_{P1} \cdot e(k) + K_{D1} \cdot \Delta e(k) + K_{I1} \cdot \delta e(k)$
$\vdots$

$R^{(j)}$: **SE** $e(k)$ é A_1^j **E** $\Delta e(k)$ é A_2^j **E** $\delta e(k)$ é A_3^j
ENTÃO $u_j = K_{Pj} \cdot e(k) + K_{Dj} \cdot \Delta e(k) + K_{Ij} \cdot \delta e(k)$

onde $j = 1, \ldots, \Omega$; A_i^j $(i = 1, \ldots, 3)$ são conjuntos nebulosos; K_{Pj}, K_{Dj} e K_{Ij} são os ganhos proporcional, derivativo e integral do controlador associado à regra j; $e(k)$ é o erro entre a saída da planta e a referência desejada, $\Delta e(k)$ é a variação do erro e $\delta e(k)$ é a integral do erro, sendo todos estes valores definidos no instante k. As equações que definem $\Delta e(k)$ e $\delta e(k)$ em função de $e(k)$ são descritas a seguir.

$$\Delta e(k) = \frac{e(k) - e(k-1)}{T} \tag{2.8}$$

$$\delta e(k) = \sum_{j=1}^{k} e(j) \cdot T \tag{2.9}$$

Nas equações acima T denota o período de amostragem.

Seja P_e o número de conjuntos nebulosos definidos para a variável erro, P_Δ o número de conjuntos nebulosos definidos para a variável variação do erro, e P_δ o número de conjuntos nebulosos definidos para a variável integral do erro, então o número total de regras do controlador nebuloso é dado por

$$\Omega = P_e \cdot P_\Delta \cdot P_\delta. \tag{2.10}$$

Conseqüentemente, o número total de parâmetros dos conseqüentes das regras é $3 \cdot \Omega$.

Por exemplo, se para cada variável de entrada forem definidos 3 conjuntos nebulosos ($P_e = P_\Delta = P_\delta = 3$), o número total de regras será 27, resultando num total de 81 parâmetros para os conseqüentes das regras.

Existem alguns estudos que procuram reduzir o número de parâmetros ajustáveis dos controladores nebulosos (Hampel & Chaker 1998, Ying 1998*b*). Estes estudos geralmente empregam propriedades de similaridade ou algumas restrições na estrutura do controlador.

O modelo do controlador nebuloso *PD* pode facilmente ser obtido a partir do modelo descrito acima, bastando apenas suprimir o termo da integral do erro.

2.5 Estabilidade de Controladores Nebulosos

Os controladores nebulosos têm demonstrado ser ferramentas poderosas quando aplicados a problemas em que as estratégias tradicionais falham.

Os métodos de projeto da maioria dos controladores nebulosos são baseados no conhecimento heurístico obtido dos engenheiros de controle e/ou operadores. Conseqüentemente, do ponto de vista da engenharia de controle, os maiores avanços na área de controle nebuloso estão concentrados no desenvolvimento de controladores nebulosos específicos para determinadas aplicações, ao invés do desenvolvimento de metodologias de análise e projeto, que são fundamentais em engenharia de controle.

A falta de técnicas analíticas satisfatórias para o estudo da estabilidade de processos envolvendo controladores nebulosos tem sido considerada a maior desvantagem destes controladores.

Sistemas de controle nebuloso são sistemas intrinsecamente não-lineares. Por esta razão, é difícil a obtenção de resultados gerais sobre as respectivas análises e projetos. Além disso, o conhecimento sobre a dinâmica do processo a ser controlado normalmente é escasso e/ou impreciso.

Existem dois paradigmas para análise de estabilidade de controladores nebulosos:

(1) **Teoria clássica de sistemas dinâmicos não-lineares:** o sistema a ser controlado é considerado um sistema não-nebuloso, e o controlador nebuloso é estudado como uma classe de controladores não-lineares.

(2) **Teoria de sistema dinâmicos nebulosos:** o sistema a ser controlado é considerado um modelo nebuloso. Pesquisas nesta área incluem critérios de estabilidade baseados

no conceito de energia ou controlabilidade de sistemas nebulosos.

Um dos trabalhos pioneiros envolvendo estabilidade em malha fechada de controladores nebulosos foi desenvolvido por *Tong* (Tong 1978). Neste trabalho, a análise de estabilidade é baseada na função não-linear do controlador, que pode ser obtida a partir da matriz relacional do sistema nebuloso.

Braae e *Rutherford* (Braae & Rutherford 1979) introduziram o conceito de trajetória lingüística de sistemas nebulosos. A partir deste conceito, pode-se estudar o comportamento do sistema de controle e realizar análises de estabilidade, através de abordagens semelhantes às empregadas na teoria geral de sistemas não lineares (planos de fase, análises espectrais, mapas de *Poincaré*, mapas de bifurcação, expoentes de *Lyapunov*, entropia de *Kolmogorov-Sinai*, e outras conjecturas) (Guckenheimer & Holmes 1983, Parker & Chua 1989, Ferrara & Prado 1994).

Contribuições importantes para projeto e análise de controladores nebulosos surgiram a partir do trabalho de *Takagi* e *Sugeno* (Takagi & Sugeno 1985). *Tanaka* e *Sugeno* (Tanaka & Sugeno 1992) introduziram o conceito de blocos nebulosos e apresentaram uma metodologia para projeto e análise da estabilidade usando o método direto de *Lyapunov*. Em (Ying 1998*a*) é apresentado um procedimento com condições necessárias e suficientes para a determinação analítica da estabilidade assintótica local (pontos de equilíbrio) de sistemas controlados por controladores nebulosos que usam regras *Takagi-Sugeno* baseada no segundo método de *Lyapunov*, sendo que há um comentário explícito onde o autor afirma que aparenta ser impossível estabelecer analiticamente as condições de estabilidade global sabendo que não há o conhecimento preciso das expressões analíticas nem da planta nem do controlador, simplesmente porque no caso é impossível encontrar funções de *Lyapunov* corretas e necessárias para a análise.

Nos problemas de controle analisados nesta tese, são feitas as seguintes considerações:

- O controlador nebuloso é tratado como uma "caixa preta" a ser otimizada.
- O conhecimento a respeito da dinâmica da planta a ser controlada é qualitativo, portanto, o seu modelo matemático é desconhecido.
- Não se utiliza nenhum método à priori para identificação da planta a ser controlada.

Com base nas considerações acima, fica portanto muito difícil estabelecer critérios

de estabilidade para o sistema controlado. Contudo, obteve-se comprovação experimental que os controladores nebulosos obtidos a partir do processo de otimização proposto apresentam estabilidade global e desempenhos bastante satisfatórios.

Capítulo 3

Algoritmos Genéticos

3.1 Introdução

A imitação de processos observados na natureza tem proporcionado inquestionáveis benefícios ao homem. Existem inúmeros exemplos em que os humanos tentaram imitar formas, estruturas e processos de outros seres vivos. Por exemplo, o homem conseguiu voar através de máquinas (aviões) que foram originalmente inspiradas nos pássaros. Será que o homem teria insistido na idéia de voar se os pássaros não existissem?

Algoritmos Genéticos (*AGs*), cujo conceito foi introduzido por *Holland* em 1975 (Holland 1975), são algoritmos de otimização nos quais os métodos de busca são inspirados na genética e nos mecanismos de seleção natural de *Darwin* (Darwin 1859).

Os algoritmos genéticos utilizam estruturas de dados, tipicamente cadeias binárias, representando possíveis soluções (indivíduos) de um problema de otimização. Essas estruturas são denominadas de cromossomos. Associado a cada indivíduo existe um valor de adaptação que indica o quanto a sua solução é "boa".

A busca por melhores soluções ocorre através da simulação de um ambiente natural onde indivíduos reunidos em uma população competem entre si por recursos limitados. Assim como ocorre na natureza, os indivíduos mais adaptados terão maior chance de sobreviver e gerar descendentes. Por sua vez, os descendentes geralmente são mais adaptados que os pais, devido a seus cromossomos serem uma combinação dos cromossomos dos indivíduos mais adaptados.

As principais vantagens dos mecanismos de busca dos *AGs* são:

- O espaço de soluções é explorado paralelamente por buscas em diferentes regiões, permitindo a realização de buscas globais no espaço que pode conter as soluções.
- Os mecanismos de busca não são baseados em informações de gradiente e, portanto, não há exigências de continuidade ou convexidade do espaço de busca.

O mundo real é não-linear e dinâmico, e está repleto de fenômenos conhecidos como caos determinístico, instabilidades e geometrias fractais. Atualmente, com a evolução tecnológica dos computadores, é possível obter modelos computacionais cada vez mais próximos da realidade sem ignorar os fenômenos não-lineares. Determinar as soluções ótimas para tais modelos através de minimização ou maximização de uma medida de qualidade, pode ser uma tarefa impossível quando se utiliza métodos fortes (linearizações, aproximações que empregam informação de ordem elevada,...). Estes métodos foram concebidos para mundos lineares, contínuos e diferenciáveis. Os algoritmos genéticos apresentam-se como métodos capazes de superar estas limitações, não por garantir a obtenção da solução ótima em poucos experimentos, mas por encontrarem boas aproximações no sentido das soluções em tempos que normalmente têm crescimento menor que o exponencial em função do aumento do número de variáveis de busca.

Vários trabalhos teóricos e experimentais têm mostrado que os algoritmos genéticos realizam buscas robustas em espaços complexos, apresentando uma abordagem válida para problemas que requerem buscas efetivas e eficazes (Goldberg 1989, Whitley 1993).

Os *AGs* têm sido aplicados com sucesso em diversos problemas de otimização, tais como, roteamento de circuitos, planejamento de tarefas, controle adaptativo, modelagem cognitiva, problemas de transporte, o problema do caixeiro viajante, problemas de controle ótimo, otimização de sistemas nebulosos, ajuste de redes neurais, etc. (Whitley *et al.* 1989, Yamada & Nakano 1992, Srikanth *et al.* 1995, Gen *et al.* 1995, Buckley *et al.* 1996, Karr *et al.* 2000)

Neste trabalho, os *AGs* são utilizados para o ajuste paramétrico de controladores nebulosos, mais precisamente o ajuste dos coeficientes dos conseqüentes das regras nebulosas. Para tanto, propõe-se um algoritmo genético com codificação em ponto flutuante e operadores genéticos especiais. O algoritmo proposto possui características que permitem sua aplicação não somente em simulações, mas em sistemas de controle práticos e em tempo real.

Neste capítulo, inicialmente faz-se uma descrição do algoritmo genético clássico e discute-se as vantagens da codificação em ponto flutuante. Em seguida, descreve-se o algoritmo genético proposto para otimização dos controladores nebulosos abordados. Finalmente, são feitas algumas considerações sobre a função de *fitness* usada para medir a qualidade dos indivíduos que participam do processo de busca por soluções.

3.2 O Algoritmo Genético Clássico

O algoritmo genético clássico utiliza cadeias binárias apropriadamente codificadas para representar as variáveis de busca. Estas cadeias binárias são denominadas de cromossomos (ver figura 3.1). O *AG* trabalha com uma população de cromossomos (ou indivíduos), cada um representando uma solução possível para o problema em questão, possuindo um valor de adaptação, determinado pela função de *fitness*, que indica o quanto a solução é boa. Em um problema de otimização, a função de *fitness* é uma função que se objetiva maximizar.

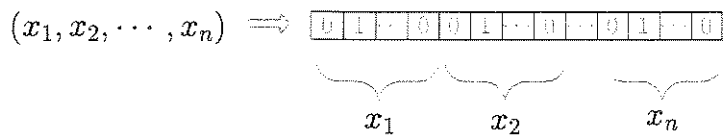


Figura 3.1: Codificação das variáveis de busca.

Suponha que se deseja maximizar uma função de k variáveis, $f(x_1, \cdots, x_k) : \mathbb{R}^k \rightarrow \mathbb{R}$, e que cada variável possa assumir valores no intervalo $D_i = [a_i, b_i] \subseteq \mathbb{R}$ e $f(x_1, \cdots, x_k) \geq 0$ para todo $x_i \in D_i$. Para obter-se uma precisão de 6 casas decimais, cada intervalo D_i deve ser dividido em $(b_i - a_i) \cdot 10^6$ partes iguais. Seja m_i o menor inteiro tal que $(b_i - a_i) \cdot 10^6 \leq 2^{m_i} - 1$. Então a representação de cada variável x_i por uma cadeia binária de comprimento m_i satisfaz a precisão requerida. A seguinte fórmula converte o código binário de uma variável para o seu valor real:

$$x_i = a_i + decimal(100\cdots0101_2) \cdot \frac{b_i - a_i}{2^{m_i} - 1} \tag{3.1}$$

onde a função *decimal*() retorna o valor decimal da cadeia binária.

Com isso, cada cromossomo (solução possível) é representado por uma cadeia binária de comprimento $m = \Sigma_{i=1}^{k} m_i$; os primeiros m_1 bits representam um valor definido em $[a_1, b_1]$, o próximo grupo de m_2 bits representam um valor definido em $[a_2, b_2]$, e assim por diante.

Em geral, os algoritmos genéticos tratam com populações contendo um número fixo de cromossomos (*tampop*). A população inicial é constituída de cromossomos cujos bits foram gerados aleatoriamente. Entretanto, se existe alguma informação a priori sobre a região que contém as possíveis soluções ótimas, esta informação pode ser utilizada na sua geração.

As buscas por melhores soluções são realizadas por operadores genéticos, tipicamente recombinação (*crossover*) e mutação. O processo de aplicação destes operadores é chamado de reprodução. Após a reprodução, os cromossomos da população resultante devem ser avaliados através da função de *fitness*. Os cromossomos que passarão para a próxima geração são selecionados seguindo uma distribuição de probabilidade associada aos seus valores de *fitness*, ou seja, o valor de *fitness* de um dado cromossomo indicará a probabilidade deste passar para a próxima geração.

Após satisfeito algum critério de parada, o processo de evolução é finalizado e a melhor solução encontrada é apresentada. A figura 3.2 ilustra o fluxograma básico de um *AG*.

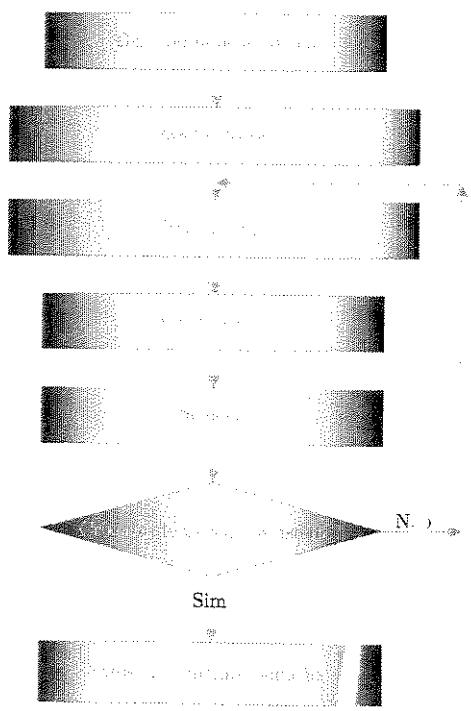


Figura 3.2: Fluxograma básico de um AG.

A evolução é um processo que opera sobre os cromossomos e não sobre os organismos. O processo evolutivo se dá durante a etapa de reprodução. Existem dois parâmetros

relacionados ao processo de reprodução: a probabilidade de *crossover* p_c e a probabilidade de mutação p_m.

Na operação de *crossover*, os cromossomos dos pais são combinados para formar os cromossomos dos filhos. Para realizar a operação *crossover* simples nos indivíduos de uma população, procede-se da seguinte maneira.

Para cada par de cromossomos na população, executa-se os seguintes passos:

1. Gera-se um número aleatório real r no intervalo $[0, 1]$.
2. Se $r < p_c$ então execute a recombinação da seguinte forma:
   - Gera-se um número aleatório (pos) no intervalo $[1, m-1]$ que indica a posição de corte no cromossomo, onde m é o número de bits do cromossomo.
   - Os dois cromossomos
     $(b_1 b_2 \cdots b_{pos} b_{pos+1} \cdots b_m)$
     $(c_1 c_2 \cdots c_{pos} c_{pos+1} \cdots c_m)$
     são substituidos por:
     $(b_1 b_2 \cdots b_{pos} c_{pos+1} \cdots c_m)$
     $(c_1 c_2 \cdots c_{pos} b_{pos+1} \cdots b_m)$

Na reprodução, o processo de cópia do material genético dos pais está sujeito a erros, e a troca aleatória de um gene por outro neste processo é chamada de mutação.

A mutação é uma conseqüência do processo de reprodução ocorrer num universo com diferencial de entropia positivo, constituindo um mecanismo importante para a implantação da diversidade nas espécies.

Nos *AGs*, a operação de mutação é realizada bit a bit nos cromossomos. Para cada cromossomo da população e para cada bit deste cromossomo realiza-se as seguintes operações.

1. Gera-se um número aleatório real r no intervalo $[0, 1]$.
2. Se $r < p_m$ então execute a mutação neste bit, ou seja, mude o seu valor de "0" para "1" ou vice-versa.

O processo de seleção é realizado através do algoritmo *Roulette Wheel*, que é construído da seguinte maneira:

1. Calcula-se o valor de *fitness* $eval(\mathbf{v}_i)$ para cada cromossomo $\mathbf{v}_i$ $(i = 1, \cdots, tampop)$. Neste caso assume-se que os valores de *fitness* são positivos, senão, poder-se-ia utilizar algum mecanismo de escalonamento.

2. Calcula-se o *fitness* total da população

$$F = \sum_{i=1}^{tampop} eval(\mathbf{v}_i). \tag{3.2}$$

3. Calcula-se a probabilidade de seleção p_i para cada cromossomo $\mathbf{v}_i$ $(i = 1, \cdots, tampop)$:

$$p_i = eval(\mathbf{v}_i)/F. \tag{3.3}$$

4. Calcula-se a probabilidade cumulativa q_i para cada cromossomo $\mathbf{v}_i$ $(i = 1, \cdots, tampop)$:

$$q_i = \sum_{i=1}^{j} p_j. \tag{3.4}$$

A seleção consiste em executar o algoritmo *roulette wheel* várias vezes até completar uma nova população. Em cada execução, um cromossomo é selecionado para formar a nova população através dos seguintes passos:

1. Gera-se um número aleatório real r no intervalo $[0, 1]$.

2. Se $r < q_1$ então selecione o cromossomo $\mathbf{v}_1$; senão selecione o i-ésimo cromossomo $\mathbf{v}_i$ $(2 \leq i \leq tampop)$ tal que $q_{i-1} \leq r \leq q_i$.

Obviamente, alguns cromossomos poderão ser selecionados mais de uma vez e outros poderão não ser selecionados. Em média, os melhores indivíduos geram várias réplicas, os indivíduos médios conseguem passar para a próxima geração, e os piores não são selecionados. Isto não implica, por exemplo, que o pior indivíduo de uma geração não possa ser selecionado para a próxima geração.

3.3 Codificação em Ponto Flutuante

A representação binária, tradicionalmente utilizada nos algoritmos genéticos, apresenta algumas desvantagens quando aplicadas em problemas multi-dimensionais que exijam alta precisão numérica. Imagine um problema no qual seja necessário ajustar 200 parâmetros. Utilizando um algoritmo genético com codificação binária e reservando 10 bits para cada parâmetro, cada cromossomo que representa uma solução teria 2000 genes. Efetuar todas as operações de reprodução de um *AG*, neste tipo de problema, significa um custo elevado em termos de recursos computacionais.

A codificação binária facilita a análise teórica dos algoritmos genéticos e permite a utilização de operadores genéticos simples e elegantes. Porém, o paralelismo implícito dos *AGs* não depende da codificação binária, conseqüentemente, pode ser interessante utilizar outros alfabetos e operadores genéticos. Neste sentido, por experiência histórica os alfabetos menores tendem a ser mais eficientes.

Como o problema de otimização tratado neste trabalho consiste no ajuste paramétrico de variáveis definidas em domínios contínuos, optou-se pela codificação em ponto flutuante com operadores genéticos especiais.

As principais vantagens da codificação em ponto flutuante são: redução da carga computacional na execução do algoritmo; há uma equivalência direta entre o espaço de representação do *AG* e o espaço de busca do problema de otimização; permite a elaboração de operadores genéticos específicos para cada problema abordado.

3.4 O Algoritmo Genético Proposto

A opção de desenvolver um algoritmo genético com codificação dos cromossomos em ponto flutuante surge da necessidade de obter-se um algoritmo genético eficiente para problemas de otimização que envolvam o ajuste de uma grande quantidade de parâmetros contínuos que assumem valores reais.

O algoritmo genético desenvolvido baseia-se no mesmo princípio de reprodução utilizado no *AG* clássico. Para a codificação em ponto flutuante, foram definidas duas operações básicas de reprodução: o *crossover* e a mutação. A codificação utilizada permite que estes operadores controlem o modo como o espaço de busca é explorado.

A forma na qual o algoritmo é iniciado depende do problema em questão. Nos casos em que o controlador a ser otimizado é testado através de simulações, a população inicial pode ser gerada aleatoriamente como na maioria dos *AGs*. Entretanto, nos casos em que o controlador é testado em um sistema físico controlado por computador, a população inicial é gerada de forma determinística para que seus respectivos controladores não causem danos ao sistema.

Antes de descrever os operadores genéticos, alguns parâmetros relacionados aos genes dos cromossomos precisam ser definidos. Seja j um gene específico que representa um parâmetro a ser otimizado, então, associado a ele existem três parâmetros a considerar:

min_j: valor mínimo que o gene j pode assumir.

max_j: valor máximo que o gene j pode assumir.

$range_j = max_j - min_j$: largura do intervalo utilizado pelo gene j.

3.4.1 Operadores Genéticos

Os tipos de operadores genéticos são muito importantes para o sucesso em uma aplicação específica de um algoritmo genético. Como foi comentado anteriormente, são utilizados dois operadores genéticos: o operador *crossover* uniforme e o operador mutação, os quais foram baseados em Michalewicz (1996).

O Operador *Crossover* Uniforme

O operador *crossover* uniforme recombina os genes dos cromossomos dos pais para formar os cromossomos dos filhos. A taxa de *crossover* (p_c) determina a probabilidade de um par de cromossomos gerar descendentes. A operação de *crossover* uniforme pode ser resumida nos seguintes passos:

1. Gera-se um número aleatório real r no intervalo $[0, 1]$.
2. Se $q < p_c$ então execute a recombinação da seguinte forma:
   (a) Sorteia-se o número de genes que serão trocados.
   (b) Gera-se uma seqüência aleatória contendo o índice dos genes que serão trocados.

(c) Efetua-se a troca dos genes sorteados.

Exemplo:

Os genes sublinhados indicam os genes que serão trocados.

cromossomo 1:$(0,10;\underline{0,45};0,32;\underline{0,87};0,65;\cdots;\underline{0,23})$

cromossomo 2:$(0,14;\underline{0,11};0,33;\underline{0,56};0,99;\cdots;\underline{0,77})$

Após o *crossover*:

cromossomo 1:$(0,10;\underline{0,11};0,32;\underline{0,56};0,65;\cdots;\underline{0,77})$

cromossomo 2:$(0,14;\underline{0,45};0,33;\underline{0,87};0,99;\cdots;\underline{0,23})$

O Operador Mutação

O operador mutação atua como uma perturbação em torno do ponto $n-$dimensional representado pelo cromossomo. A taxa de mutação (p_m) é a probabilidade de um cromossomo ser alterado, ou sofrer mutação genética. A operação de mutação é realizada da seguinte forma:

1. Gera-se um número aleatório real q no intervalo $[0,1]$.
2. Se $q < p_m$ então
   (a) Sorteia-se o número de genes que sofrerão mutação.
   (b) Gera-se uma seqüência aleatória contendo os índices dos genes que serão alterados.
   (c) Para cada gene escolhido para mutação executa-se o seguinte procedimento:
   - Gera-se um número aleatório $l \in \{0,1\}$

$$gene_j(k+1) = \begin{cases} gene_j(k) + range_j \cdot w \cdot \Delta(k+1) & \text{se } l = 0 \\ gene_j(k) - range_j \cdot w \cdot \Delta(k+1) & \text{se } l = 1 \end{cases}$$

Se $gene_j(k+1) > max_j$ então $gene_j(k+1) = max_j$

Se $gene_j(k+1) < min_j$ então $gene_j(k+1) = min_j$

onde $w \in [0,0;1,0]$ indica o valor máximo em relação a $range_j$ que pode ser adicionado ao gene j, e $\Delta(n)$ é uma função que retorna um valor aleatório no intervalo $[0,0;1,0]$, de forma que a probabilidade de $\Delta(n)$ ser próxima de zero aumenta à medida que n cresce.

Esta propriedade implica na realização de buscas uniformes no início do processo de otimização (quando n é pequeno), e em buscas locais no final do processo. A função utilizada para $\Delta(n)$ é dada por:

$$\Delta(n) = (1 - r^{(1-\frac{n}{N_g})^b}) \tag{3.5}$$

onde r é um número aleatório definido em $[0,0;1,0]$, N_g é o número máximo de gerações, e b é um parâmetro que determina o grau de dependência da função $\Delta(n)$ em relação ao número de gerações n. Nos experimentos realizados adotou-se $b = 2,0$. A figura 3.3 ilustra a o comportamento da função $\Delta(n)$ em função de r para diversos valores da razão n/N_g.

Exemplo:

Os genes sublinhados (alelos) são os genes que sofrerão mutação.

cromossomo:$(0,10;\underline{0,45};0,32;\underline{0,87};0,65;\cdots;\underline{0,23})$

Após a mutação:

cromossomo:$(0,10;\underline{0,47};0,32;\underline{0,88};0,65;\cdots;\underline{0,20})$

O parâmetro w determina o tamanho da região na qual a perturbação ocorre. Nos experimentos realizados utilizou-se $w = 0,15$ significando uma alteração máxima de $\pm 15\%$ em relação ao seu atual valor.

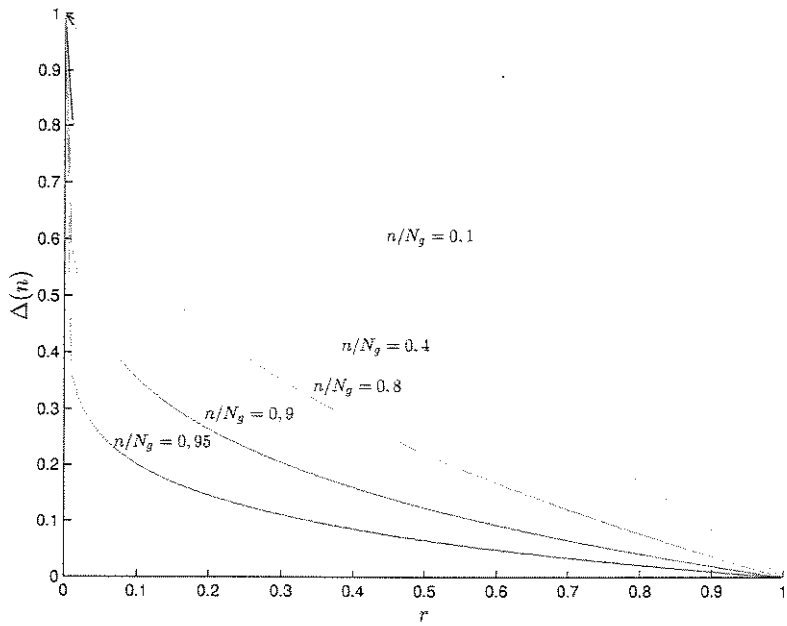


Figura 3.3: Função $\Delta(n)$

3.4.2 Reprodução e Seleção

O processo de reprodução explora o espaço de soluções através de operadores genéticos. No *AG* proposto, os processos de reprodução e seleção utilizam 4 sub-populações (sp) geradas a partir da população principal (Pop), ver figura 3.4.

Estas sub-populações possuem o mesmo tamanho da população inicial e são formadas da seguinte maneira:

- a sub-população sp1 é uma cópia da população principal.
- as sub-populações sp2 e sp3 são combinações entre o melhor indivíduo da população principal e os restantes;
- a sub-população sp4 é formada aplicando-se o algoritmo *roulette wheel* na população principal;

No processo de reprodução participam apenas as subpopulações sp2, sp3 e sp4.

Após a reprodução, o *fitness* de cada cromossomo é calculado através de uma avaliação do controlador equivalente quando aplicado na tarefa de controle em questão. A

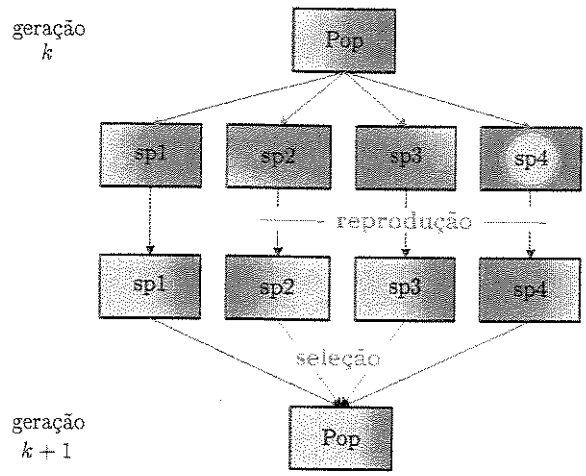


Figura 3.4: Processos de reprodução e seleção.

avaliação do controlador pode ser feita através de simulação ou execução da tarefa num sistema de controle em tempo real.

O processo de seleção consiste inicialmente na escolha de N_BEST melhores cromossomos e do pior cromossomo das sub-populações. Em seguida, aplica-se o algoritmo *roullete wheel* nas sub-populações geradas até completar uma nova população principal.

O processo completo de otimização utilizado no algoritmo genético proposto está mostrado no fluxograma da figura 3.5.

3.5 Função de *Fitness*

Um dos maiores problemas da teoria de otimização está na definição de funções multi-objetivo. Quais critérios devem ser incluídos e como ponderá-los em função de sua importância?

Definir um objetivo global que satisfaça várias metas requer um bom conhecimento do problema e uma organização das prioridades da otimização.

A função de *fitness* é uma expressão numérica que mede quantitativamente o nível de adaptação de um indivíduo do algoritmo genético. Quanto maior o valor do *fitness* mais adaptado é considerado o indivíduo. Neste contexto, em geral, um indivíduo representa

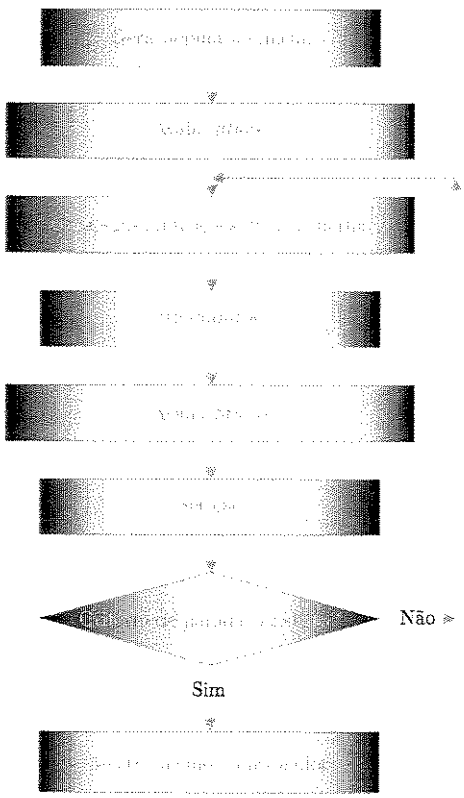


Figura 3.5: Fluxograma do *AG* proposto.

uma solução candidata do problema, portanto, a função de *fitness* pode ser entendida como a função objetivo de um problema de maximização.

Neste trabalho, os *AGs* são utilizados para otimizar controladores nebulosos, conseqüentemente, a função de *fitness* deve incluir medidas de desempenho destes controladores quando aplicados à tarefa de controle.

Em problemas de controle, várias medidas podem ser consideradas na avaliação de desempenho, por exemplo: consumo de energia, tempo de subida, tempo de estabilização, sobresinal máximo, erro de regime, etc. Para os problemas de controle abordados neste trabalho utiliza-se três medidas importantes para avaliar tal desempenho:

$t_s \in [0,1]$: Razão tempo de estabilização/tempo total da tarefa de controle.
$e_r \in [0,1]$: Erro de regime normalizado.
$M_p \in [0,1]$: Sobresinal máximo normalizado.

Estes parâmetros são combinados para formar a seguinte função de *fitness*:

$$F = 0.2(1 - M_p)^{10} + 0.3(1 - e_r)^2 + 0.5(1 - t_s). \tag{3.6}$$

A figura 3.6 mostra como cada termo da função de *fitness* contribui na medida de desempenho do controlador. O objetivo global do problema de otimização é maximizar F, ou seja, minimizar M_p, e_r e t_s simultaneamente. Os fatores que ponderam cada termo da equação podem ser modificados de acordo com o objetivo da tarefa de controle. Por exemplo, se o processo a ser controlado exige um sobresinal baixo, então o fator associado ao sobresinal deve possuir um peso maior.

