Introdução ao Scilab
Versão 1.0

Prof. Paulo Sérgio da Motta Pires

Departamento de Engenharia de Computação e Automação
Universidade Federal do Rio Grande do Norte
Natal-RN, Novembro de 2001

Resumo

O Scilab é um ambiente utilizado no desenvolvimento de programas para a resolução de problemas numéricos. Criado e mantido por pesquisadores pertencentes ao *Institut de Recherche en Informatique et en Automatique*, INRIA, através do Projeto *MÉTALAU* (*Méthods, algorithmes et logiciels pour l'automatique*) e à *École Nationale des Ponts et Chaussées*, ENPC, o Scilab é gratuito (*Free software)* e é distribuído com o código fonte (*Open Source software).*

Este é um documento sobre a utilização e as principais características deste ambiente de programação numérica. É importante ressaltar que as referências definitivas permanecem sendo os manuais que acompanham o software. Por exemplo, podemos citar *Introduction to Scilab* [1], documento no qual este texto se baseia.

O objetivo principal é apresentar um texto introdutório, em português, sobre o Scilab. Nosso interesse é fazer deste documento um complemento aos textos utilizados em disciplinas como Métodos Computacionais, Cálculo Numérico, Computação Numérica, Álgebra Linear Computacional e correlatas.

O objetivo secundário é demonstrar que a utilização de *Free/Open Source software*, do ponto de vista do usuário, traz vantagens. Algumas dessas vantagens, enumeradas em [2], são :

- A última versão do software está sempre disponível, geralmente via Internet;
- O software pode ser **legalmente** utilizado, copiado, distribuído, modificado;
- Os resultados obtidos podem ser divulgados sem nenhuma restrição;
- Os programas desenvolvidos podem ser transferidos para outras pessoas sem imposições de quaisquer natureza;
- O acesso ao código fonte, evitando surpresas desagradáveis;
- O acesso a informação de alta qualidade, e
- A certeza de estar participando de uma comunidade cujo valor principal é a irrestrita difusão do conhecimento.

A versão mais recente deste texto está disponível, no formato `pdf`, em `http://www.dca.ufrn.br/~pmotta`. Comentários ou sugestões podem ser enviados para `pmotta@dca.ufrn.br`

Agradecimentos

Ao Klaus Steding-Jessen, pelo *LaTeX demo*. A maioria das informações sobre "como se faz isso em LaTeX" podem ser encontradas no documento escrito pelo Klaus[1];

Ao Dr. Jesus Olivan Palacios, pelas "conversas" sobre o Scilab;

[1] Informações sobre o *LaTeX demo* ou sobre o LaTeX em geral podem ser obtidas em http://biquinho.furg.br/tex-br/

Histórico

- Fevereiro de 1999 - Versão 0.1 - Início.
- Julho de 2001 - Versão 0.2 - Correções e atualizações.
- Julho de 2001 - Versão 0.3 - Correções e atualizações.
- Julho/Novembro de 2001 - Versão 1.0 - Reorganização do trabalho e correções. Disponibilização deste documento no *site* do Scilab - INRIA (http://www-rocq.inria.fr/scilab/books.html)

Este trabalho foi totalmente desenvolvido utilizando *Free/Open Source* software. O Scilab 2.6[2] foi instalado no Linux distribuição Slackware 7.1[3], kernel 2.2.16[4]. A digitação foi feita usando o Xemacs[5]. Os textos em LaTeX e as figuras em *extended postscript*, **eps**, foram transformados em **pdf** através de **pdflatex** e **epstopdf**, respectivamente.

[2]Página do Scilab : http://www-rocq.inria.fr/scilab/
[3]Página do Linux distribuição Slackware : http://www.slackware.com
[4]Página do kernel Linux http://www.kernel.org
[5]Página do Xemacs http://www.xemacs.org

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

O Scilab[1] é um ambiente voltado para o desenvolvimento de software para resolução de problemas numéricos. Foi criado e é mantido por um grupo de pesquisadores do INRIA[2] e do ENPC[3]. O Scilab é gratuito, *Free software*, e distribuído com o código fonte, *Open Source software*. O software é disponibilizado, também, em versões pré-compiladas para várias plataformas.

O objetivo principal é apresentar um texto introdutório, em português, sobre o Scilab. Nosso interesse é fazer deste documento um complemento aos textos utilizados em disciplinas como Métodos Computacionais, Cálculo Numérico, Computação Numérica, Álgebra Linear Computacional e correlatas. É importante ressaltar que as referências definitivas permanecem sendo os manuais que acompanham o software. Por exemplo, podemos citar *Introduction to Scilab* [1], documento no qual este texto se baseia.

O objetivo secundário é mostrar que a utilização de *Free/Open Source software*, do ponto de vista do usuário, sempre traz vantagens. Algumas delas [2],

- A última versão do software está sempre disponível, geralmente via Internet;
- O software pode ser **legalmente** utilizado, copiado, distribuído, modificado;
- Os resultados obtidos podem ser divulgados sem nenhuma restrição;
- Os programas desenvolvidos podem ser transferidos para outras pessoas sem imposições de quaisquer natureza;
- O acesso ao código fonte, evitando surpresas desagradáveis;
- O acesso a informação de alta qualidade, e
- A certeza de estar participando de uma comunidade cujo valor principal é a irrestrita difusão do conhecimento.

Este documento, desenvolvido dentro das premissas dos parágrafos precedentes, está dividido em cinco Capítulos e dois Apêndices. Neste Capítulo, mostramos o contexto no qual o programa e este trabalho estão inseridos.

[1]Página do Scilab: http://www-rocq.inria.fr/scilab
[2]Página do INRIA : http://www.inria.fr
[3]Página do ENPC : http://www.enpc.fr

No Capítulo 2, apresentamos uma visão geral das principais características do Scilab. Descrevemos as suas diversas opções e comandos básicos utilizados na edição de linha de comandos. Mostramos, também, alguns exemplos de operações que podem ser realizadas com o software.

O Capítulo 3 é dedicado aos vários tipos de dados que podem ser manipulados pelo Scilab.

No Capítulo 4, apresentamos exemplos de desenvolvimento de programas no Scilab e, finalizando, no Capítulo 5 utilizamos diversos comandos do Scilab voltados para a geração de gráficos.

No Apêndice A, apresentamos os procedimentos para a instalação do software a partir do código fonte. Descrevemos os principais arquivos, diretórios e todos os procedimentos para a instalação em máquinas com o sistema operacional Linux (a instalação foi realizada em uma máquina com distribuição Slackware 7.1, kernel versão 2.2.16). Os procedimentos para a instalação das distribuições binárias, por serem específicos de cada plataforma, não são apresentados. O usuário é aconselhado a buscar estas informações na página do Scilab.

No Apêndice B, apresentamos um procedimento que permite executar códigos escritos em linguagem C dentro do ambiente do Scilab.

Por tratar-se de um texto introdutório, deixamos de apresentar outras características do ambiente Scilab que, entretanto, podem ser consultadas nas referências [3, 4, 5, 6, 7, 8].

Acreditamos que a maneira mais adequada de ler este documento é em frente a um computador com o Scilab instalado e funcionando. Os exemplos apresentados e a própria funcionalidade do software poderão, desta forma, ser explorados com mais eficiência. Comentários ou sugestões sobre esse documento podem ser enviados para `pmotta@dca.ufrn.br`. A última versão deste trabalho encontra-se disponível em `http://www.dca.ufrn.br/~pmotta`.

Capítulo 2

O Ambiente Scilab

A interação do usuário com o Scilab pode ocorrer de duas formas distintas. Na primeira, os comando são digitados diretamente no *prompt* do Scilab. Neste modo, o programa funciona como se fosse uma sofisticada e poderosa calculadora. Na segunda, um conjunto de comandos é digitado em um arquivo texto. Este arquivo, em seguida, é levado para o ambiente Scilab e executado. Neste modo, o Scilab funciona como um ambiente de programação.

Neste Capítulo, apresentamos algumas características do ambiente gráfico do Scilab. Através de alguns exemplos de operações que podem ser realizadas em linha de comando, mostramos o Scilab funcionando como uma sofisticada calculadora. O objetivo é a familiarização com o software.

O Scilab como ambiente de programação é apresentado no Capítulo 4.

2.1 Introdução

O Scilab é um ambiente de programação numérica bastante flexível. Suas principais características são :

1. É um software de distribuição gratuita, com código fonte disponível. Sua linguagem é simples e de fácil aprendizado;
2. É um ambiente poderoso para geração de gráficos;
3. Implementa diversas funções para manipulação de matrizes. As operações de concatenação, acesso e extração de elementos, transposição, adição e multiplicação de matrizes são facilmente realizadas;
4. Permite trabalhar com polinômios, funções de transferência, sistemas lineares e grafos;
5. Apresenta facilidades para a definição de funções. As funções podem ser passadas para outras funções como argumento de entrada ou de saída;
6. Permite interface com rotinas escritas nas linguagens FORTRAN, C ou Maple[1];

[1]O Maple é um ambiente de programação voltado para processamento simbólico. Informações sobre este software podem ser obtidas em http://www.maplesoft.com

7. Suporta o desenvolvimento de conjuntos de funções voltadas para aplicações específicas (os chamados *toolboxes*).

Além dos *toolboxes* desenvolvidos pelo Grupo Scilab [3, 4, 5, 6, 7], outros estão disponíveis também gratuitamente. Para exemplificar, destacamos o `ANN` (*Artificial Neural Network Toolbox*), para redes neurais, o `FISLAB` (*Fuzzy Logic Inference Toolbox*), para lógica difusa, e o `FRACLAB` (*Fractal, Multifractal and Wavelet Analysis Toolbox*), para análise com fractais e wavelets. Existem, ainda, trabalhos desenvolvidos tendo o Scilab como ferramenta, como o apresentado em [2], e outros documentos introdutórios [9, 10]. Também, o Scilab, através de uma extensão chamada de `Scilab Paralelo` [11], Scilab//, pode ser executado em máquinas paralelas ou em redes de estações de trabalho, as `NOWs` - *Network of Workstations*. Processos podem ser ativados, programas podem ser executados em estações remotas, com comunicação entre eles, e os resultados agregados.

2.2 O Ambiente Gráfico do Scilab

Após a realização dos procedimentos de instalação descritos no Apêndice A, podemos começar a trabalhar com o Scilab. Assumiremos que o software esteja instalado no sistema operacional Linux. No ambiente gráfico do Linux[2], basta digitar `scilab` para começar a utilizar o programa. A tela inicial do Scilab é apresentada na Figura 2.1.

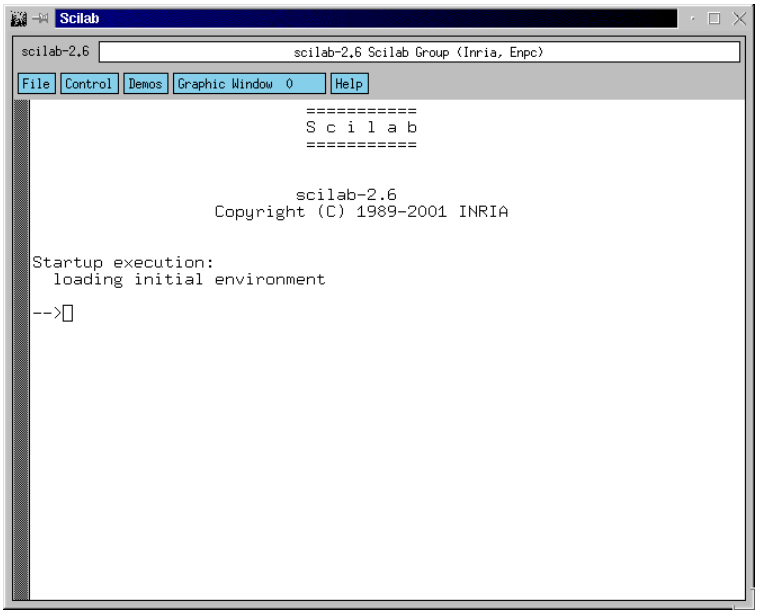

Figura 2.1: Tela inicial do Scilab

Na Figura 2.1, observamos que o *prompt* do Scilab é representado por uma seta, `-->`. Este *prompt* é chamado de *prompt* de primeiro nível. Ainda na Figura 2.1, podemos observar a existência de um menu horizontal com cinco opções: `File`, `Control`, `Demos`,

[2]O programa Scilab pode ser executado, também, em ambiente texto. Basta digitar `scilab -nw`. No ambiente texto, os gráficos que porventura forem gerados, serão apresentados no terminal gráfico, acessível via `Ctrl-Alt-F7`, caso este esteja disponível.

