Universidade do Estado do Rio de Janeiro
Faculdade de Engenharia
Departamento de Sistemas e Computação

ADEQUAÇÃO DO *Software* DE SIMULAÇÃO SCICOS PARA A AQUISIÇÃO DE DADOS E O CONTROLE EM TEMPO REAL

Autor: **ELAINE DE MATTOS SILVA**

RIO DE JANEIRO
FEVEREIRO/2010

Fevereiro–2010

SILVA, ELAINE DE MATTOS

Adequação do *Software* de Simulação Scicos para a Aquisição de Dados e o Controle em Tempo Real [Rio de Janeiro] 2010.

xii, 88 p. 29,7 cm (FEN/UERJ, Engenheiro, Engenharia Elétrica - ênfase em Sistemas e Computação, 2010)

Monografia - Universidade do Estado do Rio de Janeiro – UERJ

1. Scicos, 2. Scilab, 3. controle por computador, 4. *software* livre, 5. modelagem de sistemas híbridos

I. FEN/UERJ II.Título (série)

ADEQUAÇÃO DO *Software* DE SIMULAÇÃO SCICOS PARA A AQUISIÇÃO DE DADOS E O CONTROLE EM TEMPO REAL

Elaine de Mattos Silva

Monografia submetida ao corpo docente da Faculdade de Engenharia da Universidade do Estado do Rio de Janeiro - UERJ, como parte dos requisitos necessários à obtenção do diploma de Engenheiro Eletricista com ênfase em Sistemas e Computação.

Aprovada por:

Orientador: José Paulo Vilela Soares da Cunha, D.Sc., UERJ

Co-orientador: Orlando Bernardo Filho, D.Sc., UERJ

Membro da banca: Henrique Goldfeld, M.Sc., UERJ

Rio de Janeiro, 04 de Fevereiro de 2010.

Este trabalho é dedicado ao grande engenheiro de caminhões, José Gregório da Silva, e ao grande engenheiro de sapatos, Carlos Leopoldino de Mattos.

AGRADECIMENTOS

Aproveito este momento para agradecer ao meu querido avô Carlos, que uma vez me disse que se pode tirar qualquer coisa de um ser humano, menos seu conhecimento. Agradeço também ao meu avô José por aquela famosa troca de lâmpada.

Agradeço aos meus pais, que me ensinaram que, antes de ser importante e necessário, aprender é interessante e divertido. Obrigada por me cederem um pouco de sua inclinação aos estudos e me inspirarem a perseguir meus sonhos.

Agradeço ao meu irmão querido por estar sempre ao meu lado, nos bons e maus momentos. Sua paciência e companheirismo sempre foram muito apreciados, mas essa apreciação nem sempre foi verbalizada. Agradeço ao meu marido pelo incentivo constante, o carinho e paciêcia de sempre. Sua ajuda foi fundamental na consolidação de conceitos e técnicas de programação ao longo do curso.

Gostaria de agradecer ao professor José Paulo por acreditar em seus alunos, incentivá-los e saber aproveitar o que cada um tem de melhor. Sem suas ideias, sugestões e críticas este trabalho não seria possível. Agradeço também ao professor Orlando por me mostrar o quanto se aprende ao olhar um mesmo assunto sob perspectivas diferentes.

Agradeço também aos amigos de curso, que me ensinaram que a amizade equaciona e resolve os mais difíceis problemas. Aos funcionários da UERJ, mais precisamente às secretárias Adelaide e Silvia, além dos técnicos de laboratório Marcos e André. Sua ajuda "não tem preço"!

Por fim, agradeço a Deus, o Engenheiro de Tudo.

Resumo da Monografia apresentada à FEN/UERJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletricista com ênfase em Sistemas e Computação.

ADEQUAÇÃO DO *Software* DE SIMULAÇÃO SCICOS PARA A AQUISIÇÃO DE DADOS E O CONTROLE EM TEMPO REAL

Elaine de Mattos Silva

Fevereiro/2010

Orientador: José Paulo Vilela Soares da Cunha,UERJ

Co-orientador: Orlando Bernardo Filho,UERJ

Palavras-chave: Scicos, Scilab, controle por computador, *software* livre, modelagem de sistemas híbridos

Esse trabalho discute a viabilidade do controle por computador de sistemas híbridos utilizando um sistema de aquisição de dados descentralizado, baseado no paradigma cliente/servidor, com o uso de um sistema operacional COTS (*Comercial Off The Shelf*), sem suporte a escalonamento de tempo real e rede Ethernet.

O objetivo principal do projeto é disponibilizar o *hardware* de aquisição de dados do Laboratório de Eletrônica de Potência e Automação (LEPAT) da Faculdade de Engenharia da UERJ a diversos usuários, permitindo o compartilhamento de recursos do laboratório. Para atingir este objetivo foi desenvolvida uma integração entre a API (*Application Program Interface*) de aquisição de dados Comedilib (www.comedi.org), executada em um computador servidor, e o *software* de simulação de sistemas híbridos Scicos (www.scicos.org), executado em qualquer computador cliente.

Testes com aquisição e síntese de sinais, medição de atrasos e experimentos de controle HIL (*Hardware In the Loop*) indicam a boa aplicabilidade do sistema.

SUMÁRIO

1 Introdução **1**

1.1 Organização do trabalho . . . 3

2 Fundamentos Teóricos **5**

2.1 Sistema Operacional Linux . . . 5

2.2 Modelagem e Simulação de Sistemas Dinâmicos . . . 8

2.2.1 *Softwares* Scilab e Scicos . . . 8

2.2.2 Métodos Numéricos . . . 10

2.2.3 Equações Diferenciais Utilizadas na Modelagem de Sistemas Dinâmicos 18

2.3 Controle por Computador e o Projeto Comedi . . . 34

2.4 Programação Cliente/Servidor . . . 40

2.4.1 Programação com *sockets* . . . 40

2.4.2 Protocolos da Camada de transporte . . . 44

2.4.3 Protocolos da Camada de aplicação . . . 47

2.5 Integração de Programas Escritos pelo Usuário ao Scilab/Scicos . . . 50

2.5.1 Estrutura interna do Scilab . . . 50

2.5.2 Estrutura interna do Scicos . . . 55

3 Arquitetura e Configuração do Sistema de Aquisição Remota de Dados **63**

3.1 Definição do Sistema . . . 63

3.2 Configuração do Sistema . . . 65

3.3 Descrição dos Módulos Servidor e dos Clientes . . . 68

3.3.1 Módulo Servidor . . . 68

3.3.2 Módulo Cliente Scicos . . . 71

3.3.3 Outros Clientes . . . 74

4 Resultados Experimentais **76**
4.1 Aquisição de Sinais 77
4.2 Geração de Sinais 78
4.3 Medição do Atraso 79
4.4 Aplicação em um Servomecanismo 80

5 Conclusões **84**
5.1 Continuação Deste Trabalho 85

Referências **88**

LISTA DE FIGURAS

1.1 Janelas do RLTOOL com diagrama do lugar das raízes, resposta no domínio do tempo e frequência de um regulador de tensão. 3

2.1 Janela do Scilab onde foi definida a matriz polinomial M. 9

2.2 *Palettes* de fontes de sinais (*sources*) e de elementos elétricos (*electrical*). . . . 10

2.3 Métodos de integração numérica aplicados a um problema rígido. 16

2.4 Detalhe do gráfico a fig. 2.3. 16

2.5 Diagrama de blocos de um sistema de controle de velocidade. 19

2.6 Esquema elétrico de um motor DC, controlado por armadura. 19

2.7 Resposta ao degrau do sistema de controle de velocidade. 23

2.8 Circuito com fonte de corrente dependente. 24

2.9 Bloco Modelica do Scicos e código Modelica. 28

2.10 Resultado da simulação do circuito modelado por EAD. 29

2.11 Sistema bola-barra. 31

2.12 Controle discreto. 35

2.13 Sistema de dados amostrados. 36

2.14 Diagrama de bloco da placa AT-MIO-64E-3 da *National Instruments*. 37

2.15 Diagrama do Modelo TCP/IP. 41

2.16 Fluxo de Comunicação cliente/servidor. 42

2.17 Estrutura interna do Scilab. 51

2.18 Sinais no Scicos. 56

2.19 Diagrama esquemático de um bloco 59

3.1 Pinagem do conector DB-37 externo da placa de aquisição. 67

3.2 Fluxograma simplificado do servidor de aquisição 68

3.3 Detalhe dos parâmetros do bloco de entrada analógica. 73
3.4 Módulo de configuração da placa de aquisição no Scicos. 73
3.5 Detalhe da chamada ao servidor local para configuração da porta paralela. . . . 74
3.6 Detalhe da chamada ao servidor local para escrita de bit na porta paralela. . . . 74

4.1 Bancada para os experimentos de controle. O computador cliente, à esquerda, executa o Scilab/Scicos. No centro está o servomecanismo e à direita está o computador servidor, que hospeda a placa de aquisição de dados. 77
4.2 Diagrama do Scicos para testes de aquisição de dados pelo conversor A/D. . . . 77
4.3 Senóide 1Hz amostrada a 10ms. 78
4.4 Senóide 1Hz fornecida pelo gerador de sinais. 78
4.5 Senóide 10Hz amostrada a 10ms. 78
4.6 Senóide 10Hz fornecida pelo gerador de sinais. 78
4.7 Diagrama do Scicos para testes de geração de dados. 79
4.8 Sinal de 1Hz gerado a um período de amostragem de 10ms. 79
4.9 Sinal de 10Hz gerado a um período de amostragem de 10ms. 80
4.10 Sinal de 10Hz gerado a um período de amostragem de 1ms. 80
4.11 Diagrama do Scicos para medição do atraso de amostragem. 81
4.12 Medida do atraso de amostragem. Sinal de 10Hz. 81
4.13 Medida do atraso de amostragem. Sinal de 20Hz. 81
4.14 Servomotor Quanser SRV-02. 82
4.15 Simulação do sistema de controle de posição. Controle proporcional, por computador. 82
4.16 Sinais de referência (r) e saída (y). 83
4.17 Sinal de controle (u). 83
4.18 Implementação do sistema de controle de posição. Controle proporcional, por computador. Tempo real 83
4.19 Sinais de referência (r) e saída (y) experimentais. 83
4.20 Sinal de controle (u) experimental. 83

Lista de Tabelas

2.1 Parâmetros do sistema de controle de velocidade. 21
2.2 Simulação do sistema de controle de velocidade. 22
2.3 Funções *syslin* e *dscr*. 32
2.4 Controle em tempo contínuo. 34
2.5 Controle em tempo discreto. 34
2.6 Programa para aquisição simples. 39
2.7 Função soma. 53
2.8 Interface para a função soma. 62
2.9 Etapas da simulação. 62

3.1 Códigos do servidor e suas funções. 69
3.2 Fases da simulação e funções dos blocos. 72

CAPÍTULO 1

INTRODUÇÃO

O aumento da confiabilidade e desempenho dos sistemas computacionais observado atualmente representa um papel importante na popularização dos sistemas controlados por computador. Desde o início das pesquisas sobre o uso de computadores digitais para controle de processos, em meados de 1950, até os dias de hoje os sistemas computacionais passaram de máquinas a válvula, com programas em linguagem de montagem (*assembly*), até microprocessadores com sistemas operacionais embutidos em tempo real.

Ultimamente quase todos os sistemas de controle em áreas como geração e distribuição de energia elétrica, controle de processos, manufatura e entretenimento utilizam controle por computador (Åstrom & Wittenmark 1997). Esse desenvolvimento progressivo vem acompanhado de uma queda significativa nos custos com *hardware*, porém, a programação ainda impõe uma restrição ao desenvolvimento do controle por computador. A desvalorização que se percebe no preço do *hardware* não se observa no custo de aquisição e manutenção de *software* proprietários. Além do alto custo do *software*, a impossibilidade de modificar ou otimizar pacotes fornecidos pelo fabricante faz aumentar ainda mais os custos com programação, dificultando a ampla utilização de sistemas computacionais em instituições públicas de pesquisa e ensino brasileiras, onde os recursos financeiros são geralmente escassos (Silva & Cunha 2006).

Ainda assim percebe-se o crescente uso dos computadores no auxílio ao ensino de disciplinas ligadas às diversas áreas da engenharia. Na engenharia elétrica, faz-se uso específico de programas de computação numérica para simulação de sistemas dinâmicos em disciplinas como Eletromagnetismo, Circuitos Lineares e Engenharia de Controle, porém, o alto preço dos *softwares* proprietários tradicionalmente usados torna quase inviáveis as tentativas de desenvolvimento de instalações especiais para o ensino e pesquisa nessas áreas dentro das universidades

públicas. Por outro lado, o conceito de *software* livre vem sendo difundido e ganhando aceitação dentro dessas instituições, tanto por parte de professores como alunos. O *software* livre e de código fonte aberto promove não só a sensível diminuição com os custos de licença como também permite que o próprio usuário realize modificações.

Países como China, Índia, França e Itália vêm utilizando e desenvolvendo extensões para o *software* livre de computação numérica e simulação de sistemas dinâmicos híbridos Scilab/Scicos para fins de controle industrial e educacional. Na área industrial destacam-se projetos de simulação com *Hardware In the Loop* (HIL), em tempo real, para desenvolvimento e testes de sistemas embutidos (Ma, Xia & Peng 2008). Entre as iniciativas educacionais destaca-se o RLTOOL, uma extensão para o Scilab que facilita o projeto e análise de sistemas de controle monovariáveis (SISO - *Single In Single Out*), permitindo diversas interações, entre elas, a manipulação de pólos e zeros e a visualização dos efeitos da mudança de parâmetros do controlador no lugar das raízes e no desempenho do sistema (Pendharkar 2005). A fig. 1.1 mostra janelas do RLTOOL onde é possível visualizar o diagrama do lugar das raízes, as respostas no domínio do tempo e frequência de um regulador de tensão descrito em Silva & Cunha (2006).

A ideia para esse projeto nasceu da dificuldade em manter um sistema de aquisição de dados proprietário no Laboratório de Controle e Automação da Faculdade de Engenharia da UERJ. O laboratório possui placas de aquisição de dados mais antigas, de barramento ISA (*Industry Standard Architecture* - padrão de barramento de 8 ou 16 bits criado pela IBM na década de 1980) . Apesar da boa qualidade dessas placas sua utilização era limitada pois o *software* do fabricante impossibilitava atualizações do sistema operacional por falta de *drivers* de dispositivo e programas de aquisição apropriados, impedindo qualquer mudança de plataforma.

O objetivo principal é projetar e implementar um sistema de aquisição remota de dados totalmente baseado em *software* livre, utilizando o sistema operacional GNU/Linux, os equipamentos existentes no laboratório, fazendo modificações e desenvolvendo módulos adicionais para o Scilab/Scicos. O sistema é remoto pois a comunicação com as placas de aquisição de dados será feita via rede, num sistema cliente/servidor, onde o servidor é o responsável pela aquisição dos dados e envio ao cliente. Dessa forma, utiliza-se melhor os recursos do laboratório, visto que não há a necessidade de uma placa de aquisição em cada microcomputador.

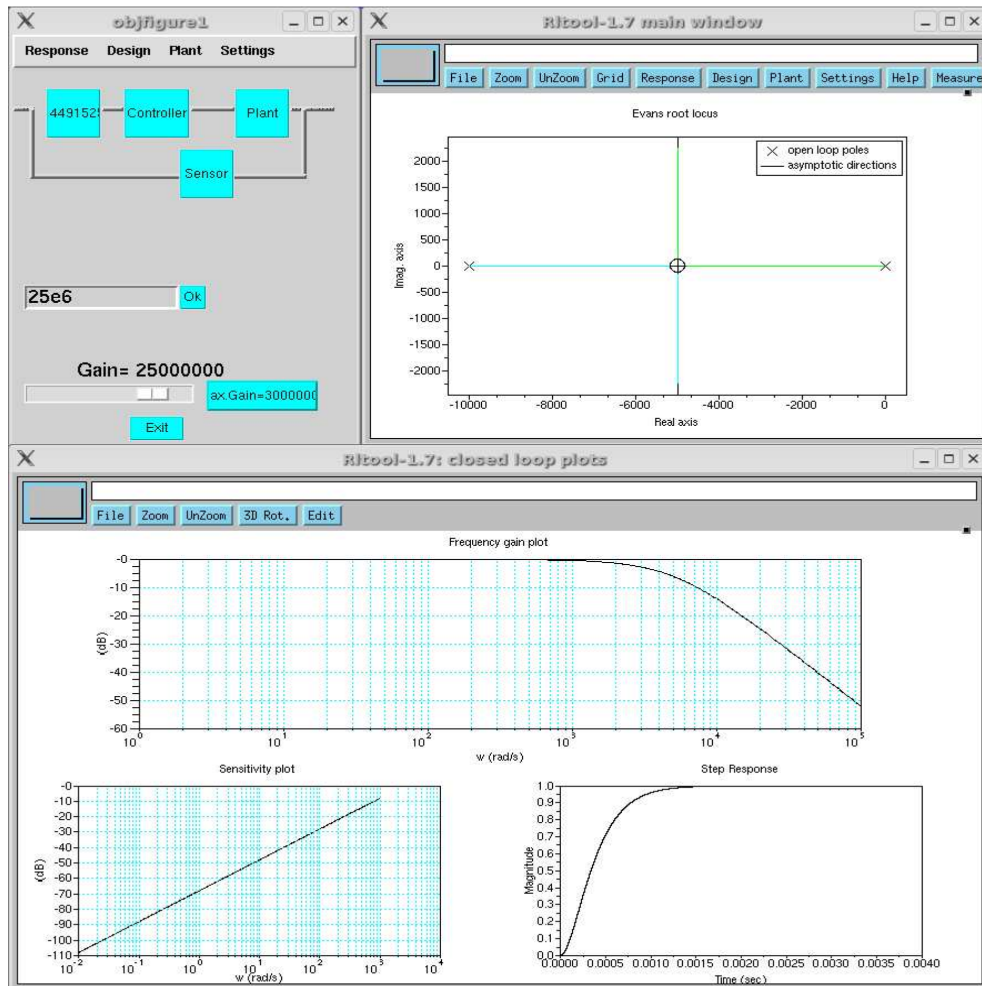

Figura 1.1: Janelas do RLTOOL com diagrama do lugar das raízes, resposta no domínio do tempo e frequência de um regulador de tensão.

1.1 Organização do trabalho

Este trabalho foi organizado da seguinte forma: o capítulo 2 descreve os principais fundamentos teóricos necessários ao desenvolvimento do trabalho. Foram abordados tópicos como o sistema operacional escolhido, as equações diferenciais utilizadas na modelagem de sistemas dinâmicos, os algoritmos que solucionam estas equações e os pacotes computacionais que implementam estes algoritmos, simulando os sistemas.Descreveu-se também o modelo de referência TCP/IP, o paradigma cliente/servidor, detalhes da API de redes para Linux na linguagem C além de detalhes sobre a inserção de novas funcionalidades aos pacotes computacionais Scilab/Scicos.

O capítulo 3 define a arquitetura e configuração do sistema de aquisição e o capítulo 4 exibe os resultados de testes com aquisição e síntese de sinais além de medições de atrasos na rede e

o comportamento do sistema no controle de um servomotor.

As conclusões são discutidas no capítulo 5.

Capítulo 2

Fundamentos Teóricos

Apresenta-se aqui os fundamentos teóricos necessários ao desenvolvimento do sistema de aquisição de dados proposto. A primeira seção faz uma breve introdução ao sistema operacional Linux e explica os principais motivos pelos quais ele foi escolhido para o desenvolvimento desse trabalho. Na primeira parte da segunda seção apresenta-se os *softwares* para computação numérica Scilab e seu simulador Scicos. A segunda parte da seção trata dos tipos de equações utilizadas na modelagem de sistemas dinâmicos, enquanto a terceira seção aborda os métodos numéricos empregados pelo Scilab/Scicos na resolução de equações diferenciais.

O controle por computador e o projeto Comedi, dedicado ao desenvolvimento de *drivers* para placas de aquisição de dados para Linux, são abordados na terceira seção. A quarta seção expõe as vantagens da utilização de um sistema de comunicação em rede e os motivos pelos quais os protocolos TCP e HTTP foram escolhidos para o desenvolvimento desse projeto. A quarta seção ainda apresenta detalhes sobre a programação para arquitetura cliente/servidor e sua implementação com a linguagem de programação C. A última seção é destinada à discussão mais detalhada sobre o funcionamento interno do Scilab e do Scicos de modo que se obtenha conhecimento suficiente para o desenvolvimento de interfaces entre o Scilab/Scicos e programas desenvolvidos pelo usuário. Ao fim da seção são mostrados exemplos de interfaces para Scilab e Scicos escritas na linguagem de programação C.

2.1 Sistema Operacional Linux

O Linux é um sistema *Unix like*, baseado em um outro sistema similar ao Unix, o Minix. A primeira versão do *kernel* do Linux foi lançada em 1991, por seu criador, Linus Torvalds. Desde então, o sistema conta com um grande número de programadores que se dedicam a

criar aplicativos e utilitários. Tanto o sistema operacional como grande parte destes aplicativos são gratuitos e licenciados de acordo com a licença GPL - *General Public License* (http://www.gnu.org/licenses/). Atualmente, com seu baixo custo de implantação e manutenção, eficácia e alta confiabilidade, o Linux tornou-se a principal alternativa aos sistemas operacionais proprietários (Guedes & Silva 2005).

O *kernel* do Linux possui todas as vantagens dos *kernels* dos sistemas Unix modernos: sistema de arquivo virtual, memória virtual, sinais, suporte a multiprocessamento simétrico (SMP - *symmetric multi processing*, arquitetura de computador onde processadores idênticos compartilham a mesma memória principal), processos leves (LWP - *light-weight processes*, linhas de execução de programa que utilizam o mínimo possível de informações sobre o contexto do processo), montagem e desmontagem dinâmica de módulos, etc. (Bovet & Cesati 1996).

O Linux não é um sistema de tempo real, pois o *kernel* sem modificações ainda não permite preempção total. Existem extensões para tempo real que, uma vez instaladas, permitem que o Linux seja executado como um sistema preparado para aplicativos que exigem execução em tempo real. Atualmente existem três projetos importantes que desenvolveram e mantém extensões para tempo real, um destes projetos é o RTAI - *Real-Time Application Interface* (https://www.rtai.org), desenvolvido pelos pesquisadores do *Dipartimento di Ingegneria Aerospaziale del Politecnico de Milano*. Essa extensão utiliza um modelo denominado *kernel* duplo. A ideia é ter um pequeno *kernel* de tempo real, que não é Linux, mas que executa o Linux como um processo de baixíssima prioridade. Nesse modelo, as aplicações de tempo real escritas para esse *kernel* usando uma API (*Application Programming Interface* - conjunto de programas e funções que permitem a interação com outros programas) são executadas com prioridade muito mais alta que o próprio Linux ou seus aplicativos (von Hagen 2005) dessa forma o escalonamento passa a ser realizado pela extensão e não mais pelo *kernel*. Entretanto a configuração desse tipo de extensão é complicada pois exige a compilação do *kernel*. Além disto, são necessários *drivers* de tempo real para os dispositivos que se pretende utilizar, como placas de aquisição de dados, e nem sempre estes *drivers* estão disponíveis como é o caso de dispositivos USB (Mannori, Nikoukhah & Steer 2006). Um outro problema na utilização de *kernel* duplo é que muitas vezes o computador utilizado para executar aplicativos de tempo real é um computador de uso geral e o usuário geralmente perde o controle da máquina para os processos

de tempo real, que executam em alta prioridade.

Existe ainda o UTIME, uma extensão para o *kernel* que programa o *timer* do microcomputador para gerar interrupções em intervalos de tempo menores que os 10ms usuais, podendo chegar até dezenas de microssegundos. A grande desvantagem desse método é que dependendo da frequência das interrupções geradas os recursos do sistema ficam alocados apenas para a geração dessas mesmas interrupções impossibilitando qualquer outra computação (Balaji, Ansari, Keimig & Sheth 2001).

Aplicativos que precisam de referências de tempo muito precisas, como o código que simule um controlador, podem ser prejudicados por incertezas nos instantes de amostragem e, os desvios podem gerar ruído e instabilidade na malha fechada de controle. Entretanto, o escalonador do *kernel* 2.6 foi reescrito por Ingo Molnar com objetivo de minimizar problemas com preempção, complexidade do algoritmo de escalonamento e melhorar o desempenho de sistemas SMP. O *kernel* 2.6 agora possui pontos de preempção que permitem que o escalonador funcione durante o tempo no qual um processo está executando em modo supervisor. Dessa forma, é possível (mas não garantido) interromper o processo corrente para escalonar um outro processo de prioridade mais alta. Além dessa modificação, o escalonador de processos foi reescrito de forma a diminuir a complexidade do algoritmo. O escalonador do *kernel* 2.4 tinha complexidade O(n), isto é, o tempo necessário para escalonar um processo era uma função do número de processos ativos no sistema. O novo escalonador teve sua estrutura de filas e listas modificada de forma que o algoritmo de escalonamento apenas precisa escolher o processo na lista de maior prioridade para executar. Agora o algoritmo é escalável pois possui complexidade O(1), o tempo de escalonamento é fixo, independente do número de processos ativos (Jones 2006).

Muito embora não seja um sistema operacional de tempo real, o Linux se mostra promissor em aplicações que exigem taxas de amostragem pequenas como o controle de sistemas eletromecânicos, com um tempo de amostragem por volta de 10ms. Årzén, Blomdell & Wittenmark (2005) demonstraram que é possível obter taxas de amostragem de até 2kHz com computadores PC, o *kernel* 2.4.18 e o UTIME. Ao topo de tudo isso soma-se o fato de o *hardware* estar cada vez mais avançado. Portanto não serão utilizados nesse projeto nenhum recurso de extensão do *kernel* para tempo real.