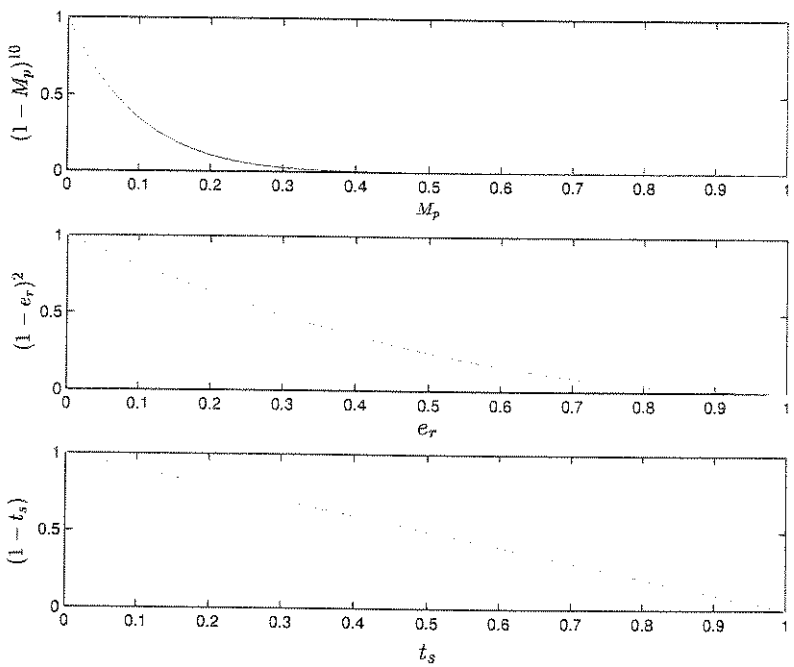


Figura 3.6: Comportamentos Individuais dos termos usados na definição da função de *fitness*.

Poder-se-ia a princípio utilizar uma infinidade de outros critérios para a definição de F, baseados na teoria da otimização multi-objetivo, que não é tão recente quanto se possa pensar (Kuhn & Tuker 1951), mas que permanece em intenso desenvolvimento até a atualidade. O critério adotado pode ser enquadrado nos chamados Métodos de Otimização com Objetivos Ponderados, onde adiciona-se todos os objetivos pretendidos, através de suas respectivas funções, para formar uma função objetivo global, ponderando com diferentes pesos os coeficientes de cada uma delas, segundo a respectiva importância perante as demais. Isso significa que o problema de otimização multi-objetivo é transformado em um problema de otimização mono-objetivo.

Do ponto de vista dos métodos numéricos que podem ser usados para fazer a varredura em busca das soluções, a localização dos respectivos pontos de ótimo depende não somente dos valores atribuídos a tais coeficientes, mas também das unidades de medida nas quais cada função que representa um objetivo particular está sendo expressada. Quando deseja-se que estes coeficientes reflitam bem a importância dos seus respectivos objetivos, suas funções associadas devem ser expressadas em unidades com valores numéricos de grandezas compatíveis.

O método utilizado para a definição da função de *fitness* é muito eficiente do ponto de vista computacional, pois pode ser processado com extrema velocidade e aplicado na busca de soluções muito eficazes e/ou alternativas para uma grande variedade de problemas.

Capítulo 4

Resultados Experimentais

4.1 Introdução

Este capítulo apresenta os principais resultados obtidos nos experimentos de otimização de controladores nebulosos realizados durante o desenvolvimento desta tese. O problema abordado consiste no controle de posição de um pêndulo acionado. Os detalhes técnicos sobre o *hardware* de controle e sistema mecânico utilizado encontram-se descritos no **Apêndice A**.

Em todas as tarefas executadas, utilizou-se controladores nebulosos de *Takagi-Sugeno* (TS) com funções de pertinência triangulares uniformemente distribuídas sobre seus universos de discurso. Os algoritmos genéticos foram aplicados no ajuste dos coeficientes dos conseqüentes das regras de *Takagi-Sugeno*.

Dentre os experimentos realizados, foram escolhidos os resultados de três exemplos muito ilustrativos para demonstrar a eficiência dos controladores nebulosos resultantes do emprego da técnica proposta:

- Controlador nebuloso tipo PD:
  - 9 regras;
  - 25 regras.
- Controlador nebuloso tipo PID com 27 regras
- Controlador nebuloso tipo PD com compensação de gravidade.

Os algoritmos desenvolvidos foram escritos em linguagem C^{++} e basicamente consistem em duas classes: *classe FuzzySystem*, utilizada para implementação de sistemas nebulosos; *classe Population*, que implementa o algoritmo genético proposto no **Capítulo 3**. O **Apêndice B** descreve as estruturas das classes que constituem as bibliotecas desenvolvidas.

O período de amostragem utilizado nos experimentos foi fixado em 5 milisegundos através de uma interrupção de *hardware*. A saída de controle (u) foi restrita a $|u| \leq 12,0$ Volts, devido às características do motor utilizado.

O algoritmo genético empregado no ajuste dos parâmetros dos controladores considera 20 indivíduos na população principal, totalizando 80 indivíduos nas subpopulações intermediárias por geração. Os parâmetros do AG utilizado nos experimentos encontram-se descritos na tabela 4.1.

Parâmetros	**Valor**
Taxa de Mutação	0,9
Taxa de *Crossover*	0,5
b	2,0
w	0,15

Tabela 4.1: Parâmetros do Algoritmo Genético (veja detalhes no Capítulo 3).

4.2 Controladores Nebulosos PD

4.2.1 Controlador Nebuloso PD com 9 Regras

O controlador nebuloso PD possui duas variáveis de entrada: o erro de posição (e) e a variação do erro de posição (Δe). Neste caso, para cada variável de entrada foram definidas três funções de pertinência triangulares, conforme ilustrado na figura 4.1. Portanto, o número total de regras para este controlador é 9. Para cada regra, existem 2 ganhos (K_P e K_D) a serem ajustados, totalizando 18 parâmetros. Os genes dos cromossomos que representam as variáveis a serem otimizadas foram divididos em 2 grupos:

- Para $j \in \{1, \ldots, 9\}$, $min_j = 0,0$ e $max_j = 48,0$ (Genes que representam os ganhos K_P do controlador PD).

- Para $j \in \{10, \ldots, 18\}$, $min_j = 0.0$ e $max_j = 4,8$ (Genes que representam os ganhos K_D do controlador PD).

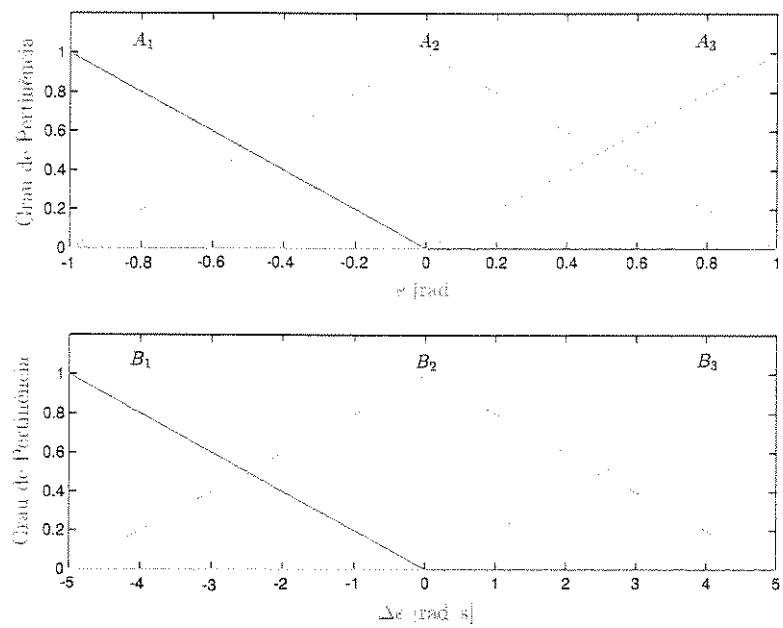


Figura 4.1: Funções de pertinência das variáveis de entrada. PD nebuloso (9 regras).

A população inicial é gerada atribuindo-se para todos os cromossomos: $gene_j = 24,0$ para $j = 1, \ldots, 9$; $gene_j = 2,4$ para $j = 10, \ldots, 18$. Ou seja, na primeira população, todos os controladores TS funcionam como um controlador PD linear com $K_P = 24,0$ e $K_D = 2,4$.

A referência de posição a ser seguida é composta por duas etapas: $\theta_{REF} = 180°$ para $t \in [0,0\,;\, 1,2)$ s e $\theta_{REF} = 0°$ para $t \in [1,2\,;\, 2,4]$ s, que são respectivamente os pontos de equilíbrio instável e estável do pêndulo. A não-linearidade intrínseca causada pela gravidade provoca comportamentos dinâmicos diferentes para cada ponto de equilíbrio. Para outras posições que não sejam os pontos de equilíbrio, este controlador nebuloso PD não é capaz de assegurar respostas de posição com erros de regime nulos (assim como também não é capaz um controlador PD linear).

Interpretar as ações dinâmicas dos controladores nebulosos TS não é tarefa trivial, em particular se a análise for baseada em dados experimentais. A intenção é que estes controladores consigam funcionar globalmente bem na tarefa de controle de sistemas não lineares, ou seja, em todo o espaço de estados de cada sistema em questão. Isso significa conseguir que suas regras (modelos locais) façam dinamicamente excelentes aproximações

lineares das trajetórias realizadas pelo sistema (na teoria, a tarefa equivalente seria encontrar as equações que representam a linearização dinâmica dos sistemas não lineares, comumente chamadas de equações variacionais associadas (Parker & Chua 1989)). Na atualidade há muitas evidências experimentais mostrando que os modelos nebulosos TS são capazes de fazer globalmente tais aproximações (Fantuzzi & Rovatti 1996, Shorten *et al.* 1999), porém verifica-se que tais modelos possuem grande sensibilidade paramétrica quando aplicados em regimes transitórios, isto é, fora das regiões vizinhas aos pontos de equilíbrio do sistema identificado/controlado.

Note que este raciocínio, seguido em muitas abordagens em que tenta-se interpretar as ações advindas da aplicação das regras TS baseando-se no senso de que estejam fazendo linearizações dinâmicas sobre o comportamento do sistema controlado, pode ser interpretado como mera especulação na tentativa de realizar estudos com maior rigor teórico sobre o assunto, no domínio da teoria de controle linear.

Shorten *et al.* (1999) argumentam que há grande sensibilidade paramétrica nos controladores baseados em regras TS devido à excessiva liberdade de movimentos que um sistema não linear pode ter fora de seus pontos de equilíbrio. Johansen *et al.* (1998) argumentam que tais aproximações podem ser excelentes desde que sejam permitidos conseqüentes não lineares nas regras TS, mas isso evidentemente pode aumentar muito a carga computacional. Também é intuitivo que se o número de regras $TS \longrightarrow \infty$, então melhor poderá ser o desempenho do respectivo controlador, mas esta idéia é contrária ao desejado do ponto de vista prático. Assim, é notório que este assunto de pesquisa ainda necessita aprofundamento teórico, e é um campo fértil para a realização de trabalhos científicos.

Para este caso específico que envolve compensação gravitacional, há uma solução engenhosa que resolve bem o problema, veja item 4.4.

A figura 4.2 mostra as respostas dos melhores controladores durante o processo de sintonia. As curvas pontilhadas indicam a referência a ser alcançada. Na primeira geração (−), observa-se que a resposta de posição possui erro de regime e alto sobre-sinal. Após 5 gerações (−), o erro de regime e o sobre-sinal já estão bem próximos de zero. Finalmente, após 50 gerações (−), o processo de sintonia já obteve uma boa solução. O melhor controlador apresenta uma resposta que pode ser considerada rápida, com erro de regime e sobre-sinal nulos.

No gráfico que ilustra o comportamento do termo $\sum u(k)^2$, nota-se que o gasto de energia do melhor controlador (−) é bastante inferior ao gasto de energia do melhor controlador da primeira geração (−).

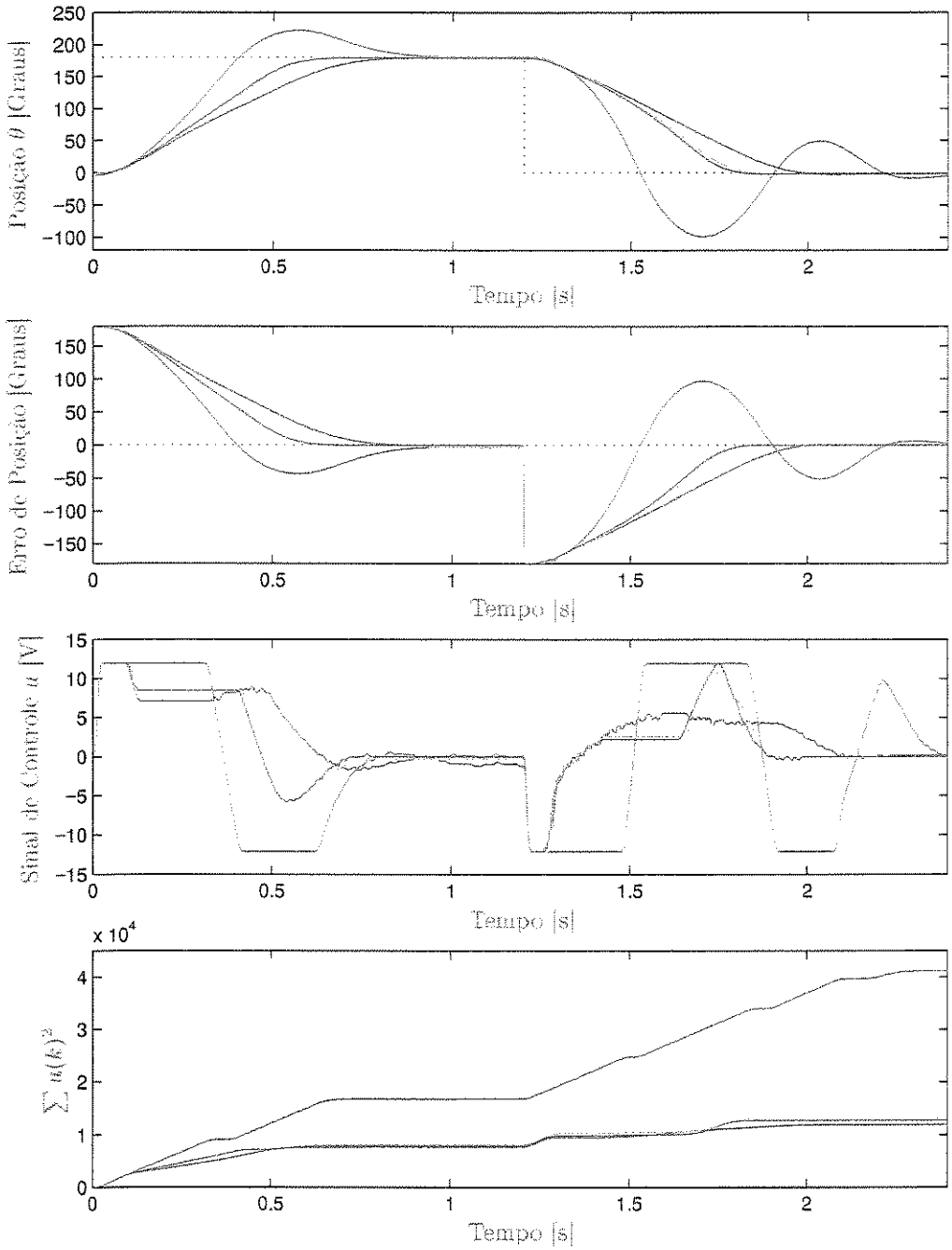


Figura 4.2: Resposta de posição do controlador nebuloso PD (9 regras): (−) $1a.$ geração, (−) $5a.$ geração, ( ) $20a.$ geração e (−) $50a.$ geração.

A figura 4.3 ilustra a evolução do *fitness* do melhor indivíduo através das gerações. Os parâmetros otimizados do melhor controlador obtido encontram-se descritos na tabela da figura 4.4. Para cada regra existe um controlador PD, e dependendo do ponto de operação (valores assumidos por e e Δe) estes controladores são considerados na produção da ação de controle. O sinal de controle resultante é uma soma ponderada dos sinais dos controladores PD das regras ativadas.

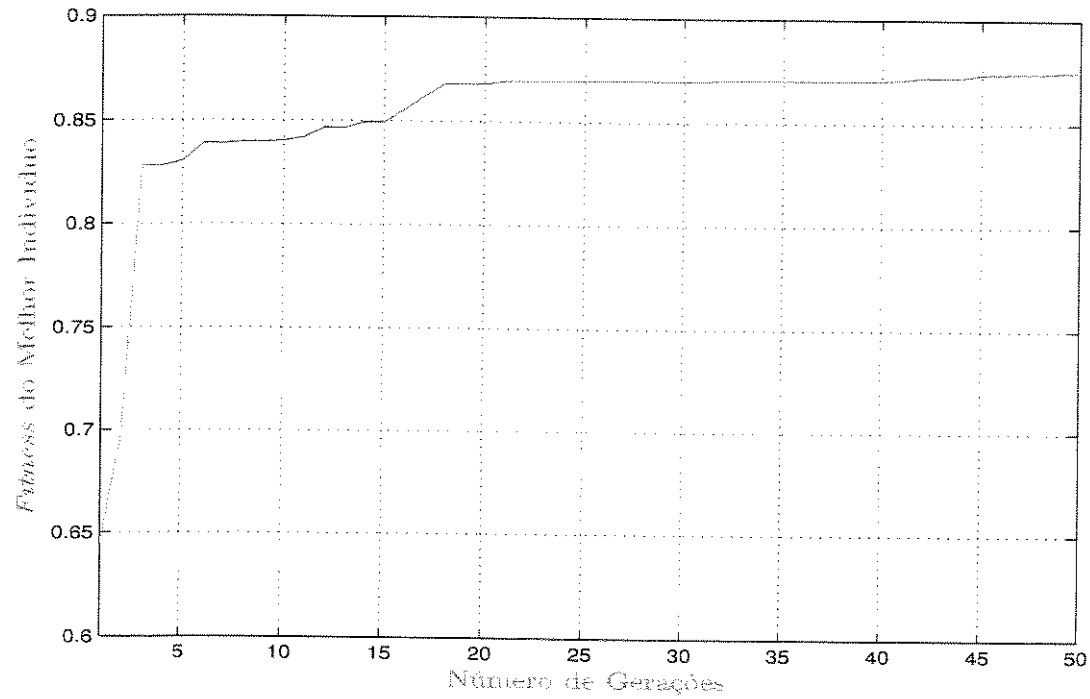


Figura 4.3: Evolução do *fitness*. PD nebuloso (9 regras).

A figura 4.5 apresenta a superfície de controle do melhor controlador obtido no processo de ajuste. Esta figura é gerada numericamente através do cálculo da saída do controlador para um conjunto de pontos definidos dentro do espaço das variáveis de entrada do controlador.

Nota-se que a superfície do controlador apresenta características não lineares, enquanto que os controladores da população inicial possuem a superfície plana de um controlador PD linear.

		e		
		A_1	A_2	A_3
Δe	B_1	$K_P = 15.0603$ $K_D = 2.4459$	$K_P = 27.7700$ $K_D = 3.5094$	$K_P = 13.7776$ $K_D = 3.2009$
	B_2	$K_P = 20.8544$ $K_D = 2.3251$	$K_P = 12.3723$ $K_D = 3.6684$	$K_P = 12.4638$ $K_D = 3.0345$
	B_3	$K_P = 21.8502$ $K_D = 2.6628$	$K_P = 29.7334$ $K_D = 2.7724$	$K_P = 23.3699$ $K_D = 2.9448$

Figura 4.4: Melhor controlador nebuloso PD (9 regras).

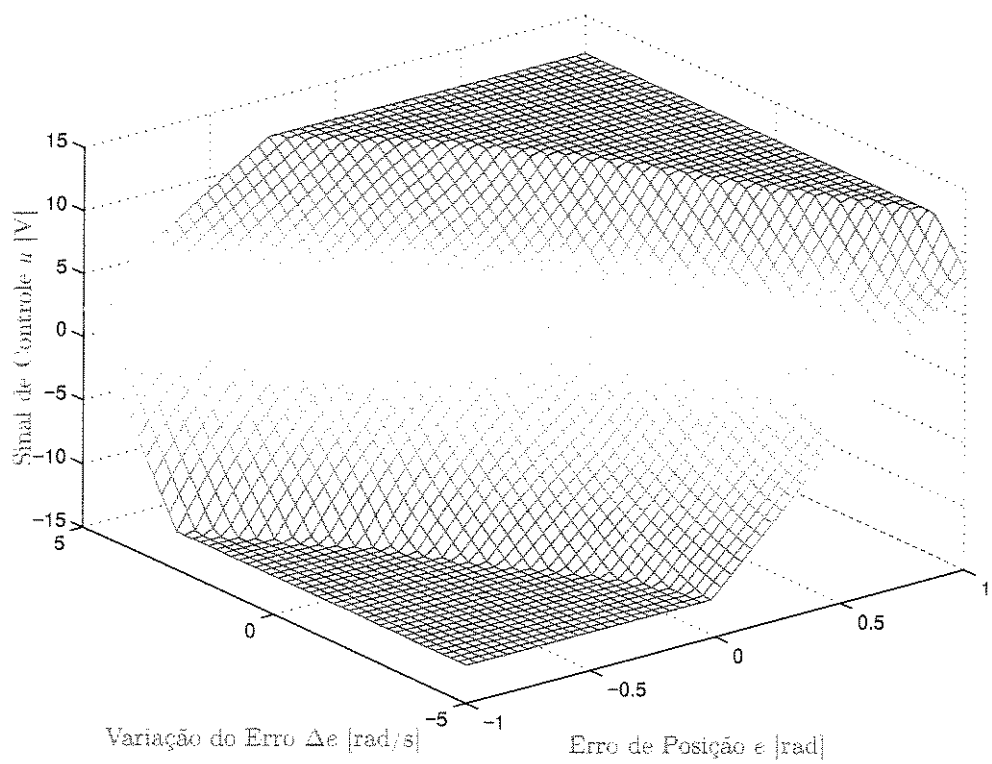


Figura 4.5: Superfície de controle do melhor controlador nebuloso PD (9 regras).

4.2.2 Controlador Nebuloso PD com 25 regras

Este exemplo é similar ao caso anterior (9 regras), entretanto, cada variável de entrada é particionada em 5 conjuntos nebulosos (ver figura 4.6), resultando em 25 regras. O controlador nebuloso PD de 25 regras possui maior flexibilidade se comparado ao de 9 regras, portanto, pode ter melhor desempenho relativo.

Os genes dos cromossomos que representam as variáveis a serem otimizadas são divididos em 2 grupos:

- Para $j \in \{1, \ldots, 25\}$, $min_j = 0,0$ e $max_j = 48,0$ (Genes que representam os ganhos K_P do controlador PD.
- Para $j \in \{26, \ldots, 50\}$, $min_j = 0,0$ e $max_j = 4,8$ (Genes que representam os ganhos K_D do controlador PD.

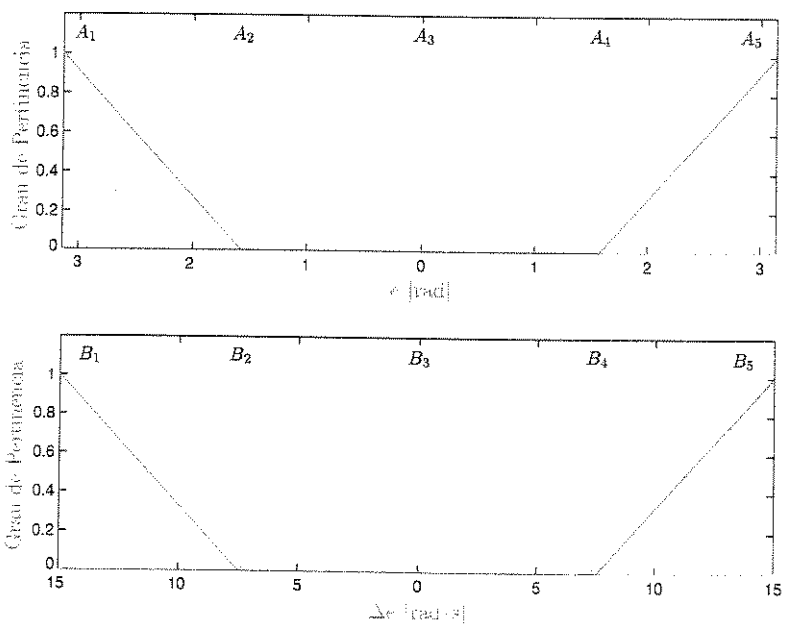


Figura 4.6: Funções de pertinência das variáveis de entrada. PD nebuloso (25 regras).

A população inicial foi gerada atribuindo-se para todos os cromossomos: $gene_j = 24,0$ para $j = 1, \ldots, 25$; $gene_j = 2,4$ para $j = 26, \ldots, 50$. Portanto, na primeira população, todos os controladores TS funcionam como controladores PD lineares com $K_P = 24,0$ e $K_D = 2,4$. A tarefa de controle utilizada para avaliar os controladores foi a mesma utilizada no caso de 9 regras. A figura 4.7 mostra as respostas de posição dos melhores controladores durante o processo de ajuste.

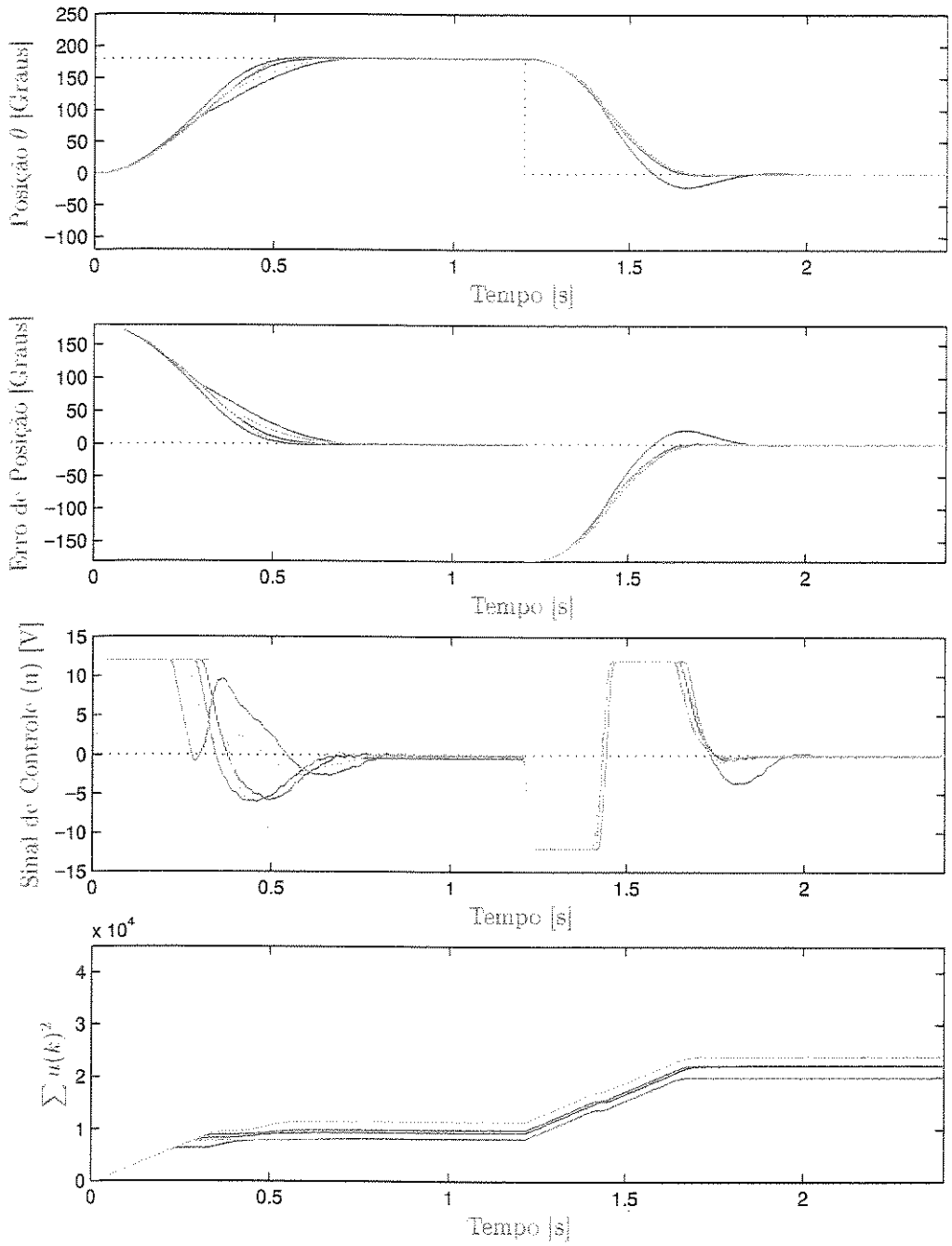


Figura 4.7: Resposta de posição do controlador nebuloso PD (25 regras): (−) $1a.$ geração, (−) $5a.$ geração, (−) $10a.$ geração, (−) $20a.$ geração e (−) $50a.$ geração.

Os resultados obtidos são semelhantes aos do caso com 9 regras, entretanto, como o controlador de 25 regras possui maior flexibilidade, a resposta de posição conseguida pelo melhor controlador obtido é mais rápida que a melhor resposta do controlador com 9 regras.

Observando a evolução de ambas as respectivas funções de *fitness*, figuras 4.3 e 4.8, nota-se que o *fitness* do melhor indivíduo da quinquagésima geração do caso com 25 regras é maior que o *fitness* do caso com 9 regras, comprovando a superioridade, ou a melhor adaptação, do controlador com 25 regras.

Os coeficientes dos conseqüentes das regras do melhor controlador obtido pelo processo de sintonia através do algoritmo genético encontram-se descritos na figura 4.9.

A superfície de controle do melhor controlador nebuloso com 25 regras é ilustrada na figura 4.10, ressaltando-se que esta apresenta não linearidades mais acentuadas que a do melhor controlador nebuloso com 9 regras.

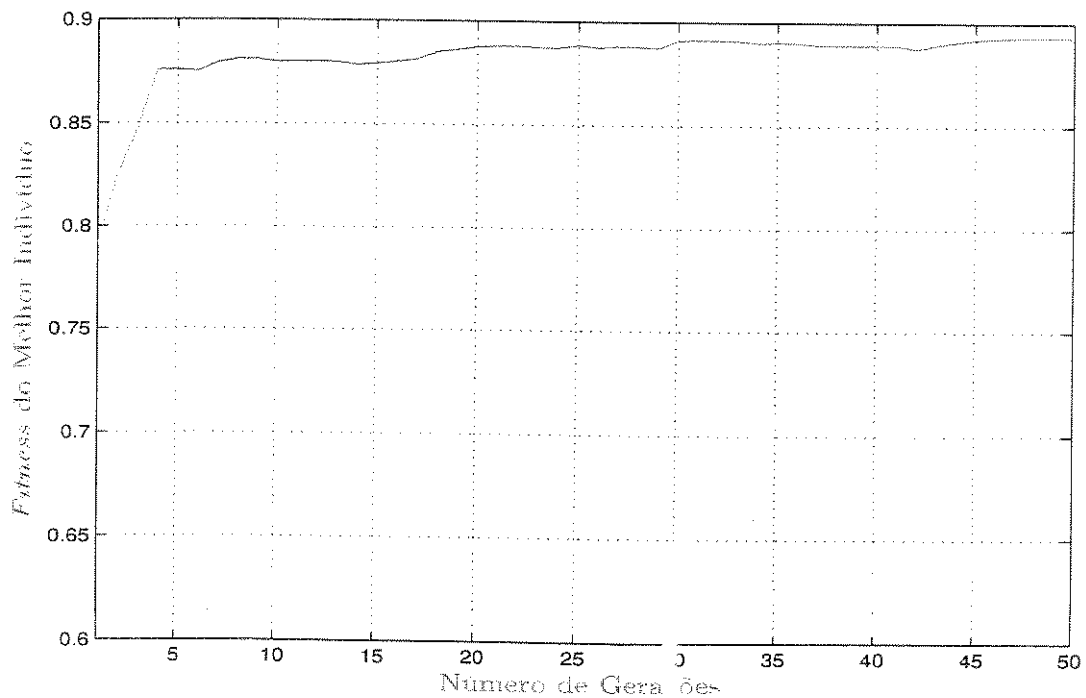


Figura 4.8: Evolução do *fitness*. *PD* n[illegible]uloso (25 regras).

		e				
		A_1	A_2	A_3	A_4	A_5
Δe	B_1	$K_P = 12.4482$ $K_D = 1.4862$	$K_P = 25.9749$ $K_D = 2.2905$	$K_P = 14.7479$ $K_D = 4.6759$	$K_P = 26.1744$ $K_D = 2.0284$	$K_P = 20.6393$ $K_D = 2.5652$
	B_2	$K_P = 23.7882$ $K_D = 2.5419$	$K_P = 24.4038$ $K_D = 0.7736$	$K_P = 11.8359$ $K_D = 2.6115$	$K_P = 19.5641$ $K_D = 3.1368$	$K_P = 2.4822$ $K_D = 4.0479$
	B_3	$K_P = 15.7508$ $K_D = 3.4097$	$K_P = 31.3956$ $K_D = 3.6846$	$K_P = 21.1333$ $K_D = 3.9601$	$K_P = 6.5005$ $K_D = 3.1054$	$K_P = 31.4285$ $K_D = 2.3702$
	B_4	$K_P = 31.1667$ $K_D = 2.9788$	$K_P = 20.9146$ $K_D = 1.6099$	$K_P = 30.9563$ $K_D = 2.1802$	$K_P = 12.9249$ $K_D = 2.7183$	$K_P = 6.9631$ $K_D = 0.6923$
	B_5	$K_P = 17.9820$ $K_D = 2.8615$	$K_P = 17.0232$ $K_D = 0.7944$	$K_P = 22.8190$ $K_D = 2.6435$	$K_P = 24.1288$ $K_D = 2.9057$	$K_P = 18.5528$ $K_D = 1.4593$

Figura 4.9: Melhor controlador nebuloso PD (25 regras).