`Graphic Window 0` e `Help`. Utilizando o *mouse*, e colocando o cursor em cima de cada uma das opções, verificamos que:

- A opção `File` possui três sub-opções :
  - Opção `File Operations`, que permite carregar arquivos, funções e executar programas, entre outras ações.
  - Opção `Kill`, que permite interromper de maneira abrupta o processamento, saindo do ambiente Scilab.
  - Opção `Quit`, que permite sair do Scilab de forma normal.
- A opção `Control` possui quatro sub-opções :
  - `Resume` - continua a execução após uma `pause` ter sido dada através de um comando em uma função ou através de `Stop` ou `Ctrl-c`.
  - `Abort` - aborta a execução após uma ou várias `pause`, retornando ao *prompt* de primeiro nível.
  - `Restart` - restaura todas as variáveis e executa os programas de inicialização.
  - `Stop` - interrompe a execução do Scilab e entra em modo `pause`.
- A opção `Demos` - permite executar os vários programas de demonstração que acompanham a distribuição Scilab. É interessante, no primeiro contato com o programa, executar algumas das rotinas de demonstração.
- A opção `Graphics Window 0` possui cinco sub-opções para manipulação de janelas gráficas :
  - `Set (Create) Window`
  - `Raise (Create) Window`
  - `Delete Graphics Window` - permite apagar uma janela gráfica,
  - +
  - -
- A opção `Help` permite obter informações sobre o Scilab. Na versão 2.6, o `Help` está disponível para :
  - `Scilab Programming` - biblioteca de comandos que podem ser utilizados na programação com o Scilab;
  - `Graphic Library` - biblioteca de comandos gráficos;
  - `Elementary Functions` - funções elementares;
  - `Input/output Functions` - funções para entrada e saída de dados;
  - `Handling of functions and libraries` - funções para manipulação de funções e bibliotecas;
  - `Character string manipulations` - funções para manipulação de *strings*;
  - `Dialogs` - funções que permitem a criação de diálogos (menus, por exemplo);

- `Utilities` - funções com utilidades diversas;
- `Linear Algebra` - biblioteca de comandos usados em álgebra linear;
- `Polynomial calculations` - biblioteca de comandos usados em cálculos com polinômios;
- `Cumulative Distribution Functions, Inverse, grand` - funções de distribuição cumulativa, inversa e geradora de números randômicos;
- `General System and Control macros` - biblioteca de funções de controle;
- `Robust control toolbox` - funções do *toolbox* de controle robusto;
- `Non-linear tools (optimization and simulation)` - biblioteca de funções não-lineares para utilização em otimização e simulação;
- `Signal Processing toolbox` - funções do *toolbox* de processamento de sinais;
- `Fractal Signal Analysis` - biblioteca de funções para análise de sinais utilizando fractais;
- `Arma modelization and simulation toolbox` - funções do *toolbox* para modelamento e simulação ARMAX;
- `Metanet : graph e network toolbox` - funções do *toolbox* Metanet para análise de grafos;
- `Scicos: Block diagram editor and simulator` - funções para modelagem e simulação de sistemas dinâmicos;
- `wav file handling` - funções para manipulação de arquivos wav;
- `Language or data translations` - funções para conversão de dados entre o Scilab e alguns aplicativos;
- `PVM parallel toolbox` - funções que permitem o gerenciamento da comunicação com outras aplicações usando máquinas paralelas virtuais;
- `GECI Communication toolbox` - funções do *toolbox* de comunicação GECI. Este *toolbox* será removido nas próximas versões do Scilab. É aconselhável, portanto, utilizar o `PVM parallel toolbox`;
- `TdCs` - outro conjunto de funções com utilidades diversas, e.
- `TCL/Tk Interface` - funções que permitem a interface com as linguagens TCL/Tk.

2.3 Algumas Funções Primitivas

O Scilab é carregado com algumas funções pré-definidas, chamadas de primitivas. Algumas destas funções :

- Funções elementares : `sum, prod, sqrt, diag, cos, max, round, sign, fft`;
- Funções para ordenação : `sort, gsort, find`;
- Matrizes específicas : `zeros, eye, ones, matriz, empty`;
- Funções de álgebra linear : `det, inv, qr, svd, bdiag, spec, schur`;
- Polinômios : `poly, roots, coeff, horner, clean, freq`;

- Sistemas lineares : `syslin`;
- Números randômicos : `rand`
- Funções de programação : `function, deff, argn, for, if, end, while, select, warning, erro, break, return`;
- Símbolos de comparação : `==, >=, = &, |, <=, >, <`;
- Execução de arquivos : `exec`;
- Depuração : `pause, return, abort`;
- Splines, interpolação : `splin, interp, interpln`;
- Manipulação de *strings* : `string, part, evstr, execstr`;
- Gráficos : `plot, xset, driver, plot2d, xgrid, locate, plot3d`;
- Resolução de equações diferenciais : `ode, dassl, dassrt, odedc`;
- Otimização : `optim, quapro, linpro, lmitool`;
- Sistemas dinâmicos : `scicos` e
- Rotinas C ou FORTRAN : `link, fort, addinter, intersi`.

2.4 Variáveis Especiais

Existem variáveis especiais que são pré-definidas no Scilab. Elas são protegidas e não podem ser apagadas. Algumas destas variáveis são prefixadas com o caracter `%` e podem ser vistas através do comando `who`,

```
-->who
your variables are...

 startup   ierr      demolist  %scicos_display_mode
 scicos_pal          %scicos_menu        %scicos_short       %helps
 MSDOS     home      PWD       TMPDIR    percentlib          soundlib
 xdesslib  utillib   tdcslib   siglib    s2flib    roblib    optlib
 metalib   elemlib   commlib   polylib   autolib   armalib   alglib
 intlib    mtlblib   SCI       %F        %T        %z        %s
 %nan      %inf      $         %t        %f        %eps      %io
 %i        %e
 using       6036 elements  out of    1000000.
          and         45 variables out of       1791


-->
```

No *prompt* do Scilab, os comandos só são interpretados após o usuário pressionar a tecla `Enter`.

A variável `%i` representa $\sqrt{(-1)}$, `%pi` é a variável que representa $\pi = 3,1415926\ldots$, e `%e` é a variável que representa a constante de Euler $e = 2.7182818\ldots$. Uma outra variável pré-definida é `%eps` que representa a precisão da máquina na qual o Scilab está instalado (`%eps` é o maior número para o qual 1+`%eps` = 1). São pré-definidas, ainda, as variáveis `%inf` que significa "Infinito" e `%nan` que significa "Não é um Número" (*NotANumber).* A variável `%s` é definida pela função `s = poly(0, 's')`.

No Scilab são definidas, também, variáveis com valores booleanos : `%T` significando "verdadeiro" (*true*) e `%F` significando "falso" (*false*).

Uma atenção especial deve ser dada às variáveis `SCI` e `PWD`. Elas representam, respectivamente, o diretório no qual o Scilab foi instalado[3] e o diretório no qual o Scilab foi lançado e está rodando. A variável `home` possui valor idêntico ao da variável `PWD`.

```
-->SCI            // Diretorio no qual o Scilab foi instalado
 SCI  =

 /usr/local/scilab-2.6

-->PWD            // Diretorio no qual o Scilab foi lancado
 PWD  =

 /home/paulo

-->home          // Mesmo valor de PWD
 home  =

 /home/paulo

-->
```

As variáveis pré-definidas e protegidas estão no arquivo de inicialização `SCI/scilab.star`. Se desejar, o usuário pode pré-definir as suas próprias variáveis colocando-as no arquivo `home/.scilab`.

2.5 Manipulação de Arquivos e Diretórios

Como foi mostrado nos exemplos anteriores, uma linha de comentários sempre começa com os caracteres `//`. É importante salientar que os comentários (e os nomes das variáveis e funções utilizadas no Scilab) **NÃO** devem ter qualquer tipo de acentuação. O Scilab possui funções que podem ser utilizadas para manipular arquivos e diretórios. A função `pwd`, não confundir com a variável `PWD` da seção anterior, mostra o diretório no qual estamos trabalhando. Assim,

```
-->pwd                // Mostra o diretorio de trabalho

 ans  =
```

[3] Ver Apêndice A

```
 /home/paulo

-->
```

Com a função `chdir`, mudamos para o diretório de trabalho`teste`,

```
-->chdir('teste')     // Mudando o diretorio de trabalho
 ans  =

    0.

-->
```

O valor de `pwd` foi alterado mas o valor da variável `PWD` permanece inalterada, como podemos verificar pela seqüência,

```
-->pwd               // Mostrando o novo diretorio de trabalho
 ans  =

 /home/paulo/teste

 -->PWD              // PWD permanece inalterado.
 PWD  =

 /home/paulo

-->
```

As variáveis criadas no ambiente Scilab podem ser armazenadas em um arquivo. Vamos considerar as variáveis,

```
-->a = 1
 a  =

    1.

-->b = 2
 b  =

    2.

-->
```

Para salvar `a` e `b` em um arquivo chamado `dados.dat`, usamos o comando `save` com a sintaxe

```
-->save('dados.dat',a,b)

-->
```

O comando `save` cria o arquivo `dados.dat` no diretório de trabalho. O arquivo `dados.dat` é um arquivo binário. Para recuperar os valores de `a` e `b`, usamos o comando `load`, conforme mostrado no exemplo,

```
-->clear

-->a
  !--error     4
undefined variable : a


-->b
  !--error     4
undefined variable : b

-->load('dados.dat','a','b')

-->a, b
 a  =

    1.
 b  =

    2.

-->
```

Outras maneiras de manipular arquivos no Scilab podem ser verificadas em [2].

A função `unix_w` permite a comunicação do Scilab com a *shell* unix. Nesta função, as respostas são apresentadss na própria janela do Scilab.

```
-->pwd
 ans  =

 /home/paulo/teste

-->unix_w('ls')      // Mostrando o conteudo de /home/paulo/teste
Makefile
Relatorio.pdf
app
app.c
app.o
chromosome.c
chromosome.h
chromosome.o

-->unix_w('mkdir outro_dir')  // Criando o diretorio outro_dir
```

```
-->unix_w('ls')
Makefile
Relatorio.pdf
app
app.c
app.o
chromosome.c
chromosome.h
chromosome.o
outro_dir

-->chdir('outro_dir')          // Mudando de diretorio
 ans  =

    0.

-->pwd
 ans  =

 /home/paulo/teste/outro_dir

-->
```

A qualquer momento, o usuário pode obter informações sobre funções digitando o comando `help <nome-da-função>` no *prompt* do Scilab ou accessando a função através da opção Help. Por exemplo, vamos usar a linha de comando para obter informações sobre a função `sqrt`. Temos,

```
-->help sqrt
```

Surgirá uma outra janela, mostrada na Figura 2.2, com todas as informações sobre a função desejada.

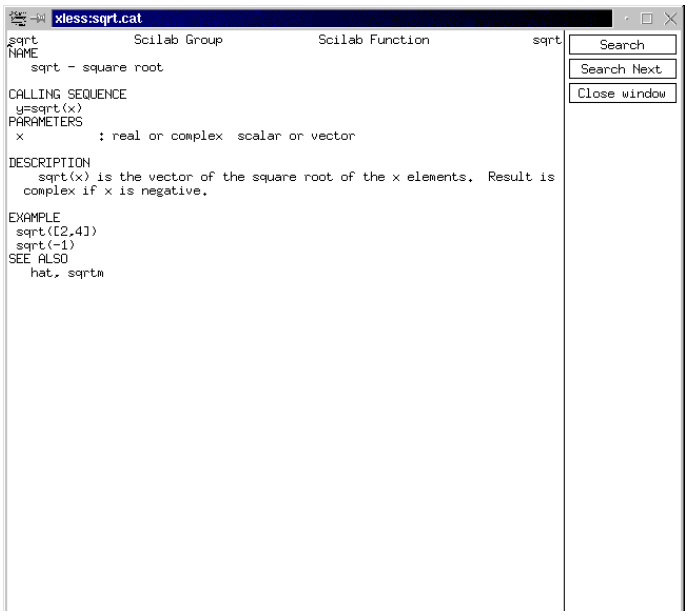

Figura 2.2: Tela de `help` sobre a função `sqrt`

A janela é fechada usando a opção Close window.

2.5.1 Comandos de Edição

Os comandos digitados a partir do *prompt* do Scilab podem ser editados. Na Tabela 2.1, mostramos algumas combinações de teclas que permitem esta edição.