Além da questão financeira e o baixo tempo de latência, o Linux foi o sistema operacional

escolhido para o desenvolvimento desse projeto pois seu código fonte aberto e escrito na popular linguagem de programação C criam um ambiente propício ao estudo e desenvolvimento de novas ferramentas. Por ter herdado o sistema de ajuda *online*, as *man pages* do Unix, o Linux é um sistema operacional muito bem documentado e como seu desenvolvimento é realizado por programadores de diversas partes do mundo a informação sobre o sistema é muito bem difundida.

2.2 Modelagem e Simulação de Sistemas Dinâmicos

Para compreender melhor sistemas complexos muitas vezes faz-se necessária a criação de um modelo matemático quantitativo desse sistema. O modelo matemático de um sistema dinâmico é uma abstração de um sistema físico geralmente complexo, que de acordo com o problema a ser resolvido, teve algumas de suas variáveis suprimidas e foi linearizado.

Nessa seção são apresentados os *softwares* de cálculo numérico Scilab e seu simulador Scicos. São abordados também os tipos de equações mais utilizadas para a descrição de sistemas dinâmicos em tempo contínuo, em tempo discreto ou híbrido; os métodos numéricos empregados para a resolução por computador destas equações e como estes métodos são implementados pelo Scilab e o Scicos.

2.2.1 *Softwares* Scilab e Scicos

O Scilab é um *software* de computação numérica, de código fonte aberto e gratuito, desenvolvido para uso científico. Inclui bibliotecas (*toolboxes*) para diversas áreas, destacando-se processamento de sinais, sistemas de controle, simulação, otimização, integração numérica e álgebra linear. Há também muitos *toolboxes* desenvolvidas por usuários e disponíveis na Internet.

O Scilab também é uma linguagem de programação interpretada e fracamente tipada (não existe necessidade de declarar o tipo das variáveis). O tipos principais de objetos são listas, matrizes reais, complexas, tipo *string*, booleanas, polinomiais, racionais e esparsas.

É possível estender as funcionalidades do Scilab com programas escritos por usuários através de *links* dinâmicos ou estáticos com bibliotecas externas. O programa também possui uma interface gráfica capaz de desenhar gráficos 2D, 3D, curvas de nível e gráficos paramétricos

que podem ser exportados para diversos formatos, os quais destacam-se GIF, Xfig e Postscript-Latex. Também é possível desenvolver novas ferramentas gráficas por conta da interface com a linguagem Tcl/Tk.

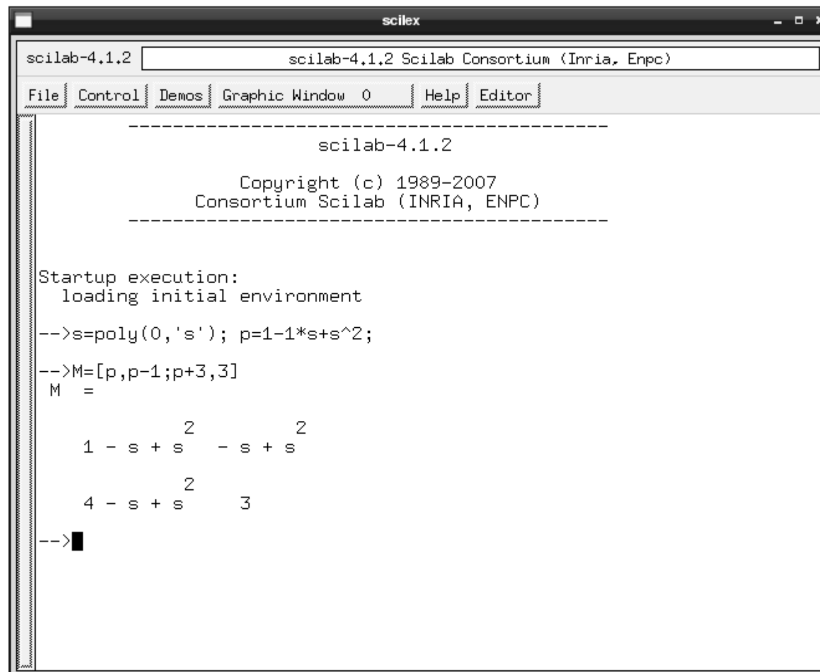

Figura 2.1: Janela do Scilab onde foi definida a matriz polinomial M.

O programa vem sendo desenvolvido em FORTRAN, C e na própria linguagem do Scilab. Foi iniciado na década de 1980 com o nome de Basile pelos pesquisadores da INRIA - *Institut National de Recherche en Informatique et en Automatique* e ENPC - *Ecole Nationale des Ponts et des Chaussées*, na França. Em 1994 teve seu nome alterado para o nome atual e tornou-se gratuito para *download* na Internet (`http://www.scilab.org`). Atualmente o Scilab é mantido por um consórcio denominado *Scilab Consortium* do qual participam empresas como EDF, Renault, Citroën e outras, além de institutos de ensino e pesquisa (Campbell, Chancelier & Nikoukhah 2005).

O Scicos - *Scilab Connected Object Simulator* é um *toolbox* do Scilab para a modelagem e simulação de sistemas dinâmicos de tempo contínuo, tempo discreto e sistemas híbridos através de diagramas de blocos interconectados. É muito apropriado em simulações de sistemas de controle, comunicação, processamento de sinais e no estudo de sistemas físicos e biológicos (Mannori et al. 2006).

Cada bloco representa uma função nativa ao Scicos ou escrita pelo usuário. Blocos nativos são organizados em grupos denominados *palettes*. Existem *palettes* com blocos especiais para sistemas lineares, geradores de sinais, elementos elétricos e hidráulicos entre outros.

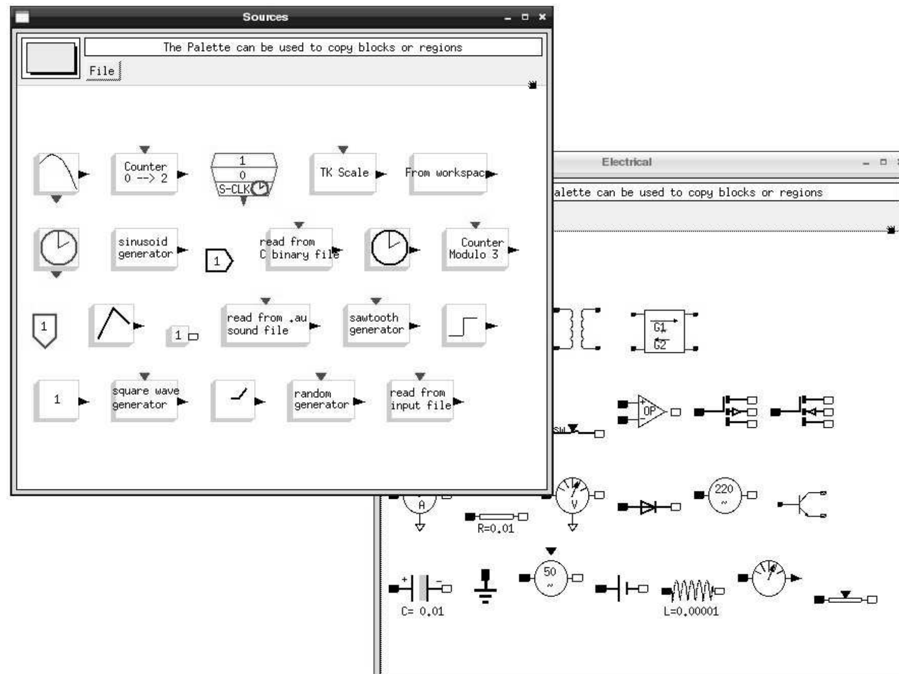

Figura 2.2: *Palettes* de fontes de sinais (*sources*) e de elementos elétricos (*electrical*).

2.2.2 Métodos Numéricos

Quando não se pode usar métodos analíticos ou expansões em série para resolver problemas de valor inicial com equações de primeira ordem do tipo:

$$\frac{dx}{dt} = f(t,x), \tag{2.1}$$

e condições iniciais $x(t_0) = x_0$, faz-se uso de aproximações da solução através de métodos numéricos. Estes métodos consistem em procedimentos envolvendo cálculos repetidos que podem ser implementados facilmente por linguagens de programação de alto nível.

Existem porém, diversos fatores a serem observados ao se recorrer aos métodos numéricos. Os algoritmos que implementam estes métodos de integração podem ter largura de passo (dis-

tância entre dois pontos onde a solução é calculada) constante ou variável (método adaptativo) de acordo com a complexidade do problema de valor inicial (PVI) a ser resolvido. Normalmente a largura de passo é escolhida tendo como base a tolerância a erros; um passo muito largo leva a resultados imprecisos e um passo muito pequeno leva à lentidão nos cálculos, aumentando o custo computacional.

Na aproximação da solução do PVI, alguns erros são introduzidos, sendo os erros de truncamento e os erros de arredondamento os mais significativos. Erros de truncamento são inerentes aos métodos numéricos, sendo introduzidos a cada passo devido ao truncamento de uma série infinita usada para aproximar funções. Se p é a ordem do método numérico e h é o passo, então o erro de truncamento por passo é da ordem de h^{p+1} (Pacitti & Atkinson 1977).

O erro de arredondamento ocorre quando determinado número é maior do que a palavra que pode ser amazenada na memória ou registradores da CPU. Esse erro é limitado por:

$$|e_r| \leq 0,5x10^{d-w}, \tag{2.2}$$

onde d é o expoente para o número na forma normalizada e w é o número de dígitos que se pode acumular (tamanho da palavra). O erro total é expresso pela soma destes dois erros (Boyce & DiPrima 2002).

Métodos de partida

Os métodos de partida ou métodos de passo simples são aqueles em que o valor aproximado num ponto depende apenas dos dados obtidos no ponto anterior, ou seja, para calcular x_i é necessário apenas o valor de x_{i-1}. Alguns destes métodos são o de Euler e de Runge-Kutta (Ruggiero & Lopes 1996).

O método de Euler é baseado na expansão da função $x(t)$ na série de Taylor simplificada:

$$x(t_0+h) = x(t_0) + h\dot{x}(t_0) + \frac{h^2}{2!}\ddot{x}(t_0) + \ldots \tag{2.3}$$

Truncando a série após o termo onde aparece a primeira derivada e substituindo $dotx$ por $f(t,x)$ obtemos a solução aproximada a partir do cálculo dos pontos sucessivos na equação:

$$x(t_0+h) \simeq x(t_0)+hf(t_0,x_0) \tag{2.4}$$

em sua forma mais genérica:

$$x_{n+1} = x_n + hf(t_n,x_n) \tag{2.5}$$

onde:

$n = 0,1,2,\ldots$

$h = (t_{n+1}) - t_n =$ passo

Para equações de primeira ordem, com intervalo de integração e passo pequenos o método de Euler apresenta bons resultados. Porém, muito embora esse seja um método de fácil implementação, apresenta a desvantagem de necessitar de muitos cálculos e uma largura de passo muito pequena para obter uma aproximação aceitável para a solução. Esse método de primeira ordem possui erro de truncamento proporcional a h^2 (Pacitti & Atkinson 1977).

O método de Euler pode ser melhorado utilizando conceitos de predição e correção e limites de convergência fixados em cada passo da iteração. Esse método é conhecido como método modificado de Euler e consiste em (i) calcular a cada passo a previsão para o valor da função $x_{i+1}(t)$ num ponto usando o método de Euler ($x_{n+1} = x_n + hf(t_n,x_n)$); (ii) calcular o valor previsto para x'_{i+1} através de $x'_{i+1} = f(t_{i+1},x_{i+1})$; (iii) corrigir os pontos calculando $x_{i+1}(t)$ através do método dos trapézios:

$$I \cong \frac{h}{2}[f(t_0)+2\sum_{i=1}^{n-1} f(t_i)+f(t_n)] \tag{2.6}$$

(iv) e sua derivada através de:

$$x'_{i+1} = f(t_{i+1},x_{i+1}) \tag{2.7}$$

dessa forma calcula-se o valor estimado para $x_{i+1}(t)$ usando o método de Euler e corrige-se esse valor com o método dos trapézios. Essa iteração continua até que o valor corrigido de

$x_{i+1}(t)$ seja diferente do valor anterior a menos de ε, que é o limite de convergência (Pacitti & Atkinson 1977).

A principal vantagem de método de Euler modificado sobre o método simples é o fato de o método modificado utilizar menos iterações para obter resultados precisos. Porém, problemas com regiões críticas (descontinuidades) requerem um maior número de iterações e, quanto maior o número destas, maior o erro de arredondamento.

Outro método frequentemente utilizado é o de Runge-Kutta. Ele pode ser usado para obter a solução completa de um PVI ou apenas pontos iniciais para métodos de passo múltiplo. Do ponto de vista computacional esse método é pouco eficiente. Seu princípio básico é o cálculo repetido da média ponderada de valores de $f(t,x)$ em pontos do intervalo $t_n \leq t \leq t_{n+1}$. O método de quarta ordem é o que conduz a soluções mais precisas a partir de uma largura de passo relativamente grande. A expressão para o método de quarta ordem é:

$$x_{n+1} = x_n + h\left(\frac{k_{n1} + 2k_{n2} + 2k_{n3} + k_{n4}}{6}\right) \tag{2.8}$$

onde:

$$k_{n1} = f(t_n, x_n)$$

$$k_{n2} = f(t_n + \tfrac{1}{2}h, x_n + \tfrac{1}{2}hk_{n1})$$

$$k_{n3} = f(t_n + \tfrac{1}{2}h, x_n + \tfrac{1}{2}hk_{n2})$$

$$k_{n4} = f(t_n + \tfrac{1}{2}h, x_n + \tfrac{1}{2}hk_{n3})$$

Métodos de passos múltiplos

Os métodos de passos múltiplos surgiram devido à deficiência dos métodos de partida e atualmente figuram como os métodos mais utilizados na resolução de PVIs por computador. Estes métodos utilizam mais do que um ponto anterior para o cálculo do próximo ponto. O número de pontos necessários é igual ao número de ordem do método. Dois destes métodos são: Adams e fórmulas inversas de diferenciação (BDFs - *backward differentiation formulas*).

O método de Adams pretende aproximar a derivada da solução de um problema de valor inicial a um polinômio $P_k(t)$ de grau $k-1$. Esse polinômio é então usado para calcular a

seguinte integral:

$$\phi(t_{n+1}) - \phi(t_n) = \int_{t_n}^{t_{n+1}} \phi'(t)dt \tag{2.9}$$

onde ϕ representa uma solução do problema de valor inicial.

Usando um polinômio de primeiro grau $P_2(t) = At + B$ com pontos (t_n, x_n) e (t_{n-1}, x_{n-1}) obtemos a fórmula de Adams-Bashforth de segunda ordem:

$$x_{n+1} = x_n + \frac{3}{2}hf_n - \frac{1}{2}hf_{n-1} \tag{2.10}$$

e um polinômio de grau três geraria:

$$x_{n+1} = x_n + \left(\frac{1}{24}h\right)(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}) \tag{2.11}$$

As fórmulas de Adams-Bashforth são explícitas para x_{n+1} e possuem erro de trucamento proporcional a h^{k+1}.

Uma pequena mudança na escolha dos pontos para a dedução da fórmula gera outro conjunto de fórmulas denominado Adams-Moulton. Considerando um polinômio de quarta ordem e pontos (t_n, x_n), (t_{n+1}, x_{n+1}), (t_{n+2}, x_{n+2}) e (t_{n+3}, x_{n+3}) obtém-se a fórmula de Adams-Moulton de quarta ordem:

$$x_{n+1} = x_n + \left(\frac{1}{24}h\right)(9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2}) \tag{2.12}$$

Como x_{n+1} aparece nos dois lados da fórmula (dentro de f_{n+1}) ela é dita implícita e são necessários mais estágios para sua solução.

O erro de truncamento é igual ao da fórmula de Adams-Bashforth, mas para ordens não muito altas as fórmulas de Adams-Moulton são mais precisas (Boyce & DiPrima 2002).

As duas fórmulas de Adams são usadas no método de previsão e correção. Esse método combina as duas fórmulas, unindo a simplicidade e precisão na aproximação da solução de PVIs. Para aplicar esse método calculam-se os valores de $x_{n-3}, x_{n-2}, x_{n-1}$ e x_n por algum método de partida, geralmente Runge-Kutta. Estes valores serão usados como valores iniciais para o método de passo múltiplo. Depois usa-se a fórmula de Adams-Bashforth para "prever", calcular uma aproximação para x_{n+1} e depois usa-se Adams-Moulton para "corrigir", refinar esse

valor.

As fórmulas inversas de diferenciação (BDF - *backward differentiation formula*) também fazem uso de um polinômio, mas, ao contrário das fórmulas de Adams, esse polinômio é aproximado da solução ϕ(t) e não de sua derivada. Depois de definido, o polinômio é derivado e igualado a $f(t_{n+1}, x_{n+1})$ para obtenção de uma fórmula implícita para x_{n+1}. Um polinômio de primeiro grau geraria a seguinte fórmula:

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}) \tag{2.13}$$

O erro de truncamento é igual ao das fórmulas de Adams, h^{k+1}.

Estabilidade e equações rígidas

Procedimentos matemáticos recursivos, como a maioria dos métodos de resolução numérica de equações diferenciais, estão sujeitos a pequenos erros que podem ser reduzidos ou aumentados a cada iteração. Soluções estáveis são aquelas que tendem a se aproximar da solução exata, portanto, ao resolver um PVI numericamente é desejável que a solução aproximada esteja mais próxima possível da solução exata. Não é possível encontrar aproximações estáveis, sem variações bruscas, para um problema naturalmente instável simplesmente resolvendo-o numericamente. Porém é possível adicionar pontos de instabilidade a um sistema estável com o uso de métodos numéricos. Essa instabilidade pode ser evitada impondo restrições à largura do passo.

Na procura pelo valor ótimo da largura de passo é necessário atingir o equilíbrio entre os erros de truncamento e arredondamento. Em geral, para tamanhos de passo grandes o erro de arredondamento começa a sobressair e se torna parte predominante do erro total.

Algumas vezes, ao analisar-se um PVI, chega-se à conclusão que a largura de passo definida para que a solução seja estável é muito menor que o tamanho de passo necessário para uma solução de boa precisão. Esse PVI é então classificado como um *problema rígido*.

Problemas rígidos são compostos de equações diferenciais para as quais a maioria dos métodos de integração numéricos produzem soluções instáveis, com variações bruscas. Os melhores métodos para simular estas equações são as BDFs.

A fig. 2.3 mostra as soluções obtidas para o PVI:

$$\dot{x} = -100y + 100t + 1 \tag{2.14}$$

Com $x(0) = 1$ através dos métodos de Euler, Runge-Kutta, BDFs e a solução exata, $x = e^{-100t} + t$, simulados através de um *script* no Scilab.

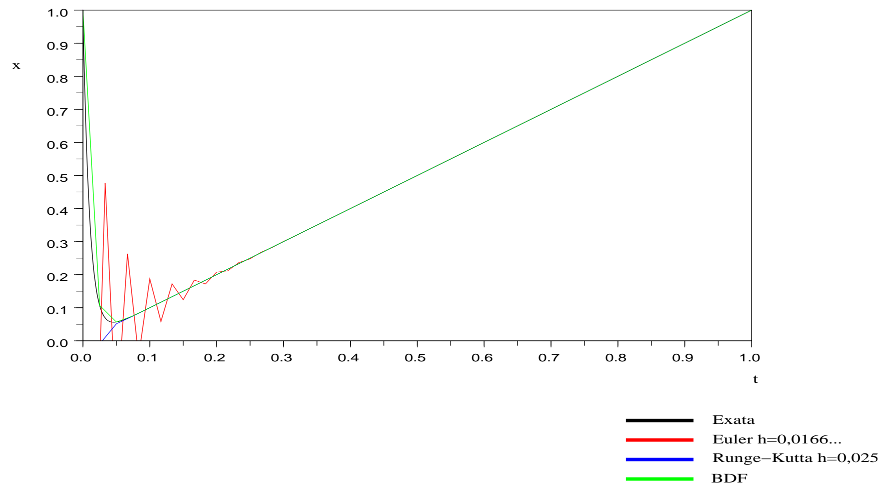

Figura 2.3: Métodos de integração numérica aplicados a um problema rígido.

A fig. 2.4 mostra o detalhe do gráfico comparativo entre os métodos numéricos (Euler, Runge-Kutta e BDF) para a resolução de um problema rígido. Note-se que a solução simulada pelas fórmulas inversas de diferenciação (BDF) é a mais precisa nesse contexto.

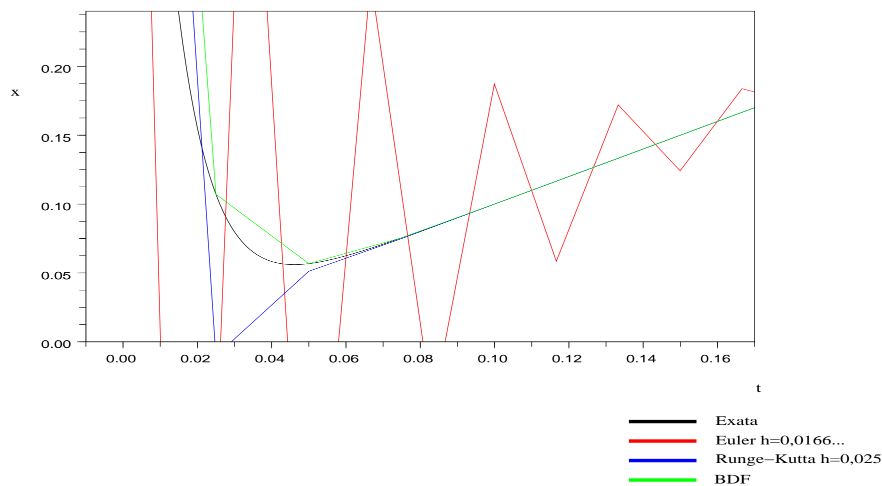

Figura 2.4: Detalhe do gráfico a fig. 2.3.

Analisando os principais métodos numéricos percebe-se que, em geral, os métodos de passos

múltiplos calculam o valor de f menos vezes a cada passo, o que os torna mais rápidos. Em compensação, se o método de Runge-Kutta é mais preciso para a solução de determinado PVI e usa menos passos, é possível que a diferença na velocidade seja superada.

Métodos que utilizam equações implícitas como as fórmulas de Adams-Moulton e as BDFs podem demandar mais tempo computacional devido à dificuldade em se resolver esse tipo de equações.

Finalmente, os métodos de passos múltiplos podem propagar erros ocorridos em passos anteriores. Sendo assim, a escolha de um método numérico não é uma tarefa simples, exigindo conhecimento prévio sobre o sistema que se pretende simular.

Existem inúmeros programas de integração numérica disponíveis para uso, sendo o pacote ODEPACK o mais famoso deles. O pacote ODEPACK foi escrito em Fortran-77 e possui nove integradores que implementam soluções para grande parte dos sistemas físicos como sistemas explícitos, sistemas linearmente implícitos ($A(t,x)\frac{dx}{dt} = g(t,x)$), sistemas rígidos, etc. A maioria destes integradores foi escrita por Alan Hindmarsh, do Laboratório Nacional de Lawrence Livermore (Livermore, Califórnia) (Hindmarsh 2001).

Um sistema que pode ser escrito na forma $\frac{dx}{dt} = f(t,x)$ é dito explícito e os programas do ODEPACK que podem ser utilizados são: LSODE, LSODES, LSODA, LSODAR, LSODPK e LSODKR. Alguns destes programas são brevemente descritos abaixo:

- LSODE (*Livermore Solver for Ordinary Differential Equations*) - é o integrador mais simples do pacote. Ele pode resolver problemas rígidos ou não-rígidos. O método de Adams (previsão-correção) é utilizado para a solução de problemas não-rígidos e as fórmulas BDF para problemas rígidos;

- LSODES - similar ao LSODE, mas para problemas rígidos a matriz Jacobiana é esparsa e tratada por rotinas especiais. Escrito em colaboração com A. H. Sherman;

- LSODA - esse programa inicia usando o método BDF para problemas rígidos e monitora os dados dinamicamente para decidir se um integrador para problemas não-rígidos é o mais adequado, isto é, dependendo do comportamento do sistema o programa escolhe o melhor método. Escrito em colaboração com Linda Petzold;

- LSODAR - variação do LSODA com capacidade para encontrar raízes. Além das características do LSODA esse programa também implementa um algoritmo que pode encontrar as raízes de qualquer conjunto de funções na forma $g(t,x)$. Esse método é particularmente eficiente quando o sistema possui pontos de descontinuidade onde o integrador precisa ser reiniciado. Também escrito em colaboração com Linda Petzold.

Os programas LSODI, LSOIBT e LSODIS são específicos para resolução de sistemas linearmente implícitos na forma $A(t,x)\frac{dx}{dt} = g(t,x)$ onde A é uma matriz quadrada. Se A é singular o sistema é modelado por equações algébrico-diferenciais (EADs) (Hindmarsh 2001).

2.2.3 Equações Diferenciais Utilizadas na Modelagem de Sistemas Dinâmicos

Modelagem por Equações Diferenciais Ordinárias

O comportamento dinâmico de sistemas em tempo contínuo, sejam eles mecânicos, hidráulicos, elétricos ou processos químicos, geralmente pode ser descrito através de modelos matemáticos envolvendo sistemas de equações diferenciais ordinárias (EDOs).

O sistema de controle de velocidade de um motor DC controlado por armadura citado em Dorf (1992), seção 2.9, e mostrado na fig. 2.5 será usado para exemplificar a modelagem por EDOs de sistemas dinâmicos em tempo contínuo.