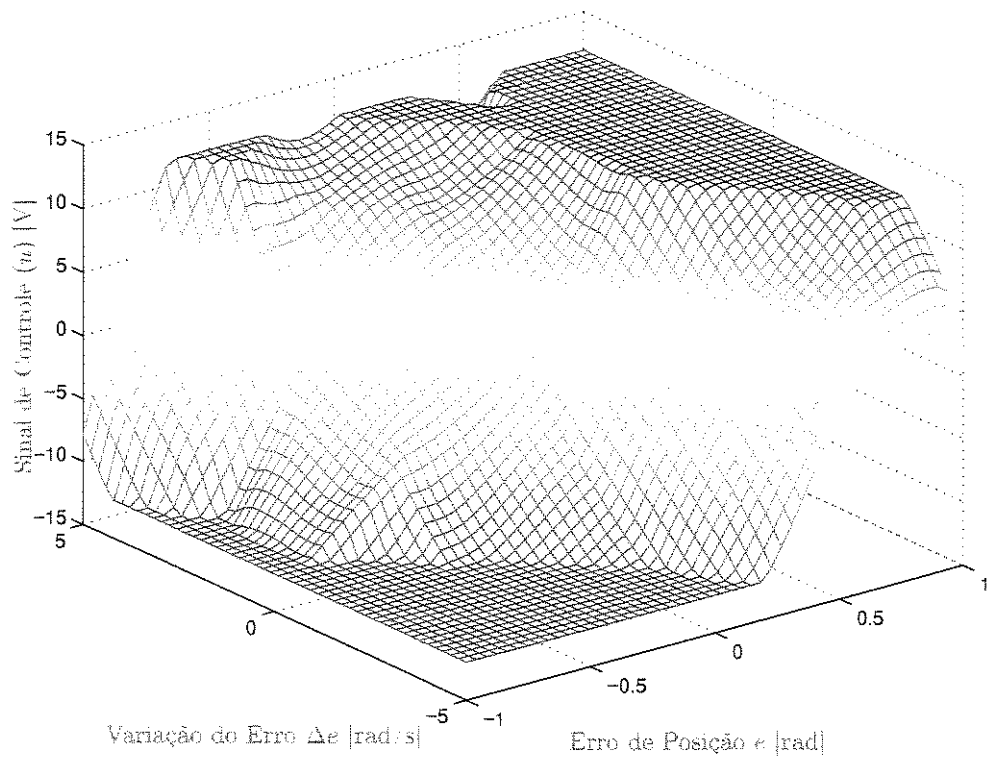


Figura 4.10: Superfície de controle do melhor controlador nebuloso PD (25 regras).

4.2.3 Comparação entre o Controlador Nebuloso *PD* e o Controlador Clássico *PD*

Como meio de avaliação das qualidades das respostas dos controladores nebulosos *PD* obtidos, utilizou-se como referêncial controladores clássicos *PD* submetidos às mesmas tarefas de controle. O ajuste do controlador *PD* foi realizado pelo mesmo algoritmo genético empregado no ajuste dos controladores nebulosos. A figura 4.11 mostra a resposta de posição do melhor controlador nebuloso *PD* com 25 regras (−) e a resposta de posição do melhor controlador clássico *PD* (−). A resposta de posição do controlador nebuloso *PD* mostrou-se bastante superior em relação ao controlador clássico *PD* para a primeira posição de referência (intervalo de 0,0 a 1,2 segundos). Nota-se que o controlador clássico *PD*, por não possuir flexibilidade suficiente, não consegue garantir uma resposta rápida do servomecanismo no rastreamento da primeira posição de referência sem prejudicar a resposta no rastreamento da posição imposta como segunda referência (intervalo de 1,2 a 2,5 segundos).

Como pode-se observar na figura 4.11, o controlador nebuloso conduz praticamente à mesma resposta de posição para posições de referência que correspondem aos pontos de equilíbrio do sistema. Para a primeira referência (180^o - ponto de equilíbrio instável), a gravidade age em sentido contrário ao torque imposto pelo motor que aciona o pêndulo, enquanto que no movimento realizado para o rastreamento da segunda referência (0^o - ponto de equilíbrio estável), a gravidade e o torque do motor atuam no mesmo sentido. Como o controlador clássico *PD* apresenta resposta linear independente do movimento realizado pelo pêndulo, nota-se claramente que ele leva a uma reposta de posição relativamente mais lenta no rastreio da primeira referência em relação ao tempo de rastreio da segunda.

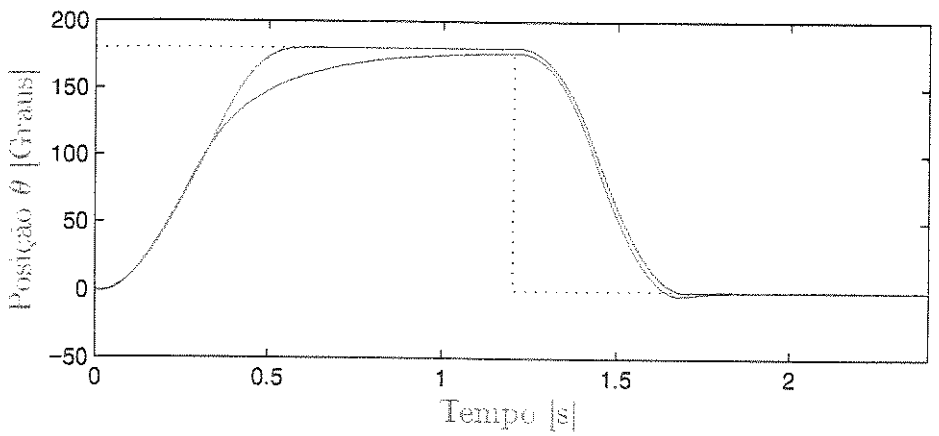


Figura 4.11: Comparação entre o controlador *PD* (−) e o controlador nebulosos *PD* (−).

4.3 Controlador Nebuloso PID

O controlador nebuloso PID possui três variáveis de entrada: erro (e), variação do erro (Δe) e integral do erro (δe). Para cada entrada foram definidas três funções de pertinência triangulares, conforme ilustrado na figura 4.12.

- Para $j \in \{1, \ldots, 27\}$, $min_j = 0,0$ e $max_j = 72,0$ (Genes que representam os ganhos K_P do controlador PD).
- Para $j \in \{28, \ldots, 54\}$, $min_j = 0,0$ e $max_j = 7,2$ (Genes que representam os ganhos K_D do controlador PD).
- Para $j \in \{55, \ldots, 81\}$, $min_j = 0,0$ e $max_j = 4,8$ (Genes que representam os ganhos K_I do controlador PD).

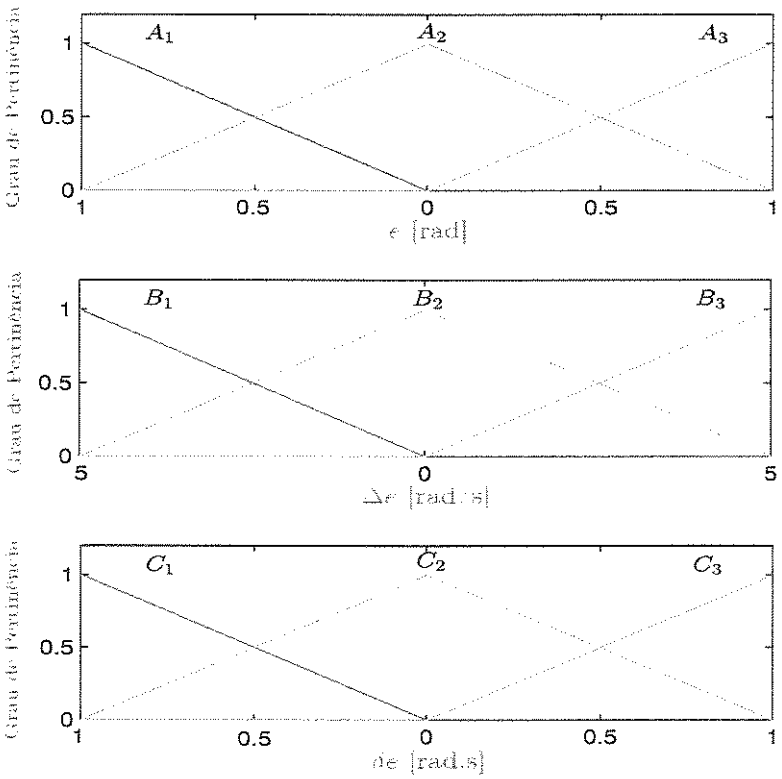


Figura 4.12: Funções de pertinência das variáveis de entrada. PID nebuloso (27 regras).

Portanto, o número total de regras é 27, resultando em 81 parâmetros a serem ajustados. Cada cromossomo do AG para este problema possui 81 genes, que estão subdivididos em 3 grupos:

A população inicial é gerada atribuindo-se a todos os cromossomos: $gene_j = 36.0$ para $j = 1, \ldots, 27$; $gene_j = 3,6$ para $j = 28, \ldots, 54$; $gene_j = 2,4$ para $j = 55, \ldots, 81$. Na população inicial todos os controladores nebulosos PID do AG, funcionam como controladores lineares PID com ganhos $K_P = 36,0$; $K_D = 3,6$ e $K_I = 2,4$.

A tarefa de controle utilizada na avaliação dos controladores foi composta pelas seguintes posições de referência: $\theta_{REF} = 60°$ para $t \in [0,0\,;\, 0,8)$ s e $\theta_{REF} = -90°$ para $t \in [0,8\,;\, 1,6]$ s, as quais não são pontos de equilíbrio naturais do sistema. Isto significa que o controlador necessariamente deve aplicar um nível de tensão diferente de zero para manter o pêndulo em regime nas posições desejadas.

Os controladores clássicos PD e sua versão nebulosa não são capazes de resolver eficazmente o problema de posicionamento do pêndulo para posições que não sejam pontos de equilíbrio. Os controladores clássicos PID possuem um termo que é proporcional à integral do erro, o que o torna apto a conseguir erro de regime nulo para qualquer posição de referência dentro do espaço de trabalho do pêndulo, sendo que os controladores nebulosos PID, por possuírem maior flexibilidade relativa, têm potencial para superarem o desempenho de tais controladores clássicos.

A figura 4.13 mostra as respostas obtidas pelos melhores controladores nebulosos PID durante o processo de sintonia.

Na primeira geração, a resposta de posição do melhor controlador (—) apresenta alto sobresinal (31,82%) e erro de regime para a primeira posição de referência rastreada, e um sobresinal médio (9,48%) para a segunda referência. Na quinta geração, a resposta de posição (—) apresenta redução nos sobresinais para ambas as posições de referência, e uma grande redução no erro de regime relativo à primeira delas. Na décima geração, a resposta de posição ( ) do melhor controlador ainda apresenta um pequeno sobresinal (5,42%) em relação à primeira referência, enquanto que para a segunda, o sobresinal é praticamente nulo.

Na vigésima geração, a resposta de posição (—) do melhor controlador não apresenta sobresinais nem erros de regime, e consegue uma melhora com respeito ao tempo de estabilização no rastreamento da segunda referência. Na quinquagésima geração, a resposta de posição ( ) apresenta ainda uma pequena redução no tempo de estabilização para o rastreamento da segunda referência.

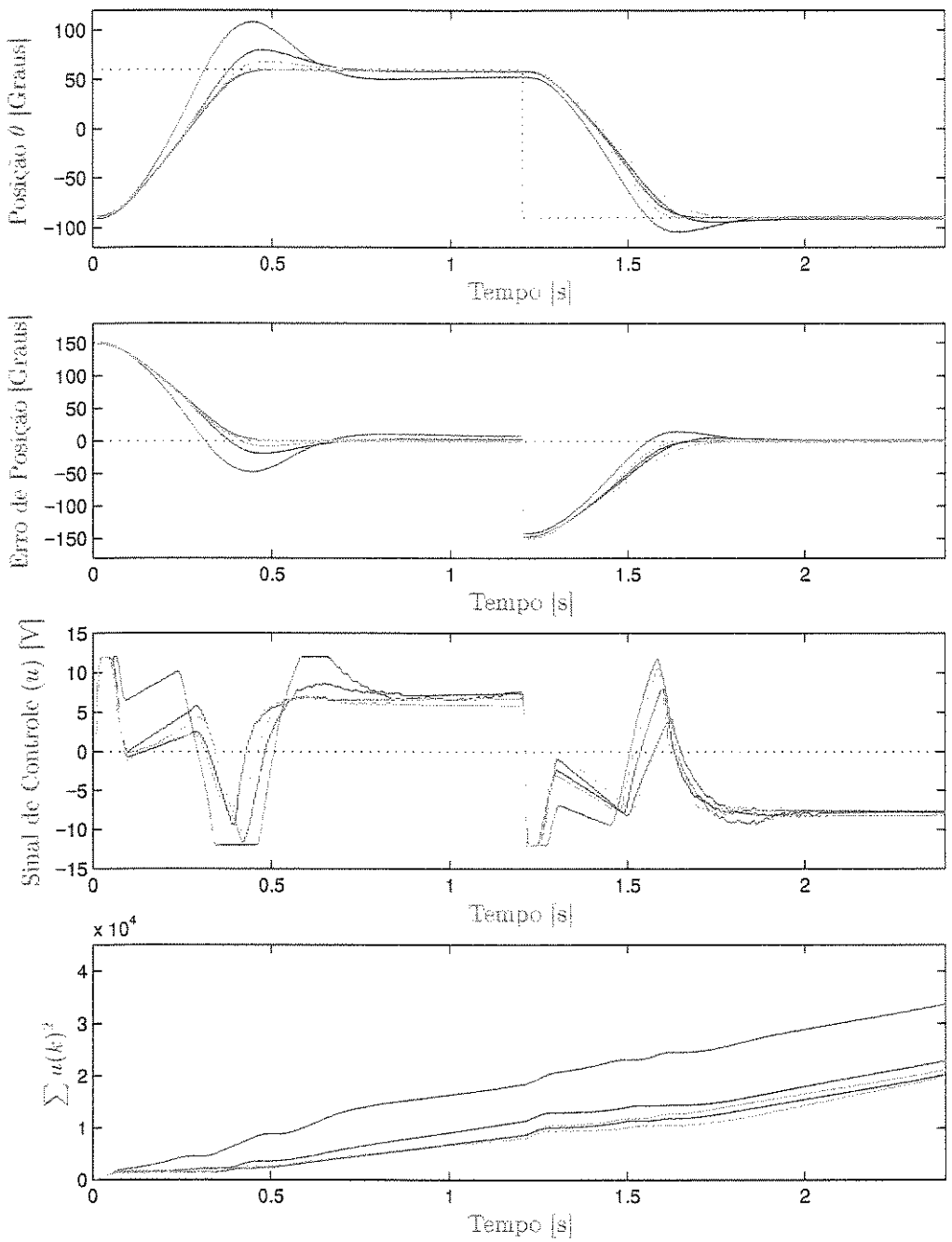


Figura 4.13: Resposta de posição do controlador nebuloso *PID* (27 regras): (–) 1a. geração, (–) 5a. geração, (··) 10a. geração, (–) 20a. geração e (··) 50a. geração.

As tabelas da figura 4.14 apresentam os parâmetros do melhor controlador nebuloso *PID* resultante do processo de otimização. Note que cada uma das três tabelas refere-se à variável lingüística do erro de posição para cada conjunto nebuloso definido (A_1, A_2 e A_3).

		Δe		
		B_1	B_2	B_3
δe	C_1	$K_P = 20.8527$ $K_D = 2.8630$ $K_I = 23.5059$	$K_P = 24.6852$ $K_D = 1.7200$ $K_I = 22.0380$	$K_P = 22.1386$ $K_D = 3.7739$ $K_I = 22.0006$
	C_2	$K_P = 15.2765$ $K_D = 3.2170$ $K_I = 37.5149$	$K_P = 19.2927$ $K_D = 2.4422$ $K_I = 38.0595$	$K_P = 11.0015$ $K_D = 2.2134$ $K_I = 34.1279$
	C_3	$K_P = 25.3700$ $K_D = 2.7753$ $K_I = 18.3624$	$K_P = 12.1026$ $K_D = 2.2836$ $K_I = 16.5421$	$K_P = 29.8221$ $K_D = 3.3361$ $K_I = 28.2987$

(a) $e = A_1$

		Δe		
		B_1	B_2	B_3
δe	C_1	$K_P = 26.0980$ $K_D = 4.0417$ $K_I = 8.3292$	$K_P = 35.7586$ $K_D = 4.8714$ $K_I = 25.6931$	$K_P = 24.5034$ $K_D = 3.6485$ $K_I = 24.3205$
	C_2	$K_P = 32.1291$ $K_D = 2.8346$ $K_I = 25.2749$	$K_P = 30.2901$ $K_D = 2.4998$ $K_I = 17.8490$	$K_P = 23.4450$ $K_D = 2.7057$ $K_I = 19.7915$
	C_3	$K_P = 11.5036$ $K_D = 2.9978$ $K_I = 24.9102$	$K_P = 13.5115$ $K_D = 4.9989$ $K_I = 13.9231$	$K_P = 31.6637$ $K_D = 2.3896$ $K_I = 25.8179$

(b) $e = A_2$

		Δe		
		B_1	B_2	B_3
δe	C_1	$K_P = 14.5987$ $K_D = 3.3360$ $K_I = 21.0651$	$K_P = 26.3800$ $K_D = 5.5314$ $K_I = 13.4832$	$K_P = 22.2337$ $K_D = 4.8398$ $K_I = 16.7232$
	C_2	$K_P = 27.8310$ $K_D = 4.3591$ $K_I = 28.5087$	$K_P = 16.2577$ $K_D = 2.9275$ $K_I = 28.3034$	$K_P = 17.0339$ $K_D = 4.0922$ $K_I = 20.9446$
	C_3	$K_P = 37.6941$ $K_D = 2.2429$ $K_I = 17.3180$	$K_P = 27.3924$ $K_D = 1.6448$ $K_I = 26.9600$	$K_P = 23.3688$ $K_D = 3.2734$ $K_I = 16.7536$

(c) $e = A_3$

Figura 4.14: Melhor controlador nebuloso PID (27 regras).

A figura 4.15 apresenta a evolução do *fitness* considerando-se os melhores indivíduos de cada geração durante o processo de sintonia. Nota-se que a partir da vigésima geração, o desempenho do controlador apresenta melhoras pouco significativas.

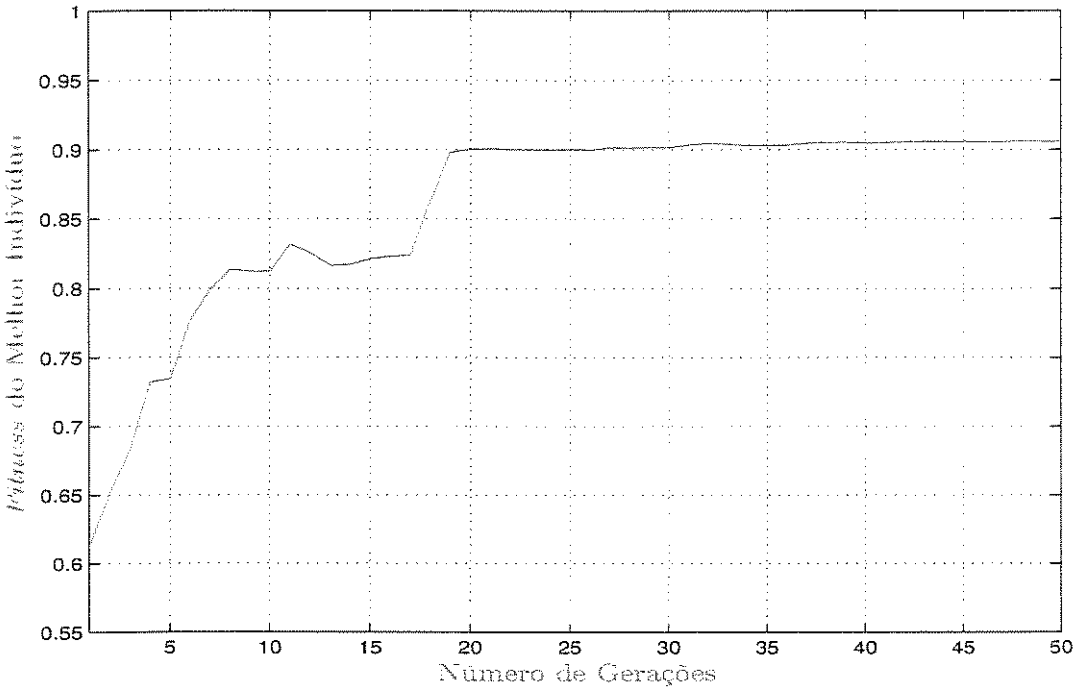


Figura 4.15: Evolução do *fitness*. Controlador nebuloso *PID*.

4.3.1 Comparação entre o Controlador Nebuloso *PID* e o Controlador Clássico *PID*

A figura 4.16 apresenta as respostas de posição do melhor controlador nebulosos *PID* com 27 regras (−) e melhor controlador clássico *PID* (−) para a tarefa de controle apresentada na seção anterior. O melhor controlador clássico *PID* foi obtido a partir do mesmo algoritmo genético utilizado para otimizar os controladores nebulosos.

Na figura 4.16, observa-se que ambos os controladores foram capazes de resolver o problema de posicionamento abordado, ou seja, as posições de referência foram alcançadas com erro de regime e sobresinais nulos num tempo relativamente curto, quando comparado com à constante de tempo natural do sistema.

Nota-se também, que o controlador nebuloso *PID* apresentou uma resposta muito

mais rápida no rastreamento da segunda posição de referência, houve uma redução de aproximadamente 0,4 segundos em relação ao controlador clássico.

Outra característica interessante apresentada pelo controlador nebuloso *PID* é que os tempos de estabilização são praticamente iguais em ambos os transitórios existentes na tarefa de controle, considerando que as distâncias percorridas nas transições são iguais, mas o comportamento dinâmico na vizinhança das posições de referência são diferentes.

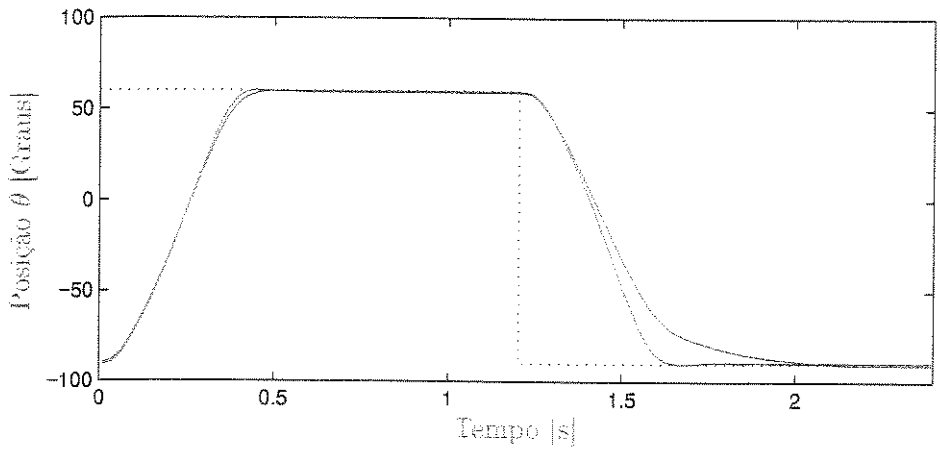


Figura 4.16: Comparação entre o controlador *PID* (−) e o controlador nebuloso *PID* (−).

4.4 Controlador Nebuloso *PD* Com Compensação de Gravidade

Como comentado anteriormente, o controlador nebuloso *PD* não é capaz de gerar uma resposta de posição com erro de regime nulo para posições de referência que não sejam pontos de equilíbrio do sistema. Para contornar este problema, é proposto um controlador nebuloso baseado na estratégia de controle *PD* com compensação de gravidade (Kelly 1997).

O controlador *PD* com compensação de gravidade para manipuladores robóticos foi introduzido por Takegaki & Arimoto (1981), com o propósito de resolver o problema de posicionamento global de manipuladores robóticos, ela fornece uma lei de controle que é capaz de governar os movimentos de um robô com cadeia cinemática serial rígida de tal maneira que as juntas são posicionadas assintoticamente nas posições desejadas independentemente das posições iniciais e velocidades.

A principal vantagem deste controlador está na sua estrutura simples. O sinal de controle é obtido através de realimentação com lei de controle PD linear, e de um *offset* com ação antecipativa que é função da posição de referência desejada (ver figura 4.17).

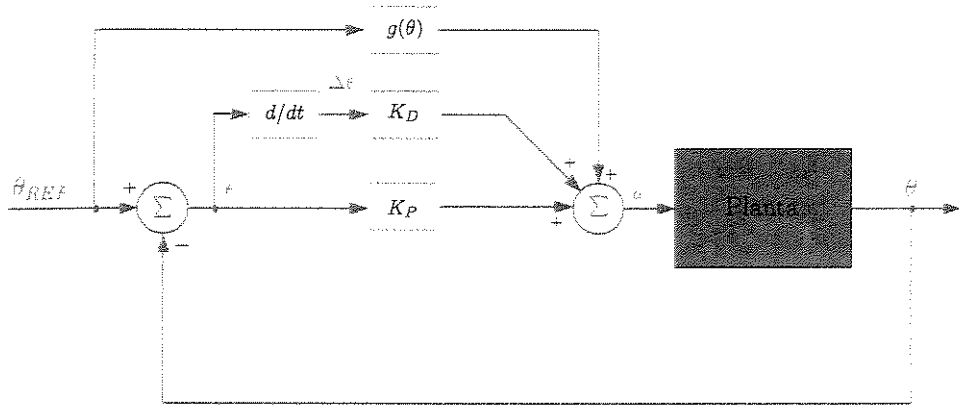


Figura 4.17: Estrutura do controlador PD com compensação de gravidade.

O controlador PD com compensação de gravidade, para o caso de uma junta robótica, possui a seguinte lei de controle:

$$u = K_P \cdot e + K_D \cdot \Delta e + g(\theta) \tag{4.1}$$

onde e é o erro de posição, Δe é a variação do erro de posição e $g(\theta)$ representa a ação de controle que atua de forma a compensar o torque causado pela gravidade. O termo $g(\theta)$ é calculado com base nas equações dinâmicas do sistema. Nesta concepção, deve-se conhecer o modelo matemático do sistema e seus parâmetros físicos.

A versão nebulosa da estratégia de controle PD com compensação de gravidade é composta por duas partes (ver figura 4.18): um controlador nebuloso PD e um aproximador de funções baseado no sistema nebuloso de *Takagi-Sugeno*.

O processo de sintonia do controlador nebuloso com compensação de gravidade envolve duas etapas:

(1) Primeiramente, deve-se realizar o ajuste do aproximador de funções que irá compensar o torque gravitacional. Para isso, utiliza-se dados experimentais do tipo $(\theta\,, g(\theta))$, obtidos de uma tarefa de controle. A partir destes dados, aplica-se o algoritmo genético proposto no aproximador de funções, de maneira a aproximar uma função que melhor se ajuste a tais dados experimentais.

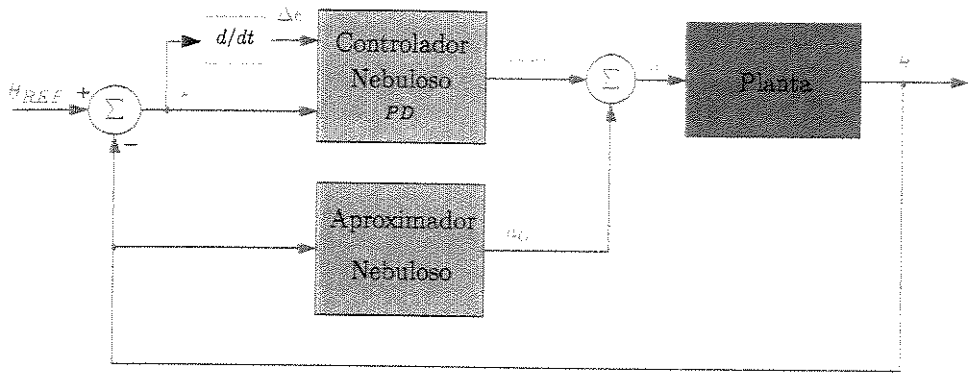


Figura 4.18: Estrutura do controlador nebuloso PD com compensação de gravidade.

(2) Após a sintonia do aproximador de funções, utilizando a estrutura da figura 4.18, aplica-se o algoritmo genético para ajustar o controlador nebuloso PD. Esta etapa de ajuste é similar à utilizada na seção 4.2.

O controlador nebuloso PD utilizado nos experimentos possui 9 regras e é similar ao controlador apresentado na seção 4.2.1. O aproximador de funções nebuloso utilizado nos experimentos possui como variável de entrada a posição angular do pêndulo acionado. Foram definidas 9 funções de pertinência triangulares para o universo de discurso da variável de entrada (ver figura 4.19), resultando um total de 9 regras de *Takagi-Sugeno*.

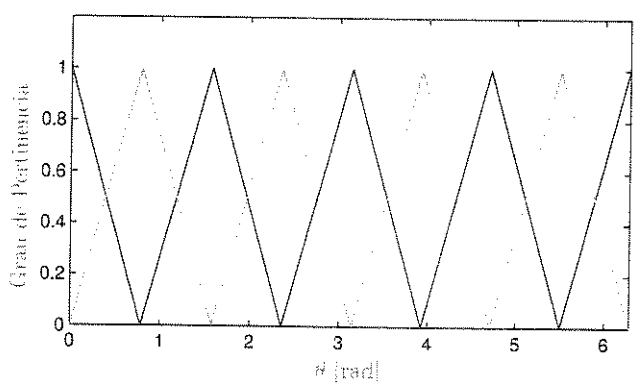


Figura 4.19: Funções de pertinência das variáveis de entrada. PD nebuloso (9 regras).

As regras de *Takagi-Sugeno* para este aproximador de funções nebuloso foram definidas da seguinte forma:

$$R^{(l)}\text{: } \underline{\textbf{SE}}\ \theta \text{ é } A^l\ \underline{\textbf{ENTÃO}}\ u_G^l = c^l + k^l \cdot \theta.$$

4.4.1 Sintonia do Aproximador Nebuloso

No processo de sintonia deste sistema nebuloso, as funções de pertinência do antecedente são fixas, restando os coeficientes das regras como variáveis de busca. Como para cada regra existem dois parâmetros livres (c^l e k^l), o número total de parâmetros ajustáveis é 18.

Considerando que a variável de entrada θ pode assumir valores no intervalo $[0,0\,;\,2\pi]$ e que o sinal de controle u_G deve assumir valores no intervalo de $[-12,0\,;\,12,0]\ V$, os genes dos cromossomos que representam as variáveis a serem otimizadas foram divididos em 2 classes:

- Para $j \in \{1,\ldots,9\}$, $min_j = -12,0$ e $max_j = 12,0$ (Genes que representam os coeficientes (c^l) dos consequentes das regras de *Takagi-Sugeno*).

- Para $j \in \{9,\ldots,18\}$, $min_j = -6,0/\pi$ e $max_j = 6,0/\pi$ (Genes representam os coeficientes (k^l) dos consequentes das regras de *Takagi-Sugeno*).

Os dados experimentais utilizados nesta etapa do processo de otimização foram gerados a partir de uma tarefa de controle onde várias posições de referência são geradas de forma a cobrir todo o espaço de trabalho.

Um controlador clássico PD é utilizado para levar o pêndulo à posição desejada. Após o sistema entrar em regime, ou seja, ficar parado, faz-se a aquisição da posição (θ) e da tensão necessária ($g(\theta)$) para mantê-lo parado. Este processo é repetido para todas as posições de referência.

Durante os experimentos, notou-se que a tensão de compensação do torque gravitacional ($g(\theta)$) assumia valores diferentes dependendo do sentido de rotação no qual o pêndulo girava. Resolveu-se então coletar os dados considerando ambos os sentidos de rotação. Os dados utilizados no treinamento ($\circ$) do sistema nebuloso podem ser visualizados na figura 4.20.