Ctrl-p ou ↑	recupera o comando digitado anteriormente
Ctrl-n ou ↓	recupera o comando seguinte (se houver)
Ctrl-b ou ←	move o cursor um caracter para trás
Ctrl-f ou →	move o cursor um caracter para a frente
Delete ou ←	apaga o caracter anterior (tecla *Backspace*)
Ctrl-h	mesmo efeito da linha anterior
Ctrl-d	apaga o caracter sob o cursor
Ctrl-a	move o cursor para o início da linha
Ctrl-e	move o cursor para o final da linha
Ctrl-k	apaga da posição do cursor até o final da linha
Ctrl-u	cancela a linha
`!prev`	recupera a linha de comando que começa com `prev`

Tabela 2.1: Teclas de edição de linha de comando

2.5.2 Exemplos de Operações

No Scilab, o ponto-e-vírgula no final de um comando inibe a apresentação do resultado.

```
-->// O ponto-e-virgula suprime a apresentacao do resultado

-->A = 1;       // Atribuindo a A o valor 1

-->b = 2;       // Atribuindo a b o valor 2

-->A + b        // Soma de A e b
 ans  =

    3.

-->
```

As grandezas no Scilab também podem ser complexas. Para atribuir a `A` o valor complexo $5 + 2i$ e a `B` o valor complexo $-2 + i$, fazemos

```
-->A = 5 + 2 * %i   // Atribuindo a A o valor 5 + 2i
 A  =

    5. + 2.i

-->B = -2 + %i      // Atribuindo a B o valor -2 + i
 B  =

  - 2. + i

-->
```

As variáveis `A` e `B` podem ser multiplicadas, divididas, somadas e subtraídas, como mostramos a seguir.

```
-->A * B           // Multiplicacao
 ans  =

  - 12. + i

-->A / B           // Divisao
 ans  =

  - 1.6 - 1.8i

-->A + B           // Adicao
 ans  =

    3. + 3.i

-->A - B           // Subtracao
```

```
 ans  =

    7. + i

-->
```

É importante observar que a resposta ao uso da função `sqrt()` com argumento negativo inclui o número complexo `i = sqrt(-1)`.

```
-->sqrt(-2)          // Funcao raiz quadrada com argumento negativo
 ans  =

     1.4142136i

-->
```

É possível digitar vários comandos em uma mesma linha,

```
-->m = 1.5; b = 35; c = 24;       // Varios comandos em uma unica linha

-->
```

Também é possível digitar um único comando em várias linhas utilizando `...` ao final do comando,

```
-->A = 3 * m ^ 2 + ...            // Um comando em varias linhas
-->    4 * 5 + ...
-->    5 * 3
 A  =

    41.75

-->
```

Um vetor de índices possui a forma geral

```
Variavel = valor_inicial:incremento:valor_final
```

Por exemplo, através do comando `I=1:3` atribuímos os valores `1, 2, e 3` à variável `I`. Quando não especificado, `incremento` é igual a `1`. Assim,

```
-->I = 1:3                  // Definindo I como um vetor com 3 posicoes
 I  =

!   1.    2.    3. !

-->
```

O valor do `incremento` pode ser negativo,

```
-->j = 5:-1:1             // Definindo j como um vetor com 5 posicoes
 j  =

!   5.    4.    3.    2.    1. !

-->
```

No Scilab existe o conceito de ambientes. Muda-se de ambiente através do comando `pause`. Todas as variáveis definidas no primeiro ambiente são válidas no novo ambiente. Observar que a mudança de ambiente modifica a forma do *prompt*. Este passa a indicar o ambiente no qual estão sendo efetuados os comandos. O retorno ao ambiente anterior dá-se através da utilização dos comandos `resume` ou `return`. Com este tipo de retorno, perde-se as variáveis definidas no novo ambiente. A utilização de ambientes é importante para a realização de testes. No exemplo a seguir, atribuímos a `a` o valor 1.5 e, através do comando `pause`, mudamos de ambiente.

```
-->// Definindo a e mudando de ambiente

-->a = 1.5; pause

-1-> // Mudanca no prompt
```

Observar que houve uma mudança no formato do *prompt*. A variável `a`, definida no ambiente anterior, ainda é válida no novo ambiente, como podemos verificar através da seqüência de comandos,

```
-1->a
 a  =

    1.5

-1->
```

Vamos definir, no novo ambiente, a variável `b` igual a 2.5,

```
-1->// Definindo b no novo ambiente

-1->b = 2.5;

-1->// Mostrando a e b no novo ambiente

-1->a, b
 a  =

    1.5
 b  =

    2.5

-1->
```

O retorno ao ambiente anterior usando o comando `resume` faz com que a variável `b` fique indefinida,

```
-1->// Retornando ao ambiente anterior

-1->resume              // Pode ser usado o comando return

--> Mostrando a e b. Observar que b foi perdido

-->a, b
 a  =

    1.5
     !--error     4
undefined variable : b
```

O valor da variável `b` pode ser preservado no ambiente original através da seqüência de comandos,

```
-->a = 1.5          // Definindo a no ambiente ooriginal
 a  =

    1.5

-->pause            // Mudando de ambiente

-1->b = 1.5         // Definindo b no novo ambiente
 b  =

    1.5

-1->b = resume(b) // Enviando b para o ambiente original

-->a, b
 a  =

    1.5
 b  =

    1.5

-->
```

Neste Capítulo, apresentamos algumas das características do Scilab. Inicialmente, mostramos o ambiente gráfico com as suas opções. Depois, vimos as variáveis pré-definidas, alguns comandos que permitem a manipulação de arquivos e diretórios e exemplos de atribuição de valores. O conceito de ambiente também foi apresentado. No Capítulo 3, mostramos os tipos de dados que podem ser manipulados pelo Scilab.

Capítulo 3

Polinômios, Vetores, Matrizes e Listas

No Scilab, podemos trabalhar com vários tipos de dados. As constantes, as variáveis booleanas, os polinômios, as *strings* e as frações envolvendo polinômios são considerados dados escalares. Com estes objetos podemos definir matrizes. Os outros tipos de dados reconhecidos pelo Scilab são as listas e as listas com definição de tipo. O objetivo deste Capítulo é apresentar alguns exemplos de utilização de cada um desses tipos de dados.

3.1 Polinômios

Os polinômios são criados no Scilab através da utilização da função `poly`. Salientamos que polinômios de mesma variável podem ser somados, subtraídos, multiplicados e divididos entre si. Por exemplo, o polinômio $p = s^2 - 3s + 2$, que possui raízes 1 e 2, pode ser criado através do comando,

```
--> // Polinomio definido pelas suas raizes

-->p = poly([1 2], 's')
 p  =

               2
    2 - 3s + s
-->
```

Com a função `roots`, comprovamos que as raízes de p são, realmente, 1 e 2,

```
-->roots(p)
 ans  =

!   1. !
!   2. !

-->
```

Um polinômio também pode ser criado a partir da especificação de seus coeficientes. Por exemplo, o polinômio $q = 2s + 1$ é criado através do comando,

```
--> // Polinomio definido pelos seus coeficientes

-->q = poly([1 2], 's', 'coeff')
 q  =

    1 + 2s

-->roots(q)      // Obtendo as raizes do polinomio q
 ans  =

  - 0.5

-->
```

Para complementar o exemplo, os dois polinômios podem ser multiplicados, divididos, somandos ou subtraídos como mostra a seqüência de comandos,

```
-->p * q                // Multiplicacao
 ans  =

              2    3
    2 + s - 5s + 2s

-->p / q                // Divisao
 ans  =

              2
    2 - 3s + s
    ----------
      1 + 2s

-->p + q                // Adicao
 ans  =

             2
    3 - s + s

-->p - q                // Subtracao
 ans  =

              2
    1 - 5s + s

-->
```

3.2 Vetores

As grandezas vetoriais são criadas colocando-se seus componentes entre colchetes, `[ ]`. Os componentes de um vetor podem ser separados por espaço, vírgula ou por ponto-e-vírgula. É importante observar que elementos entre `[ ]`, separados por espaço ou por vírgula, dão origem a vetores linha. Quando separados por ponto-e-vírgula dão origem a vetores coluna.

```
-->v = [2, -3+%i, 7]                // Vetor linha
 v  =

!   2.  - 3. + i      7. !

-->v'                               // Vetor transposto
 ans  =

!   2.       !
! - 3. - i    !
!   7.       !

-->w = [2  3  4]                    // Vetor linha : outra forma
 w  =

!   2.    3.    4. !

-->z = [ 3; 5; 7]                   // Vetor coluna
 z  =

!   3. !
!   5. !
!   7. !

-->
```

Lembrar que vetores de mesma dimensão podem ser somados ou subtraídos. Podemos, também, realizar o produto escalar de um vetor linha por um vetor coluna. A multiplicação e a divisão de um vetor (linha ou coluna) por um escalar também podem ser facilmente realizadas.

Nos exemplos a seguir, mostramos outras maneiras de construir vetores,

```
-->v = 5: -0.5: 3            // Vetor com elementos decrementados

 v  =

!   5.    4.5    4.    3.5    3. !

-->m = ones(1:4)             // Vetor constituido de elementos iguais a 1
 m  =
```

```
!   1.    1.    1.    1. !

-->z = zeros(1:5)            // Vetor constituido de elementos iguais a 0
 z  =

!   0.    0.    0.    0.    0. !

-->
```

3.3 Matrizes

Vamos considerar as matrizes :

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 5 & -8 & 9 \end{bmatrix}$$

e

$$b = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Os elementos que constituem as linhas das matrizes são separados por espaços ou por vírgulas. A indicação de término de cada linha da matriz é feita com ponto-e-vírgula[1].

Nos exemplosa seguir, para fixar conceitos, a matriz `A` é digitada com os elementos de suas linhas separados por espaços enquanto a matriz `b` é digitada com os elementos de suas linhas separados por vírgula,

```
-->// Matriz A(2 x 3) - Elementos das linhas separados por espaco

-->A = [1 2 3; 5 -8 9]
 A  =

!   1.    2.    3. !
!   5.  - 8.    9. !

-->// Matriz b (2 x 3) - Elementos das linhas separados por virgulas

-->b = [1, 2, 3; 4, 5, 6]
 b  =

!   1.    2.    3. !
!   4.    5.    6. !

-->
```

Usamos a função `ones` para criar a matriz c_{2x3}, com todos os elementos iguais a 1,

[1] Uma matriz pode ser vista como um vetor coluna formado por um ou por vários vetores linha

```
-->c = ones(2,3)
 c  =

!   1.    1.    1. !
!   1.    1.    1. !

-->
```

Finalmente, a matriz `A` é multiplicada pela matriz `c` transposta[2].

```
-->A * c'
 ans  =

!   6.    6. !
!   6.    6. !

-->
```

Podemos criar matrizes a partir de elementos de outras matrizes,

```
-->// Definido as matrizes A, B e C

-->A = [1 2; 3 4];

-->B = [5 6; 7 8];

-->C = [9 10; 11 12];

-->// Definindo a matriz D

-->D = [A B C]
 D  =

!   1.    2.    5.    6.    9.    10. !
!   3.    4.    7.    8.    11.   12. !

-->// Definindo uma matriz E a partir dos elementos de D

-->E = matrix(D,3,4)
 E  =

!   1.    4.    6.    11. !
!   3.    5.    8.    10. !
!   2.    7.    9.    12. !

-->
```

[2]Lembrar que, para que duas matrizes possam ser multiplicadas, o número de colunas da primeira matriz deve ser igual ao número de linhas da segunda matriz.

Observar que a matriz `E`, com tres linhas e quatro colunas, é criada usando a função `matrix`. Esta função gera `E` a partir da organização dos elementos da matriz `D` por colunas.

3.4 Matrizes com Polinômios

Os elementos de uma matriz podem ser polinômios,

```
-->// Definindo um polinomio

-->x = poly(0, 'x'); p = 2 + 3 * x + x ^ 2
 p  =

              2
    2 + 3x + x

-->// Definindo uma matriz polinomial, M

-->M = [p, p-1; p+1, 2]
 M  =

!             2              2 !
!   2 + 3x + x     1 + 3x + x  !
!                              !
!             2                !
!   3 + 3x + x     2           !

-->// Obtendo o determinante de M

-->det(M)
 ans  =

                2    3   4
    1 - 6x - 11x - 6x - x

-->
```

A partir de uma matriz formada por elementos que são polinômios racionais,

```
-->// Definindo uma matriz F de polinomios racionais

-->s = poly(0, 's');

-->F = [ 1/s,     (s +1)/(s + 2); ...
-->      s/(s+3),       s^2     ]
 F  =

!   1         1 + s  !
```

```
!   -         -----  !
!   s         2 + s  !
!                    !
!              2     !
!     s        s     !
!   -----      -     !
!   3 + s      1     !

-->
```

podemos criar outra matriz apenas com o numerador das frações,

```
-->F('num')  // Pegando os numeradores
 ans  =

!   1     1 + s  !
!                !
!          2     !
!   s      s     !

-->
```

ou com seus denominadores,

```
-->F('den')  // Pegando os denominadores
 ans  =

!   s         2 + s  !
!                    !
!   3 + s     1      !

-->
```

3.5 Matrizes Simbólicas

As matrizes simbólicas são constituídas por elementos compostos por *strings* de caracteres. Elas são criadas da mesma maneira que as matrizes com elementos numéricos. As *strings* são escritas entre apóstrofos ou entre aspas.

```
-->// Matriz de strings

-->A = ['x'  'y'; 'z'  'w+v']
 A  =

!x  y    !
!        !
!z  w+v  !
```

```
-->// Triangularizacao de A

-->At = trianfml(A)
 At  =

!z  w+v          !
!                !
!0  z*y-x*(w+v)  !

-->// Atribuindo valores

-->x=1;y=2;z=3;w=4;v=5;

// Valor das matrizes simbolicas

-->evstr(A)
 ans  =

!   1.    2. !
!   3.    9. !

-->evstr(At)
 ans  =

!   3.    9. !
!   0.  - 3. !

-->
```

3.6 Matrizes Booleanas

Matrizes booleanas são matrizes construídas com as constantes `%t` (`t é` *true*, `verdadeiro`) e `%f` (`f é` *false*, `falso`). Alguns exemplos de construção matrizes booleanas,

```
-->// Matriz booleana A

-->A = [%t, %f, %t, %f, %f, %f]
 A  =

! T F T F F F !

-->// Matriz booleana B

-->B = [%t, %f, %t, %f, %t, %t]
 B  =
```

```
! T F T F T T !

-->
```

Podemos realizar operações lógicas com as matrizes definidas anteriormente,

```
-->// A ou B

-->A|B
 ans  =

! T F T F T T !

-->// A e B

-->A & B
 ans  =

! T F T F F F !

-->
```

3.7 Operações com Matrizes

A Tabela 3.1 apresenta a sintaxe das operações com matrizes disponíveis no ambiente Scilab.