O principal elemento desse sistema é o motor DC de ímã permanente (PM - *permanent magnet*) controlado por armadura cujo esquema elétrico foi ilustrado na fig. 2.6. Motores DC de ímã permanente têm sido muito utilizados em aplicações de baixa potência devido à alta relação torque-volume, alta relação torque-inércia, controlabilidade da velocidade, curvas de velocidade-torque bem definidas e adaptabilidade a diversos métodos de controle (Kuo 1985). Nestes motores o enrolamento de campo é substituído pelo ímã simplificando sua construção, tornando-os menores. Além disso estes ímãs não necessitam excitação externa. As desvantagens do uso destes motores ocorrem devido ao próprio uso de ímãs: existe o risco de desmagnetização e há limitações com relação ao fluxo magnético que pode ser produzido no entreferro, o que compromete a aplicação das técnicas de análise de circuitos magnéticos. Porém estas limitações vêm sendo superadas com o emprego de novos materiais magnéticos como os ímãs permanentes de terras raras, destacando-se o samário-cobalto e o neodímio-ferro-boro. Estes

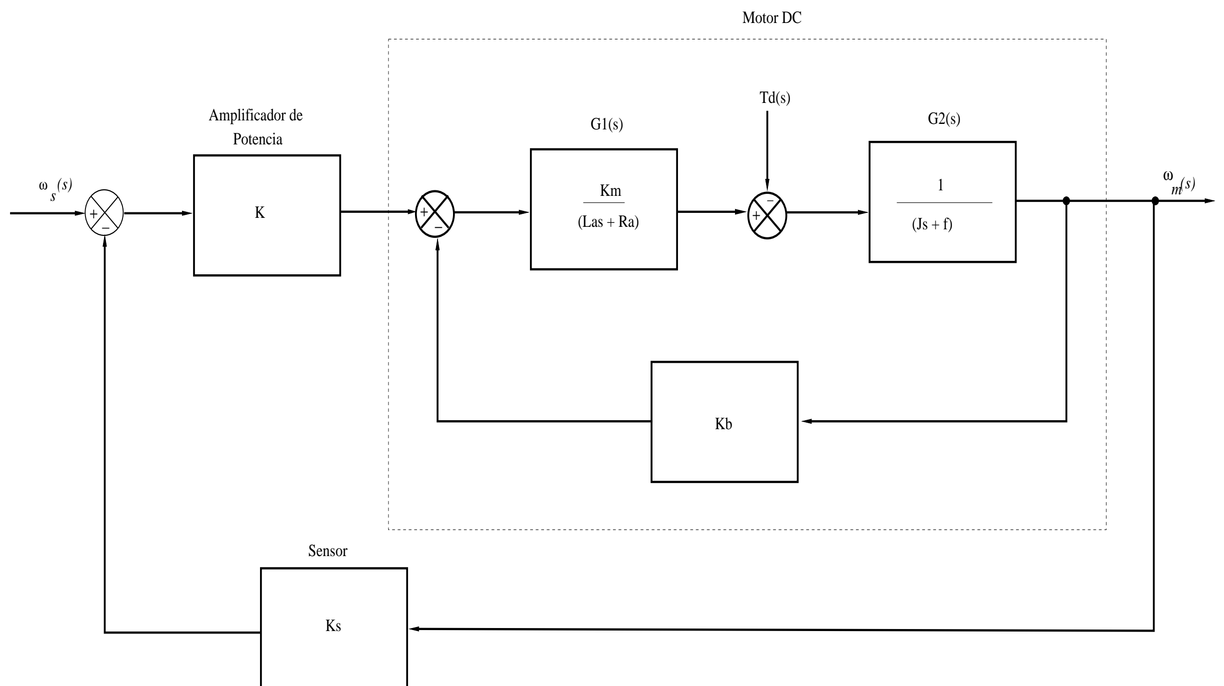

Figura 2.5: Diagrama de blocos de um sistema de controle de velocidade.

novos materiais apresentam maiores valores de densidade de fluxo residual, e coercitividade (medida da intensidade da força magnetomotriz necessária para desmagnetizar o material ou a capacidade de produção de fluxo em circuito magnético com entreferro) (Fitzgerald 2006).

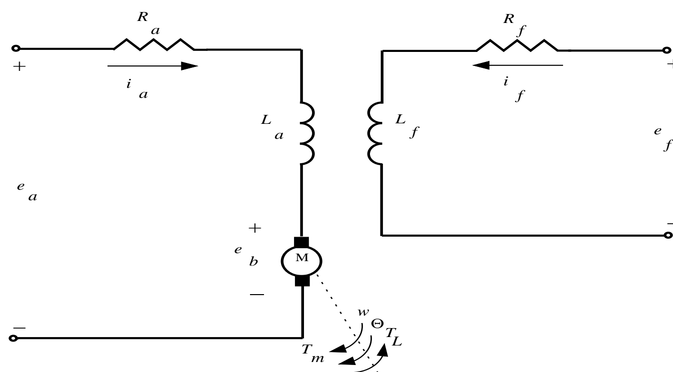

Figura 2.6: Esquema elétrico de um motor DC, controlado por armadura.

Outros elementos do sistema são o amplificador de potência cujo modelo foi linearizado, o amplificador operacional com entrada diferencial utilizado como detetor de erro e o tacômetro utilizado como sensor na malha de retroação.

No desenvolvimento desse modelo foram desprezados efeitos de segunda ordem no motor como a histerese e a queda de tensão através das escovas. A eq. 2.15 representa as equações de

estado na forma canônica.

$$\begin{bmatrix} \frac{di_a(t)}{dt} \\ \frac{d\omega_m(t)}{dt} \end{bmatrix} = \begin{bmatrix} -\frac{R_a}{L_a} & -\frac{K_b - KK_s}{L_a} \\ \frac{K_m}{J} & -\frac{f}{J} \end{bmatrix} \begin{bmatrix} i_a(t) \\ \omega_m(t) \end{bmatrix} + \begin{bmatrix} \frac{K}{L_a} \\ 0 \end{bmatrix} \omega_s(t) \tag{2.15}$$

onde:

R_a é a resistência de armadura;

L_a é a indutância de armadura;

$i_a(t)$ é a corrente de armadura;

$\omega(t)$ é a velocidade angular do rotor;

K_m é a constante de torque;

K é o ganho do amplificador de potência;

K_s é o ganho do sensor;

K_b é o constante de força contra-eletromotriz;

J é a inércia do rotor;

f é o coeficiente de atrito viscoso.

A eq. 2.16 apresenta a função de transferência do sistema.

$$\frac{KK_m}{s^2(JL_a) + s(R_aJ + L_af) + R_af + KK_sK_m + K_bK_m} \tag{2.16}$$

A eq. 2.16 é uma função de transferência de segunda ordem, portando é possível escrevê-la na forma $\frac{C(s)}{R(s)} = \frac{\omega_n^2}{s^2+2\zeta\omega_n s+\omega_n^2}$. Substituindo os valores de acordo com a Tabela 2.1 e simplificando a fração, obtém-se a eq. 2.17

$$\frac{2700}{s^2 + 1,25s + 2700,75} \tag{2.17}$$

Tabela 2.1: Parâmetros do sistema de controle de velocidade.

Parâmetro	Valor
Constante de torque	$K_m = 10$
Ganho do amplificador de potência	K = 10
Ganho do sensor	$K_s = 1$
Resistência de armadura	$R_a = 1\ \Omega$
Indutância de armadura	$L_a = 1$ H
Inércia total (rotor + carga)	$J_m = 2$ kg m^2
Constante de força contra-eletromotriz	$K_b = 0{,}1$ Vs/rad
Coeficiente de atrito viscoso total (eixo do motor + eixo da carga)	$f = 0,5$

Sendo um sistema de segunda ordem subamortecido, com $\omega_n \cong 52$ e $\zeta = 0,012$, espera-se uma resposta altamente oscilatória (Dorf 1992).

A ferramenta utilizada pelo Scilab para a solução de ODEs é a função `ode`. A forma mais simples de chamada à função `ode` é :

```
-->x = ode(x0,t0,t,f);
```

onde:

$x0$ é o valor inicial;

$t0$ é o tempo inicial de integração;

$t = [t_0, t_1, \ldots, t_n]$ é o vetor de tempos, sendo o último elemento o tempo final de integração;

$x = [x_0, x_1, \ldots, x_n]$ é o vetor de valores estimados para a solução;

f é a função que se quer integrar.

A função a ser integrada pode ser declarada no Scilab dessa forma:

```
-->function dxdt = f(t,x)

-->dxdt = -x + sin(t)

-->endfunction
```

O sistema de controle de velocidade foi simulado usando a função ode. A tab.2.2 ilustra trechos do algoritmo escrito na linguagem do Scilab e a fig. 2.7 mostra a resposta ao degrau. Note-se o sobresinal de $\approx 0,96$ e o tempo de acomodação de $\approx 6,41$ s.

Tabela 2.2: Simulação do sistema de controle de velocidade.

```
// Definição das variáveis de estado x1:=ia e x2:=wm.
t0=0;            // Tempo inicial (s).
x0=[0 ; 0];      // Estado inicial (A,rad/s).
tmax=15;         // Duração da simulação (s).
deltat=0.01;     // Passo de integração (s).

// Parâmetros do modelo:
Km=10;           // Cte torque.
K=10;           // Ganho amplificador.
Kt=1;            // Ganho tacometro.
Kb=0.1;          // Cte fcem (Vs/rad).
Ra=1;            // Resistência de armadura (ohms).
La=1;            // Indutância da armadura (H).
J=2;             // Inércia (kg*m^2).
f=0.5;           // Coef. atrito viscoso.

// Matrizes A e b da equação de estado:
A=[-Ra/La (-K*Kt - Kb)/La ; Km/J -f/J ];
b=[(K)/La; 0];

t = [t0:deltat:tmax];  // Gera o tempo.

// Definição da função a ser integrada (dx/dt):
function dxdt=xdot(t,x)
   dxdt=A*x+b;        // Equação de estado.
endfunction

// Resolve a equação diferencial
// via Runge-Kutta RKF45:
x = ode("rkf",x0,t0,t,xdot);
```

Entretanto, nem sempre a forma simples da chamada à função ode retorna os valores esperados. Para estas situações convém utilizar uma das formas mais completas do comando ode expandido. Com o comando ode expandido é possível escolher o método, a tolerância a erros relativos e absolutos, o Jacobiano (se necessário) e outros. Como mencionado anteriormente, a escolha do método numérico é delicada pois muitos fatores devem ser levados em consideração. Alguns problemas são modelados por equações com muitas descontinuidades. Estes pontos de descontinuidades são críticos para o integrador, que deve interromper a integração, retornar o instante de tempo no qual ocorreu o salto ao programa principal e reiniciar desse

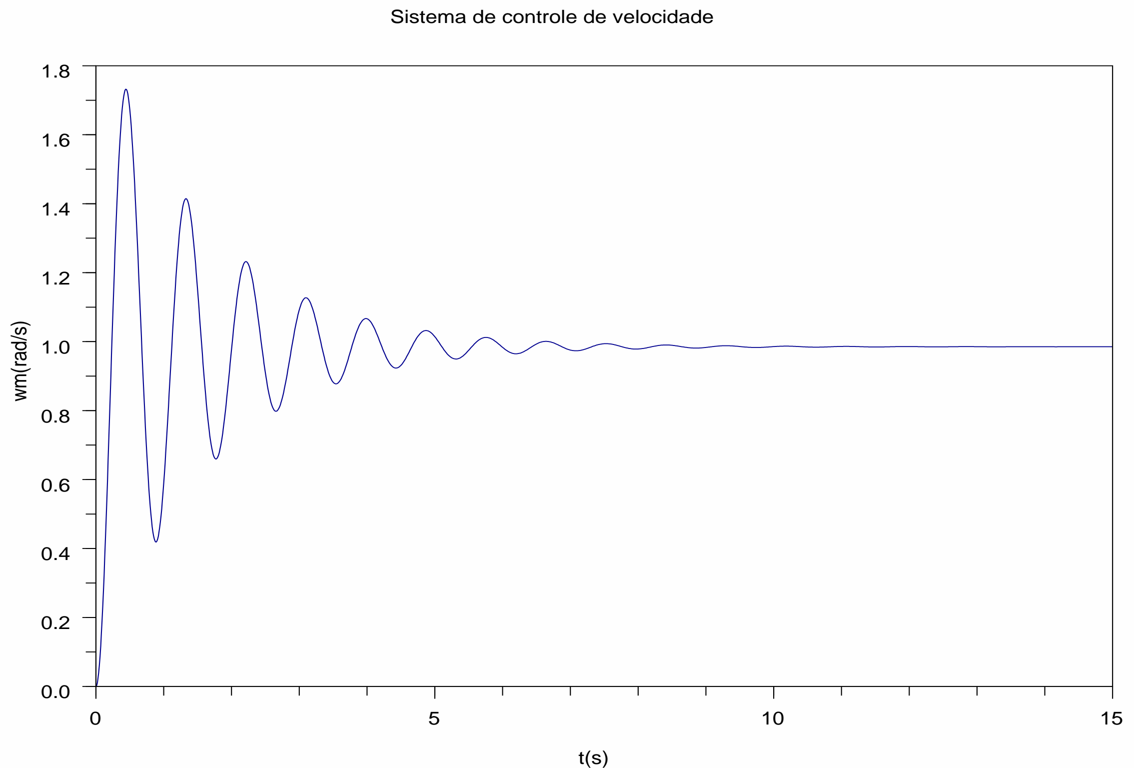

Figura 2.7: Resposta ao degrau do sistema de controle de velocidade.

ponto. Métodos de partida como Runge-Kutta têm vantagens sobre métodos de passos múltiplos na resolução destes problemas. Por outro lado, os métodos de passos múltiplos requerem menos esforço computacional, sendo mais rápidos (Campbell et al. 2005).

A função `ode` no modo expandido pode ser chamada da seguinte forma:

```
-->[x,w,iw] = ode([type],x0,t0,t[,rtol [,atol]],f [,jac] [,w,iw]);
```

Variáveis dentro de colchetes são opcionais e *type* é uma *string* que indica o método a ser utilizado. Em sua maioria, os métodos disponíveis fazem parte do pacote ODEPACK como: *lsoda, adams, stiff, rk, rkf, fix, root* e *discrete*.

As variáveis *rtol* e *atol* são os erros relativos e absolutos representadas por constantes ou vetores reais de mesmo tamanho de *x*. Valores padrão para *rtol* são 1×10^{-5} para a maioria dos casos e 1×10^{-3} para os métodos *rkf* e *fix*. Os valores para *atol* são 1×10^{-7} para a maioria dos casos e 1×10^{-4} para os métodos *rkf* e *fix*. Configurar os erros relativo e absoluto com valores muito pequenos pode levar a falhas no integrador quando as tolerâncias não puderem

ser satisfeitas. Valores muito altos de *rtol* e *atol* podem levar a soluções imprecisas.

O parâmetro *jac* é usado quando é necessário fornecer o Jacobiano e os parâmetros *w* e *iw* são usados quando é necessário armazenar dados para a reinicialização do método.

O Scilab possui um manual para suas funções. Para saber mais sobre o uso da função `ode` basta digitar `apropos ode` no *prompt* do Scilab.

Modelagem por equações algébrico-diferenciais

Ao longo dos últimos anos verificou-se a necessidade de modelagens usando equações diferenciais em conjunto com equações algébricas. As equações algébricas expressam relações entre as variáveis de estado de determinados sistemas dinâmicos. O circuito RLC com uma fonte de corrente dependente apresentado em Bazanella (2007) e mostrado na fig.2.8 é um bom exemplo desse tipo de sistema pois seu modelo matemático possui equações diferenciais e algébricas. As equações escritas a partir das leis de Kirchhoff descrevem, por exemplo, o comportamento das correntes em cada nó e suas relações com outros nós. Estas equações nem sempre são lineares e os métodos numéricos existentes para a resolução destas equações são complexos, exigindo maior tempo computacional.

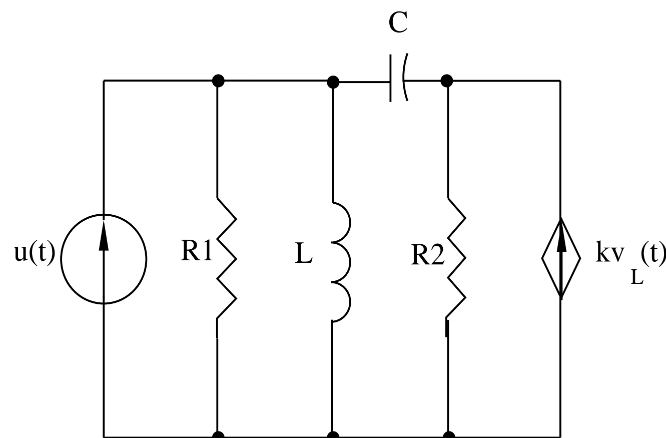

Figura 2.8: Circuito com fonte de corrente dependente.

Definimos a corrente no indutor ($i_L(t)$) e a tensão no capacitor ($v_C(t)$) como variáveis de estado. A lei das malhas de Kirchhoff na malha envolvendo o indutor, o capacitor e o resistor R_2 fornece:

$$v_{R_2}(t) = -v_C(t) + \frac{Ldi(t)}{dt}. \tag{2.18}$$

Usando a lei dos nós de Kirchhoff obtemos as seguintes equações:

$$\frac{Ldi(t)}{dt}\frac{1}{R_1} = -i_L(t) - C\frac{dv_C(t)}{dt} + u(t) \tag{2.19}$$

$$C\frac{dv_C(t)}{dt} = \left(-v_C(t) + \frac{Ldi(t)}{dt}\right)\frac{1}{R_2} - k\frac{Ldi(t)}{dt} \tag{2.20}$$

Substituindo a eq. 2.19 e eq. 2.20 na eq. 2.18 e os valores de R_1 e R_2 tem-se:

$$(2-k)\frac{dv_C(t)}{dt} = (k-1)i_L(t) - v_C(t) + (1-k)u(t). \tag{2.21}$$

Se $k = 2$, então a tensão no capacitor, a corrente no indutor e a entrada estão relacionadas pela seguinte equação algébrica:

$$i_L(t) - v_C(t) - u(t) = 0. \tag{2.22}$$

Sistemas compostos por partes dinâmicas (sistemas com memória) e partes algébricas (sem memória) são ditos sistemas singulares ou descritores (Bazanella 2007). A parte dinâmica destes sistemas é melhor representada por equações diferenciais enquanto que as restrições impostas pelas relações entre as variáveis de estado são melhor definidas por equações algébricas. Torna-se conveniente então a modelagem destes sistemas por equações algébrico-diferenciais (EADs).

O estudo destas equações teve início na década de 1960 e ainda hoje a solução através de métodos numéricos é muito discutida. A técnica de resolução mais simples desse tipo de equações é a eliminação das variáveis algébricas de forma a obter uma EDO. Porém essa eliminação nem

sempre é possível. Além disto, às vezes é difícil encontrar condições iniciais coerentes pois elas devem satisfazer o sistema de EADs e algumas derivadas da parte algébrica (Lourenço 2002).

Um sistema de EADs pode se apresentar nas seguintes formas (Najafi & Nikoukhah 2006):

- sistema semi-explícito:

$$\begin{cases} \dot{x} = f(x, y, u, t) \\ 0 = g(x, y, u, t) \end{cases} \tag{2.23}$$

  Na eq. 2.25 a parte diferencial e algébrica estão separadas. Variáveis cujas derivadas aparecem na equação são denominadas variáveis diferenciais (x) enquanto que as outras variáveis são denominadas variáveis algébricas.

- sistema totalmente implícito:

$$0 = F(\dot{z}, z, t),\ z = \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.24}$$

  Se $\frac{\partial F}{\partial \dot{z}}$ é uma matriz não-singular é possível resolver $\dot{z}$ em função de z. O sistema de EADs é então transformado em um sistema de EDOs.

Existem diversos tipos de índices que classificam sistemas de EADs quanto à complexidade de resolução, seja ela analítica ou numérica. Serão discutidos aqui o índice diferencial e o índice singular. O índice diferencial define o menor número de diferenciações em relação a t que um sistema de EADs precisa passar para tornar-se um sistema de EDOs explícito ($\dot{x} = F(t, x)$). O menor valor para esse índice é zero, caracterizando um sistema de EDOs.

O índice singular é definido como 1 mais o número de diferenciações necessárias para tornar a matriz $\frac{\partial F}{\partial \dot{z}}$ não-singular. Se o índice singular é maior que 1 o sistema começa a apresentar problemas de inicialização (quando não é possível encontrar um valor inicial que satisfaça as equações diferenciais e as equações algébricas). A estratégia aplicada na resolução de EADs com índice singular maior que 1 é a de diferenciar as equações algébricas até que todo o sistema seja constituído de equações diferenciais. Porém esse método pode trazer mais problemas pois com o surgimento de mais EDOs surgem também mais soluções dificultando a convergência

entre as soluções aproximadas e a solução exata (Lourenço 2002). Além disto, a diferenciação das equações do sistema pode resultar no surgimento de equações algébricas que estavam implícitas e o sistema ainda permanece com alguma parte algébrica. Sistemas de EADs que ao serem diferenciados apresentam mais equações algébricas são denominados sistemas estendidos.

Além dos problemas de índice, as EADs apresentam problemas de inicialização. A determinação de condições iniciais para sistemas de EADs é uma das etapas mais difíceis da resolução pois uma condição deve satisfazer não só o sistema original como também as equações intermediárias geradas pela diferenciação do sistema. A simulação de uma EDO pode começar de qualquer estado inicial, mas a simulação de uma EAD deve iniciar de um estado inicial coerente (Lourenço 2002).

O Scilab utiliza uma ferramenta denominada `dassl`, que implementa o código DASSL desenvolvido por Linda Petzold para a resolução de EADs de índice diferencial 1 e EDOs implícitas. O manual do Scilab fornece exemplos de uso desta função.

O Scicos utiliza o programa DASKR também escrito por Linda Petzold para resolver EADs de índice diferencial 1. Entretanto ainda é muito difícil para os simuladores resolver EADs descontínuas. Uma solução para esse problema seria resolver as equações simbolicamente ao invés de numericamente. A linguagem de programação Modelica foi desenvolvida com o propósito de modelar sistemas físicos através de sua descrição simbólica. Essa linguagem foi recentemente adicionada ao Scicos no intuito de simular sistemas híbridos com EADs e ODEs implícitas. A inclusão desta linguagem ao Scicos permitiu a construção de blocos não-causais, implícitos, onde as entradas e saídas não estão bem definidas. A modelagem com blocos implícitos é denominada modelagem em nível de componente.

Além dos blocos de componentes elétricos, hidráulicos e térmicos foi adicionado ao Scicos o bloco Modelica que permite que o usuário escreva seu sistema na linguagem Modelica. As eq. 2.25 foram codificadas para simular o circuito da fig. 2.8.

A fig. 2.10 mostra o resultado da simulação realizada pelo Scicos:

$$\begin{cases} \dfrac{di_l(t)}{dt} = -i_l - \dfrac{dv_c(t)}{dt} + u(t) \\ \dfrac{dv_c(t)}{dt} = -v_c - \dfrac{di_l(t)}{dt}; \end{cases} \tag{2.25}$$

com $u(t) = sen(t)$, $i_l(0) = 0$ e $v_c(0) = 0$.

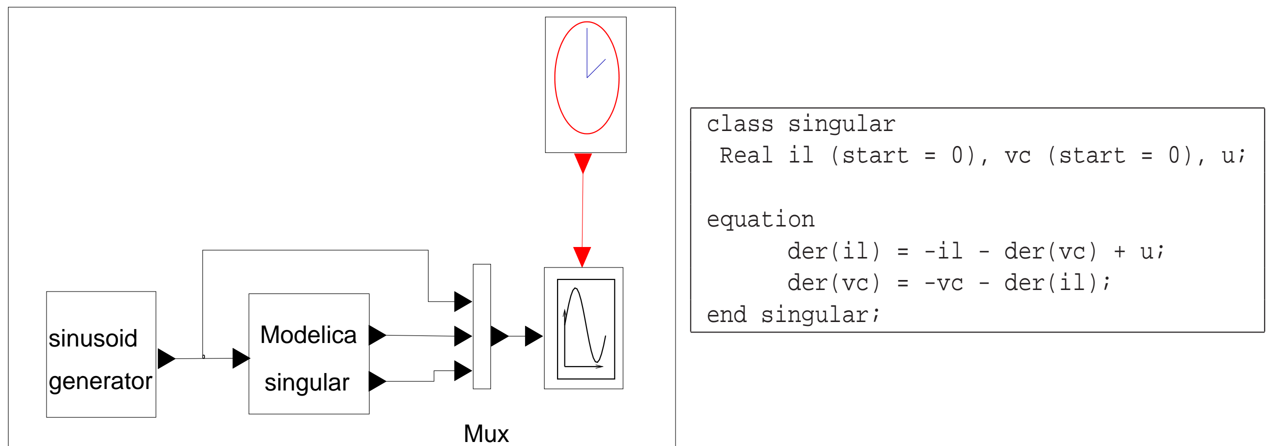

Figura 2.9: Bloco Modelica do Scicos e código Modelica.

Modelagem por equações a diferenças

Sistemas de tempo discreto são aqueles nos quais as variáveis apenas sofrem modificações em instantes de tempo isolados, ou sistemas com variáveis que sofrem modificações contínuas mas são observados em determinados instantes de tempo. Sistemas controlados por computador são um exemplo de sistema de tempo discreto uma vez que o computador lê e escreve sinais apenas em determinados instantes, denominados instantes de amostragem. Amostragem realizada por um computador é o processo pelo qual um sinal de tempo contínuo é representado por uma seqüência de bits que corresponde aos sinais em determinados intervalos (Campbell et al. 2005).