A função de *fitness* utilizada nesta etapa do treinamento é baseada no erro quadrático médio (EQM). O EQM é calculado a partir da seguinte equação:

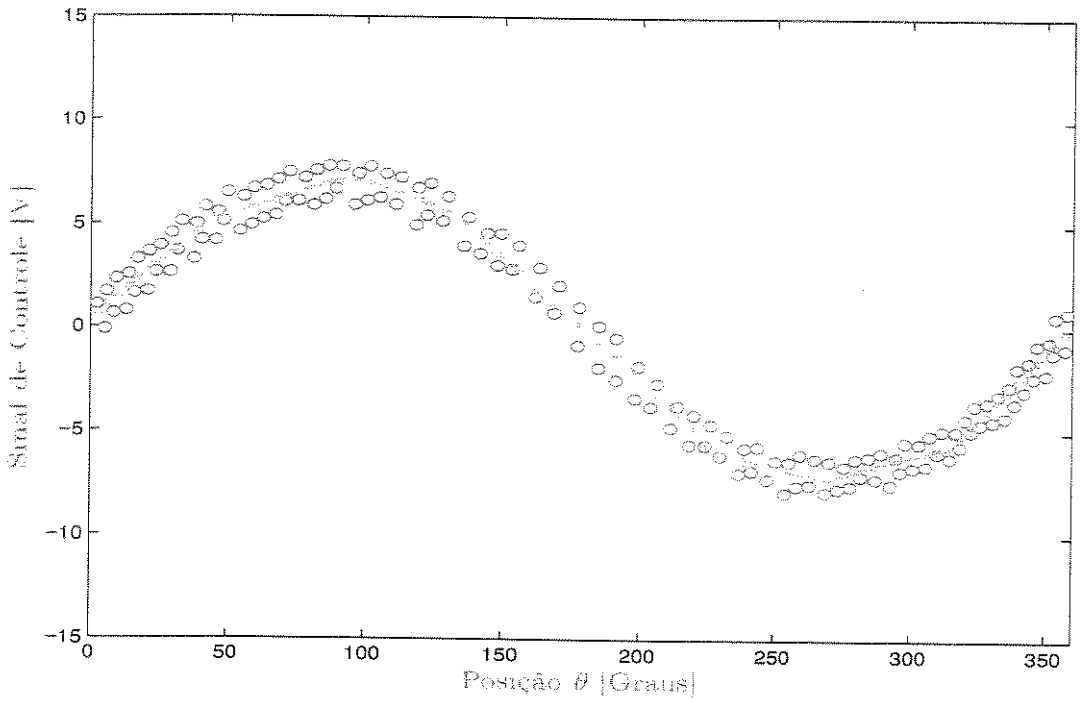


Figura 4.20: Saída do sistema nebuloso ( ), dados de treinamento (○).

$$EQM = \sqrt{\frac{\sum_{p=1}^{N}(u_G(\theta_p) - g(\theta_p))^2}{N}} \tag{4.2}$$

onde $g(\cdot)$ é a função a ser aproximada, $u_G(\cdot)$ é função aproximada pelo sistema *Takagi-Sugeno*, θ_p é a $p-$ésima amostra de treinamento e N é o número total de amostras, neste caso 144 amostras.

A função de *fitness* é definida como:

$$F = \frac{1}{EQM} \, . \tag{4.3}$$

Após executar o algoritmo genético durante 500 gerações obteve-se os resultados mostrados na figura 4.20, sendo que os dados simbolizados com ( ) representam a aproximação feita pelo sistema nebuloso. Apesar dos dados de treinamento (○) possuírem valores ambíguos, nota-se que o sistema nebuloso realizou uma excelente aproximação através de sua capacidade de fazer interpolações.

A figura 4.21 ilustra a evolução do erro quadrático médio durante o processo de treinamento. Observa-se que o processo de evolução poderia ter sido interrompido na centésima geração, já tendo obtido uma solução bastante satisfatória.

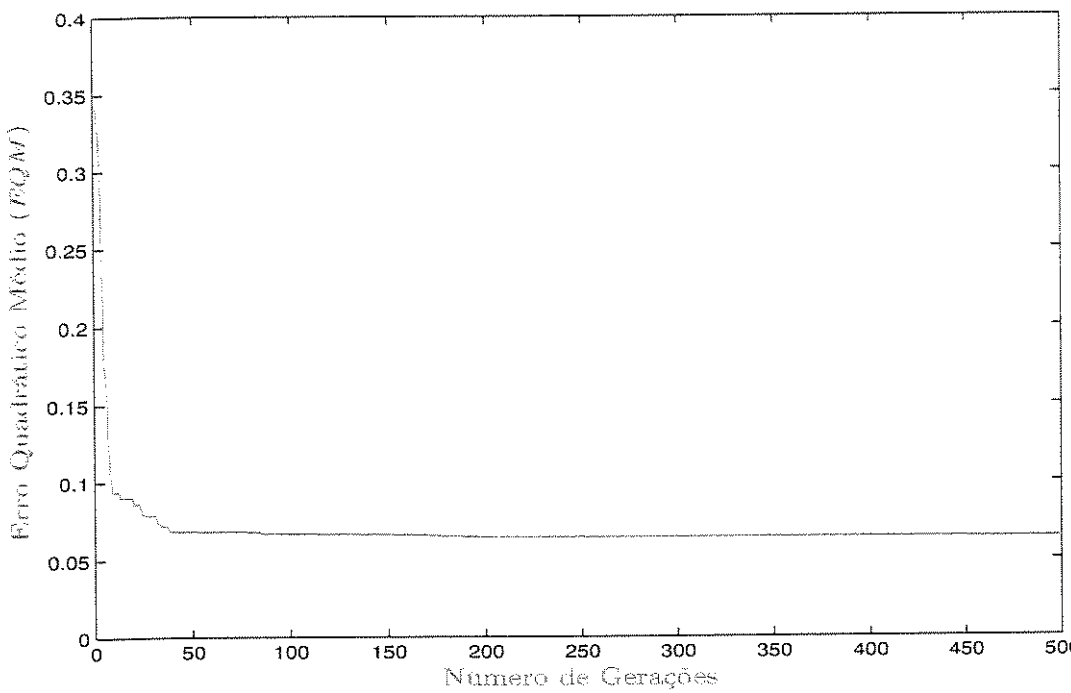


Figura 4.21: Evolução do EQM.

4.4.2 Sintonia do Controlador Nebuloso PD

Este controlador possui 18 parâmetros a serem ajustados (ver seção 4.2.1). Os genes dos cromossomos que representam as variáveis a serem otimizadas estão divididos em 2 grupos:

- Para $j \in \{1, \ldots, 9\}$, $min_j = 0,0$ e $max_j = 48,0$ (Genes que representam os ganhos K_P do controlador PD).

- Para $j \in \{10, \ldots, 18\}$, $min_j = 0,0$ e $max_j = 4,8$ (Genes que representam os ganhos K_D do controlador PD).

A população inicial foi gerada atribuindo-se a todos os cromossomos: $gene_j = 24,0$ para $j = 1,\ldots,9$; $gene_j = 2,4$ para $j = 10,\ldots,18$. Ou seja, na primeira geração, todos os controladores TS funcionam como controladores PD lineares com $K_P = 24,0$ e $K_D = 2,4$.

A referência de posição a ser seguida é composta por quatro partes: $\theta_{REF} = 60°$ para $t \in [0,0\,;\ 1,0)\ s$, $\theta_{REF} = -90°$ para $t \in [1,0\,;\ 2,0)\ s$, $\theta_{REF} = 30°$ para $t \in [2.0\,;\ 3,0)\ s$, e $\theta_{REF} = 0°$ para $t \in [3,0\,;\ 4,0]\ s$.

Em comparação com as outras tarefas de controle utilizadas para avaliação dos controladores, esta tarefa contém uma seqüência de transições que enrriquecem a análise do controlador, por exemplo, as posições de referência cobrem boa parte do espaço de trabalho do pêndulo, possibilitando a avaliação do controlador para a realização de grandes (de $\theta_{REF} = 60°$ para $\theta_{REF} = -90°$) e pequenas (de $\theta_{REF} = 30°$ para $\theta_{REF} = 0°$) excursões de movimento.

A figura 4.22 mostra as respostas dos melhores controladores durante o processo de sintonia. As curvas pontilhadas indicam a referência a ser alcançada.

Nesta etapa do treinamento, o aproximador nebuloso já está atuando na compensação do torque gravitacional, conseqüentemente, o erro de regime para cada referência é aproximadamente nulo desde a primeira geração. Este comportamento não seria possível caso apenas controladores nebulosos PD dos tipos apresentados no item 4.2 estivessem atuando.

Na figura 4.22, percebe-se que as medidas de desempenho como sobresinal, erro de regime e tempo de estabilização são otimizadas durante o processo de evolução do algoritmo genético.

Note que na última geração, o melhor controlador obtido apresenta uma resposta de posição rápida e suave com erros de regime e sobresinais nulos.

A figura 4.23 ilustra a evolução do *fitness* do melhor controlador nebuloso PD a cada geração, durante o processo de sintonia.

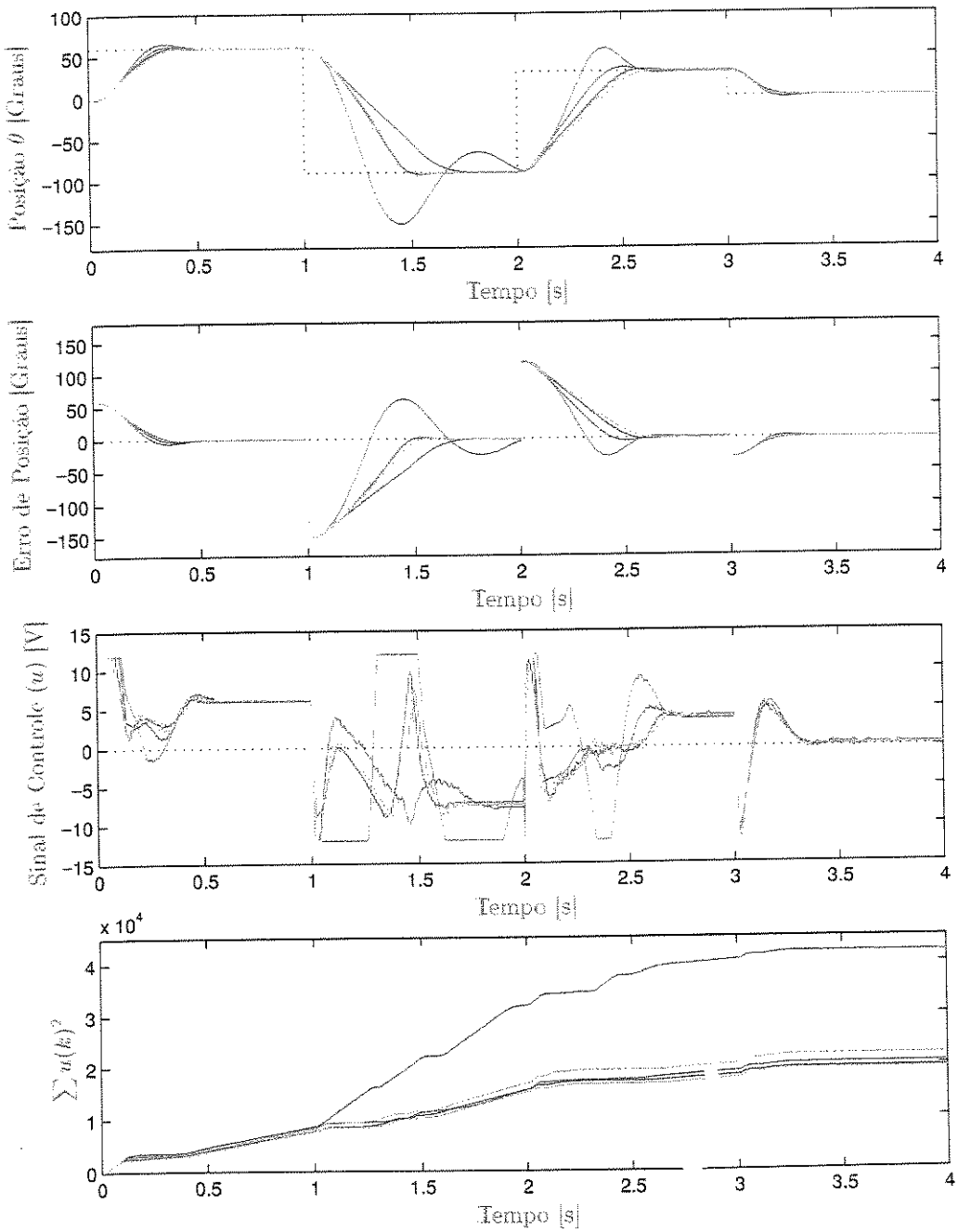


Figura 4.22: Resposta de posição do controlador nebuloso PD com compensação de gravidade: (−) 1a. geração, (−) 5a. geração, (−) 10a. geração, (−) 20a. geração e (−) 50a. geração.

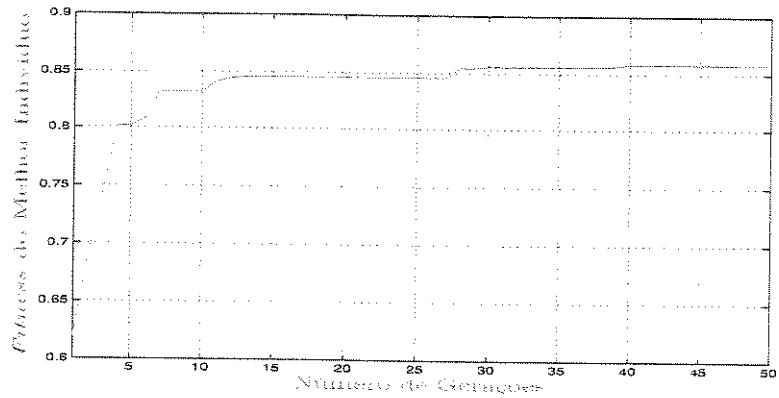


Figura 4.23: Evolução do *fitness*. PD nebuloso com compensação de gravidade.

A superfície de controle do melhor controlador nebuloso PD da última geração é ilustrada na figura 4.24, ressaltando-se que esta superfície trata apenas de uma parte do controlador nebuloso PD com compensação de gravidade.

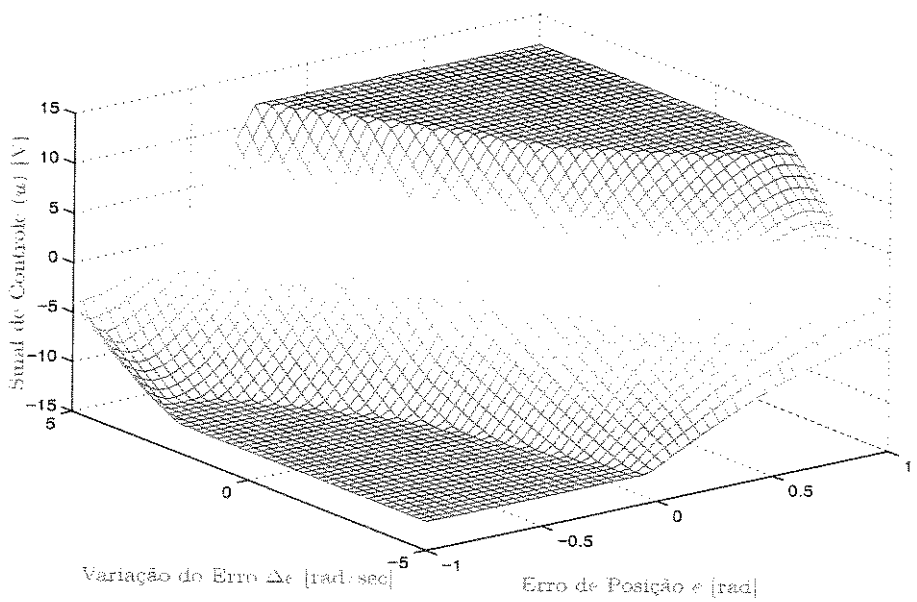


Figura 4.24: Superfície de controle do melhor controlador nebuloso PD (9 regras).

Capítulo 5

Conclusões e Direções Futuras

Nesta tese, foi abordado o problema de implementação prática de controladores nebulosos com ajuste paramétrico por algoritmos genéticos. Este estudo resultou em contribuições importantes para a síntese de controle não linear, destacando-se três pontos principais: desenvolvimento de metodologia experimental baseada em algoritmos genéticos conjugada à teoria de conjuntos nebulosos para busca numérica de controladores eficientes no sentido de otimizarem funcionais de desempenho multi-objetivo; implementação digital usando microcomputadores de baixo custo, ou seja, com boa relação custo/benefício; e construção de um exemplo que valida a aplicação prática desta abordagem. Em particular, foi criado um procedimento computacional para o projeto de controladores de servomecanismos do tipo usado em juntas robóticas, e que é extensível a uma infinidade de outros sistemas não lineares com características semelhantes existentes nos ambientes industriais.

A abordagem proposta, utilizando algoritmos genéticos para o ajuste paramétrico do controlador nebuloso de *Takagi-Sugeno*, possibilitou a obtenção de excelentes resultados experimentais, que representam um grande passo na área de algoritmos genéticos aplicados a controle, já que a grande maioria dos resultados publicados pertencem ao campo da simulação.

Os controladores nebulosos de *Takagi-Sugeno* obtidos do processo de otimização apresentaram em todos os casos experimentados uma resposta de posição rápida com erro de regime e sobresinal aproximadamente nulos, considerando-se as constantes de tempo do servomecanismo usado para prova experimental. Isto significa, que todos os objetivos incorporados na função de *fitness* foram atingidos, comprovando a adequação da função multi-objetivo proposta.

Apesar da função de *fitness* não englobar medidas de gasto de energia, observou-se que os controladores nebulosos otimizados apresentavam gastos de energia inferior aos dos controladores não otimizados. Isto deve-se a um dos objetivos que é minimizar o sobresinal, provocando indiretamente uma economia de energia.

Em todos os casos de comparação analisados, os controladores nebulosos PD e PID apresentaram desempenho superior aos dos respectivos controladores clássicos, resultado que já era esperado, uma vez que as versões nebulosas possuem maior flexibilidade. Entretanto, os resultados poderiam ser outros se não houvesse um método de busca eficiente para o ajuste paramétrico dos controladores nebulosos.

O tempo total necessário para completar o processo de sintonia dos controladores depende do intervalo de tempo gasto por cada tarefa de controle, do número de indivíduos da população do algoritmo genético e do número de subpopulações. Para o experimento do **item 4.3**, a tarefa de controle é realizada em 2,4 segundos, cada população é composta de 20 indivíduos e são geradas 4 subpopulações no processo de reprodução. Considerando que uma das subpopulações não participa do processo de reprodução, são avaliados 60 novos indivíduos (controladores) por geração, portanto o tempo gasto para avaliação dos controladores numa geração é de 144 segundos. O tempo total decorrido na execução do algoritmo genético durante 50 gerações é de aproximadamente 120 minutos. Neste cálculo aproximado não foi considerado o tempo gasto pelo algoritmo genético – da ordem de um segundo por geração – por este ser aleatório e muito pequeno se comparado ao tempo gasto pelas tarefas de controle.

Do ponto de vista prático, o processo de sintonia poderia ser interrompido a partir do ponto em que um objetivo fosse alcançado, por exemplo, quando a função de *fitness* atingisse um valor desejado. Este e vários outros critérios de parada podem ser utilizados com o objetivo de reduzir o tempo de sintonia.

Não foi interesse da pesquisa até aqui realizada comparar o desempenho do algoritmo genético proposto em função da variação de seus parâmetros, como taxa de mutação, taxa de *crossover*, etc. No entanto, estes parâmetros podem também influenciar na quantidade de gerações executadas para se obter resultados satisfatórios. Contudo, não é possível garantir que um conjunto de parâmetros seja mais adequado para todos os tipos de problemas, mesmo porque o procedimento de busca do algoritmo genético é um processo aleatório. Conseqüentemente, a estratégia adotada consistiu na escolha empírica de valores de parâmetros cujos resultados mostraram-se satisfatórios quando comparados aos de outros trabalhos envolvendo algoritmos genéticos.

Devido à generalidade estrutural do algoritmo genético proposto, ele pode ser usado em muitos tipos de aplicações práticas de controle. Mas é evidente que as funções de *fitness* devem ser adaptadas para representar adequadamente cada objetivo de controle desejado.

A montagem experimental, usada como paradigma para validar a abordagem proposta, poderá ser utilizada para muitos outros testes com novos algoritmos a serem propostos durante os avanços da pesquisa.

A seguir, destaca-se alguns pontos interessantes que foram fundamentais para obtenção dos resultados positivos apresentados.

- **Escolha apropriada do modelo de controlador nebuloso:** O controlador nebuloso de *Takagi-Sugeno* foi utilizado de forma a incorporar as qualidades dos controladores clássicos tipo PD e PID, resultando em controladores não lineares poderosos e flexíveis. Além disso, o modelo de *Takagi-Sugeno* é adequado para o processo de otimização por algoritmos genéticos.
- **O algoritmo genético proposto possui características que permitem a sua aplicação prática em problemas de controle em tempo real.**
- **A escolha de uma função de *fitness* adequada para problemas de controle.**
- **Desenvolvimento de bibliotecas em C^{++} para implementação de algoritmos genéticos e controladores nebulosos:** Este ponto foi importantíssimo para viabilizar o desenvolvimento dos algoritmos de controle em tempo real.
- **Construção de um sistema mecânico e circuitos eletrônicos para validação prática da técnica proposta.**

Visando a continuação e a extensão da pesquisa realizada, foi feita a seguinte listagem contendo idéias e sugestões para trabalhos futuros:

- Desenvolver um procedimento que possibilite a aplicação do algoritmo genético proposto para otimização dos parâmetros das funções de pertinência, garantindo que todo o espaço de entrada do sistema nebuloso sempre esteja coberto pela base de regras.
- Desenvolver a abordagem de controle nebuloso com compensação de gravidade para servomecanismos robóticos com dois ou mais graus de liberdade.

- Aplicar o algoritmo de otimização e controle para gerar controladores para outros tipos de sistemas dinâmicos.

- Embora os resultados experimentais apresentem manutenção da estabilidade em malha-fechada, as pesquisas podem ser continuadas no sentido de incluir critérios de estabilidade para as definições das funções de *fitness*.

- Realizar estudos na área de otimização multi-objetivo a fim de sintetizar funções de *fitness* que manipulem mais eficientemente os termos de ponderação entre os múltiplos critérios e que incluam outras medidas de desempenho dos controladores, como gasto de energia, complexidade numérica do algoritmo, desgaste do sistema mecânico, etc.

- Criar um algoritmo genético para configuração estrutural do controlador nebuloso, como definição do número de conjuntos nebulosos, tipo de mecanismo de inferência, número de regras, etc.

- Fazer experimentos usando funções não lineares como conseqüentes das regras nebulosas de *Takagi-Sugeno*.

- Adaptar o algoritmo para problemas em que se conhece a trajetória ótima do sistema a cada instante.

- Escolher-se ponderações fixas para a formação das funções de *fitness* ao longo de todas as gerações necessárias para atingir as soluções adequadas aos problemas pode ser razoável para muitos casos, mas certamente se tais funções sofrerem adaptações ao longo do processo evolutivo, poder-se-á conseguir ainda melhores resultados, principalmente com vistas à redução dos tempos de busca e se houver multi-objetivos em questão. Este é um assunto em aberto na corrida tecnológica, e certamente gerará grande quantidade de novos trabalhos científicos.

Os controladores *PD* e *PID* Nebulosos apresentados nesta tese podem ser vistos como sendo de propósito geral, e podem ser usados para solucionar problemas de controle de muitas áreas. Muitos problemas práticos de controle têm natureza não linear e demandam ações de controle não linear como melhor solução. Normalmente, devido à dificuldades técnicas, os problemas que envolvem controle não linear são aproximados como sendo problemas de controle linear e resolvidos com o auxílio da teoria para sistemas lineares. Os controladores tipo *PID* são muito eficientes na solução de problemas de controle linear, e muito populares devido ao fato de possuírem estruturas simples e ajustes relativamente simples, quando feitas comparações com outras classes de controladores. Neste sentido, se

um controlador *PID* linear conseguir produzir bons resultados no controle de um determinado sistema, então, certamente, os controladores nebulosos dos tipos apresentados nesta tese poderão produzir resultados ainda melhores.

Com os excelentes resultados práticos obtidos através de abordagens como as utilizadas nesta tese, é possível perceber que o aparecimento das máquinas cibernéticas evolutivas - primeiramente concebidas em ficção - está muito mais próximo da realidade, sendo que é muito razoável prever que em um futuro de curto prazo o desenvolvimento de tais máquinas com *hardware* bio-inspirado (*Bioware*) e evolutivo (*Evolware*) deverá experimentar enormes avanços (Sipper 1997). Isso começou a ficar muito notório a partir do final de 1995, com o *workshop "Towards Evolvable Hardware"* realizado em *Lausanne* na Suíça, e da primeira conferência internacional de sistemas com *hardware* evolutivo (*ICES*96) *"From Biology to Hardware"*, realizada em 1996 no Japão.

A pergunta de por que exatamente os *AGs* podem funcionar tão bem, ainda não está respondida através de uma teoria geral. Há uma emergente concentração de esforços por parte da comunidade científica internacional pela procura de respostas generalizadoras, e isso pode servir como estímulo para a proposição de novos *AGs* e respectivas aplicações.

Referências Bibliográficas

Abramovitch, D. Y. & L. G. Bushnell (1999). Report on the fuzzy versus conventional control debate. *IEEE Control Systems* pp. 88–91.

Aström, K. J., C.C Hang, P. Persson & W. Ho (1992). Towards intelligent PID control. *Automatica* **28**(1), 1–9.

Begovich, O., E. N. Sanchez & M. Maldonado (2000). T-S scheme for trajectory tracking of an underactuated robot. *2000 IEEE International Fuzzy Systems Conference, San Antonio, Texas-USA* **2**, 798–803.

Benerjee, S. & P. Y. Woo (1993). Fuzzy logic control of a robot manipulator. *Proc. 2nd IEEE/RSJ Internat. Conf. on Intelligent Robot and Systems* **1**, 87–88.

Braae, M. & D. Rutherford (1979). Theoretical and linguistic aspects of the fuzzy logic controller. *Automatica* **15**, 553–577.

Buckley, J. J. (1993). Sugeno type controllers are universal controllers. *Fuzzy Sets and Systems* **53**, 299–304.

Buckley, J. J., K. D Reilly & K. V. Penmetcha (1996). Backpropagation and genetic algorithm for training fuzzy neural. *Proc. 5 IEEE International Conference on Fuzzy Systems (FUZZIEEE'96, New Orleans-USA* **1**, 2–6.

Coleman, C. & D. Godbole (1994). A comparison of robustness: fuzzy logic, PID, & sliding mode control. *IEEE International Conference on Fuzzy Systems* **3**, 1654–1659.

Darwin, C. (1859). *The Origin of Species.* Penguin Classics,1985. John Murray.

Delgado, M. R., F. Von Zuben & F. Gomide (2000). Evolutionary design of Takagi-Sugeno fuzzy systems: a modular and hierarchical approach. *2000 IEEE International Fuzzy Systems Conference,San Antonio, Texas-USA* **1**, 447–452.

Driankov, D., H. Hellendoorn & M. Reinfrank (1996*a*). *An Introduction to Fuzzy Control.* second ed.. Springer-Verlag.

Driankov, D., R. Palm & U. Rehfuss (1996*b*). A Takagi-Sugeno fuzzy gain-scheduler. *Proc. IEEE Conf. Fuzzy Systems, New Orleans-USA* pp. 1053–1059.

Fantuzzi, C. & R. Rovatti (1996). On the approximation capabilities of the homogeneous Takagi-Sugeno model. *Proc. IEEE Conf. Fuzzy Systems, New Orleans-USA* pp. 1067–1072.

Ferrara, N. F. & C. P. C. Prado (1994). *Caos Uma Introdução.* Edgard Blücher Ltda.

Fogel, D. B. (2000). What is evolutionary computation?. *IEEE Spectrum* pp. 26–32.

Franklin, G. F., J. D. Powell & A. Emami-Naeini (1994). *Feedback Control of Dynamic Systems.* Addison-Wesley Publishing Co.. Massachusetts-USA.

Fujitech, F. (1988). *FLEX-8800 series elevator group control system.* Fujitec Co., Ltd.. Osaka-Japan.

Gen, M., K. Ida & Y. Li (1995). Solving bicriteria solid transportation problem with fuzzy numbers by genetic algorithm. *Proc. 17th International Conference on Computers and Industrial Engineering, Phoenix-USA* **29**, 537–541.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading,MA.

Guckenheimer, J. & P. Holmes (1983). *Nonlinear Oscillators, Dynamical Systems and Bifurcations of Vector Fields.* Cambridge University Press. Cambridge.

Gurocak, H. B. (1999). A genetic algorithm-based method for tunning fuzzy logic controllers. *Fuzzy Sets and Systems* **108**(1), 39–47.

Hampel, Rainer & Nasredin Chaker (1998). Minimizing the number of variable parameters for optimizing the fuzzy-controller. *Fuzzy Sets and Systems* **100**, 131–142.

Herrera, F., M. Lozano & J. L. Verdegay (1998). A learning process for fuzzy control rules using genetic algorithms. *Fuzzy Sets and Systems* **100**, 143–158.

Ho, Y. C. (1999). The No Free Lunch Theorem and The Human-Machine Interface. *IEEE Control Systems* pp. 8–10.

Holland, J. H. (1975). *Adaption in Natural and Artificial Systems.* University of Michigan Press, An Arbor. MI.

Johansen, T. A., K. J. Hunt & H. Fritz (1998). A software environment for gain scheduled local controller network design. *IEEE Control Systems Magazine* **18**(2), 48–60.

Jorde, L. B., J. C. Carey & R. L. White (1995). *Medical Genetics*. Mosby-Year Book, Inc.. St.Louis, Missouri-USA.

Karr, C. L., D. A. Stanley & B. McWhorter (2000). Optimization of hydrocyclone operation using a geno-fuzzy algorithm. *Computer Methods in Applied Mechanics and Engineering* **186**, 517–530.

Kasai, Y. & Y. Morimoto (1988). Eletronically controlled continuously variable transmission. *Proceedings of International Congress on Transportation Electronics*.

Kelly, R. (1997). PD controller with desired gravity compensation of robotic manipulators: a review. *The Intenat. Journal of Robotics Research* **16**(5), 660–672.

Kuhn, H. W. & A. W. Tuker (1951). Nonlinear programming. *Proc. 2nd Berkeley Symposium on Mathematical Statistics and Probability* pp. 481–492.

Kumbla, K. & M. Jamshidi (1993). Fuzzy control of three links of a robotic manipulator. *Proc. Internat. Fuzzy Systems Assoc. World Congress* **3**, 1410–1413. Seoul-Korea.

Kwok, D., P. Tam & P. Wang (1990). Linguistic PID controllers. *IFAC World Congress, Tallin, USRR* pp. 192–197.

Lee, C. C. (1990*a*). Fuzzy logic in control systems: Fuzzy logic controller - Part I. *IEEE Trans. Syst. Man Cybern.* **20**, 404–418.

Lee, C. C. (1990*b*). Fuzzy logic in control systems: Fuzzy logic controller - Part II. *IEEE Trans. Syst. Man Cybern.* **20**, 419–435.

Li, R. H. & Z. Yi (1996). Fuzzy logic controller based on genetic algorithms. *Fuzzy Sets and Systems* **83**(1), 1–10.

Li, T. H. S. & M. Y. Shieh (2000). Design of a GA-Based Fuzzy PID Controller for Non-Minimum Phase Systems. *Fuzzy Sets and Systems* **111**, 183–197.

Li, Y. F. & C. C. Lau (1989). Development of fuzzy algorithms for servo systems. *IEEE Control Systems Magazine* **9**, 65–72.

Lim, C. M. & T. Hiyama (1991). Application of fuzzy logic controller to a manipulator. *IEEE Trans. Robotics and Automation* **7**(5), 688–691.

Linkens, D. A. & H. O. Nyongesa (1995*a*). Genetic algorithms for fuzzy control - offline system-development and application. *IEE proceedings-control theory and applications* **142**(3), 161–176.

Linkens, D. A. & H. O. Nyongesa (1995*b*). Genetic algorithms for fuzzy control - online system-development and application. *IEE proceedings-control theory and applications* **142**(3), 177–185.

Macready, W. G. & D. H. Wolpert (1997). The No Free Lunch Theorems. *IEEE Trans. Evolutionary Computing* **1**(1), 67–82.

Mamdani, E. H. (1974). Application of fuzzy algorithms for simple dynamic plant. *Proceedings of the IEE* **121**(12), 1585–1588.

Mamdani, E. H. & N. Baaklini (1974). Descriptive method for deriving control policy in a fuzzy logic controller. *Eletronics Letters* **11**, 625–626.

Mamdani, E. H. & S. Assilian (1975). An experiment in linguistic synthesis with a fuzzy logic controller. *International Journal of Man-Machine Studies* **7**, 1–13.

Michalewicz, Z. (1996). *Genetic algorithms + data structures = evolution programs.* Springer. New York.

Nobre, F. S. M. (1997). *Projeto e Análise de Controladores Nebulosos e Sua Aplicação para Controle de Juntas Robóticas.* Tese de Mestrado. Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, Campinas, São Paulo.

Parker, T. S. & L. O. Chua (1989). *Practical Numerical Algorithms for Chaotic Systems.* Springer-Verlag. New York.

Pedrycz, W. & F. Gomide (1998). *An introduction to fuzzy sets: analysis and design.* A Bradford book - The MIT Press. London-England.

Peng, X., S. Liu, T. Yamakawa, P. Wang & X. Liu (1988). Self-regulating PID controllers and its applicatios to a temperature controller process. *Fuzzy Computing, by M. M. Gupta and T. Yamakawa, Elsevier Science Publishers.*

Qi, X. M. & T. C. Chin (1997). Genetic algorithms based fuzzy controller for high order systems. *Fuzzy Sets and Systems* **91**(3), 279–284.