SÍMBOLO	OPERAÇÃO
`[ ]`	definição de matriz, concatenação
`;`	separador de linhas
`( )`	atribuição do tipo `m = a(k)`
`( )`	atribuição do tipo `a(k) = m`
`'`	transposta
`+`	adição
`-`	subtração
`*`	multiplicação
`\`	divisão à esquerda
`/`	divisão à direita
`^`	exponenciação
`.*`	multiplicação elemento-a-elemento
`.\`	divisão à esquerda elemento-a-elemento
`./`	divisão à direita elemento-a-elemento
`.^`	exponenciação elemento-a-elemento
`.*.`	produto de Konecker
`./.`	divisão de Kronecker à direita
`.\.`	divisão de Kronecker à esquerda

Tabela 3.1: Sintaxe de comandos usados em operações matriciais

3.8 Acesso a Elementos de Matrizes

O acesso a elementos de uma matriz pode ser realizado através da indicação explícita dos índices do elemento a ser acessado, através do símbolos `:` ou `$` ou através de operações booleanas. Vamos considerar a matriz `A` com duas linhas e três colunas, $\mathtt{A}_{2x3}$,

```
-->// Definindo uma matriz A

-->A = [1 2 3; 4 5 6]
 A  =

!   1.    2.    3. !
!   4.    5.    6. !

-->
```

Observar que a não utilização do ponto-e-vírgula no final do comando permite que o resultado do mesmo seja imediatamente apresentado. O acesso a elementos individuais desta matriz é feito da maneira convencional. Por exemplo, para acessar o elemento a_{12} da matriz `A` usamos o comando `A(2,1)`,

```
-->// Acessando o elemento A(1,2)
```

```
-->A(1,2)
 ans  =

    2.

-->
```

O comando `M = A([1 2], 2)`, permite construir uma matriz, `M`, composta pelo primeiro e segundo elementos, indicados pelo vetor `[1 2]`, da segunda coluna da matriz `A`,

```
-->M = A([1 2], 2)
 M  =

!   2. !
!   5. !

-->
```

O Scilab implementa formas compactas que permitem acessar elementos de uma matriz. Vamos considerar a matriz `A` do exemplo anterior. No contexto que se segue, o símbolo `:` significa "todos os elementos". Assim, o comando `A(:,3)`, faz com que sejam acessados todos os elementos, indicado pelo símbolo `:`, da terceira coluna da matriz `A`,

```
->// Todos os elementos da terceira coluna

-->A(:, 3)
 ans  =

!   3. !
!   6. !

-->
```

O comando `A(:, 3:-1:1)` permite formar uma matriz constituída por todos os elementos das colunas três, dois e um da matriz A. Lembrar que `3:-1:2` é idêntico a `[3 2 1]`.

```
-->// Todos os elementos da terceira, segunda e primeira colunas de A

-->A(:, 3:-1:1)
 ans  =

!   3.    2.    1. !
!   6.    5.    4. !

-->A(:, [3 2 1])       // Forma equivalente
 ans  =

!   3.    2.    1. !
```

```
!   6.    5.    4. !

-->
```

Vamos considerar a utilização do símbolo $ para acessar elementos da matriz `A`. Neste contexto, o símbolo $ significa "número total de". Usando o comando `A(1:2, $-1)`, acessamos o primeiro e o segundo elementos, indicados por `1:2`, da segunda coluna, indicado por `$-1`, da matriz `A`. Lembrar que a matriz `A` possui duas linhas e três colunas. Com o comando, `A($:-1:1, 2)`, estamos acessando o segundo e o primeiro, nessa ordem, elementos da segunda coluna da matriz `A`. Escrever `$:-1:1` é equivalente, neste caso, a escrever `2:-1:1` já que a matriz `A` possui duas linhas. Com o comando, `A($)`, acessamos o último elemento de `A`.

```
-->// Primeiro e segundo elementos da segunda coluna de A

-->A(1:2, $-1)
 ans  =

!   2. !
!   5. !

-->// Segundo e primeiro elementos da segunda coluna de A

-->A($:-1:1, 2)
 ans  =

!   5. !
!   2. !

-->// Acesso ao ultimo elemento de A

-->A($)
 ans  =

    6.

-->
```

Os elementos de uma matriz são armazenados por coluna. Assim, o primeiro elemento da matriz `A` pode ser acessado através do comando `A(1)` e o quinto elemento da matriz `A` pode ser acessado através do comando `A(5)`,

```
-->// Primeiro  elemento de A

-->A(1)
 ans  =

    1.
```

```
-->// Quinto elemento de A

-->A(5)
 ans  =

    3.

-->// Todos os elementos armazenados por coluna

-->A(:)
 ans  =

!   1. !
!   4. !
!   2. !
!   5. !
!   3. !
!   6. !

--> // Mesmo efeito do comando anterior

-->A([1 2 3 4 5 6])
 ans  =

!   1. !
!   4. !
!   2. !
!   5. !
!   3. !
!   6. !

-->
```

Podemos usar variáveis booleanas para acessar elementos de uma matriz. Com o comando `A([%t %f %f %t])`, acessamos o primeiro e o quarto elementos da matriz A, indicados por `%t`, não querendo o segundo e terceiro elementos, indicados por `%f`.

```
-->// Acesso ao primeiro e quarto elementos

-->A([%t %f %f %t])
 ans  =

!   1. !
!   5. !

-->
```

Com o comando `A(%t, [2 3])`, acessamos os primeiros elementos das segunda e terceira colunas.

```
-->// Acessando os primeiros elementos da colunas 2 e 3

--> A(%t, [2 3])
 ans  =

!   2.    3. !

-->
```

É possível, caso seja necessário, alterar os valores de elementos de uma matriz. Considerando a matriz A, podemos mudar o valor do seu elemento $A(2,1)$ através do comando de atribuição `A(1,2) = 10`,

```
-->// Atribuir a A(1,2) o valor 10

-->A(1,2) = 10
 A  =

!   1.    10.    3. !
!   4.    5.     6. !

-->
```

Depois, atribuímos os valores `[-1; -2]` aos primeiro e segundo elementos da segunda coluna da matriz A,

```
-->// A(1,2) = -1 e A(2,2) = -2

-->A([1 2], 2) = [-1; -2]
 A  =

!   1.  - 1.    3. !
!   4.  - 2.    6. !

-->
```

Finalmente, modificamos os elementos $A(1,1)$ e $A(1,2)$ da matriz A.

```
-->// A(1,1) = 8 e A(1,2) = 5

-->A(:,1) = [8;5]
 A  =

!   8.  - 1.    3. !
!   5.  - 2.    6. !

-->
```

O Scilab permite a criação e manipulação de matrizes simbólicas. Vamos considerar uma matriz B_{1x2} constituída por elementos simbólicos,

```
-->// Matriz simbolica

-->B = [ 1/%s, (%s + 1)/(%s - 1)]
 B  =

!   1     1 + s  !
!   -     -----  !
!   s   - 1 + s  !

-->
```

Os elementos de uma matriz simbólicas são acessados utilizando os mesmos comandos para acessar elementos de uma matriz numérica. Nos dois comandos seguintes, apresentamos exemplos de acesso aos elementos de B,

```
-->// Acessos a elementos de B

-->B(1,1)
 ans  =

    1
    -
    s

-->B(1, $)
 ans  =

    1 + s
    -----
  - 1 + s

-->
```

Podemos, também, atribuir valores simbólicos a elementos de uma matriz, Considerando a matriz `A = [1 -1 3; 5 -2 6]`, temos,

```
-->A(1,1) = %s       // Atribuicao do valor simbolico s ao elemento A(1,1)
 A  =

!   s   - 1     3  !
!                  !
!   5   - 2     6  !

-->A($) = %s + 1    // Atribuindo s + 1 ao ultimo elemento de A
```

```
 A  =

!   s   - 1     3      !
!                      !
!   5   - 2     1 + s  !

-->
```

3.9 Listas

Uma lista é uma coleção de objetos não necessariamente do mesmo tipo. Uma lista simples é definida pela função `list`. Esta função tem a forma geral

$$\text{list}(a_1,\, a_2,\ldots,a_n)$$

onde os a_i são os elementos da lista.

Vamos criar uma lista simples, que chamamos de `L`, composta por três elementos : o elemento `1`, associado a `L(1)`, o elemento `w`, associado a `L(2)` e uma matriz 2x2 composta de 1, associada a `L(3)`,

```
-->// Uma lista simples com 3 elementos

-->L = list(1, 'w', ones(2,2))
 L  =


       L(1)

    1.

       L(2)

 w

       L(3)

!   1.    1. !
!   1.    1. !
```

É importante observar que a indexação de elementos de uma lista, no Scilab, inicia-se por 1.

Vamos transformar o elemento `L(2)` da lista do exemplo anterior em uma lista cujo primeiro elemento, `L(2)(1)`, é `w` e cujo segundo elemento, `L(2)(2)`, é uma matriz 2x2 de números aleatórios,

```
-->// Transformando o elemento L(2) em uma lista
```

```
-->L(2) = list('w', rand(2,2))
 L  =


       L(1)

    1.

       L(2)


        L(2)(1)

 w

        L(2)(2)

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

       L(3)

!   1.    1. !
!   1.    1. !

-->
```

Mostramos, também, o comando necessário para acessar o elemento (1,2) do segundo elemento de L(2),

```
-->L(2)(2)(2,1)
 ans  =

    0.7560439

-->
```

As lista tipadas são um outro tipo de dado aceito pelo Scilab. As listas tipadas são definidas através da função `tlist`. A função `tlist` possui, obrigatoriamente, como primeiro argumento um *string* ou um vetor de *strings* e os demais argumentos são os elementos da lista. A seguir, alguns exemplos de manipulação de listas tipadas.

```
-->// Definicao de uma lista tipada

-->L = tlist(['Carro'; 'Cidade'; 'Valores'], 'Natal', [2,3])
 L  =
```

```
       L(1)

!Carro    !
!         !
!Cidade   !
!         !
!Valores  !

       L(2)

 Natal

       L(3)

!   2.    3. !

-->// Acessando elementos

-->L('Cidade')
 ans  =

 Natal

-->L('Valores')
 ans  =

!   2.    3. !

-->L(1)(3)
 ans  =

 Valores

-->
```

Observar que os índices de uma lista tipada podem ser *strings* definidas no primeiro argumento da função `tlist()`.

Neste Capítulo, apresentamos os tipos de dados que podem ser manipulados pelo Scilab. Diversos exemplos foram mostrados utilizando polinômios, vetores, matrizes e listas. Os exemplos foram apresentados a partir do *prompt* do Scilab. No próximo Capítulo, vamos mostrar com podemos desenvolver programas na linguagem Scilab.

Capítulo 4

Programação

Uma característica importante do Scilab é a possibilidade do usuário criar seus próprios programas. Estes programas são portáveis e, portanto, podem ser executados em qualquer plataforma que possua o ambiente Scilab.

Apesar de simples, a linguagem do Scilab disponibiliza a maioria das estruturas das linguagens convencionais de programação. A diferença principal é que, na programação Scilab, não há a necessidade da declaração prévia do tipo das variáveis.

O Scilab é um interpretador. Os programas escritos na linguagem Scilab são, normalmente, executados em um tempo maior que os mesmos programas escritos em linguagens compiláveis. Isso acontece, principalmente, com programas utilizados em simulações e otimizações. Nesses casos, pode ser conveniente escrever o código responsável pela lentidão em uma linguagem convencional (C ou FORTRAN) e rodar esse código dentro do ambiente Scilab. No Apêndice B, mostramos os procedimentos necessários à ligação de códigos escritos em C com programas escritos em Scilab. Deve ser enfatizado, entretanto, que a vantagem na utilização dos programas Scilab advém da facilidade de prototipação e da disponibilidade de uma poderosa biblioteca de funções gráficas. Como sempre, cabe ao usuário encontrar a sua solução de compromisso.

Os comandos que formam um programa podem ser escritos diretamente no *prompt* do Scilab. Podem, também, ser escritos em um arquivo texto que, posteriormente, é carregado no ambiente do Scilab.

4.1 Comandos para Iterações

Existem dois comandos que permitem a realização de iterações, *loops*, no Scilab : o *loop* implementado com o comando `for` e o *loop* implementado com o comando `while`.

4.1.1 O *Loop* `for`

O comando `for` tem a forma geral :

```
for variavel = vetor_linha
        instrucao_1
        instrucao_2
         ...  ...
        instrucao_n
 end
```

A forma acima é equivalente à forma

```
--> for variavel=vetor_linha,instrucao_1,instrucao_2, ... ...,instrucao_n,end
```

A primeira forma é utilizada quando os programas são escritos em um arquivo (a vírgula é substituída pela mudança de linha) enquanto a segunda é utilizada quando programamos diretamente no ambiente Scilab.

No *loop* `for`, o comportamento das iterações é baseado no conteúdo de um vetor linha.

No exemplo a seguir, vamos considerar que a variável `k` do comando `for` assuma os valores estabelecidos pelo vetor linha `v = [2 3 4 5 6]`. O número de iterações será igual ao número de componentes do vetor `v`. Temos, portanto, cinco iterações. Na primeira iteração, o valor de `k` será igual a `v(1)`, que é 2, e na última iteração o valor de `k` será igual a `v(5)`, que vale 6.

```
-->v = [2 3 4 5 6]
 v  =

!   2.    3.    4.    5.    6. !

-->y = 0; for k=v, y = y + k, end
 y  =

    2.
 y  =

    5.
 y  =

    9.
 y  =

    14.
 y  =

    20.

-->
```

A variável `k` do exemplo mostrado pode assumir os valores de um vetor linha escrito de forma compacta,

```
-->y = 0; for k=2:6, y = y + k, end
 y  =

    2.
 y  =

    5.
```

```
 y  =

    9.
 y  =

    14.
 y  =

    20.

-->
```

A variável do comando `for` também pode ser uma lista. Neste caso, a variável assume os valores dos elementos da lista,

```
-->L = list(1, [1 2; 3 4], 'teste')
 L  =


       L(1)

    1.

       L(2)

!   1.    2. !
!   3.    4. !

       L(3)

 teste

-->for k=L, disp(k), end

    1.

!   1.    2. !
!   3.    4. !

 teste

-->
```

4.1.2 O *Loop* `while`

O comando `while` tem a forma geral,

```
while condicao
      instrucao_1
      instrucao_2
       ...  ...
      instrucao_n
 end
```

A forma acima é equivalente à forma

```
--> while condicao, instrucao_1, instrucao_2, ... ..., instrucao_n, end
```

Como no caso anterior, a primeira forma é utilizada quando os programas são escritos em um arquivo (a vírgula é substituída pela mudança de linha) enquanto a segunda é utilizada quando programamos diretamente no ambiente Scilab.