Para modelar sistemas desse tipo utiliza-se equações a diferenças da forma:

$$x(t_k+1) = f(t_k, x(t_k), \quad x(t_{k_0})), \tag{2.26}$$

onde t_k, $(k \geq k_0)$, é uma seqüência ou vetor de tempos.

Um problema comum em sistemas controlados por computador é a transformação de um modelo de tempo contínuo em um modelo de tempo discreto. Seja o modelo de tempo contínuo da eq. 2.27:

$$\begin{aligned} \frac{d\mathbf{x}}{dt} &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t), \\ \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t). \end{aligned} \tag{2.27}$$

onde:

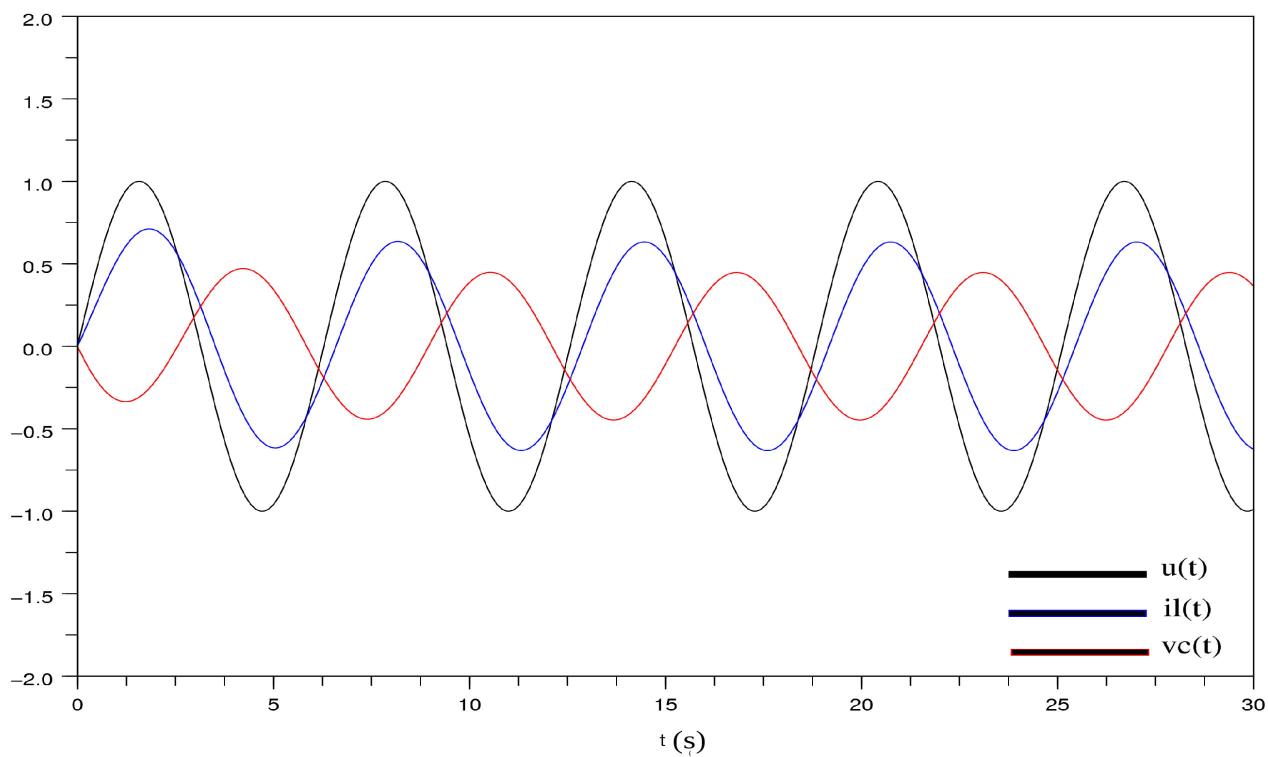

Figura 2.10: Resultado da simulação do circuito modelado por EAD.

$\mathbf{x}(t)$ é o vetor de variáveis de estado;

$\mathbf{A}$ é a matriz dinâmica do sistema;

$\mathbf{B}$ é a matriz de entrada;

$\mathbf{u}(t)$ é o vetor de variáveis de entrada;

$\mathbf{y}(t)$ é o vetor de variáveis de saída;

$\mathbf{C}$ é a matriz de saída;

$\mathbf{D}$ é a matriz de transmissão direta.

O equivalente em tempo discreto da eq.(2.27) é dado pela equação

$$
\begin{aligned}
\mathbf{x}(kh+h) &= \Phi x(kh) + \Gamma u(kh). \\
\mathbf{y}(kh) &= \mathbf{C}\mathbf{x}(kh) + \mathbf{D}\mathbf{u}(kh).
\end{aligned}
\tag{2.28}
$$

onde:

$\Phi = e^{Ah} = \mathbf{I} + \mathbf{A}\Psi$;

$\Gamma = \int_0^h e^{\mathbf{A}s} ds \mathbf{B} = \Psi\mathbf{B}$;

$\Psi = \int_0^h e^{\mathbf{A}s} ds = \mathbf{I}h + \frac{\mathbf{A}h^2}{2!} + \frac{\mathbf{A}^2h^3}{3!} + \ldots + \frac{\mathbf{A}^i h^{i+1}}{(i+1)!}$;

Nos sistemas controlados por computador o sinal de controle $u(t_k)$ é gerado em função da saída $y(t_k)$, portanto a matriz **D** é, na maioria das vezes, nula.

Para ilustrar a transformação de um modelo de tempo contínuo em tempo discreto foi utilizado o sistema bola-barra ilustrado na fig. 2.11 baseado em Åstrom & Wittenmark (1997), apêndice A. Nesse sistema uma bola está livre para se movimentar sobre os trilhos de uma barra que também se move ao redor de um eixo. O objetivo é controlar a posição da bola sobre a barra através de pequenos ajustes na inclinação da barra, controlando o torque T no motor ligado ao eixo da barra.

Nesse modelo a velocidade de rotação da bola $\dot{\theta}$ e o ângulo da barra ϕ foram considerados variáveis de estado. Linearizando o modelo ($sen(\phi) \approx \phi$), obtemos as seguintes equações:

$$
\begin{aligned}
\ddot{\theta} &= mgr\phi\frac{1}{J} \\
x &= r\theta
\end{aligned}
\tag{2.29}
$$

onde:

J é o momento de inércia rotacional ($\frac{2mr^2}{5}$);

θ é o ângulo da bola;

ϕ é o ângulo da barra;

x é a posição da bola;

m é a massa da bola;

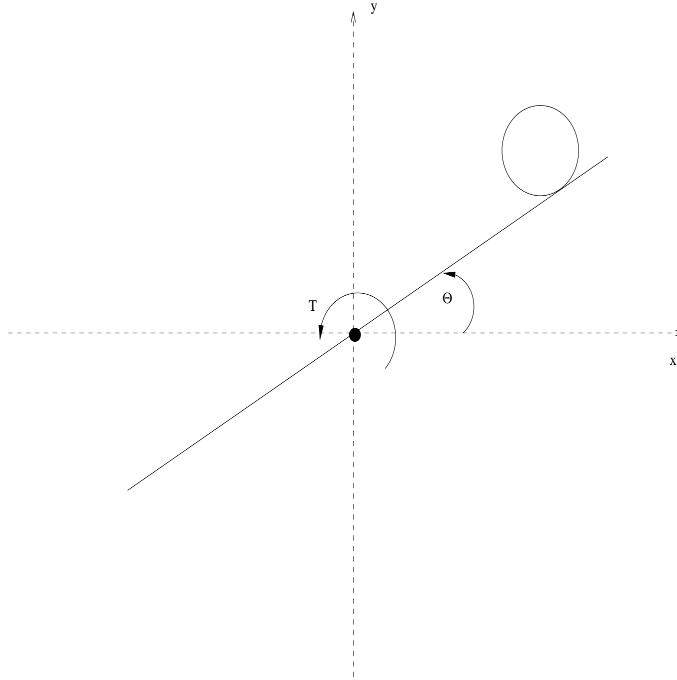

Figura 2.11: Sistema bola-barra.

g é a aceleração da gravidade;

r é o raio da bola.

A partir das eq. 2.29 é possível escrever a equação de estado na forma canônica:

$$\frac{d\varepsilon}{dt} = \begin{bmatrix} \ddot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} 0 & \frac{mgr}{J} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta} \\ \phi \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{2.30}$$

$$x = \begin{bmatrix} r & 0 \end{bmatrix} \theta \tag{2.31}$$

As eq. 2.28 foram utilizadas para transformar o sistema de tempo contínuo em tempo discreto.

$$\varepsilon(kh+h) = \begin{bmatrix} 1 & \frac{mgrh}{J} \\ 0 & 1 \end{bmatrix} \varepsilon(kh) + \begin{bmatrix} \frac{mgrh^2}{2J} \\ h \end{bmatrix} u(kh) \tag{2.32}$$

$$x(kh) = \begin{bmatrix} r & 0 \end{bmatrix} \varepsilon(kh) \tag{2.33}$$

O Scilab possui ferramentas para a transformação do sistema de tempo contínuo em sistema de tempo discreto (função `dscr`) e a resolução de sistemas lineares (função `syslin`). Essas funções foram utilizadas no *script* da tab. 2.3.

Tabela 2.3: Funções *syslin* e *dscr*.

```
//parâmetros do sistema
m=0.05;
g=9.8;
r=0.01;
J=0.02;

//matrizes
A=[0 m*g*r/J; 0 0];
B=[0;1];
C=[r 0];
D=0;

Ts=0.01;  //Ts= 10ms
t=0:19;

//modelo continuo
sist_cont=syslin('c',A,B,C,D);

//modelo discreto
sist_disc=dscr(sist_cont,Ts);

//separando as 4 matrizes discretas
[Ad,Bd,Cd,Dd]=abcd(sist_disc)
```

Ao executar o *script* o Scilab mostra as matrizes do sistema linear discreto, conforme ilustrado abaixo:

```
Dd  =


   0.
Cd  =


   0.01    0.
Bd  =


   0.0000123
   0.01
```

```
Ad  =


   1.    0.00245
   0.    1.


-->
```

Sistemas híbridos dinâmicos

Sistemas híbridos são aqueles em que a alteração das variáveis ocorre em tempo contínuo e em tempo discreto. Sistemas como esse precisam ser modelados por uma mistura entre EDOs (ou EADs) e equações a diferenças.

Os sistemas híbridos apresentam peculiaridades, entre elas o problema de inicialização. Esse problema ocorre quando um evento (instante de tempo no qual uma variável discreta muda seu valor) causa uma mudança nas equações diferenciais. Nestes casos o sistema precisa ser reiniciado e novas condições iniciais precisam ser definidas. Por esse motivo os programas de simulação de sistemas híbridos precisam ser capazes de prever quando os eventos acontecerão. Essa capacidade é denominada habilidade de encontrar raízes (*root-finding ability*).

A eq. 2.34 modela um sistema linear instável controlado pelo sinal mostrado na eq. 2.35. O sinal de controle passa a atuar somente a partir de $t = 15$. O sistema é controlado de duas formas, usando o controle em tempo contínuo e em tempo discreto. O exemplo e o algoritmo foram retirados de Campbell et al. (2005), seção 3.2.5.

$$\dot{x} = \begin{bmatrix} 0,1 & -1 \\ 1 & 0,1 \end{bmatrix} x + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u \tag{2.34}$$

$$u = \begin{bmatrix} -2 & -1 \end{bmatrix} x \tag{2.35}$$

O algoritmo da tab.2.4 mostra a definição do sistema em tempo contínuo e a chamada a função `ode` para a simulação do controle em tempo contínuo enquanto o algoritmo da tab.2.5 mostra a definição do sistema discreto e o controle com um período de amostragem de 0.9s realizado pela chamada a função `odedc`.

Tabela 2.4: Controle em tempo contínuo.

```
function z=control(t,x)
  if t<15 then z=0,
  else z=-2*x(1)-x(2);
  end
endfunction

function xdot = f(t,x)
  xdot=[0.1 -1; 10 0.1]*x+[control(t,x);0]
endfunction
x0=[1;1];
tt=[0:0.1:40];
y=ode(x0,0,tt,f);
```

Tabela 2.5: Controle em tempo discreto.

```
function ycd=fcd(t,yc,yd,flag)
  if flag == 0 then
 ycd=[0.1 -1; 1 0.1]*yc + ...
 [control(t,yd);0],
  else ycd=yc;
  end
endfunction

xd0=[1;1];
xc0=[0;0];
yy=odedc([xd0;xc0],2,[0.9,5/0.9],0,tt,fcd);
```

A fig. 2.12 mostra o resultado da simulação do sistema. O sinal em vermelho é a resposta da simulação com controle contínuo, chegando rapidamente a zero após $t = 15s$. O sinal em azul é a resposta com controle discreto. Percebe-se que com um período de amostragem de 0,9 o sinal demora para se estabilizar. É importante salientar que quanto menor o período de amostragem mais controle é aplicado ao sistema, de forma a torná-lo mais estável.

2.3 Controle por Computador e o Projeto Comedi

As pesquisas sobre a viabilidade do uso de computadores como componentes de sistemas de controle iniciaram na década de 1950. No início das pesquisas o computador era utilizado em controle apenas como um guia de operador, fornecendo instruções sobre o processo, ou como um controlador de ponto de operação (*set point*), modificando os pontos de operação de reguladores analógicos.

Durante aquele período os pesquisadores e usuários descobriram que ainda havia muito a ser desenvolvido. Havia a necessidade de melhores sensores, melhor conhecimento sobre processos e modelagem de sistemas, além do desenvolvimento de uma teoria de controle digital que estudasse alguns fenômenos próprios do processo de amostragem. Ao mesmo tempo diversas técnicas e avanços de *hardware* foram sendo desenvolvidos. O conceito de interrupção foi introduzido aos computadores, permitindo que um processo seja interrompido por um evento externo. Foram desenvolvidos também alguns computadores de uso específico como os analisadores digitais diferenciais (DDAs - *Digital Differential Analyzers*)(Åstrom & Wittenmark 1997).

Ao longo da década de 1960 foi introduzido o conceito de controle totalmente digital (DDC

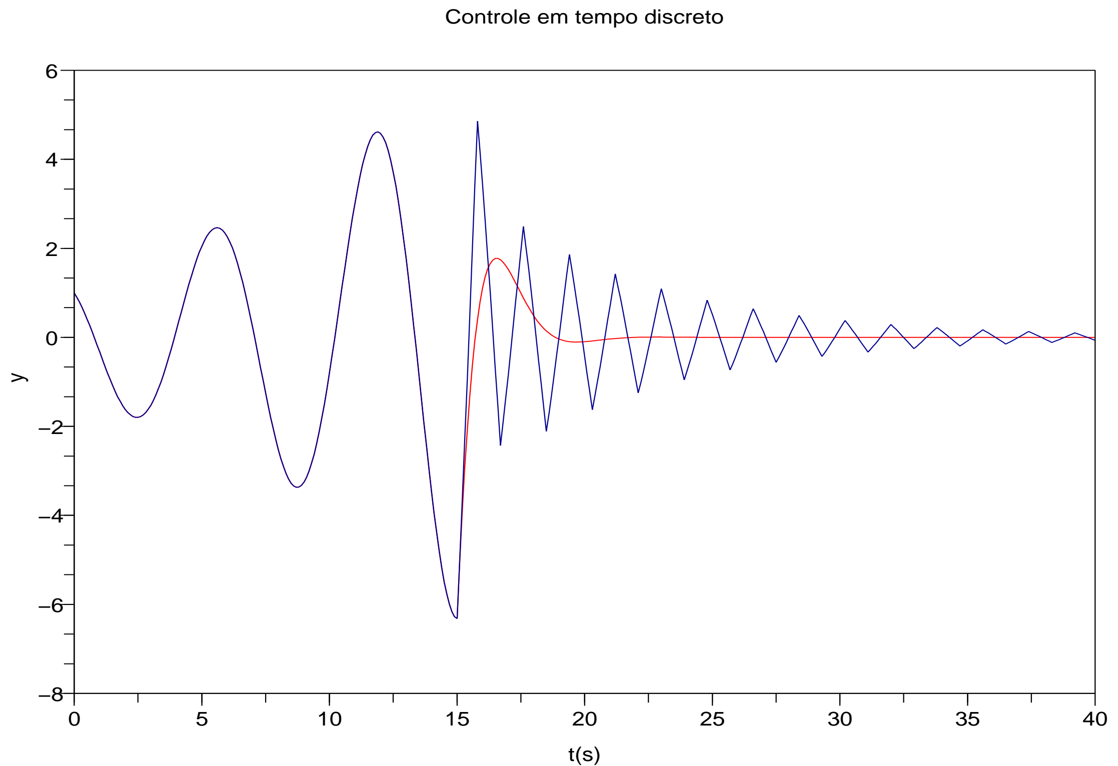

Figura 2.12: Controle discreto.

- *direct digital control*) onde o computador media variáveis e controlava válvulas diretamente.

O avanço da tecnologia de fabricação de circuitos possibilitou a queda de custos e aumento de qualidade do *hardware*, de forma que, ao fim da década de 1970 surgiram os microcontroladores, utilizados em diversas aplicações como a eletrônica automotiva, dispositivos de áudio e vídeo. Na mesma época surgiram também os controladores de lógica programável PLCs (*Programmable Logic Controllers*) que substituíram os relés na automação industrial (Åstrom & Wittenmark 1997).

Atualmente diversos tipos de indústrias utilizam o controle por computador, porém ainda existem problemas a ser resolvidos. Um destes problemas é a programação que ainda impôe certas restrições ao desenvolvimento de sistemas de controle por computador. Fabricantes de equipamentos relacionados a controle por computador como a Siemens e a *National Instruments* fornecem pacotes de *software* que permitem, até certo ponto, a programação de seus produtos. Porém esta programação possui limitações pois não é permitido ao usuário ir além de preencher tabelas com entradas e saídas do sistema, fatores de escala e parâmetros de reguladores. Isto

dificulta a aplicação de técnicas não convencionais de controle.

Uma metodologia que possibilite a escrita de códigos seguros, rápidos e de tempo real, com linguagens de programação de alto nível ainda não foi desenvolvida e os conceitos de engenharia de *software* ainda não direcionaram atenção para esse problema. Entretanto, muitos estudos sobre o desenvolvimento de *software* para sistemas de tempo real, sistemas paralelos e concorrentes têm sido realizados. As metodologias advindas desses estudos podem vir a ser de grande utilidade para o futuro do controle por computador.

A fig. 2.13 descreve um sistema de dados amostrados, ou sistema controlado por computador.

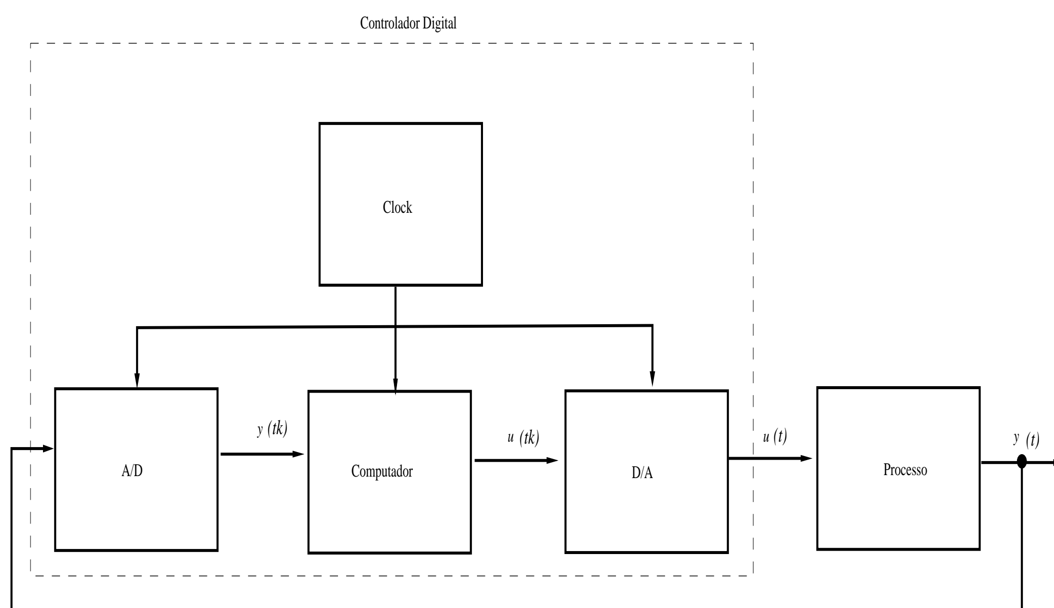

Figura 2.13: Sistema de dados amostrados.

O conversor A/D (analógico-digital) converte o sinal em tempo contínuo de saída $y(t)$ em um sinal de tempo discreto $y(t_k)$ nos instantes de amostragem definidos pelo relógio do sistema (*clock*). O sinal $y(t_k)$ é uma sequência de bits que serve de entrada para um algoritmo de controle. Esse algoritmo gera uma outra sequência de bits, o sinal $u(t_k)$ que é o sinal de controle. O sinal $u(t_k)$ é convertido para um sinal de tempo contínuo pelo conversos D/A (digital-analógico)(Åstrom & Wittenmark 1997).

Muito embora esse sistema possua variáveis em tempo contínuo e discreto basta que sua descrição seja feita nos instantes de amostragem. Essa simplificação permite o uso de equações a diferenças na representação do sistema.

Atualmente muitos sistemas computacionais usam placas de aquisição de dados na etapa de conversão A/D, D/A dos sistemas controlados por computador.

Placas de aquisição de dados são dispositivos que, acoplados a um microcomputador, fazem interface com sinais externos, sejam eles analógicos ou digitais.

De uma forma geral as placas de aquisição de dados são compostas de canais. Os canais são os componentes mais básicos e é através deles que a leitura/escrita é realizada. Um conjunto de canais afins (entrada analógica, saída analógica, *timer*, contadores, entrada digital, saída digital) forma um subdispositivo. O conjunto de subdispositivos forma o dispositivo, a placa em si. As características dos subdispositivos varia dependendo do fabricante e do tipo de placa, porém algumas especificações são comuns, os canais do subdispositivo de entrada permite, por exemplo, que o usuário escolha entradas diferenciais ou comuns (utilizam terra comum), modo de operação unipolar ou bipolar, faixa/ganho, etc.

A placa AT-MIO-64E-3 da *National Instruments* possui 64 canais para entradas analógicas, 2 canais para saídas analógicas, 8 canais para entrada/saída digital e 2 canais para contadores (NI 1997).

Figura 2.14: Diagrama de bloco da placa AT-MIO-64E-3 da *National Instruments*.

Os fabricantes da maioria das placas de aquisição de dados fornecem *drivers*, programas aplicativos e de instalação para somente um tipo de sistema operacional. Percebendo a demanda de *drivers* para o sistema operacional Linux David Schleef desenvolveu o projeto Comedi (*Common Measurement device interface* - interface comum para dispositivos de medidas). Além do desenvolvimento de *drivers* para placas de aquisição de dados para Linux, o projeto fornece ferramentas e uma biblioteca de funções (Comedilib API, escrita na linguagem de programação C) que permite a criação de aplicativos para diversas formas de aquisição: leitura e escrita de sinais digitais e analógicos, contagem de pulsos ou frequência, geração de pulsos, codificação, etc (Schleef, Hess & Bruynickx 2003).

A versão atual do Comedi (0.7.76) fornece *drivers* para 401 dispositivos. Os *drivers* foram organizados em um pacote denominado *comedi* que está disponível para *download* no site do projeto: (`http://www.comedi.org`).

O pacote *comedilib* possui uma API com funções de calibração, escrita, leitura, configuração, etc. Além desssas funcionalidades o pacote *comedilib* inclui documentação e programas demonstrativos. O projeto também distribui gratuitamente o pacote *kcomedilib*. Esse pacote possui a mesma biblioteca do *comedilib*, porém esta API pode ser executada em modo supervisor permitindo assim o seu uso em sistemas operacionais de tempo real ou extensões de *kernel* para tempo real.

As funções disponíveis na API englobam:

- configuração - a função `comedi_config` é chamada do terminal como um comando. Essa função permite relacionar um determinado *driver* a um dispositivo do Linux, por exemplo, o `/dev/comedi0`. A `comedi_config` também permite a configuração de algumas opções relacionadas a entrada e saída em tempo de execução como a interrupção (IRQ) e o endereço base (*IO Base*). Com a função `comedi_dio_config` o usuário pode configurar a direção do fluxo de sinais de um canal digital de entrada e saída para operar como entrada digital ou saída digital.

- leitura analógica - função `comedi_data_read`

- leitura digital - função `comedi_dio_read`

- escrita analógica - função `comedi_data_write`

- escrita analógica - função comedi_dio_write
- funções de conversão - a função comedi_to_phys converte valores amostrados (representados por valores inteiros não-sinalizados) em valores físicos. A função comedi_from_phys converte valores físicos em valores amostrados.

Existem também funções auxiliares que retornam informações sobre o dispositivo como comedi_get_maxdata que fornece o maior valor que pode ser representado por um determinado canal, comedi_get_range que retorna o valor máximo e mínimo da faixa de operação do canal (o valor máximo é o correspondente ao valor retornado por comedi_get_maxdata e o mínimo é valor retornado quando comedi_data_read retorna 0) e a função comedi_get_subdevice_type() que retorna o tipo do subdispositivo.

O programa da tab. 2.6 foi escrito para ler uma tensão de $\approx 0,8$ volts através da placa de aquisição de dados da *Keithley Instruments*, modelo DAS-1602.