Shorten, R., R. M. Smith, R. Bjorgan & H. Gollee (1999). On the interpretation of local models in blended multiple model structures. *Int. J. Control* **72**, 620–628.

Silva, C. W. (1995). Applications of fuzzy logic in the control of robotic manipulators. *Fuzzy Sets and Systems* **70**, 223–234.

Sipper, M. (1997). A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *IEEE Transactions on Evolutionary Computation* 1(1), 83–96.

Sousa, M. A. T. & M. K. Madrid (1999). Controlador nebuloso com sintonia automática por algoritmos genéticos aplicado a manipuladores robóticos. *4º SBAI,São Paulo-Brazil* pp. 188–193.

Sousa, M. A. T. & M. K. Madrid (2000). Optimization of Takagi-Sugeno fuzzy controllers using a genetic algorithm. *2000 IEEE International Fuzzy Systems Conference, San Antonio, Texas-USA* **1**, 30–35.

Srikanth, R., R. George, N. Warsi, D. Prabhu, F. E. Petry & B. P. Buckles (1995). A variablelength genetic algorithm for clustering and classification. *Pattern Recognition Letters* **16**, 789–800.

Takagi, T. & M. Sugeno (1985). Fuzzy identification of systems and its applications to modeling and control. *IEEE Trans. Syst., Man, Cybern.* **15**, 116–132.

Takegaki, M. & S. Arimoto (1981). A new feedback method for dynamic control of manipulators. *ASME J. Dyn. Sys. Meas. Control* **103**, 119–125.

Tanaka, K. & M. Sugeno (1992). Stability analysis and design of fuzzy control systems. *Fuzzy Sets and Systems* **45**, 135–156.

Tanscheit, R. & E. M. Scharf (1988). Experiments whith the use of a rule-based self-organising controller for robotics applications. *Fuzzy Sets and Systems* **26**, 195–214.

Tong, R. M. (1978). Analysis and control of fuzzy systems using finite discrete relations. *International Journal of Control* **27**(3), 431–440.

Wang, H. O., K. Tanaka & M. F. Griffin (1995). Parallel distributed compensation of nonlinear systems by Takagi-Sugeno fuzzy model. *FUZZ-IEEE/IFES'95 International Fuzzy Systems Conference, San Antonio, Texas-USA* pp. 531–538.

Wang, Li-Xin (1999). Automatic design of fuzzy controllers. *Automatica* **35**, 1471–1475.

Whitley, D. (1993). A Genetic Algorithm Tutorial, Technical Report CS-93-103, pp. 1-37, Department of Computer Science, Colorado State University, USA.

Whitley, D., T. Starkweather & D. Fuquay (1989). Scheduling problems and traveling salesman: The genetic edge recombination operator. *ICGA89* pp. 133–140.

Yagshita, O., O. Itoh & M. Sugeno (1985). Application of fuzzy reasoning to water purification process. *Industrial Applications of Fuzzy Control, Ed. Amsterdam: North-Holland* pp. 19–40.

Yamada, T. & R. Nakano (1992). A genetic algorithm applicable to large-scale job-shop problems. *PPSN92* pp. 281–290.

Yasunobu, S. & T. Hasegawa (1985). Automatic train operation by predictive fuzzy control. *Industrial Applications of Fuzzy Control, Ed. Amsterdan:North-Holland* pp. 1–18.

Ying, H. (1998*a*). An analitical study on structure, stability and design of general nonlinear Takagi-Sugeno fuzzy control systems. *Automatica* **34**(12), 1617–1623.

Ying, H. (1998*b*). Constructing nonlinear variable gain controllers via the Takagi-Sugeno fuzzy control. *IEEE Transaction on Fuzzy Systems* **6**(2), 226–234.

Ying, H. (2000). Theory and application of a novel fuzzy PID controller using a simplified Takagi-Sugeno rule scheme. *Information Sciences* **123**, 281–293.

Yong, M. De, J. Polson, R. Moore, C.C. Weng & J. Lara (1992). Fuzzy and adaptative control simulations for a walking machine. *IEEE Control Systems Magazine* **12**, 43–50.

Zadeh, Lofti A. (1965). Fuzzy sets theory. *Information and control* **8**, 338–353.

Zheng, Li (1992). A practical guide to tune of proportional and integral (PI) like fuzzy controllers. *Proceedings of the First IEEE International Conference on Fuzzy Systems* pp. 663–670.

Ziegler, J. G. & N. B. Nichols (1942). Optimum setings for automatic controllers. *Transactions of the ASME* pp. 759–768.

Apêndice A

Descrição do Sistema Experimental

Os resultados mais relevantes da pesquisa realizada dependeram de vários fatores técnicos que são essenciais para a realização de um trabalho de qualidade com dados experimentais confiáveis. Este Anexo tem por finalidade explicitar alguns detalhes do *hardware* e do *software* relacionados aos experimentos práticos realizados.

A.1 O Sistema Experimental

O sistema prático utilizado para a obtenção dos resultados apresentados no Capítulo 4 é composto de um pêndulo acionado, um motor de corrente contínua (CC), um codificador óptico incremental (*encoder*), um circuito de interfaceamento, um conversor Analógico/Digital (D/A), um contador $UP/DOWN$, um amplificador de potência, e um microcomputador do tipo $PC486$ com relógio de 100 MHz. O algoritmo de controle foi implementado via *software*, sendo que o sinal de controle calculado é enviado para a interface de E/S (Entrada/Saída). A saída digital de controle é convertida para um sinal analógico através de um conversor D/A de 12 bits. Em seguida, este sinal é amplificado linearmente para alimentar um motor CC que está conectado ao pêndulo. A posição do pêndulo é lida através de um contador $UP/DOWN$ cuja contagem é função dos movimentos de um *encoder* óptico convenientemente acoplado ao pêndulo. A figura A.1 ilustra como todas as partes do sistema prático foram interligadas.

A seguir, são descritas de forma mais detalhada as partes que compõem o processo.

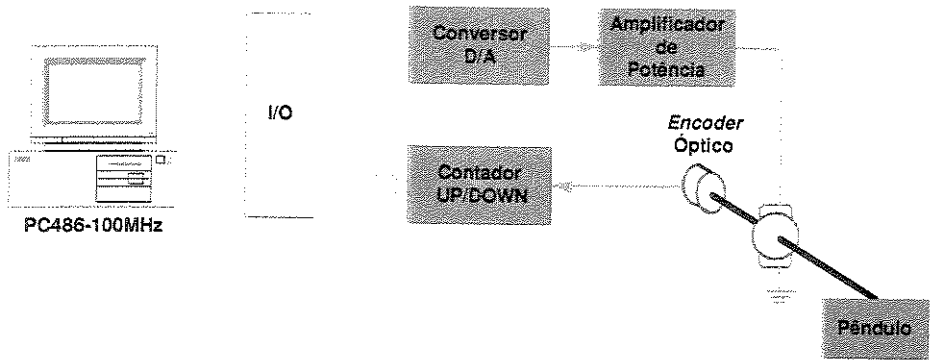


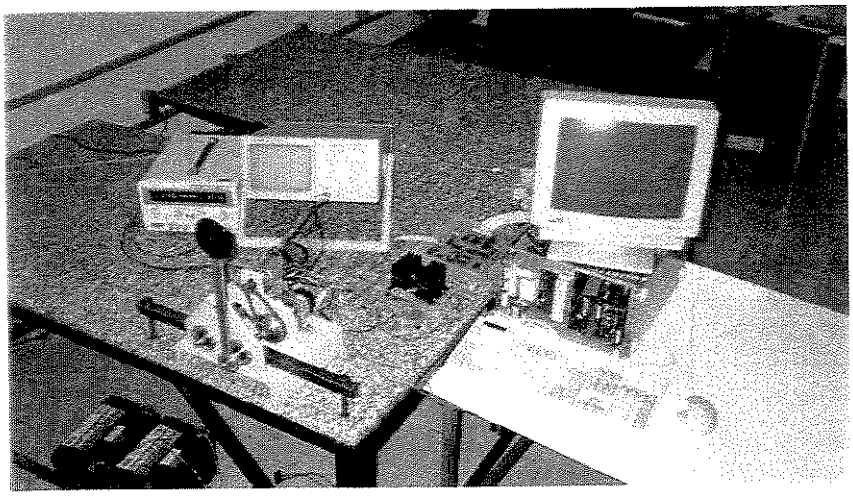
Figura A.1: Diagrama do sistema experimental.

A.1.1 Pêndulo

O pêndulo utilizado nos experimentos consiste de um sistema mecânico simples composto de dois eixos e engrenagens do tipo polias e correias sincronizadas. O motor está acoplado a um dos eixos através de uma engrenagem que reduz a velocidade por um fator de $4:1$. Este primeiro eixo está acoplado ao segundo também através de uma engrenagem com fator de redução de $4:1$. Portanto, no primeiro eixo a velocidade do motor fica reduzida de $4:1$ e no segundo de $16:1$. O braço do pêndulo pode ser conectado a qualquer um dos eixos, dependendo da redução desejada. Nos experimentos realizados, o braço foi acoplado ao primeiro eixo para obter-se efeitos mais intensos das não-linearidades causadas pela ação do torque gravitacional. Para conseguir-se medidas mais precisas da posição do pêndulo, o *encoder* foi conectado ao mesmo eixo do braço do pêndulo. As etapas de projeto, usinagem e construção deste pêndulo foram realizadas pela própria equipe do Laboratório de Sistemas Modulares Robóticos ($LSMR$). A figura A.2 mostra um vista da bancada de laboratório que foi utilizada nos testes experimentais cujos resultados estão apresentados no corpo deste trabalho, e também um *croquis* mostrando alguns detalhes da construção do pêndulo acionado.

A.1.2 Motor

O motor utilizado para acionar o pêndulo é um servomotor de corrente contínua modelo $E-530$, da *Electro-Craft*. As especificações técnicas deste motor estão descritas na Tabela A.1.



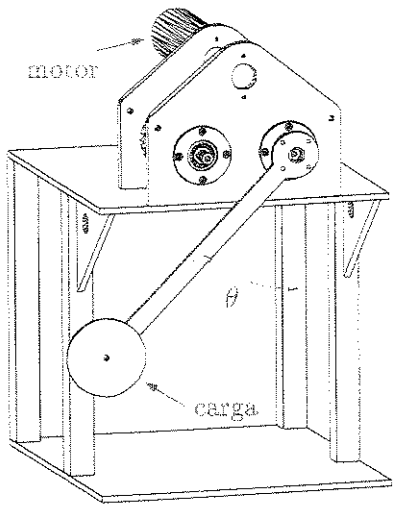


Figura A.2: Bancada experimental com a montagem do pêndulo acionado.

A.1.3 Circuito de Entrada/Saída (I/O)

Este circuito tem a função de decodificar os endereços que serão utilizados para a escrita e leitura de dados. A figura A.5, apresentada mais adiante, mostra o diagrama esquemático do circuito de Entrada/Saída.

São utilizados *buffers* em todas as linhas de comunicação externa com o microcomputador, pois segundo a descrição técnica dos *slots*, cada linha de sinal consegue acionar seguramente apenas um dispositivo TTL do tipo LS. O circuito integrado $U2$ ($74F245$) é um transmissor bidirecional para barramento (*tri-state*), que permite o tráfego de dados entre o computador e os circuitos a ele conectados. A direção do fluxo de dados é de-

Modelo	E−530
Potência Máxima	28 [Watt]
Torque Máximo	32 [oz.-in.]
Inércia da Armadura	0.0038 [oz.-in.-sec]^2
Coeficiente de Amortecimento	0.1 [oz.-in./KRPM]
Constante de Tempo Elétrica	2.06 [mseg]
Constante de Tempo Mecânica	8.3 [mseg]
Rotação Máxima (sem carga)	6000 [RPM]
Resistência Elétrica de Armadura	1.55 [Ohms]
Indutância da Armadura	3.19 [mH]
Tensão	12 [V]

Tabela A.1: Especificações técnicas do motor

terminada pelo nível lógico aplicado ao pino DIR, neste caso, conectado ao sinal $-IOR$. Os circuitos integrados $U3$ e $U4$ são *buffers* ($74LS244$) de 8 bits cada, que permitem a passagem dos sinais de controle e do barramento de endereços para o circuito. O circuito integrado $U5$ ($74F138$) é um demultiplexador/decodificador 3×8 que estabelece a faixa de endereços de E/S através dos sinais $ADR2$, $ADR3$ e $ADR4$, e é utilizado para gerar e/ou distinguir oito endereços disponibilizados para escrita/leitura ($PE1$, $PE2$, ...,$PE8$).

A.1.4 *Encoder* Óptico

O sensor utilizado para medir a posição angular do pêndulo é um *encoder* óptico modelo $HEDS-6310$ da *Hewlett Packard* (HP), que possui uma resolução de 1024 pulsos por volta. Este *encoder* dispõe de três fases de saída, sendo duas fases (FA e FB) deslocadas de 90^o entre si, cujos sinais são usados para as medições do deslocamento angular, e uma terceira fase (FS) que emite um pulso de referência por volta que pode ser utilizado para estabelecer o referencial das medições. O circuito de decodificação dos sinais do *encoder* (ver figura A.6 do Anexo A) é baseado no circuito integrado $HCTL2020$ fabricado pela própria HP. Este CI integra num só componente toda a lógica de detecção de fase e um circuito contador $UP/DOWN$ de 16 bits.

A lógica de detecção de fase é responsável pela identificação do sentido de rotação do eixo de movimento do pêndulo, ou do eixo do *encoder* que está solidário. Em cada pulso emitido pelo *encoder* é possível identificar quatro transições de estado nas fases FA e FB, com isso consegue-se uma resolução total de 4096 pulsos por volta, que equivale a uma

precisão de 0,09 graus na medida da posição angular.

A.1.5 Conversor Digital/Analógico

O conversor utilizado para transformação do sinal digital em analógico é o conversor *DAC667* da *BURR BROWN* que possui uma resolução de 12 bits. O CI *DAC667* foi adequadamente configurado para fornecer uma tensão de saída variando no intervalo [−10;+10] Volts. O diagrama esquemático do circuito de conversão encontra-se na figura A.6.

A.1.6 Amplificador de Potência

O circuito amplificador de potência compõe o estágio responsável pela amplificação dos sinais provenientes do conversor D/A. O diagrama esquemático do circuito amplificador de potência encontra-se ilustrado na figura A.7. O circuito é baseado no amplificador operacional dual *OPA2541* da *BURR BROWN*, cujas principais características são:

- alimentação de até 40 Volts;
- estágio de entrada baseado na tecnologia FET;
- capacidade de corrente de saída de até 5A por canal;
- amplificação linear;
- produto ganho-banda passante de 1,6 MHz.

O circuito desenvolvido é composto de dois estágios. O primeiro estágio, baseado no amplificador operacional $TL081$, é responsável pelo ganho de tensão e ajuste da simetria do sinal. O segundo estágio tem a função de fornecer a corrente necessária para alimentação do motor. O potenciômetro $P1$ é utilizado para ajustar o nível de tensão 0 volts do sinal de saída ($drift$). O potenciômetro $P2$ tem a função de ajustar o ganho de tensão de forma que este varie no intervalo [−12;+12] Volts.

A.2 Alguns Detalhes do *Software*

Todas as rotinas de acesso a *hardware*, geração de interrupções e controle em tempo real foram escritas em linguagem C padrão. O código fonte das funções de acesso a *hardware* e programação de interrupções encontra-se descrito no Anexo C.

A.2.1 Interrupções

Aplicações de controle em tempo-real exigem precisão e confiabilidade quanto ao período de amostragem. É necessário garantir que o sinal de controle e os dados lidos sejam atualizados a cada período de amostragem, que no caso deve ser fixo.

O sistema de amostragem utilizado é baseado na geração de interrupções. Os micro-computadores do tipo $PC-AT$ possuem contadores internos que podem ser programados para gerar interrupções. Neste trabalho, utilizou-se o relógio de sistema do PC para gerar regularmente a interrupção $INT\ 0 \times 1C$. Normalmente, este relógio está programado para gerar uma interrupção a cada $54,9$ milisegundos ($18,2$ interrupções por segundo), podendo ser reprogramado para obtenção de períodos de amostragem de até no mínimo $0,838$ microsegundos. Nos experimentos realizados utilizou-se um período de amostragem de 5 milisegundos, tempo suficiente para o cálculo do sinal de controle pelo algoritmo de controle nebuloso. Geralmente, aplicações de robótica permitem intervalos de amostragem de até 10 milisegundos para que se obtenha desempenho e precisão razoáveis.

Uma vez inicializada a rotina de interrupção, em cada instante de amostragem executam-se os seguintes passos:

1. Incrementa a variável global **tickcount**, que indica quantos intervalos de amostragem se passaram desde a inicialização da rotina de interrupção. Esta variável global é utilizada para determinar o tempo decorrido na tarefa de controle.
2. Lê o estado do contador e armazena o resultado na variável global **position**.
3. Escreve no conversor D/A a palavra digital de controle armazenada na variável global **da_value** que foi calculada no intervalo de amostragem anterior.

A seguir, é feita uma descrição das principais funções utilizadas na implementação do sistema de amostragem.

Função *SetNewCounterTime()*

Esta função programa o relógio do sistema para que este gere interrupções regulares a cada período de amostragem. O valor do período de amostragem é definido pela macro TS, a qual é expressa em milisegundos.

Função *RestoreCounterTime()*

Restaura a programação padrão do relógio do sistema, ou seja, o relógio volta a gerar $18,2$ interrupções por segundo. Esta função deve ser chamada na finalização da tarefa de controle em tempo real.

Função *InitTimer()*

Esta função inicializa o sistema de amostragem. Primeiramente, o vetor de interrupção associado à interrupção $INT\ 0 \times 1C$ é salvo para posterior restauração da rotina de interrupção padrão. Em seguida, instala-se o serviço de interrupção responsável pelo sistema de amostragem. Finalmente, o relógio de sistema é programado para gerar interrupções de acordo com o período de amostragem escolhido.

Função *FinishTimer()*

Esta função restaura a programação do relógio de sistema e instala o serviço de interrupção salvo na inicialização do sistema de amostragem.

A.2.2 Funções de acesso a *hardware*

A seguir, serão descritas as principais funções de acesso a *hardware* utilizadas na implementação das tarefas de controle em tempo real.

Função *DA(float u)*

Esta função envia para o conversor D/A a palavra digital correspondente ao parâmetro normalizado u definido no intervalo $[-1;1]$. Por exemplo, se $u = 0,5$, a tensão aplicada no motor do pêndulo é de 6 Volts.

Função *ReadCounter()*

Retorna o valor em graus da posição angular do pêndulo. Para que esta função funcione corretamente a função *ConterOffSet()* deve ser chamada uma vez, para que o contador seja inicializado na posição de referência emitida pelo *encoder*.

Função *ReadCounterPulses()*

Retorna o valor em pulsos da posição angular do pêndulo. Igualmente à função *ReadCounter()*, os contadores devem ser adequadamente inicializados para que haja uma referência para contagem dos pulsos.

Função *CounterOffSet()*

Esta função realiza um movimento no pêndulo forçando-o a passar pela posição de referência na qual o *encoder* emite um pulso de referência. Este pulso é utilizado para inicialização do contador num instante um que o pêndulo está numa posição conhecida. Deste modo, é possível obter uma medida absoluta da posição.

A.3 Filtragem Digital do Sinal de Controle

A maioria dos algoritmos de controle desenvolvidos nesta tese utiliza a informação da variação do erro (Δe) no cálculo do sinal de controle. Esta variável, por ser obtida de forma indireta e devido aos efeitos da discretização, contém muito ruído de alta frequência que pode comprometer o sinal de controle.

Uma alternativa bastante eficiente para contornar este problema consiste na uti-

lização de um filtro digital passa-baixa para eliminar as componentes de alta frequência do sinal de controle.

A solução adotada neste trabalho baseia-se num filtro *FIR* (*Finite Impulse Response*) de quarta ordem projetado através do método da janela de *Hamming*. A freqüência de corte do filtro foi estabelecida em $\omega_n = 0,1$ considerando que $\omega_n = 1,0$ seja metade da freqüência de amostragem. Com isso, a equação de diferenças que define o filtro utilizado é dada por:

$$u_{filtrado}(k) = b_1 * u(k) + b_2 * u(k-1) + b_3 * u(k-2) + b_4 * u(k-3) + b_5 * u(k-4), \tag{A.1}$$

onde $b_1 = 0,0338332$, $b_2 = 0,2401270$, $b_3 = 0,4520794$, $b_4 = 0,2401270$ e $b_5 = 0,0338332$.

A figura A.3 apresenta a resposta em frequência (diagramas de *Bode*) do filtro digital aplicado nos experimentos realizados. O efeito da filtragem digital utilizada pode ser visualizado na figura A.4.

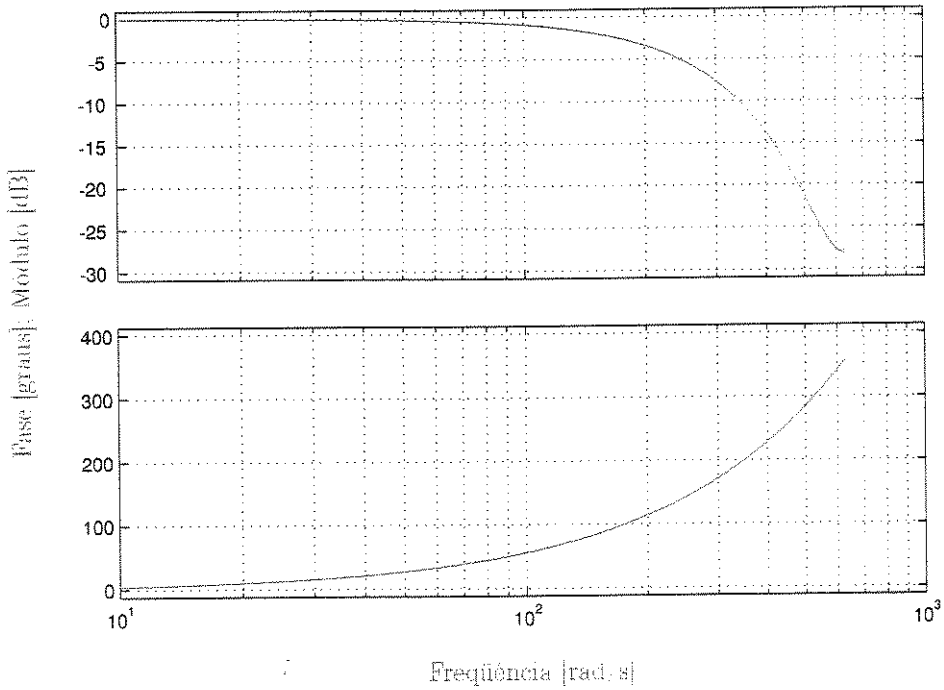


Figura A.3: Resposta em freqüência do filtro digital.

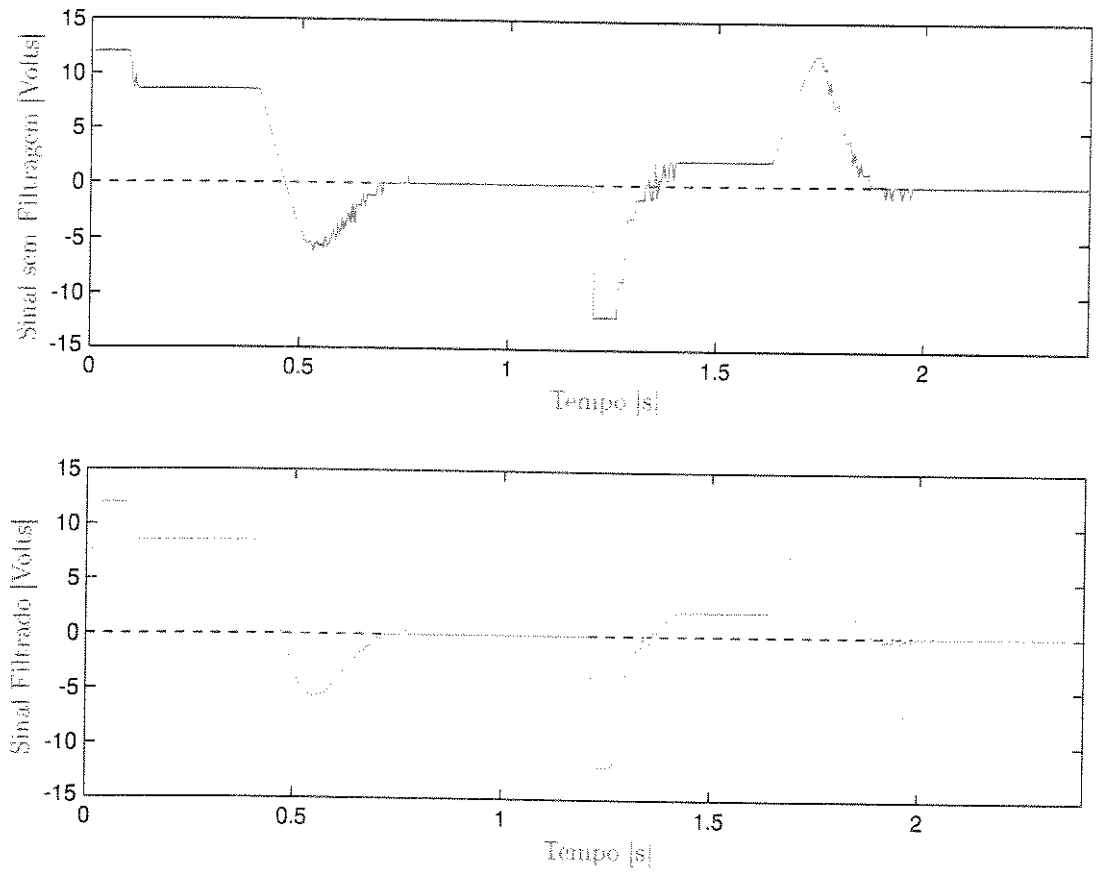


Figura A.4: Exemplo da filtragem.

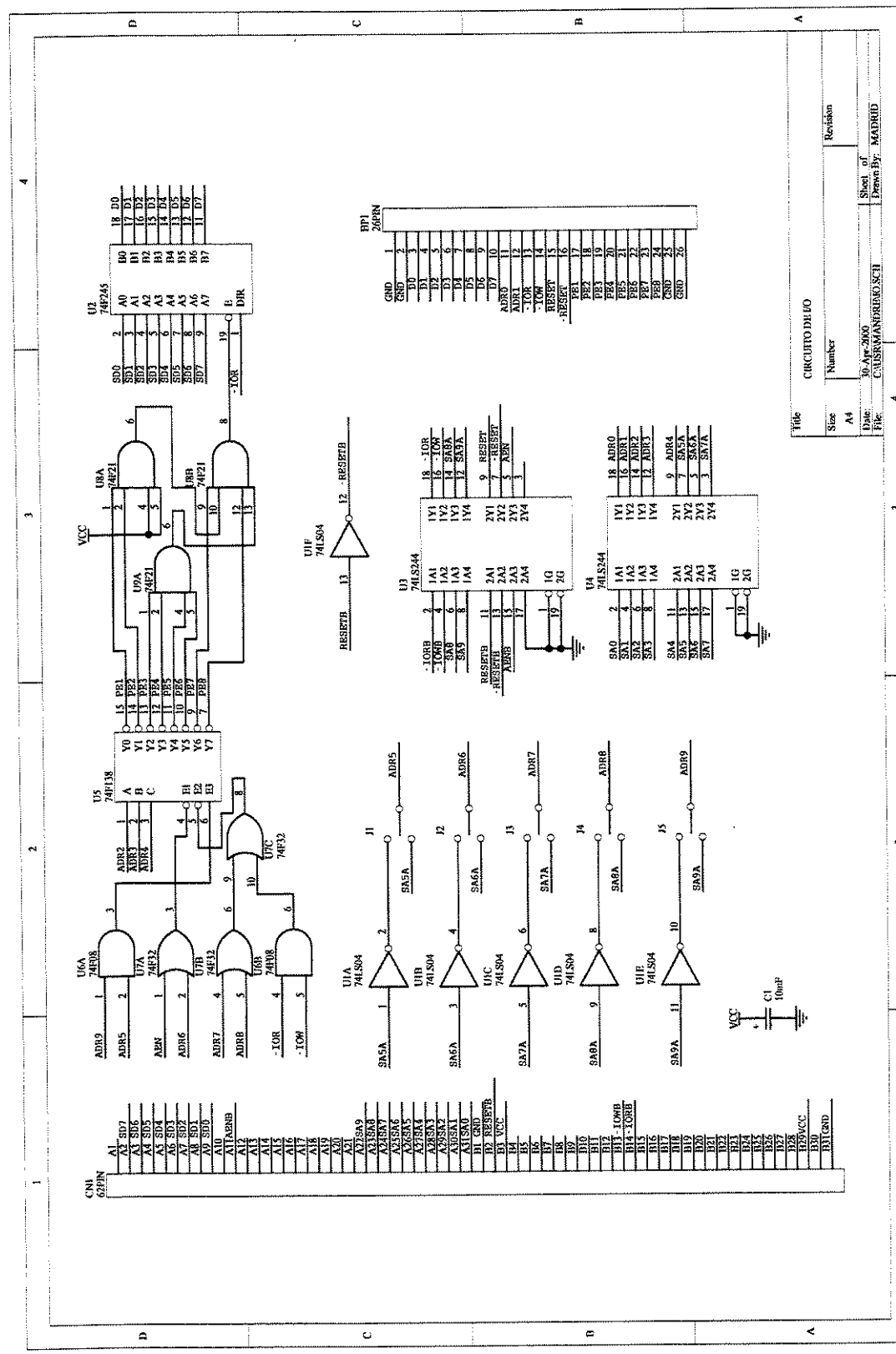


Figura A.5: Circuito de interfaceamento Entrada/Saída.

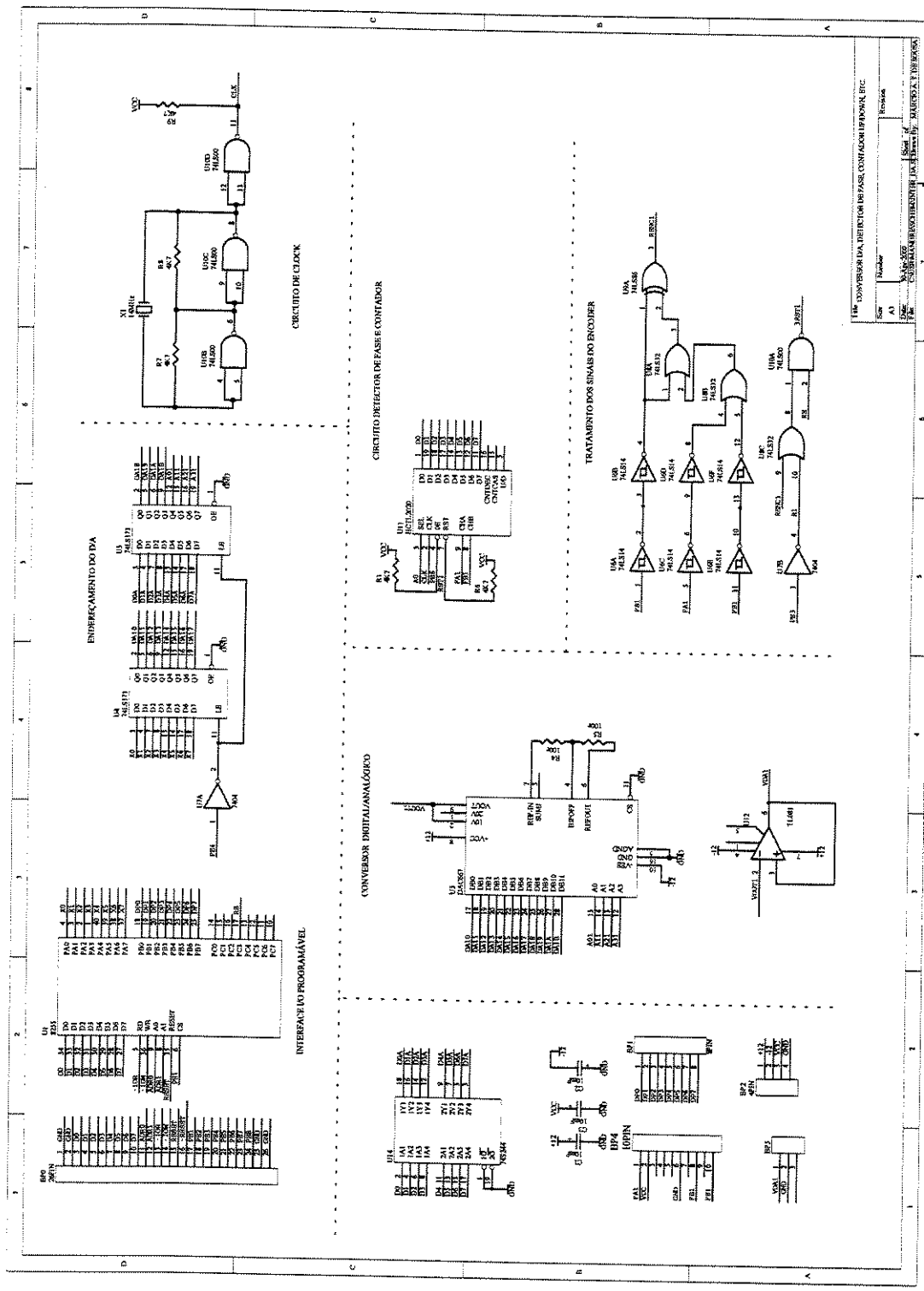


Figura A.6: Circuito detector de fase, conversor D/A e contador *UP/DOWN*.