O *loop* baseado no `while` realiza uma seqüência de instruções enquanto uma determinada condição estiver sendo satisfeita. Na Tabela 4.1, apresentamos os operadores que permitem fazer comparações entre valores de objetos no Scilab.

Operadores	Significado
== ou =	igual a
<	menor do que
>	maior do que
<=	menor ou igual a
>=	maior ou igual a
<> ou ~=	diferente

Tabela 4.1: Operadores condicionais

A seguir, apresentamos um exemplo da utilização do *loop* baseado no comando `while`,

```
-->x = 1; while x < 14, x=2*x, end
 x  =

    2.
 x  =

    4.
 x  =

    8.
 x  =

    16.

-->
```

4.2 Comandos Condicionais

O Scilab implementa dois tipos de comandos condicionais : `if-then-else` e `select-case`.

4.2.1 Comando `if-then-else`

O comando `if-then-else` tem a forma geral,

```
if condicao_1 then
     sequencia_de_instrucoes_1
elseif condicao_2
     sequencia_de_instrucoes_2
   ... ... ... ... ... ...
elseif condicao_n
     sequencia_de_instrucoes_n
else
     sequencia_de_instrucoes_n+1
end
```

A forma acima é equivalente à forma

```
--> if condicao_1 then, sequencia_de_instrucoes_1, elseif condicao_2, ...
--> sequencia_de_instrucoes_2, ...
--> elseif condicao_n, sequencia_de_instrucoes_n, ...
--> else, sequencia_de_instrucoes_n+1, end
```

A primeira forma é utilizada quando os programas são escritos em um arquivo (a vírgula é substituída pela mudança de linha) enquanto a segunda é utilizada quando programamos diretamente no ambiente Scilab. Observar que a continuação dos comandos foi feita usando `...` .

O `if-then-else` avalia uma expressão. Se esta expressão for verdadeira, *true*, executa a instrução ou instruções subseqüentes. Se for falsa, *false*, executa a instrução ou instruções após o `else` ou o `elseif`, conforme o caso. Alguns exemplos da utilização do condicional `if-then-else`,

```
  -->x = 1          // Inicializando
 x  =

    1.

-->if x > 0 then, y = -x, else, y=x, end
 y  =

  - 1.

-->x = -1
 x  =
```

```
  - 1.

-->if x > 0 then, y = -x, else, y=x, end
 y  =

  - 1.

-->
```

4.2.2 Comando `select-case`

O condicional `select-case` tem a forma geral,

```
select variavel_de_teste
case expressao_1
        sequencia_de_instrucoes_1
case expressao_2
        sequencia_de_instrucoes_2
  ... ... ... ... ... ... ... ...
case expressao_n
        sequencia_de_instrucoes_n
else
        sequencia_de_instrucoes_n+1
end
```

A forma acima é equivalente à forma

```
--> select variavel_de_teste, case expressao_1, sequencia_de_instrucoes_1, ...
--> case expressao_2, sequencia_de_instrucoes_2, ..., case expressao_n, ...
--> sequencia_de_instrucoes_n, else, sequencia_de_instrucoes_n+1, end
```

Como citamos anteriormente, a primeira forma é utilizada quando os programas são escritos em um arquivo (a vírgula é substituída pela mudança de linha) enquanto a segunda é utilizada quando programamos diretamente no ambiente Scilab. Como no caso anterior, a continuação dos comandos foi feita usando `...` .

O condicional `select-case` compara o valor de uma variável de teste com as várias expressões dos `case`. Serão executadas as instruções que possuirem o valor da expressão do `case` igual ao valor da variável de teste. Um exemplo do condicional `select-case`,

```
-->x = -1         // Inicializacao
 x  =

  - 1.

-->select x, case 1, y = x+5, case -1, y = sqrt(x), end
 y  =
```

```
    i

-->
```

4.3 Definindo Funções

É possível definir funções dentro do próprio ambiente Scilab. Entretanto, é mais conveniente utilizar um editor de textos para criar, fora do ambiente Scilab, um arquivo contendo a função. É importante observar que o nome desse arquivo não é, necessariamente, o nome que deve ser dado à função. No Scilab, funções são sinônimos de programas.

4.3.1 Estrutura das Funções

Uma função obedece ao formato,

```
function [y1, ..., yn] = foo(x1, ..., xm)
instrucao_1
instrucao_2
    ...
instrucao_p
```

onde `foo` é o nome da função, `xi`, i=1,...,m, são os seus argumentos de entrada, `yj`, j=1,...,n, são os seus argumentos de saída e `instrucao_i`, i=1,...,p, representam a seqüência de instruções que deve ser executadas pela função.

Como primeiro exemplo [12], vamos desenvolver um programa para resolver a equação diferencial ordinária,

$$\frac{dy}{dx} = (x - y)/2$$

com condição inicial $y(0) = 1$, utilizando o método de Runge-Kutta de $4^{\underline{a}}$ ordem. Vamos considerar o intervalo de integração igual a $[0, 3]$ e o passo de integração $h = 1/8$. A solução obtida por Runge-Kutta será comparada com valores da solução exata que é $y(x) = 3e^{-x/2} + x - 2$.

O método Runge-Kutta de $4^{\underline{a}}$ ordem é representado pelas equações,

$$y_{k+1} = y_k + \frac{h}{6}(f_1 + 2f_2 + 2f_3 + f_4)$$

com os coeficientes f_i definidos por :

$$\begin{aligned} f_1 &= f(x_k, y_k) \\ f_2 &= f(x_k + \frac{h}{2}, y_k + \frac{h}{2}f_1) \\ f_3 &= f(x_k + \frac{h}{2}, y_k + \frac{h}{2}f_2) \\ f_4 &= f(x_k + h, y_k + hf_3) \end{aligned}$$

e pode ser implementado através do algoritmo :

Algorithm 1: Método de Runge-Kutta de $4^{\underline{a}}$ ordem

Entrada $[a, b]$, h e $y0$
Fazer $x0 = a$
Fazer $n = (b - a)/h$
for k = 0 to n **do**
  Calcular f_1, f_2, f_3 e f_4
  Calcular $y_{k+1} = y_k + \frac{h}{6}(f_1 + 2f_2 + 2f_3 + f_4)$
  Fazer $x_{k+1} = x_k + h$
end for
Apresentar valores de x_k e y_k

Inicialmente, utilizamos um editor de textos (por exemplo, `vi`, `joe`) para, fora do ambiente Scilab, criar o programa. O programa criado é mostrado em Código 1.

```
1  function [XY] =  rk4(a, b, h, y0)
2  // Resolucao de equacoes diferenciais ordinarias por Runge-Kutta 4a. ordem
3  // Entrada : [a,b] - intervalo de integracao
4  //               h - passo da integracao
5  //              y0 - condicao inicial em x0
6  X(1) = a
7  Y(1) = y0
8  Exato(1) = f2(X(1), Y(1))
9  n = (b-a)/h
10 for k=1:n
11         xk = X(k)
12         yk = Y(k)
13         hf1 = h * f(xk, yk)
14         hf2 = h * f(xk + h/2, yk + hf1/2)
15         hf3 = h * f(xk + h/2, yk + hf2/2)
16         hf4 = h * f(xk + h, yk + hf3)
17         Y(k+1) = Y(k) + (hf1 + 2*hf2 + 2*hf3 + hf4)/6
18         X(k+1) = X(k) + h
19         Exato(k+1) = f2(X(k+1), Y(k+1))
20 end
21 XY = [X Y Exato]
```

Código 1: Programa principal, Runge-Kutta

Como podemos observar, o programa chama a função `f(x,y)`, linhas 13 a 16, e a função com a solução exata da equação diferencial dada[1], linhas 8 e 19 de Código 1. A função `f(x,y)` é mostrada em Código 2,

[1] É óbvio que nem sempre essa solução está disponível

```
1  function [fxy] = f(x,y)
2  // funcao exemplo
3  fxy = (x - y)/2
```

Código 2: A função `f(x,y)`

e a solução exata é mostrada em Código 3,

```
1  function [fexato] = f2(x,y)
2  // funcao solucao
3  fexato = 3 * exp(-x/2) + x - 2
```

Código 3: A solução exata da equação diferencial

Vamos considerar que o programa e as funções estejam em arquivos localizados no diretório onde o Scilab é lançado. Assim sendo, o comando

```
-->getf('fdexy.sci')     // arquivo com a funcao f(x,y) = (x - y)/2

-->
```

carrega a função `f(x,y)` no ambiente Scilab. Depois, com o comando

```
-->getf('fsolucao.sci')  // arquivo com a funcao solucao = 3exp(-x/2)+x-2

-->
```

a função com a solução exata é carregada. Finalmente, o comando `getf('rkutta4.sci')` carrega o arquivo que define a função `rk4`[2] e o comando `rk4(0, 3, 1/8, 1)` executa o programa. Os resultados obtidos são :

```
-->getf('rkutta4.sci')   // programa metodo de Runge-Kutta 4a. ordem

-->rk4(0, 3, 1/8, 1)     // executando o programa
 ans  =

!   0.       1.           1.        !
!   0.125    0.9432392    0.9432392 !
!   0.25     0.8974908    0.8974907 !
!   0.375    0.8620874    0.8620874 !
!   0.5      0.8364024    0.8364023 !
!   0.625    0.8198470    0.8198469 !
!   0.75     0.8118679    0.8118678 !
!   0.875    0.8119457    0.8119456 !
```

[2]Esta ordem é irrelevante. Observe que os nomes das funções e os nomes dos arquivos que as contém são, intencionalmente, diferentes.

```
!   1.        0.8195921    0.8195920 !
!   1.125     0.8343486    0.8343485 !
!   1.25      0.8557844    0.8557843 !
!   1.375     0.8834949    0.8834947 !
!   1.5       0.9170998    0.9170997 !
!   1.625     0.9562421    0.9562419 !
!   1.75      1.0005862    1.0005861 !
!   1.875     1.049817     1.0498169 !
!   2.        1.1036385    1.1036383 !
!   2.125     1.1617724    1.1617723 !
!   2.25      1.2239575    1.2239574 !
!   2.375     1.2899485    1.2899483 !
!   2.5       1.3595145    1.3595144 !
!   2.625     1.4324392    1.432439  !
!   2.75      1.5085189    1.5085188 !
!   2.875     1.5875626    1.5875625 !
!   3.        1.6693906    1.6693905 !

-->
```

Na primeira coluna são mostrados os valores de x, na segunda coluna são mostrados os valores da solução aproximada, y, e na terceira coluna são mostrados os valores da solução exata, $3e^{-x/2} + x - 2$.

Como segundo exemplo de programação, [12], vamos resolver um sistema triangular superior de equações lineares, Este exemplo requer a leitura de um arquivo externo contendo dados.

Vamos considerar o sistema triangular superior,

$$\begin{array}{rcrcrcrcrcl} 4x_1 & - & x_2 & + & 2x_3 & + & 2x_4 & - & x_5 & = & 4 \\ & - & 2x_2 & + & 6x_3 & + & 2x_4 & + & 7x_5 & = & 0 \\ & & & & x_3 & - & x_4 & - & 2x_5 & = & 3 \\ & & & & & - & 2x_4 & - & x_5 & = & 10 \\ & & & & & & & & 3x_5 & = & 6 \end{array}$$

Para resolver estes tipos de sistemas, usamos a matriz dos coeficientes aumentada, definida por

$$[A \mid \mathbf{b}] = \left(\begin{array}{ccccc|c} 4 & -1 & 2 & 2 & -1 & 4 \\ 0 & -2 & 6 & 2 & 7 & 0 \\ 0 & 0 & 1 & -1 & -2 & 3 \\ 0 & 0 & 0 & -2 & -1 & 10 \\ 0 & 0 & 0 & 0 & 6 & 6 \end{array}\right)$$

e o processo de substituição reversa, indicado peloo algoritmo :

Algorithm 2: Substituição Reversa

$x_n = \dfrac{b_n}{a_{n,n}^{(n)}}$
for $r = n-1$ até 1 **do**
  $soma = 0$
  for $j = r+1$ até n **do**
    $soma = soma + a_{r,j} * x_j$
  end for
  $x_r = \dfrac{b(r) - soma}{a_{r,r}}$
end for

A matriz aumentada, neste caso, é armazenada em um arquivo ASCII puro, que chamamos de arquivo1. O conteúdo de arquivo1, digitado com um editor qualuqer, pode ser visto no ambiente Linux usando o comando `cat`,

```
paulo:~/metodos/funcoes/aula3$ cat arquivo1

4 -1  2  2 -1  4
0 -2  6  2  7  0
0  0  1 -1 -2  3
0  0  0 -2 -1 10
0  0  0  0  3  6
```

Este arquivo é lido pelo Scilab através do comando :

```
-->Ab = read('arquivo1', 5, 6)
 Ab  =

!   4.  - 1.    2.    2.  - 1.    4.  !
!   0.  - 2.    6.    2.    7.    0.  !
!   0.    0.    1.  - 1.  - 2.    3.  !
!   0.    0.    0.  - 2.  - 1.    10. !
!   0.    0.    0.    0.    3.    6.  !

-->
```

onde `Ab` é a variável que contém a matriz aumentada. O comando `read` lê as cinco linhas e as seis colunas de dados do arquivo1 que está armazenado no diretório de trabalho.