Tabela 2.6: Programa para aquisição simples.

```
#include<stdio.h>
#include<comedilib.h>

int subdev = 0;
int range = 0;
int chan = 0;
comedi_range *range_struct;
int aref = AREF_GROUND;
int main(int argc, char *argv[]){
 comedi_t *it;
 lsampl_t data, maxdata;
 double volts;

 it = comedi_open("/dev/comedi0");
 comedi_data_read(it,subdev,chan,range,aref,&data);
 maxdata = comedi_get_maxdata(it,subdev,chan);
 range_struct = comedi_get_range(it,subdev,chan,range);
 volts = comedi_to_phys(data,range_struct,maxdata);
 printf("Faixa de operacao de %g ate %g.\n",range_struct->min,range_struct->max);
 printf("Dado: %d, maxdata: %d, volts:%g\n",data,maxdata,volts);

 return(0);

}
```

O programa lista o valor máximo que pode ser retornado pelos subdispositivos de entrada analógica, a faixa de operação do canal, o valor amostrado e o valor em volts. O valor de

referência é o terra, representado pela constante AREF_GROUND; o subdispositivo de leitura analógica da placa é o 0 e o canal escolhido para a leitura foi o canal 0. A placa foi previamente configurada para operar no dispositivo do Linux `/dev/comedi0` com o comando `comedi_config` da seguinte forma:

```
comedi_config -v /dev/comedi0 cio-das1602/16 0x390,7
```

A saída do programa listado na tabela 2.6 é mostrada abaixo:

```
Faixa de operacao de -10 ate 10.
Dado: 35392, maxdata: 65535, volts:0.800946
```

A documentação da API também está disponível no site do projeto:
(`http://www.comedi.org/doc/index.html`).

2.4 Programação Cliente/Servidor

Nesta seção são abordadas as vantagens da programação no modelo cliente/servidor, protocolos de transporte e de aplicação além de alguns aspectos específicos da programação em C para Linux. São expostos os motivos pelos quais adotou-se o protocolo de aplicação HTTP e o protocolo de transporte TCP na comunicação do sistema de aquisição remota.

2.4.1 Programação com *sockets*

Na abordagem cliente/servidor um aplicativo é dividido em um processo cliente, que envia solicitações a um servidor, e um processo servidor, que disponibiliza algum tipo de serviço ou recurso ao cliente. Esses processos são geralmente executados em computadores diferentes fornecendo flexibilidade, portabilidade e melhor utilização dos recursos.

O modelo TCP/IP, tambem conhecido como modelo Internet descreve em linhas gerais protocolos que permitem a comunicação entre computadores em uma rede. Este modelo define como os dados devem ser formatados, endereçados, transmitidos e recebidos no destino. O mdeolo TCP/IP é separado em quatro camadas: enlace, inter-rede, transporte e aplicação. A fig. 2.15 resume o modelo.

No Linux a comunicação em rede é realizada através de *sockets* de Internet, que são canais de comunicação. Esses canais fornecem uma interface entre aplicações que são executadas

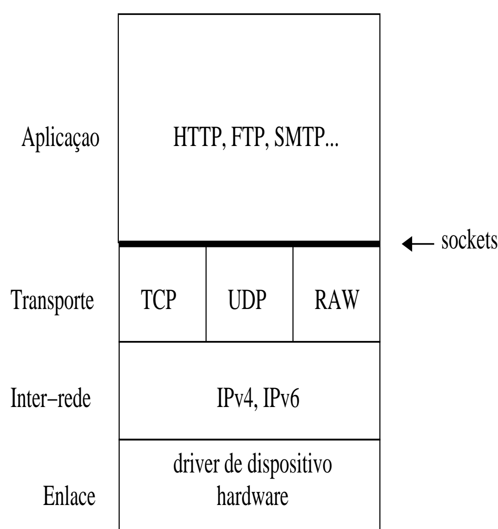

Figura 2.15: Diagrama do Modelo TCP/IP.

em modo usuário (camada superior do modelo TCP/IP) e os processos executados em modo supervisor (pelo *kernel*, três camadas inferiores do modelo TCP/IP)(Stevens 1998). Os *sockets* são tratados no Linux como descritores de arquivo, inteiros associados a um arquivo aberto. Porém esse arquivo pode ser uma conexão de rede, uma fila, um terminal ou um arquivo em um disco rígido(Hall 2001).

Existe um fluxo típico na comunicação via *sockets*. Num modelo orientado a conexão o servidor estabelece um endereço (porta) que os clientes podem usar para se conectar a ele. Esse servidor possui um *socket* que aguarda pedidos de um cliente. Quando um cliente precisa fazer requisições ao servidor, ele abre um canal (*socket*) e se comunica com o servidor a partir da troca de mensagens. O servidor executa o pedido do cliente enviando uma mensagem com o resultado da execução. Na maioria das aplicações o servidor fecha a conexão com o cliente logo após o envio da resposta (IBM 2005). Este fluxo de comunicação foi ilustrado na fig. 2.16

O canal de comunicação que um *socket* abre pode ser orientado ou não à conexão. A comunicação orientada à conexão implica que a troca de mensagens será realizada através de um canal que não será fechado até que um dos lados decida. Na comunicação não conectada os lados enviam e recebem mensagens para um endereço, sem estabelecer uma conexão. Uma boa analogia para comunicação orientada à conexão é uma conversa telefônica, onde ambas as partes estão conectadas e trocam mensagens em tempo real. A comunicação sem conexão pode ser pensada como o envio de uma carta para uma caixa postal; não existe comunicação em tempo

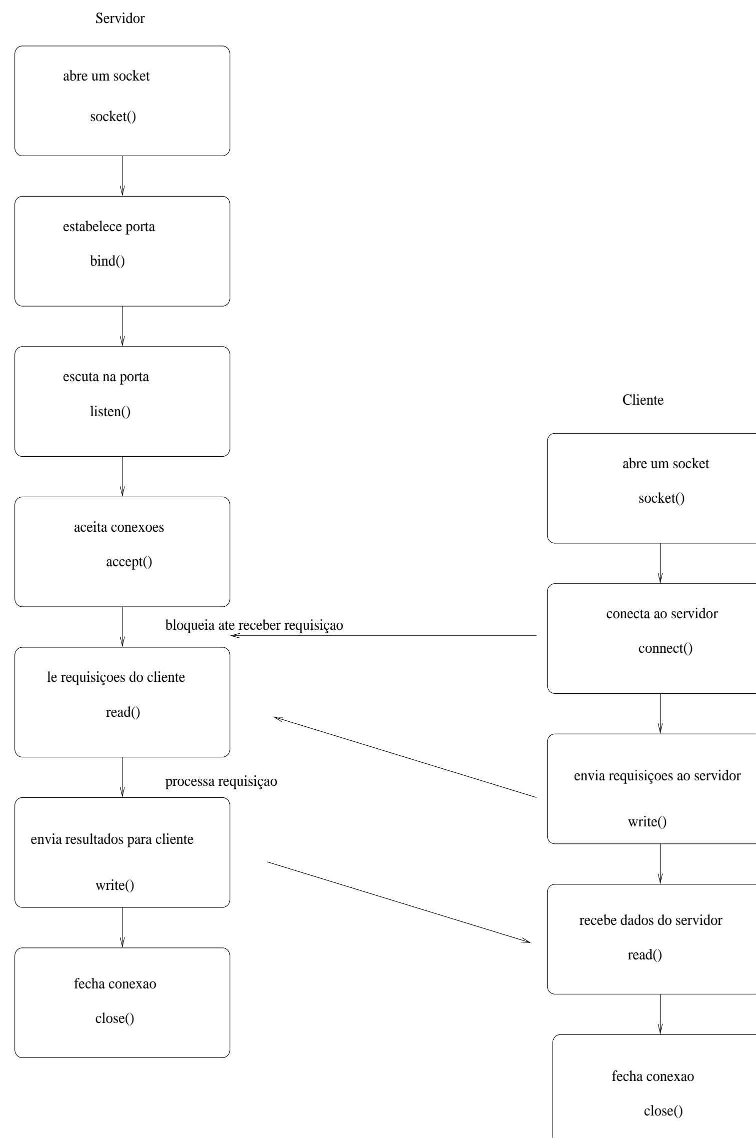

Figura 2.16: Fluxo de Comunicação cliente/servidor.

real, as partes não interagem diretamente(IBM 2005).

A fig. 2.16 também mostra algumas das *system calls* (funções) mais usadas na programação na linguagem C com *sockets* em sistemas *Unix-like*. Antes de utilizar a função `socket()` é necessário preencher a estrutura de dados que contém informações sobre os *sockets* que serão abertos. Essa estrutura é a `sockaddr_in` (de *socket address - internet*) e ela possui a seguinte forma:

```
struct sockaddr_in {
    short int           sin_family;
    unsigned short int  sin_port;
    struct in_addr      sin_addr;
```

```
    unisgned char          sin_zero[8];
};
```

Os elementos desta estrutura são detalhados abaixo:

- sin_family - esse campo recebe o tipo de família de endereços do *socket*. Alguns desses tipos são:
  - AF_UNIX - apenas para comunicação não orientada à conexão utilizando *sockets* Unix;
  - AF_INET - essa família de endereços pode ser usada para *sockets* orientados à conexão (tipo `SOCK_STREAM`- TCP) ou não (tipo `SOCK_DGRAM` - UDP);
  - AF_INET6 - AF_INET para protocolo IP versão 6 (IPv6);
- sin_port - esse campo recebe o número da porta através da qual o servidor fornece o serviçco (ex.: HTTP - porta 80, SSH - porta 22, etc.);
- sin_addr - esse campo recebe o endereço IP do *socket*;
- sin_zero[8] - esse campo é reservado. Ele deve receber zeros (no formato hexadecimal).

Uma chamada típica à função `int socket(int domain, int type,int protocol)` seria realizada da seguinta maneira:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include<netinet/in.h>

#define port 80

struct sockaddr_in myAddr;

memset(&myAddr,0,sizeof(myAddr));        //zerando toda a estrutura
myAddr.sin_family = AF_INET;             //familia AF_INET
myAddr.sin_port = htons(port);           //converte e assinala porta
myAddr.sin_addr.s_addr = INADDR_ANY;     //todas as interfaces (placas de rede)
```

```
                                        //recebem solicitacoes do cliente

//type = SOCK_STREAM (orientado a conexao).
//protocol = 0 (permite que o socket escolha o protocolo baseado no tipo - TCP)
if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
perror("socket error");
return -1;
}
```

Para relacionar o *socket* recém criado à porta do servidor que receberá as solicitações do cliente utiliza-se a função `int bind(int sockfd, struct sockaddr *my_addr, int addrlen)`. Essa função é chamada apenas do lado servidor. Depois de realizado o *bind* o servidor precisa aguardar requisições do cliente. Isso é feito através da chamada à função `int listen(int sockfd, int backlog)`. Quando o servidor recebe a requisição do cliente ele chama a função `int accept (int sockfd, void *addr, int *addrlen)`.

O cliente precisa criar um *socket*, conectar-se ao servidor e iniciar a troca de mensagens. Essa troca é realizada pelas funções `ssize_t read(int fd, void *buf, size_t count)` e `ssize_t write(int fd, const void *buf, size_t count)`.

Exemplos de clientes e servidores simples, com sockets orientados ou não à conexão, podem ser encontrados em Hall (2001), IBM (2005) e em Stevens (1998).

2.4.2 Protocolos da Camada de transporte

Dois tipos de protocolos são abordados neste trabalho: protocolos da camada de transporte e protocolos da camada de aplicação. Essas camadas referem-se tanto ao modelo TCP como ao modelo OSI de comunicação.

A camada de transporte é a responsável pelo recebimento dos dados vindos da camada de aplicação, pela sua divisão em pacotes, adição do preâmbulo (*header*) do protocolo e envio à camada de rede. Os protocolos mais utilizados dessa camada são o UDP e o TCP.

Num sistema de comunicação que utiliza o protocolo UDP (*User Datagram Protocol*) as mensagens vindas do aplicativo são escritas no *socket* UDP, encapsuladas no datagrama UDP e enviadas para a camada de rede. O UDP é um protocolo que não oferece confiabilidade por não garantir a entrega de pacotes, não possuir controle de fluxo e não fornecer um método que

permita que os pacotes que chegaram de forma desordenada sejam ordenados na recepção. A única forma de controle de um pacote UDP é o número de *bytes* enviados incluído no preâmbulo. Assim ambos o transmissor e receptor sabem o número de *bytes* enviados. Dessa forma o programador da aplicação precisa criar mecanismos para verificar se os pacotes enviados foram recebidos pelo cliente, se foram recebidos de forma ordenada e se o fluxo de pacotes é comportado pela rede e pelo cliente.

O *socket* UDP é do tipo não orientado à conexão, sem a necessidade do estabelecimento de uma conexão um cliente pode, por exemplo, enviar um datagrama para um servidor através de um *socket* e logo após utilizar o mesmo *socket* para enviar datagramas a um outro servidor. Sendo assim a aplicação do lado servidor também deve verificar se os datagramas recebidos fazem parte de um mesmo processo ou não.

O *socket* TCP (*Transmission Control Protocol*) é do tipo orientado à conexão. O protocolo TCP fornece diversos mecanismos que favorecem o envio seguro de mensagens. São eles:

- confiabilidade - quando um pacote TCP é enviado o transmissor espera um sinal de aviso de recebimento, denominado `ACK` (*acknowledgment*). Se o transmissor não recebe o `ACK` ele retransmite o pacote e espera o `ACK` novamente. A cada retransmissão o tempo de espera pelo `ACK` é mais longo. Este processo pode levar até 10 minutos até o que o protocolo desista das retransmissões e retorne um erro e termine a conexão.

- sequenciamento de pacotes - o TCP envia um número de sequência associado a cada *byte*. Se o aplicativo escreve mais dados do que podem ser enviados em um único pacote o TCP envia a informação separada em pacotes diferentes com os números de sequência. Se os pacotes chegam ao receptor fora de ordem, o repector consegue reordená-los. Além disso o receptor consegue descartar pacotes duplicados.

- controle de fluxo - um receptor trabalhando com *sockets* TCP informa ao transmissor quantos *bytes* podem ser enviados em determinado momento. Assim um transmissor não envia mais dados do que o receptor pode tratar.

As conexões TCP são estabelecidas em três etapas (*Three-Way Handshake*). Depois da chamada às funções `socket()`, `bind()` e `listen()` o servidor está pronto para receber conexões. Ao executar a função `connect()` o cliente envia ao servidor o sinal `SYN` com o número de

sequência do primeiro *byte* a ser enviado. O servidor deve enviar um sinal de `ACK` confirmando o recebimento do `SYN` do cliente. Na mesma transmissão o servidor envia um novo sinal de `SYNC` para sinalizar ao cliente o número de sequência do primeiro *byte* que será enviado pelo servidor. Depois disso o cliente envia um `ACK` ao servidor, informando que o `SYNC` foi recebido.

Uma conexão TCP é finalizada em quatro etapas. Primeiro o aplicativo termina a comunicação chamando a função `close()`. A partir desse momento o lado que termina a comunicação envia um pacote com o sinal `FIN`. O receptor recebe o sinal de finalização e, depois que todos os dados recebidos são armazenados em memória (*buffer*) o aplicativo recebe um sinal de fim de arquivo. O receptor envia ao transmissor um sinal de `ACK` informando sobre o recebimento do `FIN`). Após algum tempo o aplicativo do lado receptor também chama a função `close()` e fecha seu *socket*. Quando isso é feito o TCP do receptor envia um `FIN` para o transmissor , que responde com um `ACK` caso a mensagem seja recebida. A troca de mensagens entre transmissor e receptor durante a finalização da conexão implica no surgimento de um tempo de latência no qual a conexão fica no estado de *TIME_WAIT*. O estado de *TIME_WAIT* pode perdurar de 1 a 4 minutos (dobro do tempo máximo de vida de um segmento (*maximum segment lifetime - MSL*) que varia de 30 segundos a 2 minutos). O estado serve para:

- permitir que segmentos duplicados expirem - se houver segmentos duplicados na rede, a conexão fechasse e depois uma outra conexão do mesmo cliente para p mesmo servidor (mesmos IPs e portas) fosse aberta o pacote duplicado da conexão anterior poderia ser entregue como um pacote válido. Como antes de fechar a conexão entra no estado de *TIME_WAIT* há tempo suficiente para o segmento duplicado expirar ou ser descartado pelo receptor.

- garantir a confiabilidade da conexão *full-duplex* - se o último `ACK` enviado pelo cliente não for recebido pelo servidor, o sinal de `FIN` será retransmitido. Se o cliente não estiver no estado de *TIME_WAIT* ele responderia com um `RST`, um *reset* instantâneo nas duas direções. O servidor receberia este `RST` como um erro e avisaria ao aplicativo.

O protocolo UDP não possui as confirmações e garantias do TCP, visto que não é um protocolo orientado à conexão, e, por este motivo, não está sujeito a tempos de latência. Para aplicações em tempo real, onde os mesmos dados ou dados muito semelhantes são enviados

em grande número, a perda de pacotes não seria tão crítica e o protocolo UDP seria o mais apropriado.

O sistema de aquisição remota de dados proposto nesse trabalho utiliza uma rede *Ethernet* para a transmissão e recepção de dados amostrados em frequências de até 1KHz. Muito embora a quantidade de dados seja grande, dependendo do tipo de sinal amostrado a perda de pacotes e a troca de sua sequência poderia afetar não só a forma como os sinais são interpretados pelo usuário como também afetar a operação dos equipamentos eletromecânicos que por ventura estejam sendo controlados. Por estes motivos optou-se pelo uso de *sockets* TCP.

2.4.3 Protocolos da Camada de aplicação

Quando se escreve um programa no modelo cliente/servidor, algumas decisões devem ser tomadas. Deve-se definir qual dos lados iniciará a comunicação, como esta comunicação será realizada, qual dos lados fechará a comunicação e quando. Essas decisões compõem um protocolo de comunicação em nível de aplicação.

O protocolo da camada de aplicação HTTP (*Hyper Text Transfer Protocol*) é um protocolo bastante simples baseado em requisições enviadas pelo cliente e respostas enviadas pelo servidor. As requisições enviadas no padrão HTTP possuem um método e dados; as respostas possuem um código de retorno e dados. Depois de enviados a requisição e a resposta a conexão é fechada.

Uma requisição simples de um programa navegador de Internet seria:

```
GET http://www.lee.eng.uerj.br HTTP/1.0

User-Agent: Opera/9.50 (X11; Linux i686; U; en)
Host: 127.0.0.1:80
Accept: text/html, application/xml;q=0.9, application/xhtml+xml, image/png, imag
e/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: en-US,en;q=0.9
Accept-Charset: iso-8859-1, utf-8, utf-16, *;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Connection: Keep-Alive
```

A primeira linha da requisição é composta pela chamada ao método GET, o endereço e a versão do protoclo no cliente, neste caso o HTTP 1.0. O método GET avisa ao servidor localizado no endereço http://www.lee.eng.uerj.br que o cliente está solicitando a página principal. A linha que contém o método é sempre a primeira e ela vem seguida de uma linha em branco (um CR - *carriage return* seguido de um LF- *line feed*). Após a linha em branco seguem os *headers* com os dados que informam ao servidor detalhes sobre o programa navegador tais como nome, versão, os tipos de arquivos aceitos, entre outros.

O servidor responderia a esta requisição com uma mensagem de sucesso ou falha, da seguinte forma:

```
HTTP/1.1 302 Found
Date: Tue, 02 Feb 2010 18:52:20 GMT
Server: Apache/2.2.3 (Debian) mod_python/3.2.10 Python/2.4.4 PHP/4.4.4-8+etch1
mod_ssl/2.2.3 OpenSSL/0.9.8c mod_perl/2.0.2 Perl/v5.8.8
Location: https://master.uerj.br/webmail
Content-Length: 403
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="https://master.uerj.br/webmail">here</a>.</p>
<hr>
<address>Apache/2.2.3 (Debian) mod_python/3.2.10 Python/2.4.4 PHP/4.4.4-8+etch1 mod_ssl/2.2.3
OpenSSL/0.9.8c mod_perl/2.0.2 Perl/v5.8.8 Server at www.lee.eng.uerj.br Port 80</address>
</body></html>
```

A primeira linha da resposta do servidor indica que a versão o protocolo é HTTP 1.1 e a mensagem de retorno é *302 Found* ou *Moved temporarily* indicando que a página está localizada temporariamente em outro endereço). As linhas seguintes são similares aos *headers* da requisição, trazendo informações sobre a resposta como data, o nome do servidor *web*, o tipo do

conteúdo retornado, etc. Logo após a linha em branco segue o conteúdo solicitado pelo cliente, o código HTML da página principal, que é interpretado pelo navegador.

Outros métodos do protocolo HTTP são:

- POST - o método POST permite o envio e atualização de campos de um formulário HTML (`<form>`);
- HEAD - similar ao GET, mas somente o corpo da resposta não é enviada ao cliente, apenas os *headers*. Este método é útil quando se quer verificar se o recurso é válido e está acessível;
- OPTIONS - o método OPTIONS permite que o cliente saiba quais métodos podem ser executados por determinado servidor;
- PUT - esse método permite que um cliente envie um arquivo e salve no servidor;
- DELETE - ao contrário do método PUT o método DELETE permite que o cliente apague um arquivo localizado no servidor;
- TRACE - se habilitado no servidor, este método retorna ao cliente a requisição da forma como ela chegou ao destino. Este método serve para verificar se existem problemas relacionados ao *header* de requisição, caso haja alguma mensagem de erro sendo retornada pelos outros métodos.

Existem duas versões do protocolo: HTTP 1.0 e HTTP 1.1. A diferença básica entre os dois é que o protocolo 1.1 permite que depois da resposta do servidor a conexão permaneça aberta. Para sistemas cujo voulme de troca de mensagens é muito alto, esta é uma diferença importante. Definições detalhadas dos protocolos HTTP 1.0 (RFC 1945) e HTTP 1.1 (RFC 2616) podem ser encontradas no *site* da IETF (*Internet Engineering Task Force*): http://www.ietf.org/rfc/rfc1945.txt e http://www.ietf.org/rfc/rfc2616.txt

O protocolo HTTP é universal e muito simples. Escrever clientes diversos para servidores que utilizam HTTP é fácil e rápido, uma vez que o programador não perde muito tempo aprendendo um protocolo específico e difícil, com muitos detalhes. Essas foram as características que levaram à escolha desse protocolo para o desenvolvimento do sistema de aquisição remoto.

A essas características soma-se o fato de que qualquer navegador de Internet utiliza o protocolo HTTP e o sistema de aquisição pode se estendido para uso através da *web*.

2.5 Integração de Programas Escritos pelo Usuário ao Scilab/Scicos

Na seção 2.2.3 foi mencionado o uso de programas da biblioteca ODEPACK, escrita em Fortran, pela função `ode` do Scilab. Existem funções do Scilab que ao ser chamadas executam código externo, que não consta do pacote original do Scilab. Esse código pode ser escrito em C, Fortran e na própria linguagem do Scilab. Pode ser carregado na inicialização ou compilado como uma biblioteca compartilhada e carregado dinamicamente, em tempo de execução.

O Scicos também permite a execução de código externo. É possível fazer uma analogia entre o Scicos e uma linguagem de programação. Nesta analogia os blocos seriam comparados às funções e o diagrama (um conjunto de blocos interconectados) seria um algoritmo. Assim como as funções nas linguagens de programação, alguns blocos são pré-definidos e outros podem ser desenvolvidos pelo usuário.

Nesta seção serão discutidas as estruturas de dados que permitem a troca de informações entre os programas externos e o Scilab, as estruturas de blocos do Scicos e o funcionamento do seu editor, compilador e simulador. Ao fim da seção dois exemplos de interfaces com o Scilab e com o Scicos são mostrados.

2.5.1 Estrutura interna do Scilab

O Scilab é uma linguagem interpretada e um ambiente aberto de programação. O pacote original, disponível para *download* em (`http://www.scilab.org`), possui diversas funções nativas ou primitivas, funções que vêm no pacote original do Scilab. Essas primitivas são escritas em C ou Fortran e o conjunto delas pode ser expandido com a adição de primitivas escritas por usuários.

Os códigos das primitivas são organizados em arquivos de acordo com a área de aplicação (álgebra linear, integração numérica, otmização, interface gráfica, processamnto de sinais, etc.). Para cada código existe um programa que realiza a interface com o Scilab.

A execução do Scilab começa com a função `StartScilab`. Essa função inicializa os nomes do principais arquivos (utilizados pelos menus de *help, save* etc.) a pilha e outras tabelas. O

comando digitado no *prompt* do Scilab é uma primitiva cuja interface precisa ser identificada. A função `scirun` recebe o nome da primitiva e passa ao interpretador (função `parse`). O interpretador examina a primitiva, chama a função `callinterf` que retorna o número da interface que deve ser executada. Este número é retornado à função `scirun` que executa o comando (Scilab 2003). A fig. 2.18 resume o funcionamento do Scilab.

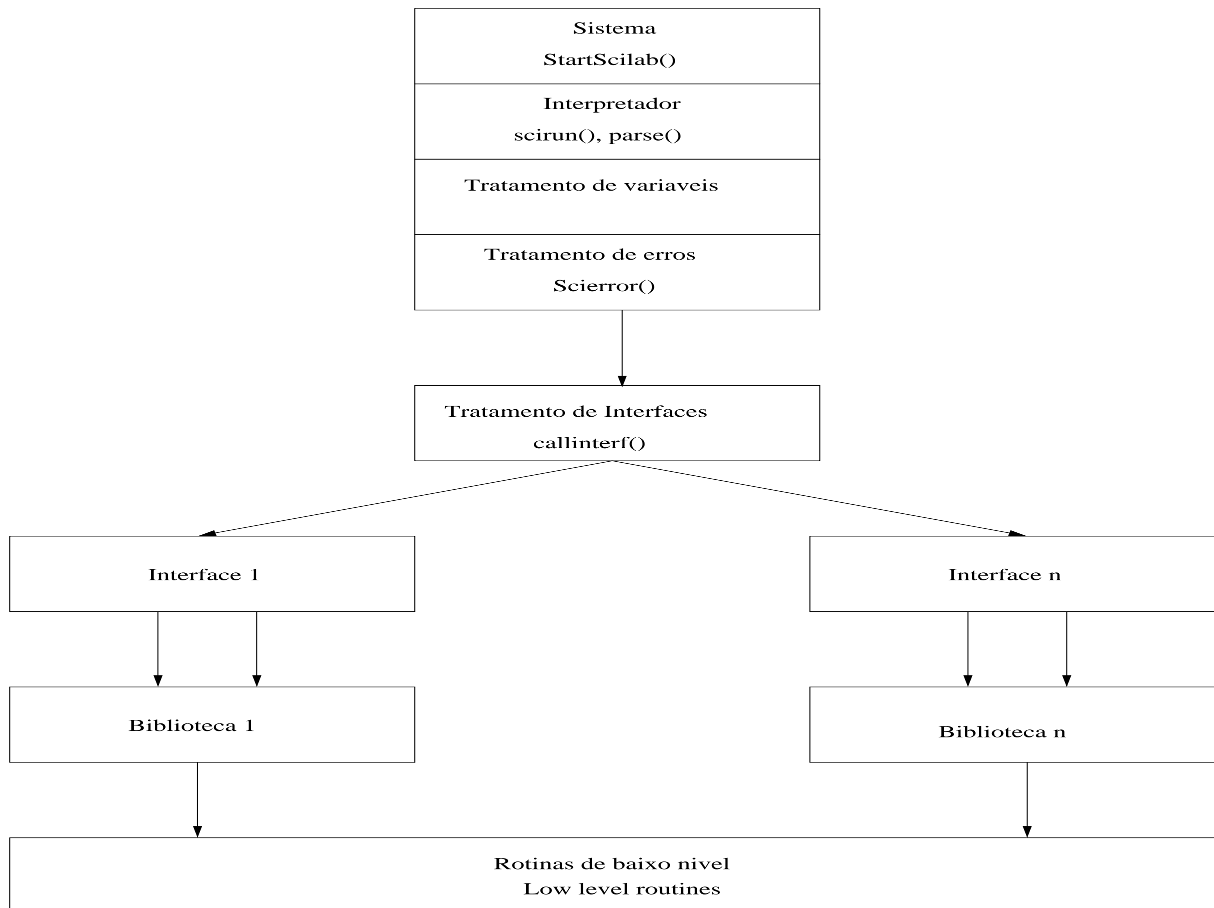

Figura 2.17: Estrutura interna do Scilab.

Novas primitivas podem ser adicionadas ao Scilab de forma permanente ou dinamicamente, em tempo de execução. Para a adição de primitivas permanentes é necessário ter instalada a versão com código fonte pois a recompilaçao dos módulos faz-se necessária. Por ser muito dependente de versão este tipo de adição não será abordada neste trabalho. Para que as primitivas escritas por usuários sejam adicionadas ao Scilab é necessário encapsular as mesmas numa interface. Embora exista a opção de inserção de códigos em Fortran e na linguagem do Scilab aborda-se neste trabalho as interfaces para programas escritos em C.

Supondo que um usuário deseje criar uma função de soma em C e inseri-la como uma primitiva do Scilab, ele deve escrever uma interface que realize os seguintes passos:

- Converter todos os argumentos que serão inseridos no *prompt* do Scilab (as entradas e saídas) em objetos que possam ser identificados pelo compilador C;
- Receber os valores dos argumentos vindos do Scilab através da pilha;
- Verificar do número e tipo dos argumentos;
- Chamar a função de soma escrita em C;
- Verificar a necessidade de passar um valor criado no código C para o Scilab. O usuário deve garantir a criação correta de variáveis na pilha do Scilab;
- Retornar os valores obtidos pelo programa em C para o Scilab.

O Scilab e o programa de interface precisam trocar dados entre si, ter acesso às mesmas variáveis. Essas variáveis são colocadas na pilha do Scilab de forma que elas possam ser utilizadas por ambos programas. Existe um conjunto de funções usadas para lidar com essa pilha. Essas funções fazem parte do pacote do Scilab e foram definidas no arquivo de *header* stack-c.h. Na versão 4.1.2 do Scilab este arquivo pode ser localizado em `/SCIDIR/routines`, onde SCIDIR é o diretório onde o Scilab foi instalado.

Ilustra-se a seguir um exemplo simples de criação da primitiva *soma*. Este exemplo foi retirado da seção de documentação e suporte do site do Scilab e pode ser encontrado em http://www.scilab.org/product/index_product.php?page=toolbox_guide#chap2.2.2.

A sintaxe desta primitiva no Scilab será `c = soma(a, b)`, onde a, b e c são matrizes. Como o Fortran foi a linguagem na qual o Scilab foi primeiramente desenvolvido, algumas características foram herdadas, como a passagem por referência. Portanto, todas as variáveis do programa escrito em C devem ser ponteiros. O código C da função soma é ilustrado na tab. 2.7. O arquivo que contém este código foi chamado `soma.c`.

Para que esta função possa ser chamada de dentro do Scilab é necessário escrever sua interface, compilar os programas como bibliotecas compartilhadas e carregar os módulos no Scilab.

Tabela 2.7: Função soma.