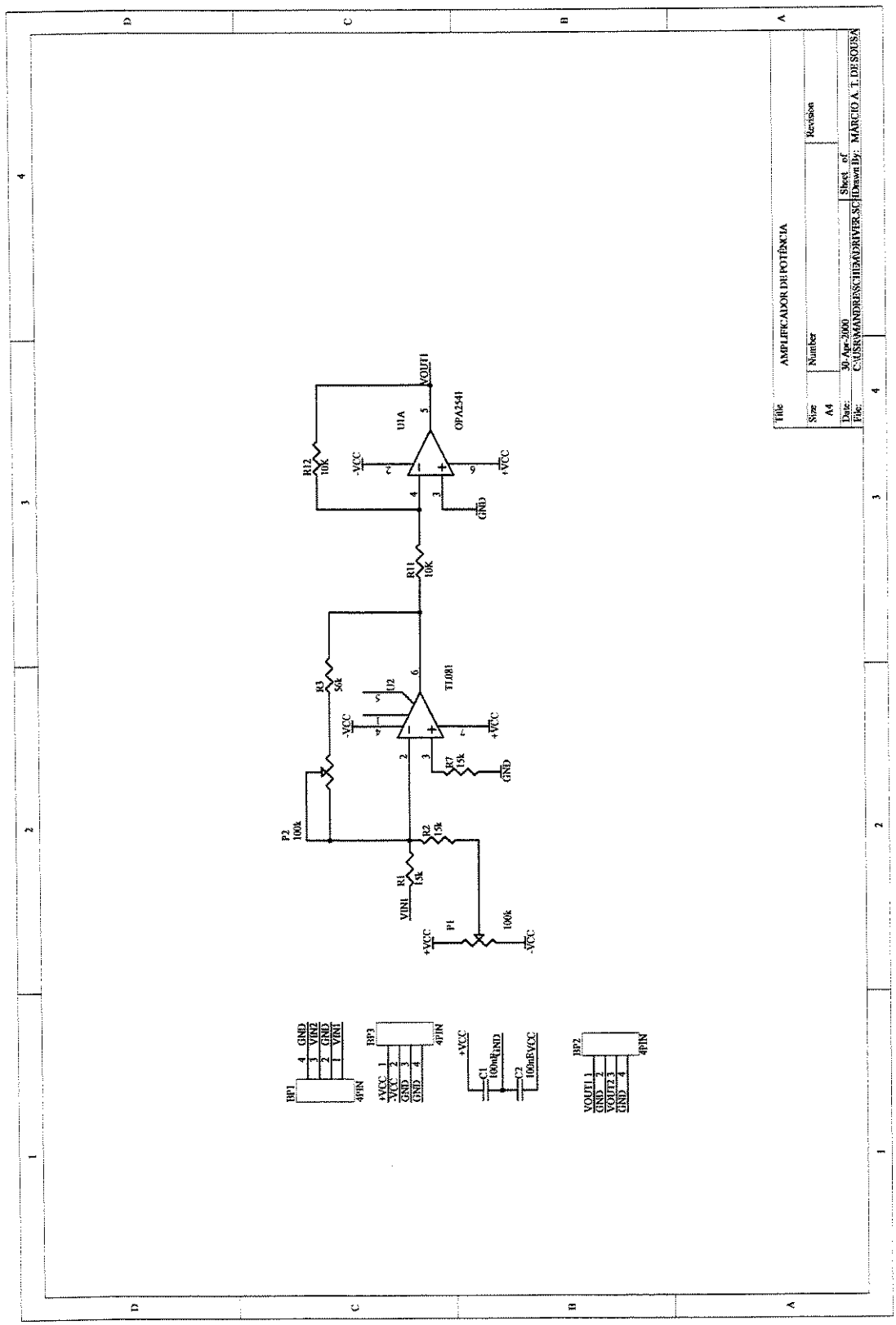


Figura A.7: Circuito Amplificador de potência.

Apêndice B

Classes em C^{++}

Neste Anexo são apresentadas as bibliotecas *FuzzySystem* e *Genetic*, desenvolvidas para implementação dos algoritmos de controle nebuloso e algoritmos genéticos.

B.1 A Biblioteca *FuzzySystem*

Com o objetivo de facilitar a implementação de programas que utilizem sistemas de controle nebuloso foi desenvolvida uma biblioteca em linguagem C^{++} chamada *FuzzySystem*. Esta biblioteca é composta basicamente de quatro classes: *Fuzzysystem*, *FuzSet*, *RuleBase* e *Universe*, que constituem a estrutura de um sistema de inferência nebulosa. A classe principal *FuzzySystem* engloba em sua estrutura objetos das classes *FuzSet* e *RuleBase*. Um objeto da classe *FuzzySystem* possui atributos e métodos de um sistema nebuloso específico para controle, tais como: base de regras, conjuntos nebulosos, método de inferência, método de defuzzificação, tipo de conjunção, etc. Como esta biblioteca foi desenvolvida para aplicações de controle, seriam necessárias algumas modificações para implementação de sistemas nebulosos mais gerais.

Basicamente, é possível trabalhar com dois tipos de sistema nebuloso: o sistema de *Mamdani* e o sistema de *Takagi-Sugeno*.

B.1.1 Classe *Universe*

Um universo de discurso é definido por um conjunto de elementos que pode ser contínuo ou discreto. Neste caso, um universo de discurso contém um número finito de elementos definidos por um intervalo. A classe *Universe* representa um universo de discurso discreto, que é utilizado como base para definição de conjuntos nebulosos.

Dados Membros

- int Card – É a cardinalidade do universo de discurso.
- float *vector – Este vetor armazena os valores discretos que compõem o universo de discurso.
- float lower – Limite inferior do intervalo que define o universo de discurso.
- float upper – Limite superior do intervalo que define o universo de discurso.

Métodos (Funções-Membro)

Universe(int card, float l, float u) – Construtor da classe *Universe*. O objeto é inicializado com base nos parâmetros que definem um universo de discurso nebuloso. A memória para o objeto é alocada de acordo com a cardinalidade, que define o tamanho do vetor que armazena os valores discretos do universo de discurso.

~Universe() – Destrutor padrão da classe que libera a memória ocupada pelo objeto.

void Print() – Imprime na tela os parâmetros elementares do universo de discurso.

B.1.2 Classe *RuleBase*

Esta classe é utilizada para definir a base de regras de um sistema nebuloso. As regras utilizadas são do tipo:

$$\underline{\text{SE}} < \text{Condição 1} > \text{E} \ldots \text{E} < \text{Condição N} > \underline{\text{ENTÃO}} < \text{Ação} >,$$

sendo possível até 3 condições para os antecedentes das regras. O conectivo padrão para avaliação dos antecedentes das regras é o operador **E**.

Dados Membros

- int NAntecedents – Número de antecedentes que compõe as regras.
- int NRules – Número total de regras.
- int Count – Esta variável indica o número de regras que já foram inicializadas.
- int *Ant – Esta matriz armazena toda informação sobre os antecedentes da base de regras.
- int *Cons – Este vetor armazena os índices que estão relacionados aos conjuntos nebulosos dos conseqüentes das regras.

Métodos (Funções-Membro)

RuleBase(int nant, int nr) – Construtor da classe *RuleBase*. A memória reservada para o objeto depende do número de antecedentes (*nant*) e do número total de regras (*nr*).

~RuleBase() – Destrutor da classe cuja finalidade é liberar toda a memória alocada para o objeto.

char AddRule(int a1, int a2, int a3, int c1) –Esta função é utilizada para inicializar as regras para a base de regras formadas por três antecedentes. As variáveis $a1$, $a2$ e $a3$ representam os índices associados aos conjuntos nebulosos dos antecedentes da regra e a variável $c1$ é o índice associado ao conjunto nebuloso do conseqüente da regra.

char AddRule(int a1, int a2, int c1) – Esta função é utilizada para inicializar as regras para a base de regras formadas por dois antecedentes.

char AddRule(int a1, int a2, int a3, int c1) – Esta função é utilizada para inicializar as regras para a base de regras formadas por um único antecedente.

Exemplo

Suponha um sistema nebuloso com duas variáveis de entrada e uma variável de saída, cujas regras sejam do tipo:

SE *pressão* é *ALTA* **E** *umidade* é *BAIXA* **ENTÃO** *fluxo* é *CONSTANTE*

onde *pressão* e *umidade* são as variáveis nebulosas de entrada, *fluxo* é a variável nebulosa de saída, e *ALTA*, *BAIXA* e *CONSTANTE* são conjuntos nebulosos.

Para definição da base de regras é necessário associar um índice a cada conjunto nebuloso. Para a base de regras adotada neste exemplo são considerados os seguintes conjuntos nebulosos.

Índice	Conjunto Nebuloso
0	BAIXA
1	MÉDIA
2	ALTA
3	CONSTANTE
4	CRESCENTE

Seja a base de regras:

1. **SE** *pressão* é *ALTA* **E** *umidade* é *BAIXA* **ENTÃO** *fluxo* é *CONSTANTE*
2. **SE** *pressão* é *ALTA* **E** *umidade* é *MÉDIA* **ENTÃO** *fluxo* é *CONSTANTE*
3. **SE** *pressão* é *ALTA* **E** *umidade* é *ALTA* **ENTÃO** *fluxo* é *CRESCENTE*
4. **SE** *pressão* é *BAIXA* **E** *umidade* é *BAIXA* **ENTÃO** *fluxo* é *CONSTANTE*
5. **SE** *pressão* é *BAIXA* **E** *umidade* é *MÉDIA* **ENTÃO** *fluxo* é *CONSTANTE*
6. **SE** *pressão* é *BAIXA* **E** *umidade* é *ALTA* **ENTÃO** *fluxo* é *CONSTANTE*

Os parâmetros necessários para alocação de memória para o objeto da classe *RuleBase* são: número de antecedentes (2) e número total de regras (6). O exemplo a seguir mostra como criar um objeto da classe *RuleBase* e como definir a base de regras.

```
//-------------------------------------
// Construção do objeto da classe RuleBase
RuleBase MinhaBase(2,6);

MinhaBase.AddRule(2,0,3);  // Definição da regra no.  1
MinhaBase.AddRule(2,1,3);  // Definição da regra no.  2
MinhaBase.AddRule(2,2,4);  // Definição da regra no.  3
MinhaBase.AddRule(0,0,3);  // Definição da regra no.  4
MinhaBase.AddRule(0,1,3);  // Definição da regra no.  5
MinhaBase.AddRule(0,2,3);  // Definição da regra no.  6
//-------------------------------------
```

B.1.3 Classe *FuzSet*

A classe *FuzSet* é utilizada para implementação de conjuntos nebulosos e algumas operações relacionadas a conjuntos nebulosos. Os métodos estabelecidos nesta classe permitem a definição de conjuntos nebulosos com função de pertinência triangular ou trapezoidal. Outros tipos de funções de pertinência podem ser obtidos através do desenvolvimento de novos métodos para a classe.

Dados Membros

- int Card — É a cardinalidade do universo de discurso.
- float *value — Este vetor armazena os valores discretos que compõem o universo de discurso.
- float *v — Este vetor armazena os valores discretos da função de pertinência do conjunto nebuloso.
- float u_lower — Limite inferior do intervalo que define o universo de discurso do conjunto nebuloso.
- float u_upper — Limite superior do intervalo que define o universo de discurso do conjunto nebuloso.
- float lower — Esta variável indica o limite inferior do intervalo no qual a função de pertinência assume valores não nulos.
- float upper — Esta variável indica o limite superior do intervalo no qual a função de pertinência assume valores não nulos.
- char *name — Variável que armazena o nome do conjunto nebuloso.

Métodos (Funções-Membro)

FuzSet() — Construtor padrão da classe utilizado para gerar uma matriz de objetos do tipo *FuzSet*. Este construtor não aloca memória para os objetos criados, sendo que a alocação deve ser feita separadamente para cada objeto do vetor ou matriz utilizando a função *Init*.

FuzSet(float l, float u, int card) – Este construtor inicializa e aloca memória para um objeto do tipo *FuzSet* com base nos parâmetros que definem o universo de discurso do conjunto nebuloso.

FuzSet(Universe *X) – Este construtor inicializa e aloca memória para um objeto do tipo *FuzSet* baseado no objeto *Universe* $*X$ que define o universo de discurso do conjunto nebuloso.

~FuzSet() – Destrutor padrão da classe que libera toda memória alocada para o objeto.

void Init(Universe *X) – Este método é utilizado para alocação de memória para um objeto do tipo *FuzSet*. O parâmetro *Universe* $*X$ define o universo de discurso do conjunto nebuloso. Geralmente esta função é utilizada para alocação de memória para objetos pertencentes a uma matriz ou vetor e que utilizaram o construtor padrão.

char Triangle(float tl, float tm, float tu) – Utilizado para definição de um conjunto nebuloso com função de pertinência triangular, onde *tl* é o ponto inferior da base , *tm* é o ponto onde a função de pertinência é igual a um, e *tu* é o ponto superior da base (ver figura B.1).

char Triangle(char *str, float tl, float tm, float tu) – Idem ao método anterior, entretanto este acrescenta o nome do conjunto nebuloso (***str**).

char Triangle(float center, float spread) – Utilizado para definição de um conjunto nebuloso com função de pertinência triangular isóceles, onde *center* é o ponto central da base onde a função de pertinência é igual a um, e *spread* é a largura da base (ver figura B.1).

char Triangle(char *str, float center, float spread) – Idem ao método anterior, entretanto este acrescenta o nome do conjunto nebuloso (***str**).

char Trapezoid(float t1, float t2, float t3, float t4) – Este método é utilizado para definição de conjuntos nebuloso com forma trapezoidal. Os parâmetros *t1*, *t2*, *t3* e *t4* definem o trapézio utilizado para gerar o conjunto nebuloso (ver figura B.2).

char Trapezoid(char *str, float t1, float t2, float t3, float t4) –Idem ao método anterior, entretanto este acrescenta o nome do conjunto nebuloso (***str**).

void MaxProd(FuzSet *x, float fire) – Este método faz a multiplicação conjunto nebuloso (**FuzSet *x**) pela constante **fire**. O resultado é armazenado em **FuzSet *x**.

void MaxMin(FuzSet *x, float fire) – Este método realiza a operação de *MINIMO* entre o conjunto nebuloso (**FuzSet *x**) e a constante **fire**. O resultado é armazenado em **FuzSet *x**.

float Defuzzyfication_COG() – Retorna o resultado da defuzzificação do conjunto nebuloso utilizando o método do centro de gravidade.

float Max(float a, float b) – Retorna o máximo entre a e b.

float Min(float a, float b) – Retorna o mínimo entre a e b.

void Print() – Imprime os principais atributos do conjunto nebuloso, como limite inferior e os valores discretos do grau de pertinência (válido apenas para aplicações *DOS* ou Console).

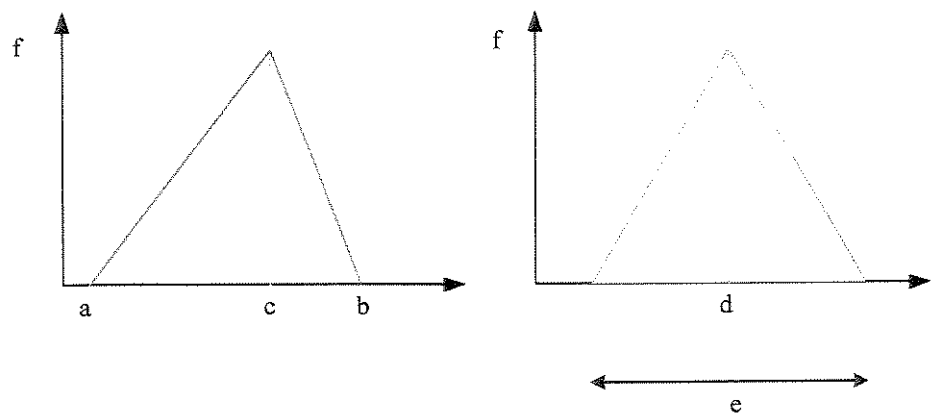


Figura B.1: Funções de pertinência triangulares.

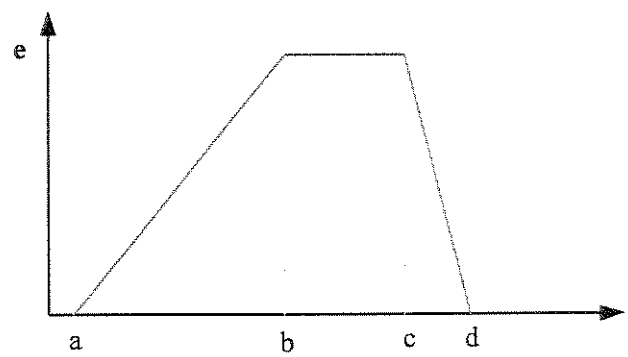


Figura B.2: Função de pertinência trapezoidal.

B.1.4 Classe *FuzzySystem*

A classe *FuzzySystem* é a principal componente da biblioteca *FuzzySystem*. Um objeto desta classe possui os atributos e métodos básicos para implementação de sistemas nebulosos.

Existem dois construtores para esta classe: um para sistemas de *Mandani* e outro para sistemas de *Takagi-Sugeno*. As características básicas dos sistemas nebulosos suportados por esta classe são:

Tipo de Sistema Nebuloso – *Mamdani* ou *Takagi-Sugeno*;

Número de Variáveis de entrada – até três entradas;

Interpretação das Regras – *Mamdani* ou *Larsen* (*default*);

T-Norma – Mínimo (*default*) ou Produto Algébrico;

Defuzzificação – Centro de Gravidade;

Agregação – Máximo;

Dados Membros

- int NSetIn1 – Número de conjuntos nebulosos definidos para a variável de entrada no. 1.
- int NSetIn2 – Número de conjuntos nebulosos definidos para a variável de entrada no. 2.
- int NSetIn3 – Número de conjuntos nebulosos definidos para a variável de entrada no. 3.
- int NSetOut – Número de conjuntos nebulosos definidos para a variável de saída. Para o caso de sistemas de *Takagi-Sugeno*, $NSetOut = 0$.
- int NRules – Número total de regras nebulosas.
- int NAnt – Quantidade de antecedentes das regras nebulosas.
- int TNormType – Tipo de *T-Norma* utilizada, podendo assumir dois valores: *MINIMUM* para o operador Mínimo, ou *ALGEBRAIC_PROD* para o operador Produto Algébrico.

- int ImpType – Esta variável indica o modo em que as regras são interpretadas. *ImpType* = *LARSEN* para implicação de *Larsen* ou *ImpType* = *MAMDANI* para implicação de *Mamdani*.
- char FISType – Esta variável especifica o tipo de de sistema nebuloso: *MAMDANI* (Sistemas de *Mamdani*) ou *TAKAGI_SUGENO* (Sistemas de *Takagi & Sugeno*).
- RuleBase *Rule – Esta variável armazena um objeto do tipo *RuleBase*, que é a base de regra do sistema nebuloso.
- FuzSet *Input – Esta matriz é utilizada para armazenar todos os conjuntos nebulosos das variáveis de entrada.
- FuzSet *Output – Este vetor é utilizado para armazenar os conjuntos nebulosos da variável de saída. Caso o sistema nebuloso seja do tipo *Takagi-Sugeno*, não há alocação de memória para esta variável na construção do objeto.
- float **Coef – Esta matriz armazena os coeficientes dos conseqüentes das regras de *Takagi-Sugeno*. Caso o sistema nebuloso seja do tipo *Mamdani*, não há alocação de memória para esta variável na construção do objeto.

Métodos (Funções-Membro)

FuzzySystem(char fistype, int nant, int nr, int nSetIn1, int nSetIn2, int nSetIn3, int nSetOut) – Este construtor é utilizado para inicializar e alocar memória para objetos que implementem sistemas nebulosos de *Mamdani*. A seguir são descritos os argumentos utilizados por este construtor.

- **fistype** = *MAMDANI*.
- *nant* - Número de antecedentes.
- *nr* - Número total de regras nebulosas.
- *nSetIn*1 - Número de conjuntos nebulosos para a variável de entrada 1.
- *nSetIn*2 - Número de conjuntos nebulosos para a variável de entrada 2.
- *nSetIn*3 - Número de conjuntos nebulosos para a variável de entrada 3.
- *nSetOut* - Número de conjuntos nebulosos para a variável de saída.

FuzzySystem(char fistype, int nant, int nr, int nSetIn1, int nSetIn2, int nSetIn3) – Este construtor é utilizado para inicializar e alocar memória para objetos que implementem sistemas nebulosos de *Takagi-Sugeno*. A seguir são descritos os argumentos utilizados por este construtor.

- $fistype = TAKAGI\_SUGENO$.
- $nant$ - Número de antecedentes.
- nr - Número total de regras nebulosas.
- $nSetIn1$ - Número de conjuntos nebulosos para a variável de entrada 1.
- $nSetIn2$ - Número de conjuntos nebulosos para a variável de entrada 2.
- $nSetIn3$ - Número de conjuntos nebulosos para a variável de entrada 3.

~FuzzySystem() – Destrutor padrão da classe que libera toda memória alocada para o objeto.

float Mamdani(float x) – Este método retorna o valor discreto da saída do sistema de *Mamdani* para a entrada x (uma variável de entrada).

float Mamdani(float x1, float x2) – Este método retorna o valor discreto da saída do sistema de *Mamdani* para as entradas $x1$ e $x2$ (duas variáveis de entrada).

float Mamdani(float x1, float x2, float x3) – Este método retorna o valor discreto da saída do sistema de *Mamdani* para as entradas $x1$, $x2$ e $x3$ (três variáveis de entrada).

float Takagi_Sugeno(float x) – Este método retorna o valor discreto da saída do sistema de *Takagi-Sugeno* para a entrada x (uma variável de entrada).

float Takagi_Sugeno(float x1, float x2) – Este método retorna o valor discreto da saída do sistema de *Takagi-Sugeno* para as entradas $x1$ e $x2$ (duas variáveis de entrada).

float Takagi_Sugeno(float x1, float x2, float x3) – Este método retorna o valor discreto da saída do sistema de *Takagi-Sugeno* para as entradas $x1$, $x2$ e $x3$ (três variáveis de entrada).

float TNorm(float a, float b) – Retorna a *T-Norma* entre os valores discretos a e b. O tipo de *T-Norma* utilizada é especificada através do método $SetTNorm()$.

float TNorm(float a, float b, float c) – Retorna a *T-Norma* entre os valores discretos a, b e c. O tipo de *T-Norma* utilizada é especificada através do método $SetTNorm()$.

char SetImplication(char type) – Este método é utilizado para especificar o modo em que as regras nebulosas serão interpretadas. A variável *type* pode assumir dois valores: *MAMDANI* ou *LARSEN*, que representam a implicação de *Mamdani* ou *Larsen*, respectivamente.

char SetTNorm(int type) – Este método é utilizado para especificar o operador utilizado na *T-Norma*. A variável *type* pode assumir dois valores: *MINIMUM* ou *ALGEBRAIC_PROD*, que representam o operador Mínimo ou Produto Algébrico, respectivamente.

B.1.5 Exemplo

A seguir, encontra-se listado um programa exemplo em linguagem *C++* que utiliza a biblioteca *FuzzySystem* para implementação de um sistema de *Takagi-Sugeno*. A implementação de sistemas de *Mamdani* é realizada de forma similar ao exemplo apresentado.

```
//-----------------------------------------------
#include "fuzzy.h"
#include <dos.h>
#include <stdio.h>
#include <conio.h>

//-----------------------------------------------
int main() {
     register int i,j;
     float x1, x2, y;

     //-----------------------------------------
     // Cria sistema de Takagi-Sugeno:
     // nO.  de entradas :2
     // número de regras:  9
     // quantidade de conjuntos nebulosos para entrada 1:  2
     // quantidade de conjuntos nebulosos para entrada 2:  2
     //-----------------------------------------

     FuzzySystem *FLC;
     FLC = new FuzzySystem(TAKAGI_SUGENO,2,4,2,2,0);

     SetImplication(MAMDANI); // Define modo de interpretação
                              // das regras.
     SetTNorm(ALGEBRAIC_PROD); // Define T-Norma.
     //-----------------------------------------
     // Cria universo de discurso para duas variáveis de entrada.
     //-----------------------------------------
     Universe Erro(101,-1.0,1.0);
     Universe DErro(101,-5.0,5.0);

     //-----------------------------------------
     // Inicializa conjuntos nebuloso.
     //-----------------------------------------

     // Entrada 1
     FLC->Input[0][0].Init(&Erro);
     FLC->Input[0][1].Init(&Erro);

     FLC->Input[0][0].Triangle(-1.0,4.0);
     FLC->Input[0][1].Triangle( 1.0,4.0);


     // Entrada 2
     FLC->Input[1][0].Init(&DErro);
     FLC->Input[1][1].Init(&DErro);
```

```
        FLC->Input[1][0].Triangle(-5.0,20.0);
        FLC->Input[1][1].Triangle( 5.0,20.0);

        //-----------------------------------------------
        // Para sistemas de Takagi-Sugeno, a base de regras deve
        // cobrir todo o espaço de entrada.  O terceiro argumento
        // é nulo, pois para sistemas de Takagi-Sugeno, os consequentes
        // das regras não estão associados a conjuntos nebulosos.
        //-----------------------------------------------
        FLC->Rules->AddRule(0,0,0);
        FLC->Rules->AddRule(0,1,0);

        FLC->Rules->AddRule(1,0,0);
        FLC->Rules->AddRule(1,1,0);

        //-----------------------------------------------
        // Inicializa os consequentes das regras de Takagi-Sugeno.
        //-----------------------------------------------
        // regra 1
        FLC->Coef[0][0] = 0.2;
        FLC->Coef[0][1] = 0.5;
        FLC->Coef[0][2] = 0.1;

        // regra 2
        FLC->Coef[1][0] = 0.6;
        FLC->Coef[1][1] = 0.3;
        FLC->Coef[1][2] = 0.8;

        // regra 3
        FLC->Coef[2][0] = 0.1;
        FLC->Coef[2][1] = 0.3;
        FLC->Coef[2][2] = 0.91;

        // regra 4
        FLC->Coef[3][0] = 0.24;
        FLC->Coef[3][1] = 0.35;
        FLC->Coef[3][2] = 0.14;

        // Loop principal.
        for(i=0; i<50; i++){
           for(j=0; j<50; j++){
              x1 = -1.0 + 2.0*(double)(i)/49.0;
              x2 = -5.0 + 10.0*(double)(j)/49.0;

              // Calcula saída do sistema nebuloso para
              // as entradas x1 e x2.
              y = FLC->Takagi_Sugeno(x1,x2);

              // Imprime saída do sistema nebuloso em função
              // das entradas.
              printf ("Entradas do sistema nebuloso:
                 x1 = %.2f, x2 = %.2 \n",
                 x1,x2);
              printf("Saída do sistema nebuloso:
                 y = %.2 \ n",y);
           }
        }
        // libera memória alocada para o sistema nebuloso.
        delete FLC;
        return 0;
}
```

B.2 A Biblioteca *Genetic*

O propósito da biblioteca *Genetic* é prover um conjunto de estruturas e métodos para implementação de Algoritmos Genéticos. O algoritmo genético tomado como referência na elaboração desta biblioteca é o Algoritmo Genético Modificado descrito no Capítulo 3. As características deste algoritmo são: codificação em ponto flutuante, operadores *crossover* uniforme e mutação não-linear, e geração de subpopulações. A biblioteca *Genetic* é composta basicamente de duas classes: a classe *Individual* e a classe *Population*

B.2.1 A Classe *Individual*

A classe *Individual* define objetos que possuem os atributos relevantes de um indivíduo de um algoritmo genético, como cadeia cromossômica, grau de adaptação e tamanho do cromossomo. Além destes atributos, pode-se adicionar outros dados membros relativos à aplicação abordada no problema de otimização. Por exemplo, nos problemas de controle descritos nesta tese acrescentaram-se dados para medição do desempenho do controlador.

Dados Membros

- int ChromossomeSize – Esta variável indica o tamanho do cromossomo do indivíduo.
- float *Chromossome – Este vetor armazena os valores dos genes da cadeia cromossômica.
- float Fitness – Grau de adaptação do indivíduo.

Métodos (Funções-Membro)

Individual() – Este construtor inicializa e aloca memória para objetos da classe com base no tamanho do cromossomo, que é definido pela macro *CROM_SIZE*.

~Individual() – Destrutor da classe, cuja função é liberar a memória alocada para o objeto.

B.2.2 A Classe *Population*

A classe *Population* tem sua estrutura formada por objetos da classe *Individual* e parâmetros relacionados a algoritmos genéticos. Um objeto da classe *Population* engloba todos os dados e operações para simulação do algoritmo genético descrito no Capítulo 3.

Dados Membros

- `short **Mark` – Esta matriz é utilizada para indicar os indivíduos que já foram escolhidos no processo de seleção. Cada elemento desta matriz está relacionado a um indivíduo das subpopulações.

- `int NTypes` – Esta variável indica em quantos grupos o cromossomo está dividido em relação ao intervalo de valores que os genes podem assumir. Para cada grupo de genes existe um intervalo que limita o universo de valores reais dos genes do grupo.

- `char *PType` – Este vetor armazena caracteres que indicam como os grupos de genes serão inicializados. O elemento $PType[0]$ refere-se ao grupo 1, $PType[1]$ refere-se ao grupo 2 e assim por diante. Este vetor é inicializado através da função **SetParRange()**.

- `int *NPar` – Este vetor armazena o número de genes contidos em cada grupo. O número de elementos deste vetor depende da quantidade de grupos de genes.

- `float *P_Max` – Este vetor armazena os valores dos limites superiores dos intervalos que definem os grupos de genes.

- `float *P_Min` – Este vetor armazena os valores dos limites inferiores dos intervalos que definem os grupos de genes.

- `float *Range` – Este vetor armazena os tamanhos dos intervalos associados aos grupos de genes, ou seja, $Range[i] = | P\_Max[i] - P\_Min[i] |$, para $i = 1, \ldots, NTypes$.

- `Individual *Pop` – Este vetor armazena os dados dos indivíduos que formam a população principal do algoritmo genético.

- `Individual **SubPop` – Esta matriz armazena os dados dos indivíduos que formam as subpopulações do algoritmo genético. O número de subpopulações é definido na construção do objeto da classe *Population*.

- `Individual Best` – Esta variável armazena os dados do melhor indivíduo obtido na etapa de seleção do algoritmo genético.

- Individual Roullete – Esta variável armazena os dados do indivíduo obtido através do algoritmo *Roullete Wheel*.
- int PopSize – Esta variável indica o tamanho da população ou subpopulações do algoritmo genético.
- int NumberSubPop – Esta variável indica a quantidade de subpopulações do algoritmo genético.

Métodos (Funções-Membro)

Population(int tPop, int nSP) – Este construtor inicializa e aloca memória para objetos da classe com base no tamanho da população ($tPop$) e no número de subpopulações (nSP).

~Population() – Destrutor da classe, cuja função é liberar a memória alocada para o objeto.

void SetParRange(int ind, int npar, double min, double max, char type) – Este método é utilizado para inicialização dos parâmetros de um grupo de genes. O argumento *ind* refere-se ao índice que designa o grupo de genes. O parâmetro *npar* indica o número de genes contido no grupo. O intervalo de valores que os genes do grupo podem assumir é limitado pelos parâmetros min e max, que definem o intervalo $[min, max]$. O parâmetro *type* indica a forma como os genes do grupo serão inicializados no algoritmo genético, sendo 'c' para que todos os genes sejam inicializados com um valor constante igual ao ponto médio do intervalo $[min, max]$, e 'r' para que os genes sejam inicializados aleatoriamente.

void InitPopulation(void) – Inicializa os genes dos cromossomos da população principal. Este método deve ser executado antes do processo de evolução do algoritmo genético. A forma como os genes são inicializados é definida através da função **SetParRange()**.

void RoulleteWheelPop(void) – Este método aplica o algoritmo *Roullete Wheel* na população principal e armazena o indivíduo sorteado na variável membro *Roullete*.

void RoulleteWheelSubPop(void) – Este método aplica o algoritmo *Roullete Wheel* nas subpopulações e armazena o indivíduo sorteado na variável membro *Roullete*.

void ExecuteCrossoverSubPop(int indice_sp) – Realiza o *crossover* uniforme entre os indivíduos da subpopulação designada por *indice_sp*.

void CrossoverSubPops(void) – Este método aplica o *crossover* uniforme em todos indivíduos das subpopulações $sp2, sp3, \ldots spn$. A subpopulação $sp1$ e não sofre *crossover* nem mutação e é uma cópia da população principal. A probabilidade de um casal de indivíduos sofrer *crossover* é dada pela taxa de *crossover*, a qual é definida pela macro *PC*.

void GenerateSubPopByRoullete(int indice_sp) – Este método aplica o algoritmo *Roullete Wheel* na população principal para formar a subpopulação designada por *indice_sp*.

void BestPop(void) – Seleciona o indivíduo de maior *fitness* da população principal e o armazena na variável *Best*. O indivíduo selecionado é registrado para que o mesmo não seja selecionado duas vezes, por exemplo, se esta função for executada duas vezes, será feita a seleção dos dois melhores indivíduos da população principal.

void BestSubPops(void) – Seleciona o indivíduo de maior *fitness* das subpopulações e o armazena na variável *Best*. O indivíduo selecionado é registrado para que o mesmo não seja selecionado duas vezes.

void SubPopBestnOthers(int indice_sp1, int indice_sp2) – Este método realiza uma combinação par-a-par entre o melhor indivíduo e todos os indivíduos da população principal. O resultado deste arranjo de indivíduos forma as subpopulações designadas por *indice_sp*1 e *indice_sp*2.

void GenerateSubPops(void) – Este método faz um arranjo entre os indivíduos da população principal para gerar as subpopulações. A subpopulação $sp1$ é uma cópia da população principal, as subpopulações $sp2$ e $sp3$ são uma combinação do melhor indivíduo e os indivíduos a população principal, e as subpopulações $sp4, \ldots, spn$ são geradas através do algoritmo *Roullete Wheel*.

void ExecuteMutationSubPop(int indice_sp, int gen) – Executa o operador mutação nos indivíduos da subpopulação designada por *indice_sp*. O parâmetro *gen* deve conter o número da geração na qual a função foi chamada.

void MutationSubPops(int gen) – Este método aplica o operador mutação em todos indivíduos das subpopulações, exceto a subpopulação $sp1$. O parâmetro *gen* deve conter o número da geração na qual a função foi chamada. A probabilidade de um indivíduo sofrer mutação é dada pela taxa de mutação, a qual é definida pela macro *PM*. Outros parâmetros relativos ao operador mutação (ver **item 3.4.1**) são definidos pelas macros *NG* (número total de gerações) e *RANGE_MUT* (define a região de busca).

void Selection(void) – Este método avalia o *fitness* de todos indivíduos das subpopulações e seleciona os indivíduos que passarão para próxima geração. A macro $NBEST$ indica o número de melhores indivíduos que sempre passarão para a geração seguinte.

void EvalFitnessSubPops(void) – Este método calcula o *fitness* de todos indivíduos das subpopulações. A função de avaliação do *fitness* deve ser definida neste método.

void EvalFitnessPop(void) – Este método calcula o *fitness* de todos indivíduos da população principal. A função de avaliação do *fitness* deve ser definida neste método.