Caso seja necessário, a matriz dos coeficientes, A, e o vetor dos termos independentes, $\mathbf{b}$, podem ser recuperados da matriz `Ab` através da seqüência de comandos :

```
-->// Numero de linhas, nl, numero de colunas, nc, de Ab

-->[nl nc] = size(Ab)
 nc  =
```

```
    6.
 nl  =

    5.

-->A = Ab(:,1:nc-1)   // Matriz dos coeficientes
 A  =

!   4.  - 1.    2.    2.  - 1. !
!   0.  - 2.    6.    2.    7. !
!   0.    0.    1.  - 1.  - 2. !
!   0.    0.    0.  - 2.  - 1. !
!   0.    0.    0.    0.    3. !

-->b = Ab(:,nc)       // Vetor dos termos independentes
 b  =

!   4.  !
!   0.  !
!   3.  !
!   10. !
!   6.  !

-->
```

O programa para resolver o sistema linear é mostrado no Codigo 4,

```
1  function X = subst(Tsup)
2  // Entrada : matriz triangular superior, Tsup
3  // Saida : Vetor solucao X
4  [nl, nc] = size(Tsup);
5  n = nc-1;
6  A = Tsup(:,1:n);  // Matriz A
7  b = Tsup(:,nc)    // Vetor  b
8  X = zeros(n,1);
9  X(n) = b(n)/A(n,n);
10 for r = n-1:-1:1,
11         soma = 0,
12         for j = r+1:n,
13                 soma = soma + A(r,j) * X(j);
14         end,
15 X(r) = (b(r) - soma)/A(r,r);
16 end
```

Código 4: Programa para resolver um sistema triangular

Usando a matriz aumentada `Ab` como entrada para este programa, obtemos como vetor solução,

```
-->subst(Ab)
 ans  =

!   5. !
!   4. !
!   1. !
! - 6. !
!   2. !

-->
```

4.3.2 Variáveis Globais e Variáveis Locais

As variáveis globais são válidas no ambiente do Scilab enquanto as variáveis locais são válidas apenas no escopo de uma função. Para exemplificar os conceitos, vamos considerar a função `f`,

```
function [y1, y2] = f(x1, x2)
y1 = x1 + x2,
y2 = x1 - x2
```

Observe que $y1$ e $y2$ são as variáveis de saída enquanto $x1$ e $x2$ são as variáveis de entrada da função `f`. Vamos considerar alguns exemplos de chamadas desta função.

Inicialmente, a função é carregada e chamada com argumentos $x1 = 1$ e $x2 = 3$ tendo seus parâmetros de retorno associados às váriáveis $m1$ e $m2$. Observe que o Scilab retorna primeiro o último valor calculado. Observe que $y1$ e $y2$, apesar de terem sido calculados dentro da função (definição local), não são definidas no ambiente do Scilab.

```
-->getf('f1.sci')      // Carregando a funcao

-->[m1, m2] = f(1,3)   // Retorno associado as variaveis [m1, m2]
 m2  =

  - 2.
 m1  =

    4.

--> // Provocando erro : y1 e y2 nao sao globais

-->y1
  !--error     4
undefined variable : y1


-->y2
```

```
  !--error     4
undefined variable : y2

-->
```

Continuando com o exemplo, a função é chamada sem a associação explícita de variáveis de retorno. Este caso é como se a função `f` tivesse sido chamada com a associação de apenas uma variável de retorno, uma chamada do tipo $z = f(1,3)$.

```
-->f(1,3)       // Chamada equivalente a z = f(1,3)
 ans  =

    4.

-->
```

O exemplo continua e um erro é provocado quando a função é chamada com apenas um argumento. Logo em seguida, o argumento é definido no ambiente (definição global) e a função é, novamente, chamada com apenas um argumento sem que haja a ocorrência de erro.

```
-->f(1)    // Erro por indefinicao de argumento
 !--error     4
undefined variable : x2
at line       2 of function f                         called by :
f(1)

-->x2 = 3   // Definindo x2 no ambiente (global)
 x2  =

    3.

-->f(1)     // Chamando a funcao com apenas um argumento
 ans  =

    4.

-->
```

Como vismos, a chamada de uma função sem que todos os seus argumentos de entrada tenham sido previamente definidos ocasiona erro. Os argumentos de entrada devem ser definidos explícitamente na chamada ou através de definições via variáveis globais. Considerando a função anterior, teremos, como casos interessante (mas não aconselháveis !), os exemplos,

```
-->x1 = 2; x2 = 3;  // Definindo x1 e x2

-->f()              // Chamando f sem nenhum argumento
 ans  =
```

```
    5.

-->[m1, m2] = f()  // Retorno associado as variaveis [m1, m2]
 m2  =

  - 1.
 m1  =

    5.

-->
```

4.3.3 Comandos Especiais

O Scilab possui alguns comandos especiais que são, exclusivamente, utilizados por funcões :

- `argn` - retorna o número de argumentos de entrada e de saída da função;
- `error` - usado para suspender a execução de uma função, apresentar uma mensagem de erro e retornar para o ambiente anterior quando um erro for detectado;
- `warning` e `pause` - suspendem, temporariamente, a execução de uma função;
- `break` - força o final de um *loop*;
- `return` ou `resume` - utilizado para passar as variáveis locais do ambiente da função para o ambiente que chamou a função.

Alguns dos comandos especiais apresentados anteriormente são utilizados na função,

```
function [z] = foo(x, y)
[out, in] = argn(0);

if x == 0 then,
   error('Divisao por zero');
end,
slope = y/x;
pause,
z = sqrt(slope);
s = resume(slope);
```

Vamos chamar esta função com argumento `x = 0`. O valor `x = 0` ocasiona um erro, apresentado ao usuário através do comando `error`, com a conseqüente interrupção das operações. Há o retorno ao *prompt* do Scilab. Estas operações são apresentadas no exemplo,

```
-->getf('f3.sci')       // Carregando a funcao

-->z = foo(0, 1)        // Provocando um erro
```

```
 !--error 10000
Divisao por zero
at line       5 of function foo                    called by :
z = foo(0, 1)

-->
```

Na segunda chamada, mostrada a seguir, desta vez com os argumentos corretos, a função suspende a operação após o cálculo de `slope`. Neste ponto, o usuário pode verificar valores calculados pela função até o ponto em que houve a interrupção. O *prompt* `-1->`, causado pelo `pause`, indica a mudança de ambiente. O controle pode ser retornado à função através do comando `return`. As operações da função podem ser encerradas usando os comandos `quit` ou `abort`. Após digitar `resume`, a função calcula o valor de `z` e disponibiliza a variável `s`, local à função, para o ambiente que a chamou.

```
-->// Mudando de ambiente

-->z = foo(2,1)

-1->resume
 z  =

    0.7071068

-->s
 s  =

    0.5

-->
```

Neste Capítulo, apresentamos alguns programas desenvolvidos na linguagem Scilab. No Capítulo 5, vamos mostrar comandos que podem ser utilizados para traçar gráficos no ambiente Scilab.

Capítulo 5

Gráficos

Apresentamos alguns comandos que podem ser utilizados para traçar gráficos bi-dimensionais e tri-dimensionais usando o Scilab. Todos os comandos disponíveis na biblioteca gráfica do Scilab podem ser acessados através da seqüência `Help`/`Graphic Library`.

5.1 Gráficos Bi-dimensionais

Gráficos bi-dimensionais simples são conseguidos através da utilização da função `plot(x,y)`. Esta função permite traçar o gráfico da variável `y` em função da variável `x`.

A forma geral do comando é :

```
plot(x, y, [xcap, ycap, caption])
```

onde `xcap, ycap e caption` são, respectivamente, os nomes dados aos eixos `x`, `y` e ao gráfico. Estes três parâmetros são opcionais (estão entre colchetes `[ ]`). Um exemplo de utilização deste comando,

```
-->// Definindo abcissas e ordenadas

-->t = (0: 0.05: 1);

-->y = sin(2 * %pi * t);

-->// Comando para tracar o grafico

-->plot(t,y,'tempo', 'f(t)=sin(2*%pi*t)', 'Seno')
```

Após a execução desta seqüência de comandos, o gráfico da função `sin(2πt)` é apresentado em uma janela gráfica do Scilab. Esta janela é mostrada na Figura 5.1.

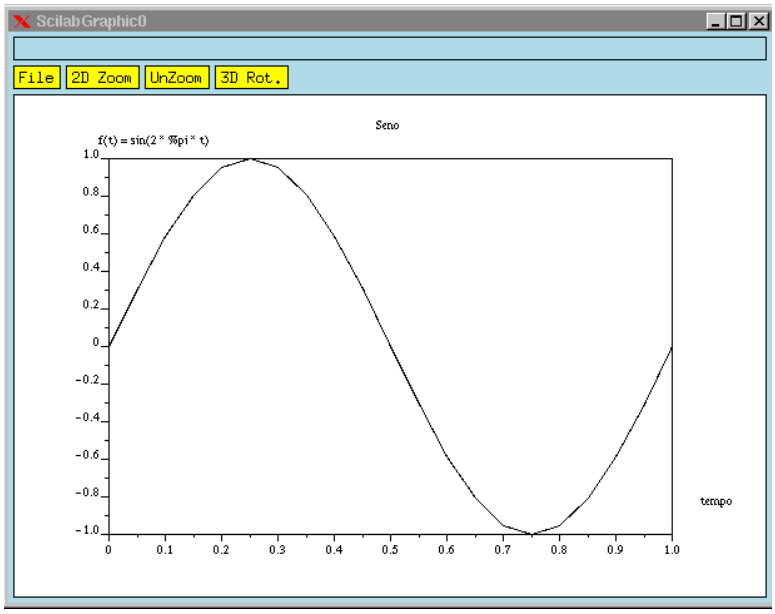

Figura 5.1: Janela gráfica do Scilab

Observar a existência, na Figura 5.1, de um menu horizontal com quatro opções : `File`, `2D Zoom`, `UnZoom` e `3D Rot.`. Utilizando o mouse para colocar o cursor sobre cada uma das opções, verificamos que :

- A opção `File` possui sete opções que permitem manipular o gráfico gerado : `Clear`, `Select`, `Print`,`Export`, `Save`, `Load` e `Close`.
- A opção `2D Zoom` - permite a ampliação de uma parte do gráfico. Escolhendo esta opção e delimitando uma área, a parte do gráfico dentro da área escolhida será expandida. Esta opção não tem efeito em gráficos tri-dimensionais.
- A opção `unZoom` - desfaz as manipulações realizadas através da opção `2D Zoom`
- A opção `3D Rot` - permite efetuar rotações em gráficos tri-dimensionais. Esta opção não produz nenhum efeito em gráficos bi-dimensionais.

O gráfico resultante pode, então, ser armazenado em um arquivo. Neste caso, a aparência do gráfico é a mostrado na Figura 5.2.

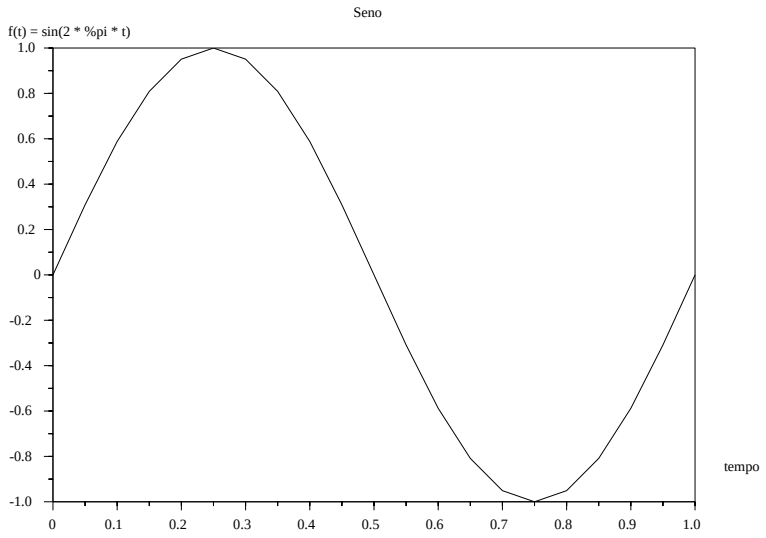

Figura 5.2: Gráfico de sen(2πt)

A largura da linha, e outros elementos de um gráfico, são controlados por parâmetros globais. Estes parâmetros definem um contexto no qual o gráfico está inserido. Outros parâmetros dos gráficos são controlados através de argumentos dos próprios comandos usados para traça-los. O comando `xset` sem nenhum argumento, `xset()`, apresenta um conjunto de opções, chamadas de "Contexto gráfico da janela gráfica 0", (*Graphic context of graphic window 0*), que permitem alterar parâmetros através da utilização do mouse.

O comando genérico para traçar gráficos bi-dimensionais é o comando :

```
plot2di(str, x, y, [style, strf, leg, rect, nax])
```

Diferentes valores de `i`, índice da função `plot2di()`, determinam um tipo diferente de gráfico. Na Tabela 5.1, apresentamos os valores permitidos para `i` e o correspondente tipo de gráfico associado.