```
void soma(int n, double * a, double * b, double * y)
{
 int k;
 for (k = 0; k < n; ++k)
 y[k] = a[k] + b[k];
}
```

A tab. 2.8 mostra a interface escrita para a função soma. O arquivo que contém o código de interface foi chamado `intersoma.c`.

As funções `CheckRhs` e `CheckLhs` verificam o número de argumentos do lado direito e esquerdo da primitiva passados pelo Scilab. O número deve ser igual ou maior que `minrhs` e `minlhs` e menor ou igual a `maxrhs` e `maxlhs`. A função `GetRhsVar` verifica se os tipos e dimensões das entradas estão corretos. Esta função também recebe o endereço das variáveis de entrada na pilha do Scilab. Os tipos permitidos são `d` para double, `c` para caracter, `r` para real e `i` para inteiro. `CreateVar` cria a variável na pilha do Scilab que receberá o valor de saída. `LhsVar` assinala à saída do Scilab a variável criada pela função `CreateVar`.

Depois de escritos os programas principal e a interface eles devem ser compilados, adicionados a uma biblioteca compartilhada e inseridos no ambiente do Scilab. Seguindo as orientações descritas na documentação do Scilab, é importante que os arquivos de interface e do programa principal estejam em diretórios separados. Neste exemplo o arquivo `soma.c` foi criado no diretório `../src`. Neste diretório foi criado o *script* `builsrc.sce`:

```
libname='somasrc';                   //nome da biblioteca a ser construida
files=['soma.o'];                    //lista de arquivos a ser compilados
libs=[];                             //bibliotecas externas necessarias
flag="c";                            //programa escrito em C
ilib_for_link(libname,files,libs,flag)
```

O comando *ilib_for_link* compila o programa `soma` na biblioteca `libsomasrc.so`. Ao executar este *script* do diretório `../src` o Scilab gera a seguinte saída:

```
generate a loader file
generate a Makefile: Makelib
running the makefile
compilation of soma
```

```
  building shared library (be patient)
ans  =

libsomasrc.so
```

Depois de criada a biblioteca contendo o código objeto do programa `soma` é necessário criar uma nova biblioteca contendo a interface. O arquivo contendo o código de interface `intsoma.c` foi criado no diretório `../sci_gateway` e o *script* para criar a biblioteca de interface foi chamado `buildsci_gateway.sce`.

Como a função de interface chama a função primitiva, a biblioteca a ser criada precisa ser *linkada* à biblioteca `libsomasrc.so`. O *script* abaixo usa o comando `ilib_build` para construir essa nova biblioteca:

```
ilib_name = 'libsoma'                  //nome da biblioteca de interface a ser criada
files = ['intsoma.o'];                 //lista de arquivos a ser compilados (codigo da interface)
libs = ["../src/libsomasrc"]           //biblioteca a ser linkada (biblioteca de primitivas)
table = ['soma', 'intsoma'];           //tabela de correspondencia entre primitiva e interface
ilib_build(ilib_name, table, files, libs)
```

A saída gerada no *prompt* do Scilab é:

```
ilib_name  =

 libsoma
 libs  =

 ../src/libsomasrc
   generate a gateway file
   generate a loader file
   generate a Makefile: Makelib
   running the makefile
   compilation of intsoma
   building shared library (be patient)
```

O *script* acima cria a biblioteca compartilhada `libsoma.so` e outros arquivos, entre eles o *script* `loader.sce`. Ao ser executado o arquivo `loader.sce` carrega a biblioteca compartilhada, de forma que o comando `soma` pode ser utilizado como uma primitiva:

```
-->shared archive loaded
Link done
shared archive loaded


-->A = [1 2];

-->B = [4 6];

-->C = soma(A,B)
 C  =

    5.    8.
```

A cada vez que o Scilab é iniciado é necessário executar o `loader.sce` para carregar a biblioteca de interfaces.

2.5.2 Estrutura interna do Scicos

Para simular sistemas híbridos dinâmicos o Scicos precisa de um formalismo que permita lidar com um ambiente onde sistemas de tempo contínuo e discreto coexistem. Tendo como base a linguagem SIGNAL o formalismo do Scicos foi definido da seguinte forma (Gomez 1999):

- Cada sinal $x(t)$ está associado a um vetor com intervalos e instantes de tempo, denominado tempo de ativação T_x;
- Sinais estão sempre presentes, mesmo fora do escopo de seus tempos de ativação.

Um sinal X foi definido como uma função vetorial de tamanho n com o tempo como variável independente x(t), associada ao conjunto de tempos T_x. T_x é um conjunto de intervalos e instantes de tempo isolados (eventos) nos quais o sinal é ativado. Um sinal de ativação é um sinal X que teve sua função x(t) igualada a zero, mas que ainda possui o conjunto de ativações T_x.

Cada bloco pode ter entradas e saídas regulares ou entradas e saídas de ativação. As entradas e saídas regulares são representadas no Scicos por triângulos pretos e as entradas e saidas de

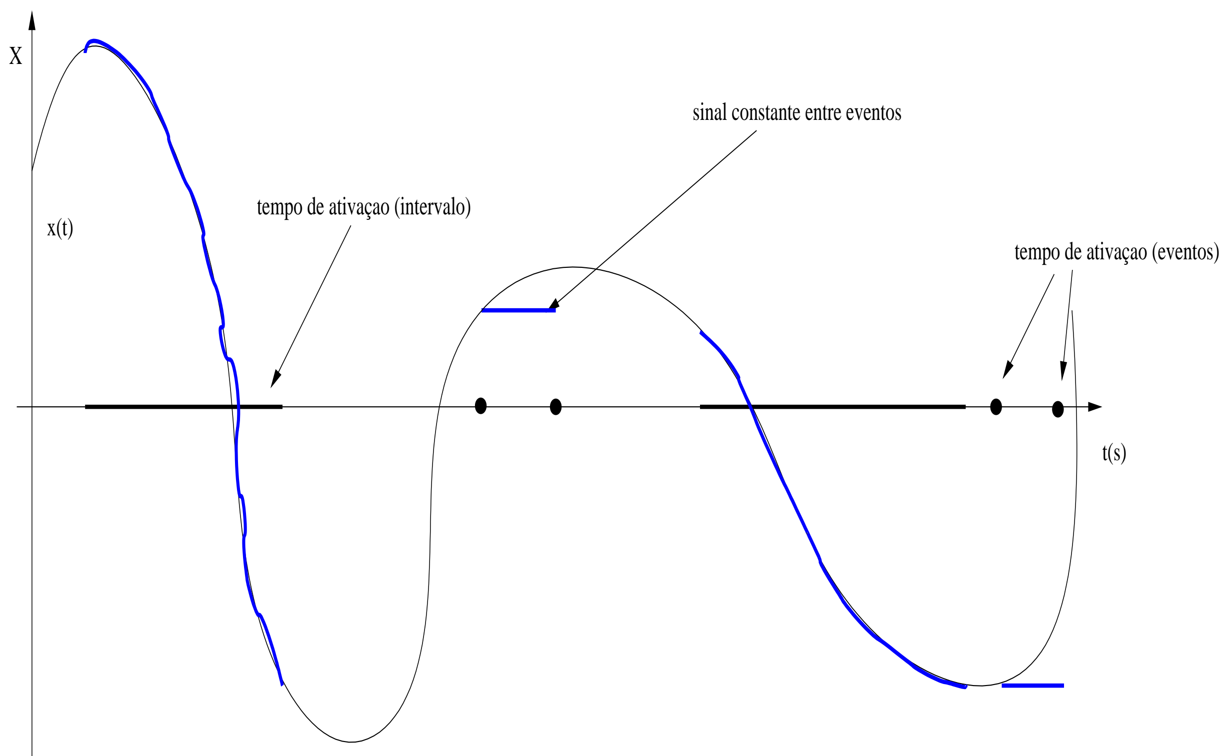

Figura 2.18: Sinais no Scicos.

ativação são representadas por triângulos vermelhos. (Gomez 1999). Cada bloco do Scicos é composto por duas funções: computacional, que define o comportamento do bloco, e de interface, que trata da parte gráfica. Durante a simulação o simulador necessita que o bloco realize certas computações. Para isso ele chama a função computacional do bloco com um determinado valor, um *flag* que simboliza um determinado estágio da simulação. As funções computacionais são geralmente escritas como laços usando estruturas do tipo *if-then-else* ou *switch-case*.

As etapas (*flags*) existentes durante a simulação foram listadas na tab. 2.9 juntamente com uma breve descrição. O número do *flag* corresponde ao índice das funções específicas que calculam estados e saídas dos blocos.

A simulação funciona em três etapas:

- inicialização - a função `cosini` inicializa entradas e saídas de todos os blocos chamando suas funções computacionais com *flag* = 4 ou *flag* = 6.
- cálculo de saídas e estados - a maior parte da simulação é realizada pela função `cossim`.

Durante este estágio o tempo de simulação pode estar correndo continuamente ou de forma discreta. Quando a ativação de um bloco é contínua, a função `cossim` chama o integrador `LSODAR` com *flag* = 0 para calcular o estado contínuo do bloco. Esse estado é calculado de acordo com a seguinte eq. 2.36:

$$\dot{x} = f_0(t, x(t), z(t), u(t), \mu(t)), \tag{2.36}$$

onde:

f_0 é uma função específica do bloco;

t_e é um vetor de tempos de ativação;

$u(t)$ é o vetor de entradas;

μ é uma função de n_e e do modo m;

n_e é o código (ou índice) de ativação;

O índice de ativação determina as portas que ativam o bloco. Ele é definido por:

$$n_e = \sum_{j=1}^{n} i_j 2^{j-1}, \tag{2.37}$$

onde:

n é o número da entrada de ativação;

i_j é o sinal que determina ativação ou não-ativação (1 ou 0) na porta j.

Se um bloco possui quatro entradas de ativação e o sinal de ativação é recebido nas portas i_1, i_3 e i_4 o índice n_e será 1101_b, 13_d (Campbell et al. 2005).

Quando a função sendo integrada possui descontinuidades o integrador encontra dificuldades, como por exemplo, controlar o tamanho do passo de integração. Para minimizar estes problemas foram introduzidos os conceitos de modo $m(t)$ e de superfície de cruzamento de zeros $s(t)$.

Supondo que a função a ser integrada é a função módulo:

$$\begin{cases} y(t) = u(t), u \geq 0 \\ y(t) = -u(t), u < 0 \end{cases} \tag{2.38}$$

Esta função não é diferenciável em $t = 0$. Neste caso um modo pode ser definido para especificar o período no qual $u(t)$ é positiva.

No ponto onde $t = 0$ a superfície de cruzamento de zeros é introduzida. Durante a integração a saída é calculada fazendo $y(t) = u(t)$ se $m = 1$, e $y(t) = -u(t)$ caso contrário. Ao encontrar uma superfície de cruzamento de zeros a integração pára, o bloco calcula o modo e a integração continua.

Se o *flag* é 9, o modo e a superfície são calculados:

$$[m(t), s(t)] = f_9(t, x(t), z(t), u(t), \mu(t)) \tag{2.39}$$

Neste caso $n_e = -1$, pois não existe nenhuma entrada ativada, o bloco é ativado internamente.

Quando a ativação é gerada por um bloco discreto a função `cossim` chama os blocos com *flags* 1 e 3. Então as saídas regulares e de ativação são atualizados antes dos estados discretos (que são chamados com *flag* 2). As saídas discretas são computadas pela equação

$$y(t_e) = f_1(t_e, x(t_{e^-}), z(t_{e^-}), u(t_e), \mu(t_e)) \tag{2.40}$$

e as saídas contínuas são computadas por

$$y = f_1(t, x(t), z(t), u(t), \mu(t)), \tag{2.41}$$

onde:

f_1 é uma função específica do bloco;

$x(t_{e^-})$ e $z(t_{e^-})$ são os estados contínuo e discreto antes do evento.

Os estados internos são atualizados de acordo com

$$[z(t_e), x(t_e)] = f_2(t_e, x(t_{e^-}), z(t_{e^-}), u(t_e), \mu(t_e)), \tag{2.42}$$

onde:

f_2 é uma função específica do bloco.

- finalização - durante a etapa de finalização da simulação a função `cosend` chama cada função computacional com o *flag* = 5. Neste momento geralmente são fechados arquivos, liberada memória alocada, etc.

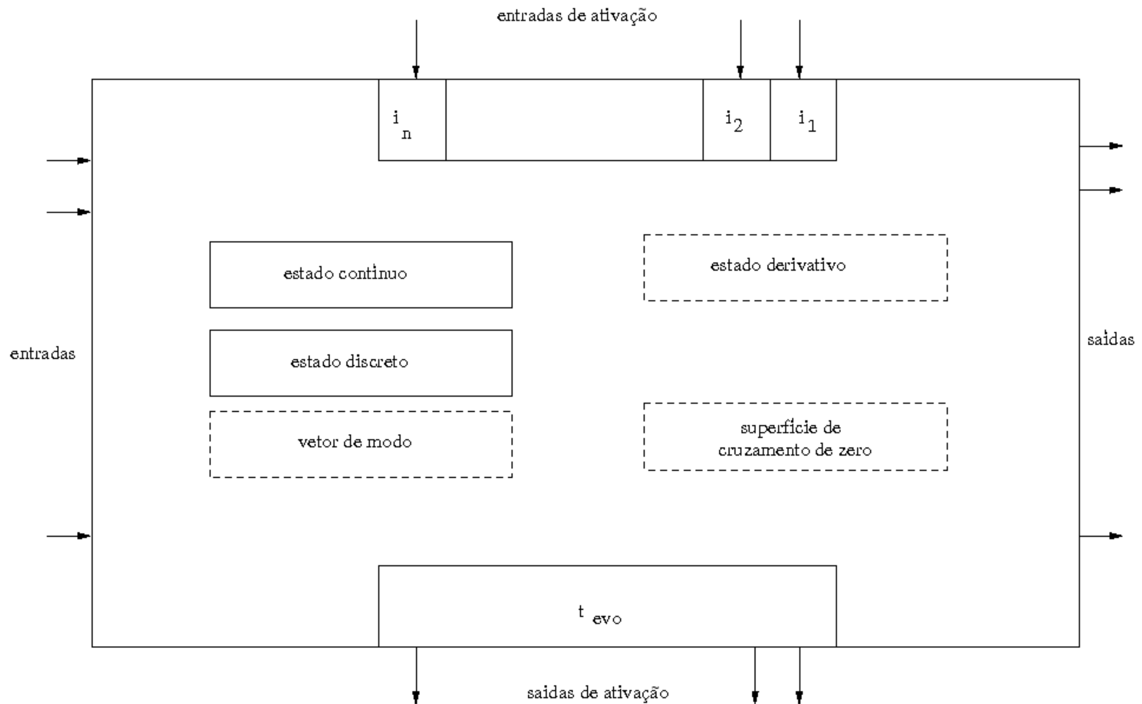

Figura 2.19: Diagrama esquemático de um bloco

Para criar um bloco é necessário escrever uma função computacional e uma função de interface. O Scilab fornece uma estrutura de dados (*struct* em C) que representa cada aspecto computacional de um bloco. Alguns dos elementos da estrutura foram listados abaixo:

- estado interno e sua derivada com relação a t: `double *x` e `double *xd`;
- tabela de ponteiros para as entradas: `double **inptr`;

- tamanho das entradas regulares: `int *insz`;
- tabela de ponteiros para as saídas: `double **outptr`;
- número de parâmetros reais: `int rpar`;
- parâmetros reais: `double *rpar`;
- número de superfícies de cruzamento de zeros: `int ng`.

O trecho de código ilustrado abaixo faz parte de `absolute_value.c`, função computacional do bloco ABS, que calcula valores absolutos.

```
for(i=0;i<block->insz[0];++i){
if (block->inptr[0][i]<0){
  block->outptr[0][i]=-block->inptr[0][i];
}else{
  block->outptr[0][i]=block->inptr[0][i];
}
```

O trecho de código mostra um laço que define as saídas `block->outptr[0][i]` a partir das entradas `block->inptr[0][i]`. Se a entrada é negativa a saída recebe o valor da entrada com sinal trocado, senão a saída simplesmente copia a entrada. A lista completa dos elementos da estrutura pode ser consultada no arquivo de header onde ela foi definida, `../scicos_block.h`.

A função de interface é segundo código necessário para se escrever um bloco. Essa função é escrita na linguagem do Scilab e é a responsável pela definição da geometria, cor, número e tamanho de portas e outros atributos gráficos de um bloco.

Assim como na função computacional a de interface também realiza computações baseadas em *flags*, denominados *jobs*. A assinatura da função é `[x,y,typ] = nome_bloco(job,arg1,arg2)`. O parâmetro *job* pode assumir os seguintes valores:

plot - desenha o bloco e seu rótulo. A função `standard_draw` pode ser usada para desenhar um retângulo simples.

getinputs - retorna informações sobre as portas de entrada (posição e tipo);

getoutputs - retorna informações sobre as portas de saída (posição e tipo);

getorigin - retorna coordenadas do ponto inferior esquerdo do bloco;

set - recebe parâmetros digitados pelo usuário em uma caixa de diálogo;

define - inicializa os parâmetros do bloco.

Os parâmetros `set` e `define` são os mais complexos. No trecho de código da função de interface do bloco ABS (`ABS_VALUE.sci`) ilustrado abaixo o parâmetro `set` utiliza a função `getvalue` para receber os valores digitados pelo usuário; o valor 0 ou 1 define se haverá cruzamentos de zero ou não durante a simulação.

```
 case 'set' then
    x=arg1;
    graphics=arg1.graphics;exprs=graphics.exprs
    model=arg1.model;
    while %t do
      [ok,zcr,exprs]=..
  getvalue('Set block parameters',..
   ['use zero_crossing (1: yes) (0:no)'],..
   list('vec',1),exprs)
      if ~ok then break,end
      graphics.exprs=exprs
      if ok then
if zcr<>0 then
  model.nmode=-1;model.nzcross=-1;
else
  model.nmode=0;model.nzcross=0;
end
x.graphics=graphics;x.model=model
break
      end
    end
```

Tabela 2.8: Interface para a função soma.