B.2.3 Exemplo

A seguir é apresentado o código fonte em C^{++} de um aplicativo baseado na biblioteca *Genetic*. O programa utiliza o algoritmo genético proposto para maximizar uma função de duas variáveis. A função de *fitness* é dada pela própria função a ser maximizada:

$$F(x,y) = 0.5 + \frac{\sin(6\pi\sqrt{(x-0.4)^2+(y-0.2)^2})}{6\pi\sqrt{(x-0.4)^2+(y-0.2)^2}}. \tag{B.1}$$

Neste caso, a função a ser maximizada deve ser definida dentro das funções *EvalFitnessPop* e EvalFitnessSubPop, conforme o exemplo:

```
void Population::EvalFitnessSubPops(void) {
    register int i, j;
    double d;
    // calcula fitness das subpopulações
    for(i=0; i<NumberSubPop; i++)
    {
        for(j=0; j<PopSize; j++)
        {
            d = sqrt( pow(SubPop[i][j].DoubleCromossome[0]-0.4,2.0)+
                pow(SubPop[i][j].DoubleCromossome[1]-0.2,2.0) );
            if(d > 0) SubPop[i][j].Fitness = 0.5 + sin(6*M_PI*d)/(6*M_PI*d);
            else      SubPop[i][j].Fitness = 1.5;
        }
    }
}
void Population::EvalFitnessPop(void) {
    register int i;
    double d;
    // Calcula o fitness de todos os indivíduos
    for(i=0; i<PopSize; i++)
    {
        d = sqrt( pow(Pop[i].DoubleCromossome[0]-0.4,2.0)+
            pow(Pop[i].DoubleCromossome[1]-0.2,2.0) );
```

```
        if(d > 0) Pop[i].Fitness = 0.5 + sin(6*M_PI*d)/(6*M_PI*d);
        else      Pop[i].Fitness = 1.5;
    }
}
```

A seguir é listado o código do programa exemplo que utiliza a biblioteca *Genetic*. A listagem completa dos arquivos que compõem as bibliotecas *Individual* e *Genetic* encontra-se no Anexo C.

```
                                    // Taxa de mutação.
                                    // Taxa de crossover.
                                    // Define tamanho da região de busca para mutação.
                                    // Quantidade de melhores indivíduos a serem preservados.
                                    // Número máximo de gerações.
                                    // Quantidade de grupos de genes.
                                    // Tamanho do cromossomo.

//-------------------------------------------------
int main() {
     int n;
     /*-----------------------------------------
     Cria objeto do tipo Poulation com 4 subpopulações contendo
     30 elementos cada.
     -----------------------------------------*/

     Population *MyPop;
     MyPop = new Population(30,4);

     /*-----------------------------------------
     Define grupo de genes:  Intervalo = [0.1] e genes são
     inicializados de forma aleatória.
     -----------------------------------------*/

     MyPop->SetParRange(0, 2, 0.0, 1.0, 'r');
     initmyrand();
     MyPop->InitPopulation();

     // Avalia o fitness da primeira população.
     MyPop->EvalFitnessPop();
     // Laço principal:  processo de evolução.
     for(n=0;n<NG;n++){
        MyPop->GenerateSubPops();  //Gera subpopulações.
        MyPop->CrossoverSubPops(); //Executa crossover nas subpopulações.
        MyPop->MutationSubPops(n); //Executa mutacao nas subpopulações.
        MyPop->Selection();        //Seleção.
        // Imprime dados do melhor indivíduo.
        printf ("Melhor x = %.6f, y = %.6f,
           Fitness = %.6f \n",
           MyPop->Best.DoubleCromossome[0],
           MyPop->Best.DoubleCromossome[1],
           MyPop->Best.Fitness);
     }
     getch();
     return 0;
}
```

Apêndice C

Código Fonte

C.1 Arquivos de Cabeçalho

C.1.1 hardfun.h

```
/*-------------------------------------------------------------------------------

    LISTAGEM DAS FUNÇÕES DE ACESSO A HARDWARE E PROGRAMACAO DE INTERRUPÇÕES.

    AUTOR: MÁRCIO A. T. DE SOUSA.

    OUTUBRO DE 1999, LSMR, DSCE, FEEC, UNICAMP.

    OBS: PARA QUE A ROTINA DE INTERRUPÇÃO FUNCIONE CORRETAMENTE A OPÇÃO
    STACK OVERFLOW DO COMPILADOR DEVE SER DESLIGADA.

-------------------------------------------------------------------------------*/

#include <dos.h>
#include <time.h>
#include <conio.h>
#include <stdio.h>

#define INTR 0X1C    // Tick´s do relógio de interrupção

#ifdef __cplusplus
     #define __CPPARGS ...
#else
     #define __CPPARGS
#endif

void CounterOffSet(void);
float ReadCounter(void);
unsigned int ReadCounterPulses(void);
void DA(float u);
void SetNewCounterTime(void);
void RestoreCounterTime(void);
void InitTimer(void);
void FinishTimer(void);
void interrupt ( *oldhandler)(__CPPARGS);

unsigned long tickcount = 0;
float da_value=0.0, position;

/*-------------------------------------------------------------------------------
Leitura da posição angular (graus).
-------------------------------------------------------------------------------*/
```

```
float ReadCounter(void)
{
    unsigned int MS8, LS8, value;
    float deg;

    MS8 = inportb(0x234);
    LS8 = inportb(0x235);
    value = LS8 + (MS8 & 0x00FF)*256;
    deg = (int) (value-297) * 180.0 / (2000.0);
    return deg;
}

/*-------------------------------------------------------------------------------
Leitura do contador (pulsos).
-------------------------------------------------------------------------------*/
unsigned int ReadCounterPulses(void)
{
    unsigned int MS8, LS8, value;

    MS8 = inportb(0x234);
    LS8 = inportb(0x235);
    value = LS8 + (MS8 & 0x00FF)*256;
    return value;
}

/*-------------------------------------------------------------------------------
Esta função inicializa o contador.
-------------------------------------------------------------------------------*/
void CounterOffSet(void)
{
    unsigned long old;

    InitTimer();
    old = 0;
    tickcount = 0;
    DA(-1.0);
    while (tickcount < (int) (250.0/TS) )
    {
        da_value = -1.0;
        outportb(0x222,0x08);  // habilta reset dos contadores
    }
    FinishTimer();
    DA(0.0);
    outportb(0x222,0x00);      // desabilta reset dos contadores
}

/*-------------------------------------------------------------------------------
Rotina de Interrupção.
-------------------------------------------------------------------------------*/
void interrupt handler(__CPPARGS)
{
    disable();
    tickcount++;               // incrementa contador de período de amostragem
    position = ReadCounter(); // lê dados do encoder
    DA(da_value);              // escreve dados no conversor D/A
    enable();
}

/*-------------------------------------------------------------------------------
Esta função escreve dados no conversor D/A.
-------------------------------------------------------------------------------*/
void DA(float u)
{
  unsigned int MS8,LS8;
  int  da;
  u = -1.0*u;
```

```
  if(u > 1.0)  u = 1.0;
  else if(u < -1.0) u = -1.0;
  da = 2048 + (int)(2047*u);
  LS8=  da & 0x00FF;
  MS8=  (da/256) & 0x000F;
  outportb(0x220,LS8);
  outportb(0x22C,MS8);
}
/*-------------------------------------------------------------------------
Esta função programa o timer do sistema.
-------------------------------------------------------------------------*/
void SetNewCounterTime(void)
{
    int MSB, LSB, n;
    n = (int)(1193*TS);
    MSB = (0xFF00 & n) / 256;
    LSB = 0x00FF & n;
    outportb(0x43,0x34);
    outportb(0x40,LSB);
    outportb(0x40,MSB);
}
/*-------------------------------------------------------------------------
Esta função restaura a programação padrão do timer do sistema.
-------------------------------------------------------------------------*/
void RestoreCounterTime(void)
{
    outportb(0x43,0x34);
    outportb(0x40,0xFF);
    outportb(0x40,0xFF);
}
/*-------------------------------------------------------------------------
Esta função é utilizada na intalação da rotina de interrupção.
-------------------------------------------------------------------------*/
void InitTimer(void)
{
// salva o vetor de interrupção antigo
    oldhandler = getvect(INTR);
// instala o novo handler de interrupção
    setvect(INTR, handler);
    SetNewCounterTime();
}
/*-------------------------------------------------------------------------
Esta função finaliza o sistema de amostragem e restaura a rotina de interrupção
gravada.
-------------------------------------------------------------------------*/
void FinishTimer(void)
{
    RestoreCounterTime();
    // reseta o antigo handler de interrupção
    setvect(INTR, oldhandler);
}
```

C.1.2 fuzzysys.h

```
#ifndef FuzzySystemH
#define FuzzySystemH

#include <math.h>
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <conio.h>

#include "Universe.h"
#include "Fuzset.h"
#include "RuleBase.h"

#define MINIMUM 0
#define ALGEBRAIC_PROD 1
#define LARSEN 2
#define MAMDANI 3
#define TAKAGI_SUGENO 4

//---------------------------------------------------------------------------
// Classe FuzzySystem
//
// Esta classe define o sistema fuzzy de inferência. Há dois tipos disponíveis:
// Sistema Mamdani e Sistema Takagi-Sugeno de inferência Fuzzy
// Esta Classe Suporta até três entradas e tem somente uma saída.
//
// T-Norm padrão: Min
// Agregação padrão: Max
// Função de implicação padrão: produto (LARSEN)
//
// Márcio A. T. de Sousa 06/22/99.
//---------------------------------------------------------------------------
class FuzzySystem
{
private:
    int NSetIn1;        // Número de conjuntos fuzzy definidos para entrada 1.
    int NSetIn2;        // Número de conjuntos fuzzy definidos para entrada 2.
    int NSetIn3;        // Número de conjuntos fuzzy definidos para entrada 3.
    int NSetOut;        // Número de conjuntos fuzzy definidos para a saída.
    int NRules;         // Número de regras.
    int NAnt;           // Número de antecedentes.
    int TNormType;      // Tipo de T-Norma.
    int ImpType;        // Tipo de função implicação.
    char FISType;       // Tipo de sistema fuzzy: Mamdani ou Takagi-Sugeno.

public:
    RuleBase *Rules;
    FuzSet **Input;
    FuzSet *Output;
    float **Coef;

    FuzzySystem(char fistype, int nant, int nr, int nSetIn1, int nSetIn2,
                int nSetIn3, int nSetOut);
    FuzzySystem(char fistype, int nant, int nr, int nSetIn1, int nSetIn2,
                int nSetIn3);
    ~FuzzySystem();

    float Mamdani(float x);
    float Mamdani(float x1, float x2);
    float Mamdani(float x1, float x2, float x3);

    float Takagi_Sugeno(float x);
    float Takagi_Sugeno(float x1, float x2);
    float Takagi_Sugeno(float x1, float x2, float x3);

    float TNorm(float a, float b);
    float TNorm(float a, float b, float c);

    char SetImplication(char *type);
    char SetTNorm(char *type);
};
#endif\\
```

C.1.3 fuzset.h

```
#ifndef FuzsetH
#define FuzsetH
//---------------------------------------------------------------------------
// Classe FuzSet
//
// Esta Classe define um conjunto fuzzy.
//
// Márcio A. T. de Sousa 06/21/99.
//---------------------------------------------------------------------------
class FuzSet{
public:
    int Card;           // Cardinalidade.
    float *value;       // Este vetor armazena os valores discretos do
                        // universo de discuro.
    float *y;           // Este vetor armazena os valores dos graus de
                        // pertinência.

    float u_lower;      // Limitante Inferior.
    float u_upper;      // Limitante Superior.
    float lower;
    float upper;
    char *name;         // Nome do conjunto Fuzzy.

    FuzSet();
    FuzSet(float l, float u, int card);
    FuzSet(Universe *X);
    ~FuzSet();

    void Init(Universe *X);

    char Triangle(float tl, float tm, float tu);
    char Triangle(float center, float spread);

    char Triangle(char *str, float tl, float tm, float tu);
    char Triangle(char *str, float center, float spread);

    char Trapezoid(float t1, float t2, float t3, float t4);
    char Trapezoid(char *str, float t1, float t2, float t3, float t4);

    void MaxProd(FuzSet *x, float fire);
    void MaxMin(FuzSet *x, float fire);
    float Defuzzyfication_COG();
    float Max(float a, float b);
    float Min(float a, float b);
    void Print();
};
#endif\\
```

C.1.4 rulebase.h

```
#ifndef RuleBaseH
#define RuleBaseH
//---------------------------------------------------------------------------
// Classe RuleBase
//
// Esta classe define a base de dados de regras.
//
// Márcio A. T. de Sousa 06/21/99.
//---------------------------------------------------------------------------
class RuleBase
{
public:
    int NAntecedents;       // Número de antecedentes.
    int NRules;             // Número de regras.
    int Count;              // Contador que indica o número de regras
```

```
                                // inicializadas.
    int **Ant;                   // Variável do Antecedente
    int *Cons;                  // Variável do Consequente.(sistemas de só uma saída)
    RuleBase(int nant, int nr);
    ~RuleBase();
    char AddRule(int a1, int c1);
    char AddRule(int a1, int a2, int c1);
    char AddRule(int a1, int a2, int a3,int c1);
};
#endif\\
```

C.1.5 universe.h

```
#ifndef UniverseH
#define UniverseH

//---------------------------------------------------------------------------
// Classe Universe
//
// Esta classe define o universo discreto de discurso a ser utilizado na
// inicialização de um conjunto fuzzy, na classe FuzSet.
//
// Márcio A. T. de Sousa 06/21/99.
//---------------------------------------------------------------------------
class Universe
{
public:
    int Card;           // Define a cardinalidade.
    float *value;       // Este vetor armazena os valores discretos do
                        // universo de discurso.
    float lower;        // Limitante Inferior.
    float upper;        // Limitante Superior.

    Universe(int card, float l, float u);
    ~Universe();
    void Print();
    friend class FuzSet;
};
#endif
```

C.1.6 random.h

```
#include <time.h>
#include <math.h>
#include <sys\timeb.h>

long myrand();
long myrandom(long Max);
long myrandom(long Min , long Max);
float myrandomf(float Min, float Max);
void random_seq(long *vector,int Max, int n);
void initmyrand();
```

C.1.7 pop.h

```
#ifndef PopulationH
#define PopulationH

class Population
{
private:
    short **Vaux; // vetor auxiliar para verificar se algum indivíduo já foi
                  //escolhido.
public:
    char  *PType;    // tipo do parâmetro
    int   NTypes;  // número de tipos de parâmetros
```

```
    int   *NPar;      // armazena número de parâmetros por tipo
    float *P_Max;
    float *P_Min;
    float *Range;

    Individual *Pop;          // vetor de populações
    Individual **SubPop;      // vetor de subpopulações
    Individual Best, Roullete;
    int PopSize;
    int NumberSubPop;

    Population(int tPop, int nSP);
    ~Population(void);
    void SetParRange(int ind, int npar, float min, float max, char type);
    void InitPopulation(void);
    void RoulleteWheelPop(void);
    void RoulleteWheelSubPop(void);
    void ExecuteCrossoverSubPop(int indice_sp);
    void CrossoverSubPops(void);
    void GenerateSubPopByRoullete(int indice_sp);
    void BestSubPops(void);
    void BestPop(void);
    void SubPopBestnOthers(int indice_sp1, int indice_sp2);
    void GenerateSubPops(void);
    void ExecuteMutationSubPop(int indice_sp, int gen);
    void MutationSubPops(int gen);
    void Selection(void);
    void EvalFitnessSubPops(void);
    void EvalFitnessPop(void);
};
#endif
```

C.1.8 indiv.h

```
#ifndef IndividualH
#define IndividualH

#define CROM_SIZE 2

class Individual
{
public:
    int DoubleCromSize;
    float DoubleCromossome[CROM_SIZE];
    float Fitness;

    Individual(void);
    ~Individual(void);
};
#endif
```

C.2 Classes

C.2.1 fuzzysys.cpp

```
//---------------------------------------------------------------------------
// Classe FuzzySystem
//
// Esta classe define o sistema fuzzy de inferência. Há dois tipos disponíveis:
// Sistema Mamdani e Sistema Takagi-Sugeno de inferência Fuzzy
// Esta Classe Suporta até três entradas e tem somente uma saída.
//
// T-Norm padrão: Min
// Agregação padrão: Max
// Função de implicação padrão: produto (LARSEN)
//
// By Márcio A. T. de Sousa 06/22/99.
//---------------------------------------------------------------------------
#include "fuzzysys.h"
//---------------------------------------------------------------------------
// Class Constructor: For Mamdani Systems
//
// fistype =    "MAMDANI" (Comandante)
// nant:        número de regras antecedentes
// nr:          número total de regras
// nSetIn1:     Número de conjuntos fuzzy definidos para entrada 1.
// nSetIn2:     Número de conjuntos fuzzy definidos para entrada 2.
// nSetIn3:     Número de conjuntos fuzzy definidos para entrada 3.
// nSetOut:     Número de conjuntos fuzzy definidos para a saída.
//---------------------------------------------------------------------------
FuzzySystem::FuzzySystem(char fistype, int nant, int nr, int nSetIn1,
                         int nSetIn2, int nSetIn3, int nSetOut)
{
    NSetIn1 = nSetIn1;
    NSetIn2 = nSetIn2;
    NSetIn3 = nSetIn3;
    NSetOut = nSetOut;
    NRules = nr;
    NAnt = nant;

    if(fistype!=MAMDANI)
    {
        printf("\nError na inicialização do FIS: número de parâmetros em /
                desacordo com o tipo de FIS");
        getch();
        exit(0);
    }
    FISType = fistype;
    TNormType = MINIMUM;
    ImpType = LARSEN;

    Rules = new RuleBase(NAnt,NRules);

    if(NAnt == 1)
    {
        Input = new FuzSet*[1];
        Input[0] = new FuzSet[NSetIn1];
    }
    else if(NAnt == 2)
    {
        Input = new FuzSet*[NAnt];
        Input[0] = new FuzSet[NSetIn1];
        Input[1] = new FuzSet[NSetIn2];
```

```
    }
    else if(NAnt ==3)
    {
        Input = new FuzSet*[NAnt];
        Input[0] = new FuzSet[NSetIn1];
        Input[1] = new FuzSet[NSetIn2];
        Input[2] = new FuzSet[NSetIn3];
    }
    Output = new FuzSet[NSetOut];
}
//-------------------------------------------------------------------------
// Constructor de Classe: Para Sistemas Takagi-Sugeno
//
// fistype =    "TAKAGI_SUGENO" (Comandante)
// nant:        número de regras antecedentes
// nr:          número total de regras
// nSetIn1:     Número de conjuntos fuzzy definidos para entrada 1.
// nSetIn2:     Número de conjuntos fuzzy definidos para entrada 2.
// nSetIn3:     Número de conjuntos fuzzy definidos para entrada 3.
//-------------------------------------------------------------------------
FuzzySystem::FuzzySystem(char fistype, int nant, int nr, int nSetIn1,
                         int nSetIn2, int nSetIn3)
{
    NSetIn1 = nSetIn1;
    NSetIn2 = nSetIn2;
    NSetIn3 = nSetIn3;
    NSetOut = 0;
    NRules = nr;
    NAnt = nant;

    if(fistype!=TAKAGI_SUGENO)
    {
        printf("\nError na inicialização do FIS: número de parâmetros em /
                desacordo com o tipo de FIS");
        getch();
        exit(0);
    }

    FISType = fistype;
    TNormType = MINIMUM;

    Rules = new RuleBase(NAnt,NRules);

    if(NAnt == 1)
    {
        Input = new FuzSet*[NAnt];
        Input[0] = new FuzSet[NSetIn1];
    }
    else if(NAnt == 2)
    {
        Input = new FuzSet*[NAnt];
        Input[0] = new FuzSet[NSetIn1];
        Input[1] = new FuzSet[NSetIn2];
    }
    else if(NAnt ==3 )
    {
        Input = new FuzSet*[NAnt];
        Input[0] = new FuzSet[NSetIn1];
        Input[1] = new FuzSet[NSetIn2];
        Input[2] = new FuzSet[NSetIn3];
    }

    Coef = new float*[NRules];

    for(int i=0; i<NRules; i++)
        Coef[i] = new float[NAnt+1];
```

```
}
//-------------------------------------------------------------------------------
// Destrutor de Classe
//-------------------------------------------------------------------------------
FuzzySystem::~FuzzySystem()
{
    if(FISType == MAMDANI)
    {
        if(NAnt == 1)
        {
            delete[] Input[0];
            delete[] Input;
        }
        else if(NAnt == 2)
        {
            delete[] Input[0];
            delete[] Input[1];
            delete[] Input;
        }
        else if(NAnt == 3)
        {
            delete[] Input[0];
            delete[] Input[1];
            delete[] Input[2];
            delete[] Input;
        }
        delete[] Output;
    }
    else if(FISType == TAKAGI_SUGENO)
    {
        if(NAnt == 1)
        {
            delete[] Input[0];
            delete[] Input;
        }
        else if(NAnt == 2)
        {
            delete[] Input[0];
            delete[] Input[1];
            delete[] Input;
        }
        else if(NAnt == 3)
        {
            delete[] Input[0];
            delete[] Input[1];
            delete[] Input[2];
            delete[] Input;
        }
        for(int i=0; i<NRules; i++)
            delete[] Coef[i];

        delete[] Coef;
    }
}
//-------------------------------------------------------------------------------
// Avalia a saída do Sistema de Inferência Fuzzy - Tipo Mamdani (uma entrada)
//-------------------------------------------------------------------------------
float FuzzySystem::Mamdani(float x)
{
    register int j;
    int Rf;
    float aux, crisp, fire;
    int ind_crisp;
```

```
    aux = Input[0][0].u_upper - Input[0][0].u_lower;

    ind_crisp = abs( (x - Input[0][0].u_lower)/(aux)*(Input[0][0].Card-1)+ 0.5);
     crisp = Input[0][0].value[ind_crisp];
     Rf = 0;
    FuzSet Action(Output[0].u_lower, Output[0].u_upper, Output[0].Card);
    for(j=0; j<NRules; j++)
     {
        if(crisp >= Input[0][Rules->Ant[j][0]].lower &&
                                        crisp <= Input[0][Rules->Ant[j][0]].upper)
         {
            fire = Input[0][Rules->Ant[j][0]].y[ind_crisp];
            if(ImpType == LARSEN)
            {
                Action.MaxProd(&Output[Rules->Cons[j]],fire);
            }
            else if(ImpType == MAMDANI)
            {
                Action.MaxMin(&Output[Rules->Cons[j]],fire);
            }
            Rf = 1;
        }
     }
     if(Rf == 0)  return 0.0;
     else return Action.Defuzzyfication_COG();
}
//-------------------------------------------------------------------------
// Avalia a saída do Sistema de Inferência Fuzzy - Tipo Mamdani (duas entradas)
//-------------------------------------------------------------------------
float FuzzySystem::Mamdani(float x1, float x2)
{
    register int j;
    int Rf;
    float aux1, aux2, crisp1, crisp2, fire;
    int ind_x1, ind_x2;
    aux1 = Input[0][0].u_upper - Input[0][0].u_lower;
     aux2 = Input[1][0].u_upper - Input[1][0].u_lower;
    ind_x1 = abs( (x1-Input[0][0].u_lower)/(aux1)*(Input[0][0].Card-1) + 0.5 );
     ind_x2 = abs( (x2-Input[1][0].u_lower)/(aux2)*(Input[1][0].Card-1) + 0.5 );

    crisp1 = Input[0][0].value[ind_x1];
     crisp2 = Input[1][0].value[ind_x2];
    Rf = 0;
    FuzSet Action(Output[0].u_lower, Output[0].u_upper, Output[0].Card);
    for(j=0; j<NRules; j++)
     {
        if ( (crisp1 >= Input[0][Rules->Ant[j][0]].lower &&
              crisp1 <= Input[0][Rules->Ant[j][0]].upper) &&
             (crisp2 >= Input[1][Rules->Ant[j][1]].lower &&
              crisp2 <= Input[1][Rules->Ant[j][1]].upper ))
        {
            fire = TNorm(Input[0][Rules->Ant[j][0]].y[ind_x1],
                                            Input[1][Rules->Ant[j][1]].y[ind_x2]);
            if(ImpType == LARSEN)
            {
                Action.MaxProd(&Output[Rules->Cons[j]],fire);
            }
            else if(ImpType == MAMDANI)
            {
```

```
                Action.MaxMin(&Output[Rules->Cons[j]],fire);
            }
            Rf = 1;
        }
    }
    if(Rf == 0)  return 0.0;
    else return Action.Defuzzyfication_COG();
}
//-------------------------------------------------------------------------
// Avalia a saída do Sistema de Inferência Fuzzy - Tipo Mamdani (três entradas)
//-------------------------------------------------------------------------
float FuzzySystem::Mamdani(float x1, float x2, float x3)
{
    register int j;
    int Rf;
    float aux1, aux2, aux3, crisp1, crisp2, crisp3, fire, out;
    int ind_x1, ind_x2, ind_x3;

    aux1 = Input[0][0].u_upper - Input[0][0].u_lower;
     aux2 = Input[1][0].u_upper - Input[1][0].u_lower;
    aux3 = Input[2][0].u_upper - Input[2][0].u_lower;

    ind_x1 = abs( (x1-Input[0][0].u_lower)/(aux1)*(Input[0][0].Card-1) + 0.5 );
     ind_x2 = abs( (x2-Input[1][0].u_lower)/(aux2)*(Input[1][0].Card-1) + 0.5 );
    ind_x3 = abs( (x3-Input[2][0].u_lower)/(aux3)*(Input[2][0].Card-1) + 0.5 );

    crisp1 = Input[0][0].value[ind_x1];
     crisp2 = Input[1][0].value[ind_x2];
    crisp3 = Input[2][0].value[ind_x3];

    Rf = 0;

    FuzSet Action(Output[0].u_lower, Output[0].u_upper, Output[0].Card);

    for(j=0; j<NRules; j++)
     {
        if ( (crisp1 >= Input[0][Rules->Ant[j][0]].lower &&
              crisp1 <= Input[0][Rules->Ant[j][0]].upper) &&
             (crisp2 >= Input[1][Rules->Ant[j][1]].lower &&
              crisp2 <= Input[1][Rules->Ant[j][1]].upper ) &&
             (crisp3 >= Input[2][Rules->Ant[j][2]].lower &&
              crisp3 <= Input[2][Rules->Ant[j][2]].upper ))
        {
            fire = TNorm(Input[0][Rules->Ant[j][0]].y[ind_x1],
                         Input[1][Rules->Ant[j][1]].y[ind_x2],
                         Input[2][Rules->Ant[j][2]].y[ind_x3]);
            if(ImpType == LARSEN)
            {
                Action.MaxProd(&Output[Rules->Cons[j]],fire);
            }
            else if(ImpType == MAMDANI)
            {
                Action.MaxMin(&Output[Rules->Cons[j]],fire);
            }
            Rf = 1;
        }
    }
    if(Rf == 0)  return 0.0;
    else  out = Action.Defuzzyfication_COG();
    return out;
}
//-------------------------------------------------------------------------
// Avalia a saída do Sistema de Inferência Fuzzy / Tipo Takagi-Sugeno
// (uma entrada)
//-------------------------------------------------------------------------
```

```
float FuzzySystem::Takagi_Sugeno(float x)
{
    register int j;
    int Rf;
    float aux, crisp, den, num;
    float *fire, *Y;
    int ind_crisp;

    fire = new float[NRules];
    Y = new float[NRules];

    aux = Input[0][0].u_upper - Input[0][0].u_lower;

    ind_crisp = abs( (x - Input[0][0].u_lower)/(aux)*(Input[0][0].Card-1)+0.5);
     crisp = Input[0][0].value[ind_crisp];
     Rf = 0;
    for(j=0; j<NRules; j++)
     {
        if(crisp >= Input[0][Rules->Ant[j][0]].lower &&
          crisp <= Input[0][Rules->Ant[j][0]].upper)
         {
            fire[j] = Input[0][Rules->Ant[j][0]].y[ind_crisp];
            Y[j] = Coef[j][0] + Coef[j][1]*x;
            Rf = 1;
       }
       else
       {
            fire[j] = 0.0;
            Y[j] = 0.0;
       }
    }

    if(Rf == 0)
    {
        delete[] fire;
        delete[] Y;
        return 0;
    }

    den = 0.0;
    num = 0.0;

    for(j=0; j<NRules; j++)
    {
        den += fire[j];
        num += fire[j]*Y[j];
    }

    delete[] fire;
    delete[] Y;

    if( den == 0.0 ) return 0.0;
    else return num/den;
}
//-------------------------------------------------------------------------
// Avalia a saída do Sistema de Inferência Fuzzy / Tipo Takagi-Sugeno
// (Duas entradas)
//-------------------------------------------------------------------------
float FuzzySystem::Takagi_Sugeno(float x1, float x2)
{
    register int j;
    int Rf;
    float aux1, aux2, crisp1, crisp2, num, den;
    float *fire, *Y;
    int ind_x1, ind_x2;

    fire = new float[NRules];
    Y = new float[NRules];
```

```
    aux1 = Input[0][0].u_upper - Input[0][0].u_lower;
     aux2 = Input[1][0].u_upper - Input[1][0].u_lower;
    ind_x1 = abs( (x1-Input[0][0].u_lower)/(aux1)*(Input[0][0].Card-1) + 0.5 );
     ind_x2 = abs( (x2-Input[1][0].u_lower)/(aux2)*(Input[1][0].Card-1) + 0.5 );

    crisp1 = Input[0][0].value[ind_x1];
     crisp2 = Input[1][0].value[ind_x2];
    Rf = 0;
    for(j=0; j<NRules; j++)
     {
        if ( (crisp1 >= Input[0][Rules->Ant[j][0]].lower &&
              crisp1 <= Input[0][Rules->Ant[j][0]].upper) &&
             (crisp2 >= Input[1][Rules->Ant[j][1]].lower &&
              crisp2 <= Input[1][Rules->Ant[j][1]].upper ))
        {
            fire[j] = TNorm(Input[0][Rules->Ant[j][0]].y[ind_x1],
                            Input[1][Rules->Ant[j][1]].y[ind_x2]);
            Y[j] = Coef[j][0] + Coef[j][1]*x1 + Coef[j][2]*x2;
            Rf = 1;
        }
        else
        {
            fire[j] = 0.0;
            Y[j] = 0.0;
        }
    }
    if(Rf == 0)
    {
        delete[] fire;
        delete[] Y;
        return 0;
    }
    den = 0.0;
    num = 0.0;
    for(j=0; j<NRules; j++)
    {
        den += fire[j];
        num += fire[j]*Y[j];
    }
    delete[] fire;
    delete[] Y;
    if( den == 0.0 ) return 0.0;
    else return num/den;
}
//---------------------------------------------------------------------------
// Avalia a saída do Sistema de Inferência Fuzzy / Tipo Takagi-Sugeno
// (três entradas)
//---------------------------------------------------------------------------
float FuzzySystem::Takagi_Sugeno(float x1, float x2, float x3)
{
    register int j;
    int Rf;
    float aux1, aux2, aux3, crisp1, crisp2, crisp3, num, den;
    float *fire, *Y;
    int ind_x1, ind_x2, ind_x3;
    fire = new float[NRules];
    Y = new float[NRules];
    aux1 = Input[0][0].u_upper - Input[0][0].u_lower;
    aux2 = Input[1][0].u_upper - Input[1][0].u_lower;
```

```
    aux3 = Input[2][0].u_upper - Input[2][0].u_lower;
    ind_x1 = abs( (x1-Input[0][0].u_lower)/(aux1)*(Input[0][0].Card-1) + 0.5 );
    ind_x2 = abs( (x2-Input[1][0].u_lower)/(aux2)*(Input[1][0].Card-1) + 0.5 );
    ind_x3 = abs( (x3-Input[2][0].u_lower)/(aux3)*(Input[2][0].Card-1) + 0.5 );

    crisp1 = Input[0][0].value[ind_x1];
     crisp2 = Input[1][0].value[ind_x2];
    crisp3 = Input[2][0].value[ind_x3];
    Rf = 0;
    for(j=0; j<NRules; j++)
     {
        if ( (crisp1 >= Input[0][Rules->Ant[j][0]].lower &&
              crisp1 <= Input[0][Rules->Ant[j][0]].upper) &&
             (crisp2 >= Input[1][Rules->Ant[j][1]].lower &&
              crisp2 <= Input[1][Rules->Ant[j][1]].upper ) &&
             (crisp3 >= Input[2][Rules->Ant[j][2]].lower &&
              crisp3 <= Input[2][Rules->Ant[j][2]].upper ))
        {
            fire[j] = TNorm(Input[0][Rules->Ant[j][0]].y[ind_x1],
                            Input[1][Rules->Ant[j][1]].y[ind_x2],
                            Input[2][Rules->Ant[j][2]].y[ind_x3]);
            Y[j] = Coef[j][0] + Coef[j][1]*x1 + Coef[j][2]*x2 + Coef[j][3]*x3;
            Rf = 1;
        }
        else
        {
            fire[j] = 0.0;
            Y[j] = 0.0;
        }
     }
     if(Rf == 0)
     {
         delete[] fire;
         delete[] Y;
         return 0;
     }
     den = 0.0;
     num = 0.0;
     for(j=0; j<NRules; j++)
     {
         den += fire[j];
         num += fire[j]*Y[j];
     }
     delete[] fire;
     delete[] Y;
     if( den == 0.0 ) return 0.0;
     else return num/den;
}
//-------------------------------------------------------------------------
// Seta a T-Norma.
//-------------------------------------------------------------------------
char FuzzySystem::SetTNorm(char *type)
{
    if(!strcmp(type,"MINIMUM"))
    {
        TNormType = MINIMUM;
    return 1;
    }
    else if(!strcmp(type, "PRODUCT"))
    {
        TNormType = ALGEBRAIC_PROD;
```

```
    return 1;
    }
    else return 0;
}
//---------------------------------------------------------------------------
// Seta a função de "implicação" fuzzy.
//---------------------------------------------------------------------------
char FuzzySystem::SetImplication(char *type)
{
    if(!strcmp(type,"MAMDANI"))
    {
        ImpType = MAMDANI;
    return 1;
    }
    else if(!strcmp(type, "LARSEN"))
    {
        ImpType = LARSEN;
        return 1;
    }
    else return 0;
}

//---------------------------------------------------------------------------
// Retorna a T-Norm entre a e b.
//---------------------------------------------------------------------------
float FuzzySystem::TNorm(float a, float b)
{
   if(TNormType == MINIMUM)
   {
            if(a>=b) return b;
            else return a;
   }
   else if(TNormType == ALGEBRAIC_PROD)
   {
        return a*b;
   }
   else return 0.0;
}
//---------------------------------------------------------------------------
// Retorna a T-Norma entre a, b e c.
//---------------------------------------------------------------------------
float FuzzySystem::TNorm(float a, float b, float c)
{
   if(TNormType == MINIMUM)
   {
            if(a>=b)
            {
                if(b>=c) return c;
                else return b;
            }
            else
            {
                if(a>=c) return c;
                else return a;
            }
   }
   else if(TNormType == ALGEBRAIC_PROD)
   {
       return a*b*c;
   }
   else return 0.0;
}
```