Valor do Índice	Tipo de Gráfico
sem o índice	gráfico linear por partes
`i = 1`	anterior com escala logarítmica
`i = 2`	constante por partes
`i = 3`	gráfico de barras verticais
`i = 4`	gráfico com setas

Tabela 5.1: Valores de `i` para o comando `plot2di()`

O parâmetro `str` é uma *string* de formato `abc`. Nesta *string*, `a` pode assumir os valores `e`, `o` ou `g` enquanto `b` e `c` podem assumir os valores `l` ou `n`. Os significados de cada um destes valores são :

- `a = e` - o valor de `x` não é usado.
- `a = o` - os valores de `x` são os mesmos para todas as curvas.
- `a = g` - geral.
- `b = l` - uma escala logarítmica deve ser usada no eixo `X`.
- `c = l` - uma escala logarítmica deve ser usada no eixo `Y`.

Os parâmetros `x, y` representam duas matrizes de mesmo tamanho, `[nl, nc]` onde `nc` é o número de curvas e `nl` é o número de pontos de cada curva. Para uma curva simples, este vetor pode ser um vetor linha ou um vetor coluna. Os comandos `plot2d(t', cos(t)')` e `plot2d(t, cos(t))` produzem o mesmo gráfico.

Nos exemplos a seguir, mostramos a utilização do comando `plot2di()`.

```
// Demo dos comandos plot2di()

// Funcao plot2d()
t = (1: 0.1: 8)'; xset("font", 2, 3);
xsetech([0.,0.,0.5,0.5], [-1,1,-1,1]);
plot2d([t t], [1.5+2*sin(t) 2+cos(t)]);
xtitle('plot2d');
titlepage('Linear por partes');

// Funcao plot2d1()
xsetech([0.5,0.,0.5,0.5], [-1,1,-1,1]);
plot2d1('oll', t, [1.5+2*sin(t) 2+cos(t)]);
xtitle('plot2d1');
titlepage('Escalas Logaritmica');

// Funcao plot2d2()
xsetech([0.,0.5,0.5,0.5], [-1,1,-1,1]);
plot2d2('onn', t, [1.5+2*sin(t) 2+cos(t)]);
xtitle('plot2d2');
titlepage('Constante por partes');

// Funcao plot2d3()
xsetech([0.5,0.5,0.5,0.5], [-1,1,-1,1]);
plot2d3('onn', t, [1.5+2*sin(t) 2+cos(t)]);
xtitle('plot2d3');
titlepage('Barras Verticais');

xset('default');
```

Os gráficos resultantes são mostrados na Figura 5.3.

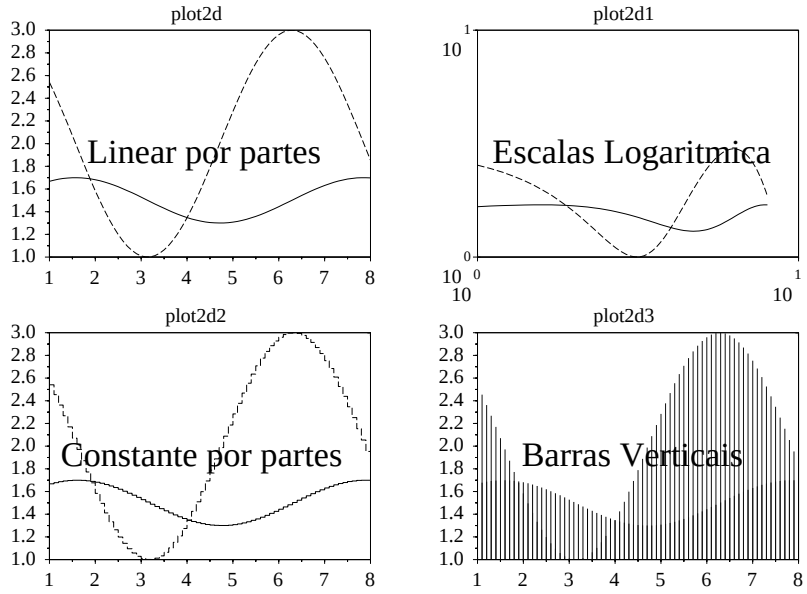

Figura 5.3: Gráficos com `plot2di()`

Constinuando a descrição dos argumentos da função `plot2di()`, o parâmetro `style` define o estilo a ser usado pela curva. Este parâmetro é um vetor real de tamanho `(1, nc)`.

O parâmetro `strf` é um *string* de comprimento três, `xyz`, correspondendo a :

- `x = 1` - mostra rótulos
- `y = 1` - o argumento `rect` é utilizado para especificar os contornos do gráfico. É um vetor com a especificação : `rect = [xmin, xmax, ymin, ymax]`.
- `y = 2` - os contornos do gráfico são calculados.
- `y = 0` - indica o contorno atual.
- `z = 1` - um eixo é traçado e o número de marcas (“tics’) neste eixo pode ser especificado pelo argumento `nax`.
- `z = 2` - o gráfico é circundado por um quadrado.

O parâmetro `leg` é uma *string* de nomes para diferentes curvas. Esta *string* é composta de campos separados por `@`. As *strings* são apresentadas abaixo dos gráficos. A seqüência de comandos mostra um exemplo da utilização do parâmetro `leg`.

```
// Identificacao de curvas

x = -%pi:0.3:%pi;
y1 = sin(x); y2 = cos(x); y3 = x;
X=[x;x;x]; Y=[y1;y2;y3];
plot2d1("gnn", X', Y', [-1 -2 -3]', "111", "nome1@nome2@nome3", ...
        [-3, -3, 3, 2], [2, 20, 5, 5]);
```

As curvas identificadas são apresentadas na Figura 5.4.

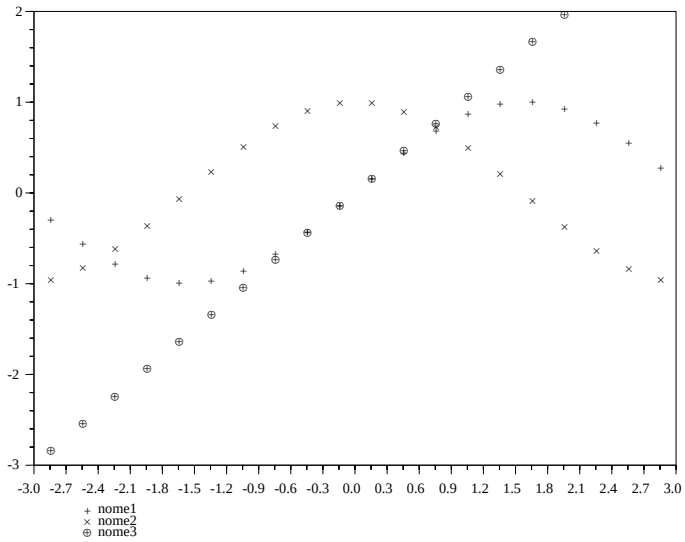

Figura 5.4: Curvas identificadas

O parâmetro `rect` é um vetor que especifica os contornos do gráfico. O parâmetro `rect` tem a forma `rect = [xmin,xmax,ymin,ymax]`.

O parâmetro `nax` é um vetor `[nx,Nx,ny,Ny]` onde `nx` e `ny` indicam o número de sub-gráficos no eixo `x` ou `y` e `Nx` e `Ny` indicam o número de graduações no eixo `x` ou `y`.

É importante ressaltar que um demo é apresentado se usarmos a função sem nenhum argumento,

```
--> plot2d1()

-->
```

5.1.1 Outros Comandos

Existem alguns comandos que podem ser utilizados para melhorar a apresentação de um gráfico. Dentre eles, ressaltamos :

- `xgrid` - coloca uma grade em um gráfico bi-dimensional.
- `xtitle` - coloca um nome acima e nos eixos de um gráfico bi-dimensional.
- `titlepage` - coloca um título no meio de um gráfico.

5.1.2 Gráficos 2D Especiais

O Scilab dispõe de alguns comandos que permitem traçar gráficos bi-dimensionais especiais. Por exemplo,

- `champ` - permite traçar o gráfico de um campo vetorial em R^2.
- `fchamp` - permite traçar o gráfico de um campo vetorial em R^2 definido por uma função.
- `fplot2d` - permite traçar o gráfico de uma curva 2D definida por uma função.
- `grayplot` - permite traçar o gráfico de uma superfície em escala cinza. A superfície deve ser definida por uma matriz de valores.
- `fgrayplot` - permite traçar o gráfico de uma superfície em escala cinza. A superfície deve ser definida por uma função.
- `errbar` - permite traçar um gráfico com barras de erro.

A seqüência de comandos a seguir represente um programa que permite traçar o gráfico de um campo vetorial utilizando o comando `champ`.

```
// Grafico de um campo vetorial
x=[-1:0.1:1];y=x;u=ones(x);
fx=x.*.u';fy=u.*.y';
champ(x,y,fx,fy);
xset("font",2,3);
xtitle(['Campo Vetorial';'(com o comando champ)']);
xset('default');
```

O gráfico correspondente é apresentado na Figura 5.5.

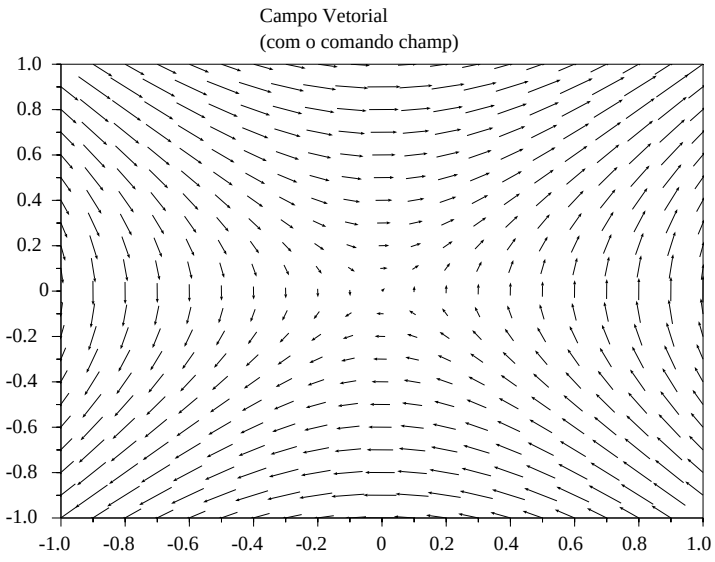

Figura 5.5: Gráfico de um campo vetorial

Outros comandos utilizados para traçar gráficos da área de controle de processos :

- `bode` - permite traçar o gráfico de módulo e fase da resposta em freqüência de um sistema linear.
- `gainplot` - permite traçar o gráfico do módulo da resposta em freqüência de um sistema linear.
- `nyquist` - permite traçar o gráfico da parte imaginária versus parte real da resposta em freqüência de um sistema linear.
- `m_cicle` - gráfico M-círculo usado com o gráfico de Nyquist.
- `chart` - permite traçar a diagrama de Nichols.
- `black` - permite traçar o diagrama de Black para um sistema linear.
- `evans` - permite traçar o o lugar das raízes pelo método de Evans.
- `plzr` - permite traçar o diagrama de polos e zeros para um sistema linear.

5.2 Gráficos Tri-dimensionais

Algumas funções permitem traçar gráficos tri-dimensionais :

- `plot3d` - permite traçar gráficos de uma matriz de pontos.
- `plot3d1` - permite traçar gráficos de uma matriz de pontos em escala cinza.
- `fplot3d` - permite traçar gráficos de funções do tipo `z = f(x,y)`.
- `fplot3d1` - permite traçar gráficos de funções do tipo `z = f(x,y)` em escala cinza.

5.2.1 Gráficos 3D Especiais

As seguintes funções permitem traçar gráficos tri-dimensionais especiais :

- `param3d` - permite traçar curvas paramétricas.
- `contour` - permite traçar curvas de nível para uma função 3D descrita por uma matriz.
- `grayplot10` - permite traçar gráficos com níveis de cinza em 2D.
- `fcontour10` - permite traçar curvas de nível para uma função 3D descrita por uma função.
- `hist3d` - permite traçar histogramas 3D.
- `secto3d` - conversão de uma descrição de superfície para dados compatíveis com a função `plot3d`.
- `eval3d` - avalia uma função em uma grade regular.

Ressaltamos que a sintaxe desses comandos pode ser verificada usando o `help` do Scilab.

Apêndice A

Instalação do Scilab

O objetivo deste Apêndice é apresentar os procedimentos necessários à instalação do Scilab a partir de seu código fonte. A instalação é feita no ambiente Linux (Slackware 7.1, kernel 2.2.16). Apesar de ser dado um exemplo de instalação em uma plataforma específica, ele é válido para qualquer ambiente operacional que disponha dos requisitos mínimos apresentados a seguir. Os procedimentos necessários à instalação de distribuições binárias, pré-compiladas, podem ser encontrados na *homepage* do Scilab[1].

A.1 Instalação no Linux Slackware - Código Fonte

O código fonte para a versão 2.6 do software Scilab é disponibilizado através do arquivo `scilab-2.6-src.tar.gz`. Para instalar o software a partir do código fonte, devemos garantir que:

- O X-Window esteja instalado (X11R4, X11R5 ou X11R6), e
- O sistema disponha de compiladores C e FORTRAN (cc e g77).