```
#include "stack-c.h"
extern int vectsum(int n, double * a, double * b, double * y);

void sci_sumab(char *fname){
int l1, m1, n1, l2, m2, n2, l3, n;

/* verifica numero de argumentos  */
int minlhs=1, maxlhs=1, minrhs=2, maxrhs=2;
CheckRhs(minrhs,maxrhs) ;
CheckLhs(minlhs,maxlhs) ;

/* verifica tipo dos argumentos de entrada e os coloca
       na pilha de entrada
*/
GetRhsVar(1, "d", &m1, &n1, &l1);
GetRhsVar(2, "d", &m2, &n2, &l2);

/* verifica o tamanho dos argumentos de entrada
*/
n=m2*n2;
if( n1!=n2 || m1!=m2)
{
 cerro("argumentos de entrada devem ter o mesmo tamanho");
 return 0;
}
if(n1!=0 && m1!=0)
 if(n1!=1 && m1!=1)
 {
 cerro("entradas devem ser vetores");
 return(0);
 }

/* cria variável que corresponde ao argumento de saida*/
CreateVar(3,"d",&m2,&n2,&l3);

/* chama funcao soma, retorna a soma para stk(l3)*/
soma(n,stk(l1),stk(l2),stk(l3));

/* especifica argumento de saida */
LhsVar(1) = 3;
return 0;
}
```

Tabela 2.9: Etapas da simulação.

flag	entradas	saidas	descrição
0	t, n_e, x, z, inptr, modo, fase	xd	calcula a derivada do estado contínuo
1	t, n_e, x, z, inptr, modo, fase	outptr	calcula as saídas do bloco
2	$t, n_e > 0, x, z$, inptr	x, z	atualiza estados devido a ativações externas

CAPÍTULO 3

ARQUITETURA E CONFIGURAÇÃO DO SISTEMA DE AQUISIÇÃO REMOTA DE DADOS

Este capítulo expõe as principais características do sistema de aquisição remota de dados proposto. Na primeira seção são abordados os objetivos do sistema, seus componentes principais e como eles interagem entre si. A segunda seção lista a configuração do sistema, detalhes sobre os microcomputadores e os programas envolvidos. A última seção descreve os módulos servidor e os diferentes clientes desenvolvidos para o sistema de aquisição.

3.1 Definição do Sistema

Este projeto surgiu devido a dificuldade na utilização dos sistemas de aquisição de dados do Laboratório de Controle e Automação da Faculdade de Engenharia da UERJ. Esses sistemas eram baseados em placas de aquisição de dados antigas, mas de excelente qualidade, de barramento ISA de 8 e 16 bits, com microcomputadores também mais antigos que acomodavam essas placas. A limitação do sistema devia-se ao fato de que tanto *hardware* como *software* não podiam ser atualizados.

Os problemas com o *hardware* iniciaram quando os fabricantes de placas-mãe de computadores começaram a excluir os *slots* de barramento ISA. Os microcomputadores foram evoluindo e ficando mais velozes, mas as placas de aquisição de dados existentes no laboratório não puderam ser migradas para outros computadores. Os problemas com o *software* surgiram por conta dos *drivers* e programas de aquisição e calibração dos fabricantes das placas de aquisição. Os fabricantes não fornecem *drivers* para outros tipos nem versões de sistemas operacionais. Isto fez com que o sistema de aquisição do laboratório ficasse quase impossível de utilizar pois as aquisições e a análise dos dados eram realizadas em computadores lentos, executando sistemas

operacionais igualmente lentos.

A solução mais simples para o problema seria adquirir novas placas, desta forma seria possível o uso de novos computadores e sistemas operacionais mais avançados. Porém, ainda assim as placas de aquisição antigas e de qualidade superior não seriam utilizadas e, mesmo com sistemas novos, sempre haveria a dificuldade imposta pelo *software* distribuído pelo fabricante, para apenas uma plataforma.

Uma outra solução seria modificar toda a estrutura do sistema de aquisição. Ao invés de utilizar um computador com baixa capacidade de processamento para aquisitar e analisar os dados o sistema poderia ser ampliado se a aquisição e a análise fossem descentralizadas, realizadas em computadores diferentes. Desta forma o computador antigo, onde estão instaladas as placas de aquisição, realizaria apenas a etapa de aquisição, e um outro computador mais avançado receberia os dados e nele seria feita a análise. Esta possibilidade foi reforçada com o lançamento do projeto Comedi, quando uma grande quantidade de *drivers* de placas de aquisição foi escrita para o sistema operacional Linux. A partir de então foi possível reformular o sistema de aquisição de dados do laboratório.

A nova proposta foi baseada no modelo cliente/servidor. A ideia basica era ter um servidor, um computador onde estariam instaladas as placas de aquisição. Esse computador também estaria ligado a uma rede padrão *Ethernet* e possuiria o sistema operacional Linux instalado. O servidor executaria um programa que receberia mensagens do cliente. Essas mensagens seriam os parâmetros necessários para a execução de algumas funções de leitura, escrita e configuração da API do Comedi. O servidor retornaria ao cliente duas mensagens: um código de retorno que informa se a função foi executada corretamente e os valores ou mensagens retornados por ela. O cliente poderia ser qualquer computador localizado na mesma rede do servidor. Para executar as funções da API do Comedi esse computador poderia utilizar o Scilab/Scicos com o pacote escrito para este fim, ou um navegador de Internet pois a troca de mensagens seria realizada através do protocolo HTTP.

O sistema foi definido da seguinte forma:

- Servidor
  - aguarda solicitações de clientes;

- abre conexão com cliente;
- recebe mensagem do cliente;
- interpreta a mensagem e executa a função Comedi correspondente;
- envia confirmação e valores retornados ao cliente dentro do padrão HTTP, com um *header* de resposta e corpo.
- fecha a conexão com o cliente.

• Cliente Scilab/Scicos

- inicia comunicação com o servidor;
- envia requisições ao servidor através de mensagens no formato HTTP. O método `GET` deve ser utilizado e as mensagens devem conter o nome da função Comedi a ser executada e os parâmetros;
- interpreta mensagens recebidas do servidor;
- fecha a conexão;
- envia dados para que sejam tratados pelo Scilab/Scicos (desenho de gráficos, impressão de informações na tela).

• Cliente navegador de Internet

- inicia comunicação com o servidor enviando mensagem no formato HTTP;
- recebe a resposta do servidor e mostra o corpo da resposta na tela.

3.2 Configuração do Sistema

Segue a descrição da configuração de *hardware* e *software* do sistema de aquisição remota de dados:

• Servidor

- processador Intel Pentium II, 300MHz;
- memória principal de 256MB;

  - placa de aquisição de dados *Keithley Instruments* DAS-1602;
  - sistema operacional Linux, distribuição Vector Linux VL 5.9 - Light Edition. Kernel 2.6.22;
  - pacote Comedi comedi-0.7.76;
  - pacote Comedilib comedilib-0.8.1.

- Cliente
  - processador Intel Pentium 4, 2,40GHz;
  - memória principal de 1GB;
  - sistema operacional Linux, distribuição Mandriva 2010. Kernel 2.6.31;
  - Scilab 4.1.2.

As especificações da placa *Keithley Instruments* DAS-1602 são:

- Entrada analógica
  - 16 canais simples ou 8 canais diferenciais;
  - Entrada unipolar ou bipolar, configuráveis a partir de *jumpers* localizados na placa;
  - Resolução de 12 bits;
  - ganho/faixa de operação
    * ganho:1, faixa unipolar: 0 a $+10V$, faixa bipolar: $-10V$ a $+10V$;
    * ganho:2, faixa unipolar: 0 a $+5V$, faixa bipolar: $-5V$ a $+5V$;
    * ganho:4, faixa unipolar: 0 a $+2,5V$, faixa bipolar: $-2,5V$ a $+2,5V$;
    * ganho:8, faixa unipolar: 0 a $+1,25V$, faixa bipolar: $-1,25V$ a $+1,25V$;
- Saída analógica
  - 2 canais;
  - Resolução de 12 bits;
  - faixa de operação

    * faixa unipolar: 0 a $+5V$ ou $0V$ a $+10V$;
    * faixa bipolar: $-5V$ a $+5V$ ou $-10V$ a $+10V$;

- Entrada digital
  - 4 canais;
  - Resolução de 4 bits;
  - faixa de operação unipolar 0 a $+5V$;
- Saída digital
  - 4 canais;
  - Resolução de 4 bits;
  - faixa de operação unipolar 0 a $+5V$;

A pinagem do conector externo da placa foi ilustrada na fig. 3.1.

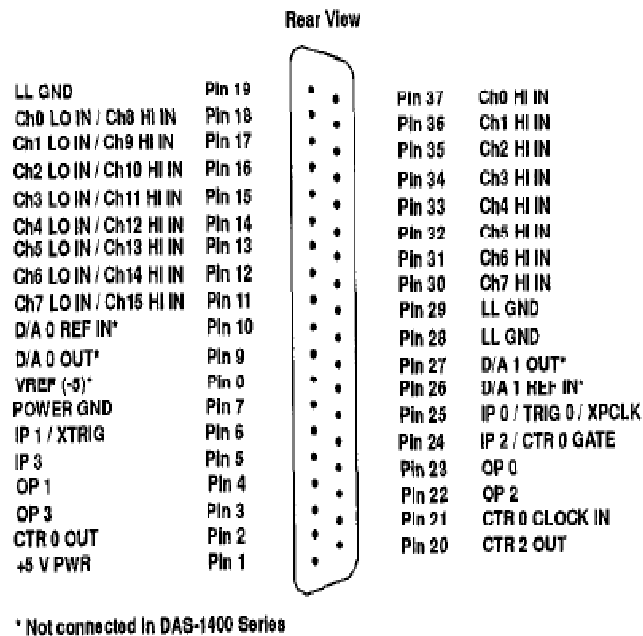

Figura 3.1: Pinagem do conector DB-37 externo da placa de aquisição.

3.3 Descrição dos Módulos Servidor e dos Clientes

Nesta seção são detalhados os módulos desenvolvidos para o servidor e os módulos desenvolvidos para o cliente.
Todos os módulos foram disponibilizados para *download* em www.lee.eng.uerj.br/ẽlaine/projeto.html.

3.3.1 Módulo Servidor

O módulo servidor foi todo desenvolvido na linguagem de programação C. As mensagens que chegam do cliente devem ter o seguinte formato:

```
GET <ip>:<porta>/<função_comedi>/<parametro1>/<parametro2>/<parametro3>
```

A mensagem enviada ao servidor localizado no endereço IP 192.168.1.2, porta 3490, solicitando o valor máximo que pode ser representado pelo subdispositivo de entrada analógica 1 da placa configurada como segundo dispositivo Comedi, `/dev/comedi2`, teria esta forma:

```
GET 192.168.1.2:3490/comedi_get_maxdata/2/1
```

A fig. 3.2 mostra o fluxograma do programa principal denominado `acqd` e a tab. 3.1 relaciona os códigos e suas funções principais.

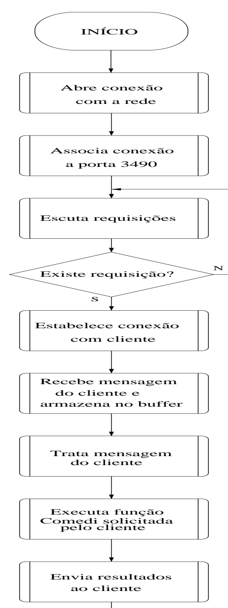

Figura 3.2: Fluxograma simplificado do servidor de aquisição

Tabela 3.1: Códigos do servidor e suas funções.

Arquivo	Função
acqd.c	Módulo principal. Controla a execução do servidor.
getLocalSocket.c	Abre um *socket* para o servidor e conecta à porta 3490.
getClientData.c	Recebe as mensagens do cliente e armazena em um *buffer*.
parser.c	Realiza o tratamento das mensagens que estão no *buffer*. Separa as mensagens em função Comedi e parâmetros.
execComedi.c	Recebe o nome da função Comedi e seus parâmetros. Executa as funções Comedi.
execCmd.c	Utiliza a função *execve* para executar funçoes de linha de comando.
sendMsg.c	Encapsula as mensagens a ser enviadas para o cliente com um *header* de resposta HTTP e transmite.
getDriverName.c	Recebe o nome da placa e procura o nome do *driver* para ser utilizado pelo comando `modprobe` e modelo para ser utilizado pelo comando `comedi_config`.
getInAddr.c	Realiza o tratamento de IPs no formato IPv4 ou IPv6.

Não é demais comentar detalhes sobre a função que executa as funções do Comedi, `execComedi`. Esta função recebe o nome da função Comedi e seus parâmetros, executa essa função e retorna o valor ao módulo principal, `acqd`. Porém existem dois comandos que precisam ser executados via linha de comando. Esses comandos são o `modprobe` que carrega o *driver* para o *kernel* do Linux e *comedi_config* que assinala um dispositivo físico a um dos 16 dispositivos Comedi disponíveis em `/dev/comedi`. Para a execução desses comandos e envio do valor retornado por eles ao cliente foi necessário utilizar a função `fork()` para bifurcar a execução do código (num processo filho) e a função `execve()` para chamar os dois comandos como se estivessem sendo executados da linha de comando. Além disso , ao abrir o processo filho também a saída padrão e o erro padrão (`stdout` e `stderr`) foram redirecionados para o *socket*, com isso foi possivel enviar as mensagens dos comandos diretamente para o cliente. Antes da execução do comando `modprobe` o programa precisa trocar seus privilégios de execução de um usuário comum para superusuário; isso é feito através da função `setuid()`. O trecho de código abaixo mostra os detalhes descritos:

```
/*
 *
 * Starts forking in 2 processes to call modprobe from the command line of the child process
```

```
 * First fork() is called to create a child process. Inside the child process the setuid()
 * command is called to give root power. This is needed for the modprobe command.
 * Right after that the close() commands are issued to close the stdout and sterr
 * the dup command redirects the 2 sysouts to the socket. Error messages and out messages
 * are now sent directly to the client.
 *
 */
pid = fork();
   if (pid == -1){
       /*
        * Error on fork():
        */
        fprintf(stderr,"%s: Failed to fork()\n", strerror(errno));
        exit(13);
        } else if (pid == 0){
       /*
        * Child process
        */
        uid_t uIDsu = 0;        /* sets superuser ID */
        uid_t uIDu = getuid();  /* gets user ID */
        uidStatusSU = setuid(uIDsu); /* sets superuser ID to issue modprobe */

        if(uidStatusSU < 0){   /* checks setuid execution for superuser setting */
         break;
        }
        else{
               close(1); /* closes stdout and redirects to socket */
               close(2); /* closes the stdout and redirects to socket */
               dup2(newSockfd,1);
               dup2(newSockfd,2);
               execStatusMod =  execCmd(MOD_PATH,argsMod);

               if(execStatusMod != 0){
                         break;
               }
```

```
            uidStatusU = setuid(uIDu); /* backs the user ID */
            if(uidStatusU < 0){
                    close(newSockfd);
            }
    }
```

A função `execve()` foi encapsulada na função `execCmd()` para incluir tratamento de erro.

A instalação do módulo servidor se dá através da execução do *script* `./install`. O *script* executa um *Makefile* que compila e faz a *linkagem* dos programas. O *script* ainda assinala o bit de usuário para `s` garantindo ao executável `acdq` acesso aos recursos do sistema, com isso é possível a execução dos comandos `modprobe` e `comedi_config`. Depois de executado o *script* `./install` o comando `ls acqd -la` deve retornar a seguinte linha:

```
-rwsrwxrwx 1 root   root     41595 2010-02-03 09:01 acqd*
```

3.3.2 Módulo Cliente Scicos

Para o cliente Scicos foram desenvolvidos quatro novos blocos. Cada bloco possui uma função computacional escrita em C, que faz as requisições para a execução das funções do Comedi durante as etapas da simulação do Scicos, e uma função de interface escrita na lingauem do Scilab, que define o número de entradas e saídas regulares, entradas de ativação e o visual gráfico do bloco. As funcionalidades de cada bloco foram listadas abaixo:

- *Comedi Network Analog Input* - Lê dos dados de um subdispositivo de entrada analógica (A/D);
- *Comedi Network Analog Output* - Escreve dados em um subdispositivo de saída analógica (D/A);
- *Comedi Network Digital Input* - Lê dados de um subdispositivo de entrada digital;
- *Comedi Network Digital Output* - Escreve dados de um subdispositivo de saída digital.

A tab. 3.2 mostra as ações executadas pelos blocos durante a simulação no Scicos. Apenas três fases são necessárias: inicialização, atualização das saídas e finalização.

Um diagrama simples com o bloco de leitura analógica é mostrado na fig. 3.3.

Tabela 3.2: Fases da simulação e funções dos blocos.

Fase	Inicialização	Atualização das saídas	Finalização
Entrada analógica (A/D)	Recebe parâmetros digitados pelo usuário e aloca ponteiros. Abre o dispositivo, verifica se o subdispositivo informado é do tipo entrada analógica, verifica o valor máximo que pode ser escrito e as faixas de operação.	Solicita ao servidor que execute a função `comedi_data_read()` para leitura de determinado canal e a função `comedi_to_phys()` que converte um valor amostrado para um valor real.	Desaloca ponteiros.
Saída analógica (D/A)	Recebe parâmetros digitados pelo usuário e aloca ponteiros. Abre o dispositivo, verifica se o subdispositivo informado é do tipo saída analógica, verifica o valor máximo que pode ser escrito e as faixas de operação. Inicializa valor seguro em 0 (zero).	Solicita ao servidor que execute a função `comedi_from_phys()` que faz a conversão de um valor real para um valor amostrado e a função `comedi_data_write()` que escreve o valor amostrado em um determinado canal.	Solicita que o servidor escreva valor seguro (zero) no canal especificado e desaloca ponteiros.
Entrada digital	Recebe parâmetros digitados pelo usuário e aloca ponteiros. Abre o dispositivo, verifica se o subdispositivo informado é do tipo entrada digital.	Solicta ao servidor que leia determinado canal através da função `comedi_dio_read()`.	Desaloca ponteiros.
Saída digital	Recebe parâmetros digitados pelo usuário e aloca ponteiros. Abre o dispositivo, verifica se o subdispositivo informado é do tipo saída digital.	Solicta ao servidor que leia determinado canal através da função `comedi_dio_write()`.	Desaloca ponteiros.

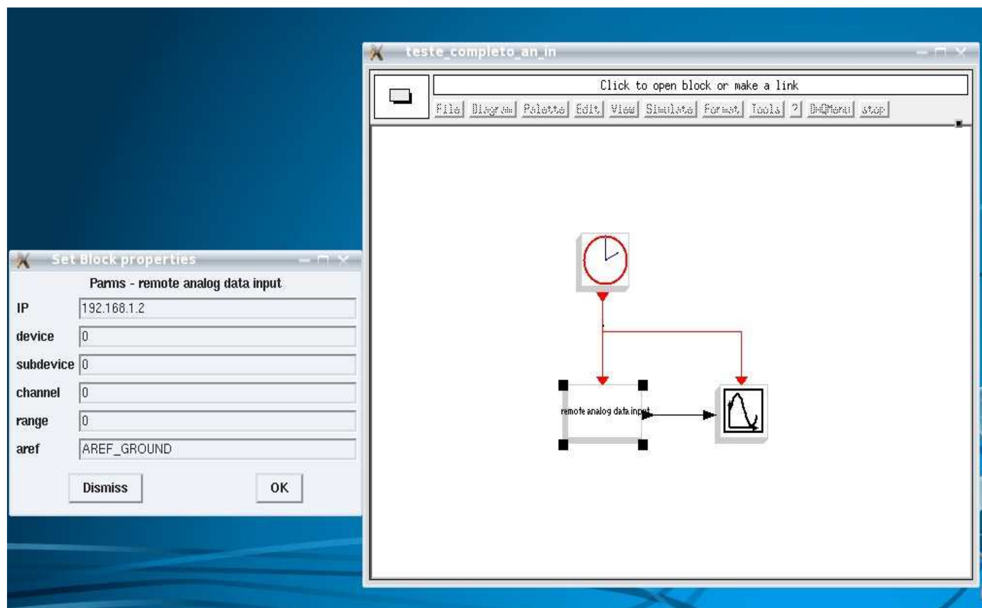

Figura 3.3: Detalhe dos parâmetros do bloco de entrada analógica.

Foram desenvolvidas ferramentas para configuração das placas de aquisição do servidor. Essas ferramentas estão localizadas no menu no Scicos denominado *DAQMenu*. O usuário insere dados como o nome da placa de aquisição, o IP do servidor, o número de interrupção e o endereço de base (se necessários) e a ferramenta envia um pedido de execução dos comandos `modprobe` e `comedi_config`. A fig. 3.4 mostra as respostas dos dois comandos.

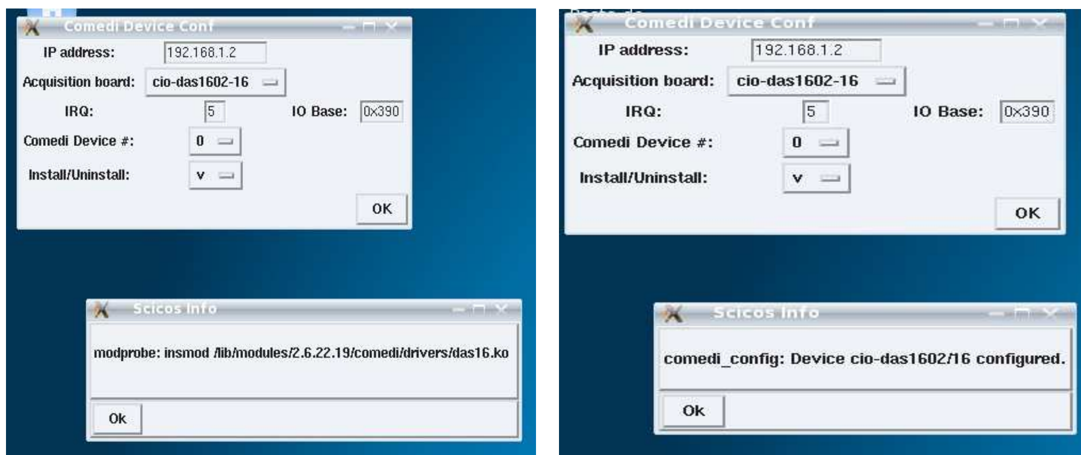

Figura 3.4: Módulo de configuração da placa de aquisição no Scicos.

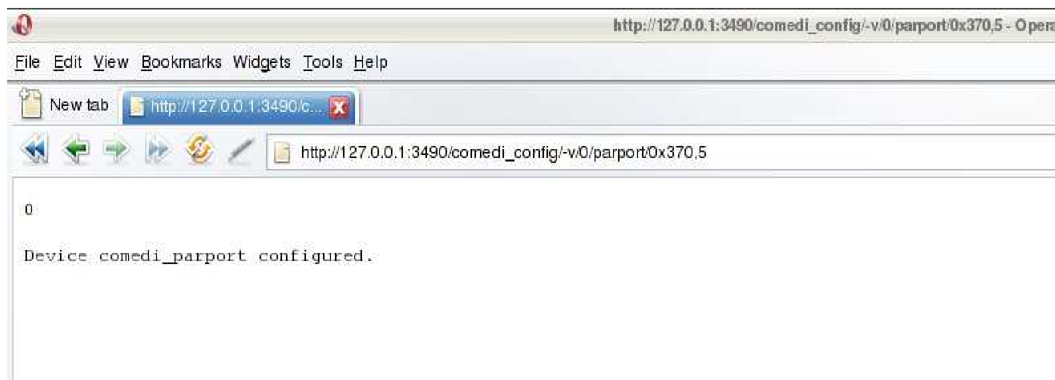

Figura 3.5: Detalhe da chamada ao servidor local para configuração da porta paralela.

Figura 3.6: Detalhe da chamada ao servidor local para escrita de bit na porta paralela.

3.3.3 Outros Clientes

A escolha do protocolo HTTP tornou possível o acesso de algumas funções do Comedi por meio de um navegador de Internet. É possível obter informações sobre um determinado dispositivo, realizar algumas leituras e escritas. Porém ainda é necessário o desenvolvimento de uma interface amigável para que a utilização do sistema seja mais simples.

Os comandos para a execução de funções a partir de um navegador seguem o padrão:

```
http://<endereco IP>:<porta>/<função_comedi>/<opções>/<parametro1>/<parametro2>/<parametro3>
```

A fig. 3.5 ilustra a configuração da porta paralela com a função `comedi_config` com a opção `-v` (*verbose* exibe mensagens) no dispositivo `/dev/comedi0` com as opções de tempo de execução endereço base= 0x370 e `IRQ` = 5.

O exemplo ilustrado na fig. 3.6 escreve o bit 1 no subdispositivo de saída digital (subdispositivo 2) da porta paralela.

Nas figuras acima, o valor 0 retornado na primeira linha significa que a função solicitada foi executada com sucesso. A segunda linha é o retorno da função do Comedi.

O *toolbox* de aquisição do Scicos deve ser instalado executando dois *scripts* do Scilab, `builder.sce` e `loader.sce`. O `builder.sce` compila todas as função do *toolbox* e só precisa ser executado uma única vez. O *script* `loader.sce` carrega as bibliotecas compartilhadas criadas pelo `builder.sce` ao ambiente de execução do Scilab, tornando as interfaces do *toolbox* disponíveis para uso. O `loader.sce` precisa ser executado a cada vez que se inicia o Scilab.

CAPÍTULO 4

RESULTADOS EXPERIMENTAIS

Este capítulo discute os testes realizados com o sistema de aquisição remota de dados. Foram realizados quatro testes no Laboratório de Controle e Automação da Faculdade de Engenharia da UERJ com o objetivo de avaliar o atraso de amostragem, a precisão da temporização na geração e aquisição de sinais. Os resultados são mostrados e discutidos nas seções 4.1, 4.2 e 4.3.

Por fim o sistema foi utilizado para controlar a posição de um servomotor. O principal objetivo deste teste foi o de mostrar a aplicabilidade do sistema em uma situação real. Os resultados foram comparados aos resultados obtidos através da simulação do modelo matemático do sistema de controle.

Todos os testes foram realizados com a placa de aquisição da *Keithely Instruments* modelo DAS-1602. Os computadores servidor e cliente, cuja configuração foi detalhada na seção 3.2, foram conectados através de um *link* Ethernet direto de 100Mbps. A Fig. 4.1 mostra a bancada com os computadores e o servomecanimo.

O comando *nice* foi utilizado no servidor para priorizar o processo *acqd* e no cliente para priorizar o processo *scilab*. A sintaxe do comando é `nice -n prioridade comando` onde a prioridade varia de -20 (maior prioridade) a 20 (menor). Os comandos foram digitados desta forma:

```
sudo nice -n -20 ./scilab
sudo nice -n -20 ./acqd
```

Figura 4.1: Bancada para os experimentos de controle. O computador cliente, à esquerda, executa o Scilab/Scicos. No centro está o servomecanismo e à direita está o computador servidor, que hospeda a placa de aquisição de dados.

4.1 Aquisição de Sinais

Para os testes de aquisição de sinais um gerador de sinais Techtronix foi acoplado ao subdispositivo de entrada analógica, canal 0 da placa de aquisição de dados. Foram realizadas dois testes. No primeiro, o gerador de sinais fornecia uma senóide de amplitude 3 (6Vpp) e frequência 1Hz. O diagrama de blocos do Scicos mostrado na fig. 4.2 foi configurado para realizar uma leitura a cada 10ms e exibir os dados graficamente.

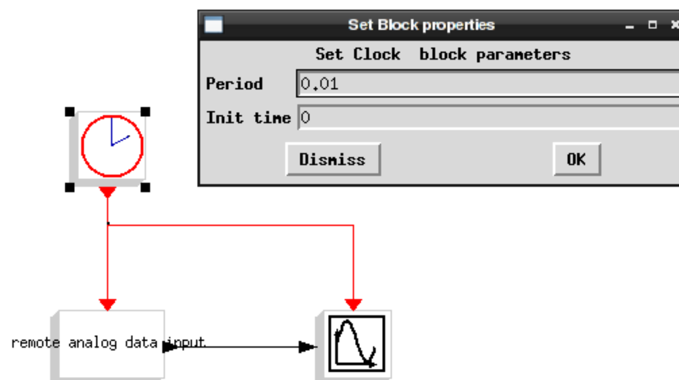

Figura 4.2: Diagrama do Scicos para testes de aquisição de dados pelo conversor A/D.

O resultado da simulação foi mostrado na fig. 4.3. O sinal gerado foi medido com o osciloscópio da Agilent DSO3102A e capturado usando o *software* do fabricante. Esta imagem é mostrada na fig. 4.4.

O segundo teste foi realizado aumentando a frequência do sinal gerado e mantendo o período

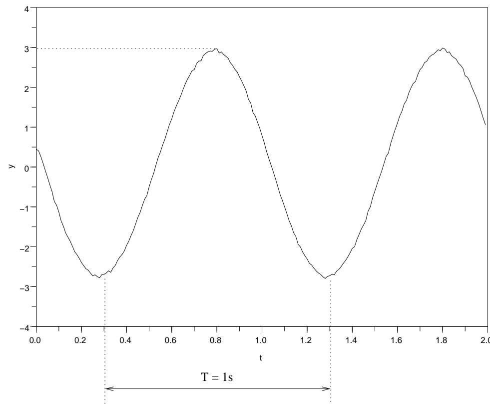

Figura 4.3: Senóide 1Hz amostrada a 10ms.

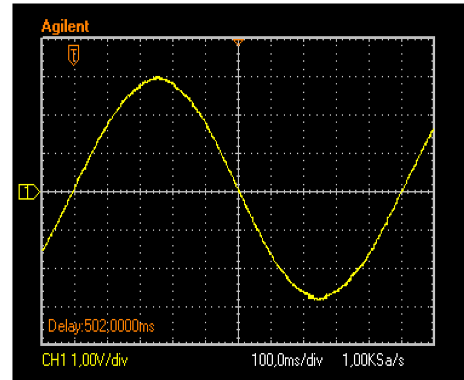

Figura 4.4: Senóide 1Hz fornecida pelo gerador de sinais.

de amostragem de 10ms. A fig. 4.5 apresenta o sinal amostrado e a fig. 4.6 ilustra o sinal medido pelo osciloscópio.

É possível medir a amplitude e período dos sinais amostrados a partir das formas de onda apresentadas nas fig. 4.3 e fig. 4.5. A amplitude do sinal amostrado é de 3V e o período de aproximadamente 1s na fig. 4.3 e 0,1s na fig. 4.5.

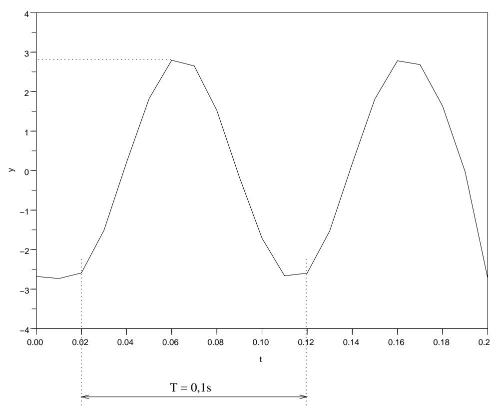

Figura 4.5: Senóide 10Hz amostrada a 10ms.

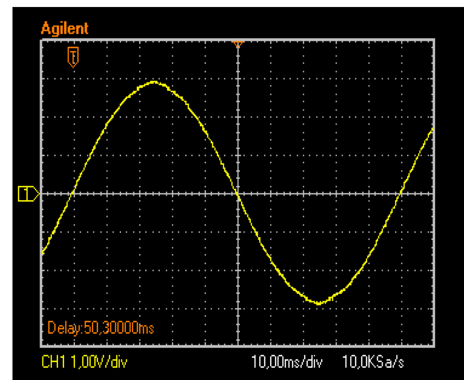

Figura 4.6: Senóide 10Hz fornecida pelo gerador de sinais.

4.2 Geração de Sinais

Os testes de geração de sinais foram realizados com o subdispositivo de saída analógica da placa de aquisição, canal 0. Um bloco gerador de senóides do Scicos foi ligado ao bloco de escrita analógica conforme apresentado na fig. 4.7.

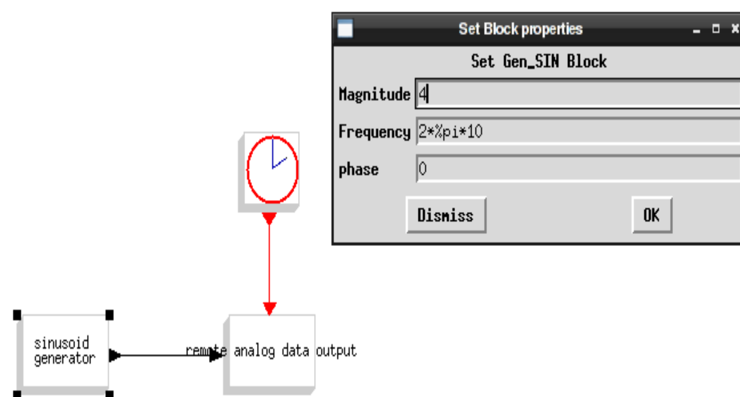

Figura 4.7: Diagrama do Scicos para testes de geração de dados.

Foram testadas a geração dos seguintes sinais:

- senóide de amplitude 4 V_p, frequência de 10Hz a uma taxa de 10ms;
- senóide de amplitude 4 V_p, frequência de 10Hz a uma taxa de 1ms e
- senóide de amplitude 4 V_p, frequência de 1Hz a uma taxa de 10ms.

Os sinais gerados pelo conversor D/A da placa e medidos pelo osciloscópio são apresentados nas fig. 4.8, fig. 4.9 e fig. 4.10.

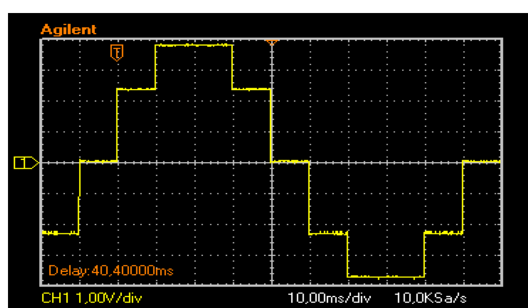

Figura 4.8: Sinal de 1Hz gerado a um período de amostragem de 10ms.

4.3 Medição do Atraso

A medição do atraso devido à amostragem, comunicação e processamento foi realizada da seguinte forma: uma senóide de 3V_p e 10Hz uma de 20Hz eram fornecidas por um gerador ana-

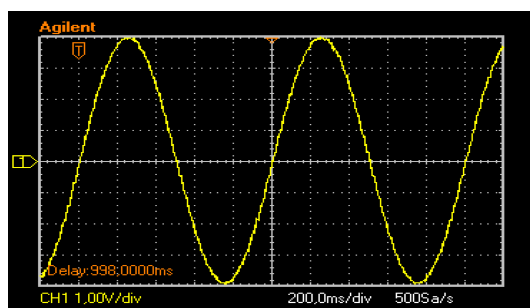

Figura 4.9: Sinal de 10Hz gerado a um período de amostragem de 10ms.

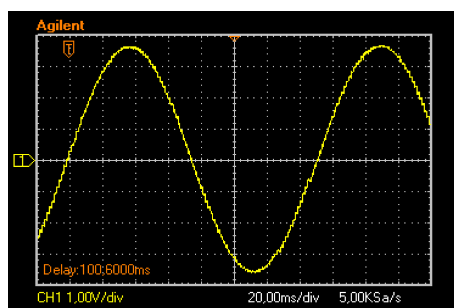

Figura 4.10: Sinal de 10Hz gerado a um período de amostragem de 1ms.

lógico conectado ao conversor A/D da placa de aquisição. O sinal digitalizado pelo conversor A/D foi enviado ao conversor D/A conforme o diagrama do Scicos na fig. 4.11

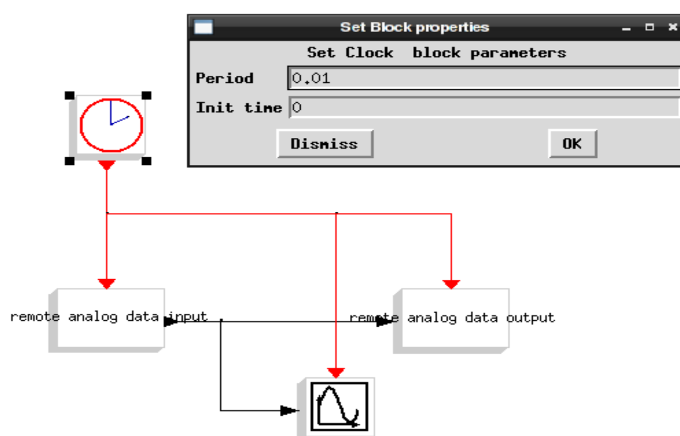

Figura 4.11: Diagrama do Scicos para medição do atraso de amostragem.

O período de amostragem era de 10ms. Os sinais gerados pelo D/A e pelo gerador de sinais foram medidos por um osciloscópio e apresentados nas fig. 4.12 e fig. 4.13.

Ao comparar os sinais constata-se que o atraso máximo entre o sinal analógico e o amostrado é 4ms.

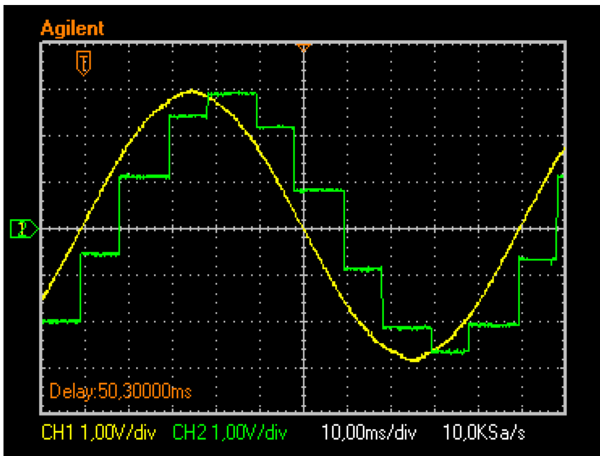

Figura 4.12: Medida do atraso de amostragem. Sinal de 10Hz.

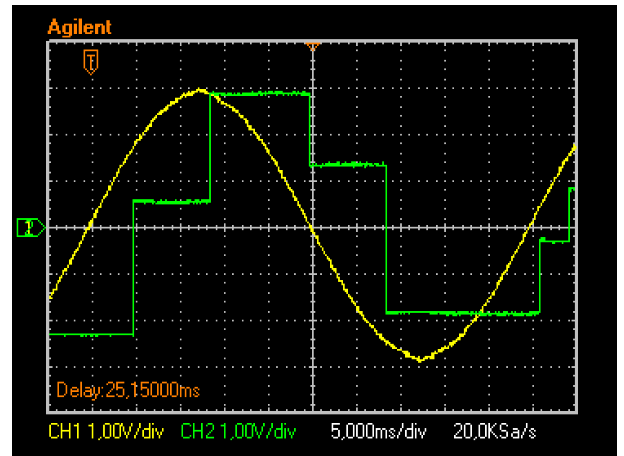

Figura 4.13: Medida do atraso de amostragem. Sinal de 20Hz.

4.4 Aplicação em um Servomecanismo

O sistema de aquisição remota de dados foi utilizado para controlar a posição do servomotor na fig. 4.14. Foi utilizado o servomecanismo de posicionamento rotativo SRV-02 em conjunto com o módulo amplificador de potência PA0103, ambos da *Quanser consulting*. O servomecanismo é composto de um motor DC com caixa de redução. Um potenciômetro mede a posição angular da carga acoplada ao eixo do servomecanismo.

Figura 4.14: Servomotor Quanser SRV-02.

O objetivo do teste era mostrar a viabilidade da sua aplicação em sistemas reais.

Para realizar simulações, o sistema foi modelado pelo diagrama de blocos implementado no

Scicos, que pode ser visto na fig. 4.15, na qual o servomecanismo é modelado pela função de transferência:

$$G(s) = \frac{1,68 \times 10^6}{s(s+57)(s+14,4 \times 10^3)} \tag{4.1}$$

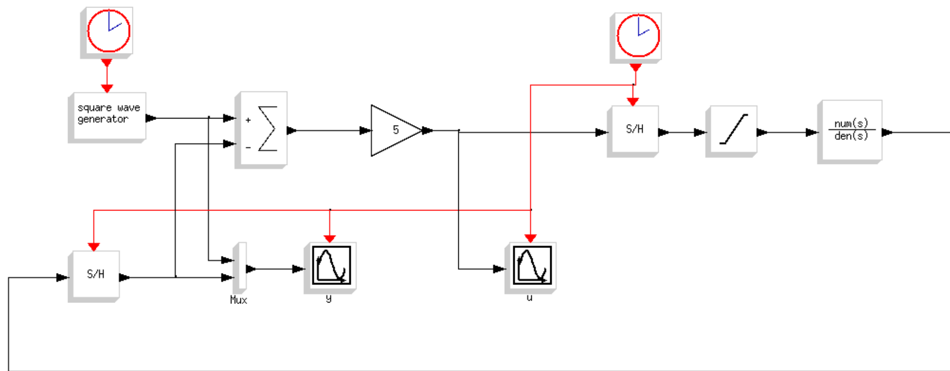

Figura 4.15: Simulação do sistema de controle de posição. Controle proporcional, por computador.

O sinal de referência é uma onda quadradada de 2 Vp de amplitude e período de 4 s.

Os conversores A/D e o D/A foram modelados pelos blocos de *sample and hold* (S/H) e um saturador que limita os sinais a uma faixa de -5 V a +5 V. A simulação foi realizada com período de amostragem de 10ms e controlador proporcional com ganho $K_p = 5$, ajustado para que a resposta seja rápida, mas sem *overshoot*. O sinal de saída y e de referência r são exibidos na fig. 4.16. O sinal de controle u é apresentado na fig. 4.17.

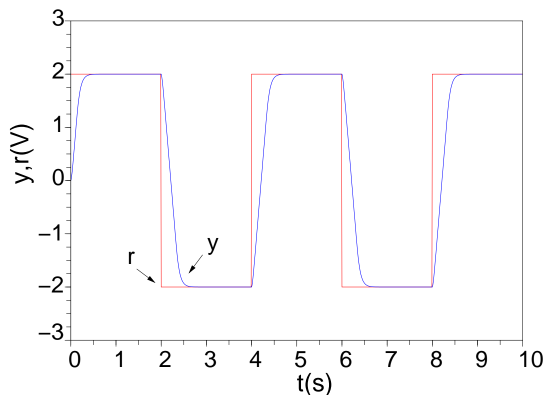

Figura 4.16: Sinais de referência (r) e saída (y).

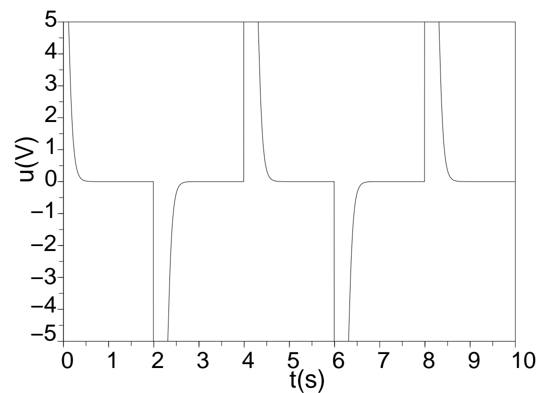

Figura 4.17: Sinal de controle (u).

Para realizar o experimento com o servomecanismo, os blocos S/H e o saturador foram substituídos pelos blocos *remote analog data input*, configurado para adquirir sinais do A/D da placa DAS-1602, e o *remote analog data output*, configurado para enviar sinais ao D/A. A fig. 4.18 mostra o novo diagrama.

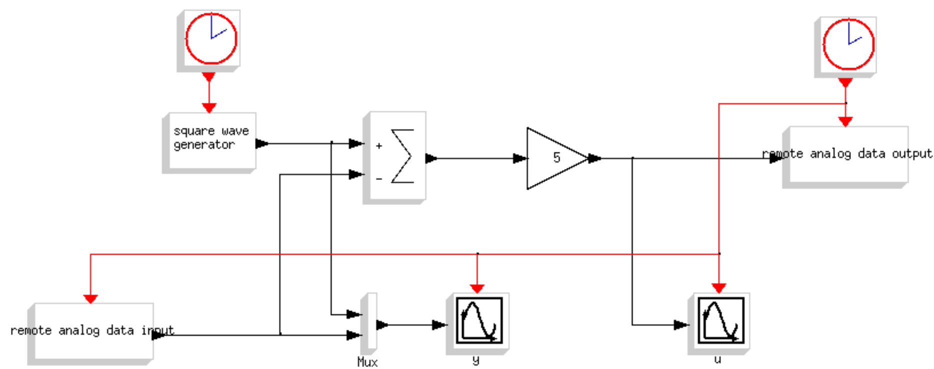

Figura 4.18: Implementação do sistema de controle de posição. Controle proporcional, por computador. Tempo real

Os resultados experimentais são apresentados nas figs. 4.19 e 4.20.

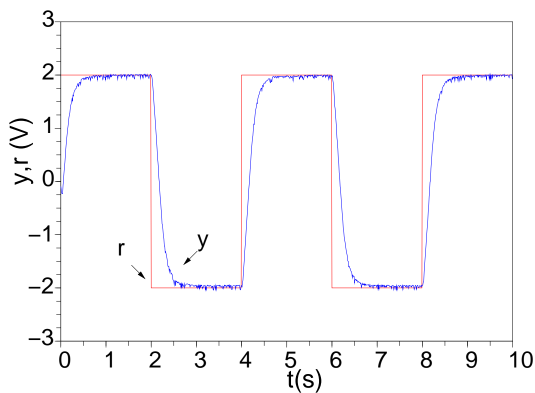

Figura 4.19: Sinais de referência (r) e saída (y) experimentais.

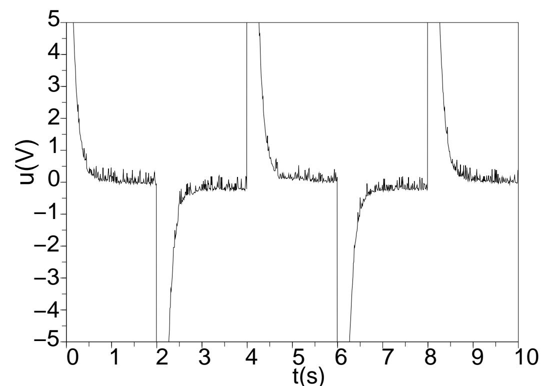

Figura 4.20: Sinal de controle (u) experimental.

Comparando-se os resultados experimentais (figs. 4.19 e 4.20) com os simulados (figs. 4.16 e 4.17), nota-se que são bastante parecidos, diferindo apenas pelo ruído de medição presente nos experimentos. O sistema de aquisição de dados desenvolvido para realizar os experimentos *hardware in the loop* teve um bom desempenho no controle deste servomecanismo.

Capítulo 5

Conclusões

O sistema de aquisição de dados desenvolvido neste trabalho permitiu o reaproveitamento de equipamentos legados existentes no Laboratório de Controle e Automação da Faculdade de Engenharia da UERJ. A arquitetura cliente/servidor melhorou a utilização dos recursos, visto que não há a necessidade de uma placa de aquisição em cada microcomputador, disponibilizando o sistema de aquisição a mais alunos.

Com os testes realizados em laboratório e apresentados verifica-se a possibilidade de uso do sistema de aquisição desenvolvido para a realização de experimentos clássicos na área de controle. Sem nenhum recurso de extensão do *kernel* do Linux para a execução de programas em tempo real foi possível realizar simulações com período de amostragem de até 10 ms alcançando bons resultados. Para períodos de amostragem menores que 10 ms o sistema apresenta problemas de temporização e precisão dos valores amostrados. Parte do problema é devido ao grande número de conexões abertas entre cliente e servidor causando um esgotamento do número de portas disponíveis no cliente e um *overhead* causado pelo estabelecimento de múltiplas conexões, que são abertas a cada instante de amostragem.

O sistema continua em desenvolvimento e novas versões do servidor darão suporte à versão 1.1 do protocolo HTTP, permitindo que a troca de mensagens seja realizada através de conexões persistentes, resolvendo assim o problema de múltiplas conexões. Além disso o código do servidor está sendo paralelizado para permitir o tratamento de mais solicitações ao mesmo tempo.

O principal limite do sistema é o atraso inserido pelo tempo de *round-trip* dos pacotes na rede Ethernet. Com o computador servidor e cliente conectados através de um *link* direto o tempo de *round-trip* médio medido pelo comando `traceroute` foi de 0,24 ms.

5.1 Continuação Deste Trabalho

Futuramente o sistema poderá ser usado como base para um laboratório remoto (*weblab*). Um servidor *web* pode ser implementado depois de definidas políticas de acesso e segurança. Assim, o aluno poderá acessar os experimentos via *browser*.

Além disso o sistema de aquisição de dados poderá ser utilizado para experimentos com projeto de controladores para uma planta desconhecida. Um modelo da planta pode ser construído no Scicos, em um computador ao qual os alunos não têm acesso. Esse modelo envia sinais para o servidor e recebe sinais dele. Esses sinais são capturados pelos computadores dos alunos que podem então construir seus controladores baseados no comportamento da planta.

Todo o código-fonte do sistema de aquisição desenvolvido encontra-se em

http://www.lee.eng.uerj.br/~elaine/acqd.html.

Referências Bibliográficas

Årzén, K.-E., Blomdell, A. & Wittenmark, B. (2005), 'Laboratories and real-time computing', *IEEE Contr. Sys. Mag.* **25**(1), 30–34.

Åstrom, K. & Wittenmark, B. (1997), *Computer-Controlled systems: theory and design*, 3ª edn, Prentice Hall.

Balaji, R. M., Ansari, F., Keimig, J. & Sheth, A. (2001), Utime - micro-second resolution timers for linux. Disponível em http://www.ittc.ku.edu/utime/.

Bazanella, e. a. (2007), *Enciclopédia de Automática - Controle & Automação - vol. 2*, 1ª edn, Blucher.

Bovet, D. & Cesati, M. (1996), *Understanding the Linux Kernel*, 1ª edn, O'Reilly.

Boyce, W. E. & DiPrima, R. C. (2002), *Equações diferenciais elementares e problemas de valores de contorno*, 7ª edn, LTC editora.

Campbell, S. L., Chancelier, J.-P. & Nikoukhah, R. (2005), *Modeling and Simulation in Scilab/Scicos*, Springer-Verlag.

Dorf, R. C. (1992), *Modern Control Systems*, 6ª edn, Addison-Wesley Publishing Company, Inc.

Fitzgerald, A. E. (2006), *Máquinas Elétricas*, 6ª edn, Bookman.

Gomez, C. e. a. (1999), *Engineering and Scientific Computing with Scilab*, 3ª edn, Prentice Hall.

Guedes, R. M. & Silva, E. M. (2005), Introdução ao uso do Linux. Disponível em http://www.lee.eng.uerj.br/~elaine.

Hall, B. (2001), Beej's guide to network programming - using internet sockets. Disponível em http://beej.us/guide/bgnet/.

Hindmarsh, A. C. (2001), 'Brief description of odepack - a systematized collection of ode solvers double precision version'.

IBM (2005), iseries - socket programming. Disponível em http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/rzab6/rzab6mst.pdf.

Jones, M. T. (2006), Inside the linux scheduler. Disponível em http://www.ibm.com/developerworks/linux/library/l-scheduler/.

Kuo, B. C. (1985), *Sistemas de Controle Automático*, 4ª edn, Prentice Hall.

Lourenço, M. (2002), 'Solução de sistemas de equações algébrico-diferenciais ordinárias de índice superior'. Disponível em http://www.lume.ufrgs.br/handle/10183/2526.

Ma, L., Xia, F. & Peng, Z. (2008), 'Integrated design an implementation of embedded control systems with scilab', *Sensors International Open Access Journal* pp. 5501–5515.

Mannori, S., Nikoukhah, R. & Steer, S. (2006), 'Scicos-hil - scicos hardware in the loop'. Disponível em http://www.scicos.org/scicoshil.html.

Najafi, M. & Nikoukhah, R. (2006), Modeling and simulation of differential equations in scicos, *in* '6th Modelica International Conference', Viena, Austria, pp. 177–185. Disponível em http://www.modelica.org/events/modelica2006/Proceedings/sessions/Session2c1.pdf.

NI, C. (1997), *AT E Series User Manual*, National Instruments Corporation.

Pacitti, T. & Atkinson, C. P. (1977), *Programação e Métodos computacionais - vol 2.*, 2ª edn, LTC.

Pendharkar, I. (2005), 'Rltool for Scilab', *IEEE Contr. Sys. Mag.* **25**(1), 23–25.

Ruggiero, M. A. G. & Lopes, V. L. R. (1996), *Cálculo Numérico*, 2ª edn, Makron Books.

Schleef, D., Hess, F. & Bruynickx, H. (2003), The control amd measurement device interface handbook. Disponível em http://www.comedi.org/doc/index.html.

Scilab, G. (2003), Guide for developers - scilab internals. Disponível em http://www.snv.jussieu.fr/~wensgen/Doc/scilab-2.6/internals/node1.html.

Silva, E. M. & Cunha, J. P. V. S. (2006), Scilab, scicos e rltool: Softwares livres no ensino de engenharia elétrica, *in* 'Proc. 16° CBA Congresso Brasileiro de Automática.', Salvador, BA, pp. 1620–1625.

Stevens, R. (1998), *Unix Network Programming. Vol. 2 - Networking APIS: Sockets and XTI*, 2ª edn, Prentice Hall.

von Hagen, W. (2005), Real-time and performance improvements in the 2.6 linux kernel. Disponível em http://www.linuxjournal.com/article/8041.