C.2.2 fuzset.cpp

```
#include "universe.h"
#include "fuzset.h"
#include <stdio.h>

//---------------------------------------------------------------------------
// Classe FuzSet
//
// Esta Classe define um conjunto fuzzy.
//
// Márcio A. T. de Sousa 06/21/99.
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Destrutor de Classe Padrão (para arrays)
// sem alocação de memória
// cada objeto deve ser inicializado individualmente pela função Init().
//---------------------------------------------------------------------------
FuzSet::FuzSet(){}

//---------------------------------------------------------------------------
// Construtor de Classe 2(baseado em um dado universo de discurso X)
// X:       universo de discurso.
//---------------------------------------------------------------------------

FuzSet::FuzSet(Universe *X)
{
    register int i;
    u_lower = X->lower;
    u_upper = X->upper;
    Card = X->Card;
    value = new float[Card];
    y = new float[Card];

    for(i=0; i<Card; i++)
        value[i] = X->value[i];

}
//---------------------------------------------------------------------------
// Construtor de Classe 3
// str:     Nome do objeto.
// l:       Limitante inferior do universo de discurso.
// u:       Limitante superior do universo de discurso.
// card:    cardinalidade.
//---------------------------------------------------------------------------

FuzSet::FuzSet(float l, float u, int card)
{
    register int i;
    u_lower = l;
    u_upper = u;
    Card = card;

    value = new float[Card];
    y = new float[Card];

    for(i=0; i<Card; i++)
    {
        value[i] = l + (u-l)*(i/(float)(Card-1));
        y[i] = 0.0;
    }
}
//---------------------------------------------------------------------------
// Destrutor de Classe
//---------------------------------------------------------------------------

FuzSet::~FuzSet()
{
    delete[] value;
    delete[] y;
```

```
}
//-------------------------------------------------------------------------------
// Inicializa o objeto baseado em um dado universo de discurso X.
// Também aloca memória para o objeto.
// X:       universo de discurso.
//-------------------------------------------------------------------------------

void FuzSet::Init(Universe *X)
{
    register int i;
    u_lower = X->lower;
    u_upper = X->upper;
    Card = X->Card;
    value = new float[Card];
    y = new float[Card];

    for(i=0; i<Card; i++)
        value[i] = X->value[i];
}

//-------------------------------------------------------------------------------
// Imprime os atributos do FuzSet. (Só para aplicações de console)
//-------------------------------------------------------------------------------
void FuzSet::Print()
{
    register int i;
    printf(" Lower = %.6f     Upper = %.6f  \n", u_lower, u_upper);

    for(i=0; i<Card; i++)
        printf("Value[%d]  =  %.6f  Y[%d] = %.6f \n",i ,value[i],i,y[i]);
}

//-------------------------------------------------------------------------------
// Inicializa o conjunto fuzzy como um conjunto fuzzy triagular.
// onde
// center: é o centro do triângulo isóceles.
// spread: é o comprimento da base do triângulo.
//-------------------------------------------------------------------------------
char FuzSet::Triangle(float center, float spread)
{
    return Triangle(center-(spread/2.0), center, center+(spread/2.0));
}

//-------------------------------------------------------------------------------
// Inicializa o conjunto fuzzy como um conjunto fuzzy triagular.
// tl: Limitante inferior do triângulo.
// tu: Limitante superior do triângulo.
// tm: ponto onde o grau de pertinência é 1.(máximo)
//-------------------------------------------------------------------------------
char FuzSet::Triangle(float tl, float tm, float tu)
{
    register int i;
    if(tl >= tm || tm >= tu) return 0;

    if( tl < u_lower ) lower = u_lower;
    else lower = tl;

    if( tu > u_upper ) upper = u_upper;
    else upper = tu;

    for(i=0; i<Card; i++)
    {
        if(value[i] < tl || value[i] > tu) y[i] = 0.0;
        else
            if(value[i] >= tl && value[i] < tm) y[i] = (value[i] - tl)/(tm - tl);
            else  y[i] = (tu - value[i])/(tu - tm);
```

```
    }
    return 1;
}
//-------------------------------------------------------------------------
// Inicializa o conjunto fuzzy como um conjunto fuzzy triagular.
// onde
// str: nome do conjunto fuzzy.
// X:  universo de discurso.
// center: é o centro do triângulo isóceles.
// spread: é o comprimento da base do triângulo.
//-------------------------------------------------------------------------
char FuzSet::Triangle(char *str, float center, float spread)
{
    name = str;
    return Triangle(center-(spread/2.0), center, center+(spread/2.0));
}

//-------------------------------------------------------------------------
// Inicializa o conjunto fuzzy como um conjunto fuzzy triagular e aloca
// memória para o objeto.
// str: nome do conjunto fuzzy.
// X:  universo de discurso.
// tl: limitante inferior do triângulo.
// tu: limitante superior do triângulo.
// tm: ponto no qual o grau de pertinência é 1.(máximo)
//-------------------------------------------------------------------------

char FuzSet::Triangle(char *str, float tl, float tm, float tu)
{
    name = str;
    return Triangle(tl, tm, tu);
}
//-------------------------------------------------------------------------
// Inicializa o conjunto fuzzy como um conjunto fuzzy trapezoidal.
// onde
// t1, t2, t3, t4: parâmetros do trapézio.
//-------------------------------------------------------------------------
char FuzSet::Trapezoid(float t1, float t2, float t3, float t4)
{
    register int i;
    if(t1 > t2 || t2 > t3 || t3 > t4) return 0;

    if( t1 < u_lower ) lower = u_lower;
    else lower = t1;

    if( t4 > u_upper ) upper = u_upper;
    else upper = t4;

    for(i=0; i<Card ; i++)
    {
        if(value[i] < t1 || value[i] > t4) y[i] = 0.0;
        else
            if(value[i] >= t1 && value[i] < t2) y[i] = (value[i] - t1)/(t2 - t1);
            else
                if(value[i] >= t2 && value[i] <= t3) y[i] = 1.0;
                else    y[i] = (t4 - value[i])/(t4 - t3);
    }
    return 1;
}
//-------------------------------------------------------------------------
// Inicializa o conjunto fuzzy como um conjunto fuzzy trapezoidal.
// onde
// str: nome do conjunto fuzzy.
```

```
// X:  universo de discurso.
// t1, t2, t3, t4: parâmetros do trapézio.
//-------------------------------------------------------------------------
char FuzSet::Trapezoid(char *str, float t1, float t2, float t3, float t4)
{
    name = str;
    return Trapezoid(t1, t2, t3, t4);
}
//-------------------------------------------------------------------------
// Executa a defuzzificação do conjunto fuzzy baseado no método do
// Centro-de-Gravidade(COG).
//-------------------------------------------------------------------------
float FuzSet::Defuzzyfication_COG(void)
{
    register int i;

    float den, num;

    den = 0.0;
    num = 0.0;

    for(i=0; i<Card; i++)
    {
        num += value[i]*y[i];
        den += y[i];
    }

    return (num/den);
}
//-------------------------------------------------------------------------
// Faz operação de Máximo entre o conjunto fuzzy que chama esta função
// e o conjunto fuzzy é multiplicado por fire.
//-------------------------------------------------------------------------
void FuzSet::MaxProd(FuzSet *x, float fire)
{
    register int i;
    for(i=0; i<Card; i++)
        y[i] = Max(y[i], fire*x->y[i]);
}
//-------------------------------------------------------------------------
// Faz a operação de máximo entre o conjunto fuzzy que chama esta função
// e o mínimo entre o conjunto fuzzy x e fire.
//-------------------------------------------------------------------------
void FuzSet::MaxMin(FuzSet *x, float fire)
{
    register int i;
    for(i=0; i<Card; i++)
        y[i] = Max(y[i], Min(fire,x->y[i]));
}

//-------------------------------------------------------------------------
// Retorna o valor máximo entre a e b.
//-------------------------------------------------------------------------
float FuzSet::Max(float a, float b)
{
    if(a<=b) return b;
    else return a;
}

//-------------------------------------------------------------------------
// Retorna o valor mínimo entre a e b.
//-------------------------------------------------------------------------
float FuzSet::Min(float a, float b)
{
   if(a>=b) return b;
```

```
   else return a;
}
```

C.2.3 rulebase.cpp

```
//-------------------------------------------------------------------------------
// Classe RuleBase
//
// Esta classe define os dados da base de regras.
//
// Márcio A. T. de Sousa 06/21/99.
//-------------------------------------------------------------------------------
#include "rulebase.h"
//-------------------------------------------------------------------------------
// Construtor de Classe 1
// onde
// nant: número de antecedentes.
// nr: número de regras.
//-------------------------------------------------------------------------------
RuleBase::RuleBase(int nant, int nr)
{
    Count = 0;
    NAntecedents = nant;
    NRules = nr;

    Ant = new int*[NRules];

    for (int i = 0; i < NRules; i++)
          Ant[i] = new int[NAntecedents];

    Cons = new int[NRules];
}
//-------------------------------------------------------------------------------
// Destrutor de Classe
//-------------------------------------------------------------------------------
RuleBase::~RuleBase()
{
    delete[] Cons;

    for (int i = 0; i < NRules;  i++)
       delete[] Ant[i];

    delete[] Ant;
}
//-------------------------------------------------------------------------------
// Adiciona uma nova regra na base de dados. (uma entrada)
// a1: antecedente 1
// c1: consequente
//-------------------------------------------------------------------------------
char RuleBase::AddRule(int a1, int c1)
{
    if( NAntecedents != 1)
    {
    return 0;
    }
    if( Count == NRules )
    {
    return 0;
    }
    else
    {
        Ant[Count][0] = a1;
        Cons[Count] = c1;
        Count++ ;
    }
    return 1;
}
```

```
//-------------------------------------------------------------------------------
// Adiciona uma nova regra na base de dados. (duas entradas)
// a1: antecedente 1
// a2: antecedente 2
// c1: consequente
//-------------------------------------------------------------------------------
char RuleBase::AddRule(int a1, int a2, int c1)
{
    if( NAntecedents != 2)
    {
    return 0;
    }
    if( Count == NRules )
    {
    return 0;
    }
    else
    {
        Ant[Count][0] = a1;
        Ant[Count][1] = a2;
        Cons[Count] = c1;
        Count++ ;
    }
    return 1;
}
//-------------------------------------------------------------------------------
// Adiciona uma nova regra na base de dados. (três entradas)
// a1: antecedente 1
// a2: antecedente 2
// a3: antecedente 3
// c1: consequente
//-------------------------------------------------------------------------------
char RuleBase::AddRule(int a1, int a2, int a3,int c1)
{
    if( NAntecedents != 3)
    {
    return 0;
    }
    if( Count == NRules )
    {
    return 0;
    }
    else
    {
        Ant[Count][0] = a1;
        Ant[Count][1] = a2;
        Ant[Count][2] = a3;
        Cons[Count] = c1;
        Count++;
    }
    return 1;
}
```

C.2.4 universe.cpp

```
#include "universe.h"
#include <stdio.h>

//-------------------------------------------------------------------------------
// Classe Universe
//
// Esta classe define o universo de discurso discreto que é utilizado na
// inicialização do conjunto fuzzy, na classe FuzSet.
//
// Márcio A. T. de Sousa 06/21/99.
//-------------------------------------------------------------------------------
```

```
//-------------------------------------------------------------------------
// Construtor de Classe
// l:       limitante inferior do universo de discurso.
// u:       limitante superior do universo de discurso.
// card:    cardinalidade.
//-------------------------------------------------------------------------
Universe::Universe(int card, float l, float u)
{
    register int i;
    lower = l;
    upper = u;
    Card = card;

    value = new float[Card];

    for(i=0; i<Card; i++)
        value[i] = l + (u-l)*(i/(float)(Card-1));
}
//-------------------------------------------------------------------------
// Destrutor de Classe
//-------------------------------------------------------------------------
Universe::~Universe()
{
    delete[] value;
}
//-------------------------------------------------------------------------
// Imprime os atributos do Universo. (Só para aplicações de console)
//-------------------------------------------------------------------------
void Universe::Print()
{
    register int i;
    printf(" Lower = %.6f     Upper = %.6f  \n", lower, upper);

    for(i=0; i<Card; i++)
        printf("Value[%d]  =  %.6f \n",i , value[i]);
}
```

C.2.5 random.cpp

```
#include "random.h"
#include <time.h>
#include <math.h>
#include <sys\timeb.h>

static long seed; // variável global
//-------------------------------------------------------------------------
//  Retorna um número aleatório com range definido no intervalo [1,7FFFFFFF].
//-------------------------------------------------------------------------
long myrand(void)
{
    long rn, a, q, r;

    a = 16807;
    q  = 127773;
    r = 2836;
    rn = a*seed-(a*q+r)*(long)(seed/q);
    if(rn  < 0) rn += a*q+r;
    seed = rn;
    return rn;
}
//-------------------------------------------------------------------------
//  Gera semente.
//-------------------------------------------------------------------------
void initmyrand(void)
{
    struct timeb t;
```

```
    long  maior, menor, range_res;
    float range_orig,temp, lim_i, lim_s;
    ftime(&t);
    menor = 1;
    maior = (long)pow(2,31) - 1;
    temp = float(t.time%60) + float(t.millitm)*0.001;
    lim_i = 0.0;
    lim_s = 60.0;
    range_orig = lim_s - lim_i;
    range_res = maior - menor;
    seed = (long)( ((temp-lim_i)*(float)(range_res)) / range_orig ) + menor ;
    seed = (long)1000000;//1000;
}

//---------------------------------------------------------------------------
//  Retorna um número aleatório inteiro com range definido no
//    intervalo [0,Max-1].
//---------------------------------------------------------------------------
long myrandom(long Max)
{
    long rn, a, q, r;
    a = 16807;
    q  = 127773;
    r = 2836;
    rn = a*seed-(a*q+r)*(long)(seed/q);
    if(rn  < 0) rn += a*q+r;
    seed = rn;
    return   (long)(((float)(rn-1)/(float)(0x7FFFFFFF))*Max);
}
//---------------------------------------------------------------------------
//  Retorna um número aleatório inteiro com range definido no
//    intervalo [Min,Max].
//---------------------------------------------------------------------------
long myrandom(long Min, long Max)
{
    long rn, a, q, r;
    a = 16807;
    q  = 127773;
    r = 2836;
    rn = a*seed-(a*q+r)*(long)(seed/q);
    if(rn  < 0) rn += a*q+r;
    seed = rn;
    return   (long)(((float)(rn-1)/(float)(0x7FFFFFFF))*(Max-Min + 1)) + Min ;
}
//---------------------------------------------------------------------------
//  Retorna um número aleatório (float) com range definido no
//    intervalo [Min,Max].
//---------------------------------------------------------------------------
float myrandomf(float Min, float Max)
{
    long rn, a, q, r;
    a = 16807;
    q  = 127773;
    r = 2836;
    rn = a*seed-(a*q+r)*(long)(seed/q);
    if(rn  < 0) rn += a*q+r;
    seed = rn;
    return   ((float)(rn-1)/(float)(0x7FFFFFFF-0x1))*(Max-Min) + Min ;
}
//---------------------------------------------------------------------------
```

```
//  Gera uma sequência aleatória sem repetição de n números inteiros
//  contidos no intervalo [0,Max-1].
//  A sequência será armazenada em vector
//--------------------------------------------------------------------------
void random_seq(long *vector,int Max, int n)
{
    register int i, k;
    float soma, limite_inferior;
    float *intervalo;
    int *teste;
    long r;

    intervalo = new float[Max];
    teste = new int[Max];

    for(i=0; i<Max; i++)
    {
        teste[i] = 1;
    }

    for(k=0; k<n; k++)
    {
        soma = 0.0;
        limite_inferior = 1.0;

        for(i=0; i<Max; i++)
        {
            soma += teste[i];
        }

        for(i=0; i<Max; i++)
        {
            intervalo[i] = ((float)teste[i]/(float)soma)*(float)(0x7FFFFFFF-0x1)+
                           (float)limite_inferior;
            limite_inferior = intervalo[i];
        }
        limite_inferior = 1;

        r = myrand();

        for(i=0; i<Max; i++)
        {
            if( ( r >= limite_inferior ) && ( r < intervalo[i] ) )
            {
                vector[k] = i;
                teste[i] = 0;
                break;
            }
            limite_inferior = intervalo[i];
        }
    }
    delete[] teste;
    delete[] intervalo;
}
```

C.2.6 pop.cpp

```
#include "random.h"
#include "indiv.h"
#include "pop.h"

Population::Population(int tPop, int nSP)
{
    register int i,j;
    NumberSubPop = nSP;
    PopSize = tPop;
    NTypes = NTYPES;

    SubPop = new Individual*[NumberSubPop];
    Vaux = new short*[NumberSubPop];
```

```
    for (i = 0; i < NumberSubPop; i++)
    {
      SubPop[i] = new Individual[PopSize];
      Vaux[i] = new short[PopSize];
    }
    Pop = new Individual[PopSize];
    NPar  = new int[NTypes];
    P_Max = new float[NTypes];
    P_Min = new float[NTypes];
    Range = new float[NTypes];
    PType = new char[NTypes];
    for(i=0; i<NumberSubPop; i++)
    for(j=0; j<PopSize; j++)
        Vaux[i][j] = 0;     //inicializa vetor de indices
}
Population::~Population( )
{
    for (int i = 0; i < NumberSubPop; i++)
    {
       delete[] SubPop[i];
       delete[] Vaux[i];
    }
    delete[] Vaux;
    delete[] SubPop;
    delete[] Pop;
    delete[] P_Max;
    delete[] P_Min;
    delete[] Range;
}
void Population::SetParRange(int ind, int npar, float min, float max, char type)
{
    P_Max[ind] = max;
    P_Min[ind] = min;
    Range[ind] = max - min;
    NPar[ind]  = npar;
    PType[ind] = type;
}
void Population::InitPopulation( void )
{
    register int i, j, k,count;
    for(i=0; i<PopSize; i++)
    {
    count = 0;
    for(j=0; j<NTypes; j++)
    {
       for(k=0; k<NPar[j]; k++)
       {
        if( PType[j] == 'c')      // constante - metade do range
        {
            Pop[i].DoubleCromossome[count] = P_Min[j] + Range[j]*0.5;
        }
        else if ( PType[j] == 'u')  // uniformemente distribuída no range
        {
            Pop[i].DoubleCromossome[count] = P_Min[j] +
                          (P_Max[j]-P_Min[j])*(float)(k)/(float)(NPar[j]-1);
        }
        else if ( PType[j] == 'r')  // números aleatórios contidos no range
        {
            Pop[i].DoubleCromossome[count] = myrandomf(P_Min[j],P_Max[j]);
        }
```

```
        count ++;
      }
    }
    }
    count = 0;
}
void Population::ExecuteCrossoverSubPop(int indice_sp)
{
    register int i, k;
    const int dcrom_size = SubPop[indice_sp][0].DoubleCromSize;
    long *vector;
    int  indice_gene, num_genes;
    float z;
    vector = new long[dcrom_size];

    for( k = 0; k<PopSize/2 ; k++ )
    {
        // Baseado na probabilidade de crossover, rolam-se os dados para
        // verificar se haverá crossover entre os cromossomos.
        if( myrandomf(0.0,1.0) >= PC ) continue;
        // Determina aleatoriamente o número de genes que serão trocados.
        num_genes = (int) myrandom(1L,(long)dcrom_size);
    random_seq(vector, dcrom_size, num_genes);
        for( i = 0; i<num_genes; i++ )
        {
        indice_gene = (int)vector[i];
        //if( indice_gene >= dcrom_size || indice_gene < 0 ) Beep();
            // Efetua o crossover do gene escolhido.
            z = SubPop[indice_sp][2*k].DoubleCromossome[indice_gene];
            SubPop[indice_sp][2*k].DoubleCromossome[indice_gene] =
                        SubPop[indice_sp][2*k +1].DoubleCromossome[indice_gene];
            SubPop[indice_sp][2*k +1].DoubleCromossome[indice_gene] = z;
        }
    }
    delete[] vector;
}
void Population::CrossoverSubPops(void)
{
    register int i;
    // População no.1 não sofre crossover
    for(i = 1; i<NumberSubPop; i++)
        ExecuteCrossoverSubPop(i);
}

void Population::RoulleteWheelPop(void)
{
    register int i;
    float sum, lower;
    float *interv;
    long r;
    sum = 0.0;
    lower = 1.0;
    for(i = 0; i<PopSize; i++)
        sum += Pop[i].Fitness;

    interv = new float[PopSize];
    for(i = 0; i<PopSize; i++)
    {
        interv[i] = (Pop[i].Fitness/sum)*(float)(0x7FFFFFFF-0x1) + lower;
        lower = interv[i];
    }
```

```
    lower = 1.0;
    r = myrand();
    for(i = 0; i<PopSize; i++)
    {
        if( ( r >= lower ) && ( r < interv[i] ) ) break;
        lower = interv[i];
    }
    delete[] interv;
    Roullete = Pop[i];
}
void Population::RoulleteWheelSubPop(void)
{
    register int i, j;
    int aux = 0;
    float sum, lower;
    float **interv;
    long r;
    sum = 0.0;
    lower = 1.0;
    interv = new float*[NumberSubPop];
    for(i = 0; i<NumberSubPop; i++)
    interv[i] = new float[PopSize];
    for(i = 0; i<NumberSubPop; i++)
        for(j = 0; j<PopSize; j++)
            sum += SubPop[i][j].Fitness;
    for(i = 0; i<NumberSubPop; i++)
        for(j = 0; j<PopSize; j++)
        {
            interv[i][j] = (SubPop[i][j].Fitness/sum)*(float)(0x7FFFFFFF-0x1)+
                                                                    lower;
            lower = interv[i][j];
        }
    lower = 1.0;
    r = myrand();
    for(i = 0; i<NumberSubPop; i++)
    {
        for(j = 0; j<PopSize; j++)
        {
            if( ( r >= lower ) && ( r < interv[i][j] ) )
        {
        aux = 1;
        break;
        }
            lower = interv[i][j];
        }
    if(aux==1) break;
    }
    Roullete = SubPop[i][j];
    for (i = 0; i < NumberSubPop; i++)
       delete[] interv[i];
    delete[] interv;
}
void Population::GenerateSubPopByRoullete(int indice_sp)
{
    register int i;
    for(i=0; i<PopSize; i++)
    {
```

```
        RoulleteWheelPop();
    SubPop[indice_sp][i] = Roullete;
    }
}
void Population::BestPop(void)
{
    register int j;
    float Max = Pop[0].Fitness;
    int ind = 0;
    for(j=0; j<PopSize; j++)
    {
        if((Pop[j].Fitness > Max))
        {
            ind = j;
            Max = Pop[j].Fitness;
        }
    }
    Best = Pop[ind];
}

void Population::BestSubPops(void)
{
    register int i, j;
    float Max = 0.0;
    int sp = 0 ,ind = 0;
    for(i=0; i<NumberSubPop; i++)
    {
        for(j=0; j<PopSize; j++)
        {
            if((SubPop[i][j].Fitness > Max) && (Vaux[i][j] == 0))
            {
                sp = i;
                ind = j;
                Max = SubPop[i][j].Fitness;
        }
        }
    }
    Vaux[sp][ind] = 1;
    Best = SubPop[sp][ind];
}
void Population::SubPopBestnOthers(int indice_sp1, int indice_sp2)
{
    register int i;
    BestPop();
    for(i=0; i<(PopSize/2); i++)
    {
        SubPop[indice_sp1][2*i] = Best;
        SubPop[indice_sp1][2*i+1] = Pop[i];
    }
    for(i=0; i<(PopSize/2); i++)
    {
        SubPop[indice_sp2][2*i] = Best;
        SubPop[indice_sp2][2*i+1] = Pop[i+(PopSize/2)];
    }
}
void Population::GenerateSubPops(void)
{
    register int i;
    for(i=0; i<PopSize; i++)
        SubPop[0][i] = Pop[i];    // Primeira SubPopulação é uma cópia da
                                  // População principal
```

```
    SubPopBestnOthers(1,2);
    for(i=3; i<NumberSubPop; i++)
        GenerateSubPopByRoullete(i);
}
void Population::ExecuteMutationSubPop(int indice_sp,int gen)
{
    register int i, j;
    long *vector;
    int num_genes, indice_gene, aux, k;
    float delta;
    const int dcrom_size = SubPop[indice_sp][0].DoubleCromSize;
    vector = new long[dcrom_size];
    for(i=0; i<PopSize; i++)
    {
    if( myrandomf(0.0,1.0) >= PM )  continue;
    // Determina aleatoriamente o número de genes que serão mutados.
    num_genes = (int)myrandom(1L,(long)dcrom_size);
    random_seq(vector, dcrom_size, num_genes);
    /*
    printf("\n");
    for(j=0; j<10; j++)
       printf("%.2f\t", SubPop[indice_sp][i].DoubleCromossome[j]);
    printf("\n Numero de genes mutados = %d \n",num_genes);
    for(j=0; j<num_genes; j++)
        printf("%d\t",vector[j]);    */

    for(j=0; j<num_genes; j++)
    {
        indice_gene = (int)vector[j];
        aux = 0;
        for(k=0; k<NTypes; k++)
        {
        if(indice_gene >= aux && indice_gene < aux+NPar[k]) break;
        aux += NPar[k];
        }
        //if( indice_gene >= dcrom_size || indice_gene < 0 ) Beep();
        // Efetua a mutacao do gene escolhido.
        delta =  1.0 - pow(myrandomf(0.0,1.0),
                           pow( (1.0 -(float)(gen)/(float)(NG)),2.0));
        if( myrandomf(0.0,1.0) > 0.5)
        {
        SubPop[indice_sp][i].DoubleCromossome[indice_gene] +=
                                                  Range[k]*RANGE_MUT*delta;
        if(SubPop[indice_sp][i].DoubleCromossome[indice_gene] > P_Max[k] )
                  SubPop[indice_sp][i].DoubleCromossome[indice_gene] = P_Max[k];
        else if (SubPop[indice_sp][i].DoubleCromossome[indice_gene] < P_Min[k])
                  SubPop[indice_sp][i].DoubleCromossome[indice_gene] = P_Min[k];
        }
        else
        {
        SubPop[indice_sp][i].DoubleCromossome[indice_gene] -=
                                                  Range[k]*RANGE_MUT*delta;
        if(SubPop[indice_sp][i].DoubleCromossome[indice_gene] > P_Max[k] )
                  SubPop[indice_sp][i].DoubleCromossome[indice_gene] = P_Max[k];
        else if (SubPop[indice_sp][i].DoubleCromossome[indice_gene] < P_Min[k])
                  SubPop[indice_sp][i].DoubleCromossome[indice_gene] = P_Min[k];
        }

     //SubPop[indice_sp][i].DoubleCromossome[indice_gene] +=
```

```
      //                        myrandomf(-Range[k]/2.0,Range[k]/2.0)*RANGE_MUT;
      //if(SubPop[indice_sp][i].DoubleCromossome[indice_gene] > P_Max[k] )
      //        SubPop[indice_sp][i].DoubleCromossome[indice_gene] = P_Max[k];
      //else if (SubPop[indice_sp][i].DoubleCromossome[indice_gene] < P_Min[k])
      //          SubPop[indice_sp][i].DoubleCromossome[indice_gene] = P_Min[k];

    }
    /*
    printf("\n");
    for(j=0; j<10; j++)
       printf("%.2f\t", SubPop[indice_sp][i].DoubleCromossome[j]);
    getch();          */
    }
    delete[] vector;
}
void Population::MutationSubPops(int gen)
{
    register int i;
    for(i=1; i<NumberSubPop; i++) // SubPopulação no.1 não sofre mutação.
        ExecuteMutationSubPop(i,gen);
}
void Population::Selection(void)
{
    register int i, j;
    // Inicializa vetor de indices.
    for(i=0; i<NumberSubPop; i++)
        for(j=0; j<PopSize; j++)
            Vaux[i][j] = 0;

    EvalFitnessSubPops();
    for(i=0; i<NBEST; i++)
    {
    BestSubPops();
        Pop[i] = Best;
    }
    Best = Pop[0];
    for(i=NBEST; i<PopSize; i++)
    {
        RoulleteWheelSubPop();
        Pop[i]= Roullete;
    }
}
void Population::EvalFitnessSubPops(void)
{
    register int i, j;
    double d;
    // calcula fitness das subpopulações
    for(i=0; i<NumberSubPop; i++)
    {
        for(j=0; j<PopSize; j++)
        {
            d = sqrt( pow(SubPop[i][j].DoubleCromossome[0]-0.4,2.0)+
                                pow(SubPop[i][j].DoubleCromossome[1]-0.2,2.0) );
            if(d > 0) SubPop[i][j].Fitness = 0.5 + sin(6*M_PI*d)/(6*M_PI*d);
            else      SubPop[i][j].Fitness = 1.5;
        }
    }
}
void Population::EvalFitnessPop(void)
{
```

```
    register int i;
    double d;
    // Calcula o fitness de todos os indivíduos
    for(i=0; i<PopSize; i++)
    {
        d = sqrt( pow(Pop[i].DoubleCromossome[0]-0.4,2.0)+
                                    pow(Pop[i].DoubleCromossome[1]-0.2,2.0) );
        if(d > 0) Pop[i].Fitness = 0.5 + sin(6*M_PI*d)/(6*M_PI*d);
        else      Pop[i].Fitness = 1.5;
    }
}
```

C.2.7 indiv.cpp

```
#include "indiv.h"

Individual::Individual()
{
    DoubleCromSize = CROM_SIZE;
}

Individual::~Individual()
{
}
```