Tendo garantido os requisitos mínimos, escolhemos o diretório no qual o software será instalado. Geralmente, utilizamos o diretório `/usr/local`. Copiamos, então, a distribuição fonte do Scilab para este diretório. Descompactamos o arquivo através do comando `tar -zxvf scilab-2.6-src.tar.gz`. Será criado um diretório `/usr/local/scilab-2.6` onde estarão colocados todos os arquivos que compõem o software. O diretório `/usr/local/scilab-2.6` é chamado de diretório principal do Scilab, referenciado pela variável de ambiente `SCIDIR`. Estes procedimentos devem ser realizados como `root`.

No diretório `SCIDIR=/usr/local/scilab-2.6` estão localizados, entre outros, os seguintes arquivos :

- `scilab.star` - arquivo de inicialização do Scilab. As instruções deste arquivo são executadas quando o Scilab é ativado. O usuário pode ter um arquivo de inicialização, `.scilab`, no seu próprio diretório,
- `license.txt` - arquivo contendo informações relativas ao uso do software,

[1] http://www-rocq.inria.fr/scilab/

- `configure` - arquivo de preparação para a instalação. Este arquivo modificará outros arquivos do Scilab , os Makefile, de acordo com a configuração do Sistema Operacional no qual o Scilab será instalado.

Em `SCIDIR` são criados os seguintes diretórios :

- `bin/` - onde estão localizados os arquivos executáveis. Nas distribuições Unix/Linux o script `scilab` aciona o executável `scilex`. Neste diretório estão, também, localizados programas para manipulação de arquivos Postscript e LaTeX gerados pelo Scilab.
- `demos/` - onde estão localizados os arquivos de demonstração.
- `examples/` - onde estão localizados arquivos com exemplos de como ligar o Scilab com programas externos.
- `tests/` - onde estão localizados arquivos para a realização de testes da instalação do Scilab.
- `doc/` - onde estão localizados arquivos com a documentação do Scilab no formato LaTeX, arquivos `.tex`. Para gerar documentos em formato adequado para impressão, devemos processar os arquivos `.tex`. Por exemplo, para gerar o documento *Introduction to Scilab*, no diretório `SCIDIR/doc/intro` fazer `make`. Será gerada a documentação em formato .dvi. Para gerar a documentação em formato `.ps`, fazer `dvips intro.ps`. Para gerar a documentação em formato `.pdf` a partir da `.ps` gerada no passo anterior, fazer `ps2pdf intro.ps`. A documentação no formato html não faz parte da distribuição. Podemos, entretanto, obtê-la nas `URLs` mencionadas anteriormente.
- `macros/` - onde estão localizadas as funções do Scilab. Este diretório contém diversos sub-diretórios correpondendo, cada um, a um tópico específico. Por exemplo, no diretório `comm` estão localizadas as funções que permitem trabalhar com aspectos relacionados com a área de comunicações . Cada sub-diretório contém o código fonte das funções (arquivos `.sci`).
- `man/` - onde estão localizados diversos sub-diretórios contendo arquivos man (help *on-line)* e ASCII sobre o Scilab.
- `pvm3/` - implementação 3.4 do PVM (*Parallel Vitual Machine System*) para o Scilab.
- `geci/` - onde estão localizadas as rotinas que permitem o gerenciamento de execuções remotas de programas possibilitando a troca de mensagens entre eles.
- `tcl/` - implementação tcl/tk para o Scilab.
- `libs/` - onde estão localizados os arquivos objeto que permitem a ligação de Scilab com outros sistemas.
- `routines/` - onde estão localizadas as rotinas numéricas em FORTRAN e C, divididas em sub-diretórios.
- `util/` - onde estão localizadas rotinas e arquivos para gerenciamento do Scilab
- `maple/` - onde estão localizados arquivos que permitem a ligação do Scilab com o Maple.

- `imp/` - onde estão localizados programas que permitem a manipulação de arquivos Postscript.
- `intersi/` - onde estão localizados os arquivos que permitem a construção das interfaces necessárias para adicionar novas primitivas escritas em FORTRAN ou C ao Scilab.
- `scripts/` - onde estão localizados os fontes dos arquivos *script*.
- `xmetanet` - onde estão localizados os arquivos para apresentação de grafos.

A instalação do software propriamente dita é bastante simples. No diretório `SCIDIR` digita-se `.\configure -with-tk` e, depois, `make all`. O usuário deve, também, acrescentar à variável `PATH` o caminho `/usr/local/scilab-2.6/bin` para que o Scilab possa ser executado de qualquer diretório[2]. Mais detalhes sobre opções de configuração podem ser encontrados no arquivo `/usr/local/scilab-2.6/README`

Para executar o programa, basta digitar `scilab` no ambiente gráfico ou `scilab -nw` no ambiente texto.

[2]No Linux-Slackware, editar o arquivo `/etc/profile` adicionando o caminho necessário à variável de ambiente `PATH`

Apêndice B

Ligação do Scilab com Programas em C

Como colocamos no Capítulo 4, os programas escritos na linguagem Scilab são interpretados. Estes programas, principalmente os que realizam cálculos numéricos utilizados em simulações ou otimizações, podem ser lentos em comparação com os programas compilados escritos em linguagens convencionais. Uma maneira de acelerar a execução desses programas, sem perder a flexibilidade disponibilizada pelo Scilab, é escrever o código lento em uma linguagem convencional e utilizar este código dentro do ambiente Scilab.

O Scilab permite que rotinas ou funções escritos em FORTRAN, Maple ou C sejam utilizados dentro de seu ambiente. Neste Apêndice, apresentamos os procedimentos necessários à ligação de programas Scilab com funções escritas na linguagem C. Os procedimentos para as outras linguagens podem ser encontrados em [1]

Uma função escrita na linguagem C pode ser ligada ao Scilab de três maneiras distintas :

- através do comando `link`, em um processo chamado de ligação dinâmica;
- através de programas de interface, os chamados *gateways*, escritos pelo usuário ou gerados por `intersi`, ou
- através da adição de uma nova função ao código do Scilab.

Apenas a primeira maneira será analisada. Os demais casos podem ser verificados em [1].

B.1 A Ligação Dinâmica

O comando

```
--> link('foo.o','foo','c')

-->
```

liga o arquivo objeto `foo.o`, escrito na linguagem C, indicado pelo terceiro argumento, `c`, ao Scilab. O segundo argumento de `link`, `foo`, é o nome da função a ser executada. Um arquivo objeto pode conter várias funções. O nome de cada uma delas, se necessário, deve ser indicado como segundo argumento de `link` na forma de um vetor de *strings*, `['prog1', 'prog2']`.

Como exemplo, vamos re-escrever em C o programa Scilab que implementa o método de Runge-Kutta de 4ª ordem apresentado no Capítulo 4. Algumas observações devem ser feitas :

- O programa transforma-se em uma função;
- Além do intervalo de integração, `[a,b]`, do passo de integração, `h` e da condição inicial em y, `y0`, que são os argumentos de entrada da função, os vetores `x` e `y` são explicitados na chamada da função e são os seus argumentos de retorno ou de saída;
- As variáveis de entrada da função principal, `rk4`, são passadas como apontadores;
- A ordem dos argumentos na chamada da função é relevante;
- A função a ser integrada faz parte do arquivo que contém a função principal;

O programa, transformado em uma função C, e a função com a equação a ser integrada, são mostrados em Codigo 5.

```
1  /*
2          Exemplo de utilizacao do comando link.
3          Resolucao de equacoes diferenciais ordinarias por Runge-Kutta
4          de 4a. ordem.
5
6          Entrada : [a,b] - intervalo de integracao
7                       h - passo da integracao
8                      y0 - condicao inicial em x0
9  */
10
11 double rk4(x, y, a, b, h, y0)
12 double x[], y[], *a, *b, *h, *y0;
13 {
14         int n, k;
15         double hf1, hf2, hf3, hf4;
16         double f();
17
18         n = (*b - *a) / (*h);
19
20         x[0] = *a;
21         y[0] = *y0;
22
23         for (k = 0; k < n; ++k)
24         {
25             hf1 = (*h) * f(x[k], y[k]);
26             hf2 = (*h) * f(x[k] + (*h)/2, y[k] + hf1/2);
27             hf3 = (*h) * f(x[k] + (*h)/2, y[k] + hf2/2);
28             hf4 = (*h) * f(x[k] + (*h), y[k] + hf3);
29
30             y[k+1] = y[k] + (hf1 + 2*hf2 + 2*hf3 + hf4)/6;
31             x[k+1] = x[k] + (*h);
32         }
33 }
34
35 /*
36         Funcao de integracao
37 */
38
39 double f(x,y)
40 double x, y;
41 {
42         return( (x - y)/2 );
43 }
```

Código 5: Função Runge-Kutta escrita em C

Usando o compilador `gcc`, geramos o código objeto `runge.o` para o programa `runge.c` usando

```
paulo@pmotta:~$ gcc -c runge.c -o runge.o
```

Em seguida, o código objeto é ligado ao Scilab através do comando `link`,

```
-->link('runge.o','rk4','c')
linking files runge.o  to create a shared executable
shared archive loaded
Linking rk4
Link done
 ans  =

    0.

-->
```

Observar que `rk4`, segundo argumento de `link` é o nome da rotina que resolve o problema.

Para acessar a função `rk4`, devemos inicializar seus parâmetros de entrada. A inicialização desses parâmetros é,

```
->a = 0           // Valor inicial do intervalo

    0.

-->b = 3          // Valor final do intervalo

    3.

-->h = 1/8        // Passo da integracao

    0.125

-->y0 = 1        // Valor da condicao inicial em y

    1.

-->
```

Em seguida, usamos a função `call` para rodar a função `rk4`,

```
-->[X,Y]=call('rk4',a,3,'d',b,4,'d',h,5,'d',y0,6,'d','out', ...
-->[25,1],1,'d',[25,1],2,'d');
```

O primeiro argumento da função `call` é o nome da função `rk4`. Cada argumento de entrada da função `rk4` deve ser acompanhado da posição em que ele ocupa na lista de argumentos da função chamada, `rk4`, e o seu tipo de dado. Assim, considerando

```
double rk4(x, y, a, b, h, y0),
```

`a` é o terceiro argumento na lista dos parâmetros de `rk4` e seu tipo de dado é `double`, indicado por `'d'`. Do mesmo modo, `b` é o quarto argumento, tipo de dado[1] `double`, e assim por diante.

[1]Os outros possíveis tipos de dados são real, `r` e inteiro, `i`

Os argumentos de saída, especificados após `'out'`, são vetores do tipo `double`, indicado por `'d'`, com vinte e cinco linhas e uma coluna, indicados por `[25,1]`, ocupando as posições 1 e 2 da lista de argumentos da função `rk4`[2].

Finalmente, os valores de retorno de `rk4` são associados às variáveis `[X, Y]` do ambiente Scilab. A resposta, então, é

```
-->[X Y]
 ans  =

!   0.       1.        !
!   0.125    0.9432392 !
!   0.25     0.8974908 !
!   0.375    0.8620874 !
!   0.5      0.8364024 !
!   0.625    0.8198470 !
!   0.75     0.8118679 !
!   0.875    0.8119457 !
!   1.       0.8195921 !
!   1.125    0.8343486 !
!   1.25     0.8557844 !
!   1.375    0.8834949 !
!   1.5      0.9170998 !
!   1.625    0.9562421 !
!   1.75     1.0005862 !
!   1.875    1.049817  !
!   2.       1.1036385 !
!   2.125    1.1617724 !
!   2.25     1.2239575 !
!   2.375    1.2899485 !
!   2.5      1.3595145 !
!   2.625    1.4324392 !
!   2.75     1.5085189 !
!   2.875    1.5875626 !
!   3.       1.6693906 !

-->
```

Observe que a função `call` foi chamada com um ponto-e-vírgula no final, suprimindo a apresentação imediata dos resultados. Só depois, os vetores `X` e `Y` foram mostrados. Neste caso, o ponto-e-vírgula evitou que primeiro fosse mostrado o vetor `Y` e, depois, o vetor `X`.

[2]Observar que são $n = (b - a)/h$ iterações mais o valor da condição inicial, $(x0, y0)$

Referências Bibliográficas

[1] Scilab Group, *Introduction to Scilab - User's Guide*. Esta referência, e as outras escritas pelo Scilab Group, podem ser obtidas em http://www-rocq.inria.fr/scilab

[2] Jesus Olivan Palacios, *An Introduction to the Treatment of Neurophysiological Signals using Scilab*, disponível em http://www.neurotraces.com em julho de 2001.

[3] Scilab Group, *Scicos: a Dynamic System Builder and Simulator*

[4] Scilab Group, *LMITOOL: Linear Matrix Inequalities Optimization Toolbox*

[5] Scilab Group, *Intersi: Automatically Interfacing C and FORTRAN Subroutines*

[6] Scilab Group, *Signal Processing Toolbox*

[7] Scilab Group, *Communication Toolbox*

[8] Scilab Group, *Scilab Internal Structure*

[9] Bruno Piçon, *Une introduction à Scilab, version 0.996*, obtida em http://www-rocq.inria.fr/scilab/books em maio de 2001.

[10] L.E. van Dijk, C.L. Spiel, *Scilab Bag of Tricks, sci-BOT*, obtida em http://www.hammersmith-consulting.com/scilab/sci-bot/ em maio de 2001.

[11] Scilab//, http://www,ens-lyon.fr/~desprez/FILES/RESEARCH/SOFT/SCILAB em junho de 2001.

[12] Paulo S. Motta Pires, *Métodos Computacionais - Notas de Aula, Versão 0.2*, obtida em http://www.dca.ufrn.br/~pmotta em junho de 2001