Universidade Federal de Minas Gerais

Departamento de Ciência da Computação do ICEx

Fundamentos de SCILAB

edição 2010.08

Frederico F. Campos, filho

agosto de 2010

Prefácio

O SCILAB é um software para computação científica e visualização, gratuito, com código fonte aberto e interface para as linguagens FORTRAN e C. Ele permite a solução de problemas numéricos em uma fração do tempo que seria necessário para escrever um programa em uma linguagem como FORTRAN, Pascal ou C, devido às suas centenas de funções matemáticas.

O SCILAB é desenvolvido pelo **INRIA** (*Institut National de Recherche en Informatique et en Automatique*) e **ENPC** (*École Nationale des Ponts et Chaussées*) da França. Em www.scilab.org estão disponíveis várias informações, documentação e instruções de como baixar o programa. As versões do SCILAB estão disponíveis na forma pré-compilada para diversas plataformas: Linux, Windows, HP-UX e Mac OSX. Como o código fonte também está disponível, ele pode ser compilado para uso em um computador específico.

O objetivo deste texto é apresentar o SCILAB como uma linguagem de programação dotada de funções não disponíveis nas linguagens convencionais. Por isto, este material pode ser utilizado em disciplinas, tais como, Programação de Computadores, Cálculo Numérico, Análise Numérica, Álgebra Linear e quaisquer outras dos cursos de Engenharia e das áreas de Ciências Exatas. Também, são exploradas algumas características próprias que mostram por que o SCILAB é uma poderosa ferramenta de apoio ao aprendizado e utilização da Computação Científica.

O Capítulo 1 Lógica de programação apresenta uma revisão de lógica de programação explicando as estruturas básicas: sequencial, condicional e de repetição, além das estruturas de dados fundamentais. No Capítulo 2 Ambiente de programação é descrito o ambiente de programação do SCILAB mostrando a janela de comandos e como obter informações de comandos durante a sessão. O Capítulo 3 Estruturas de dados apresenta as estruturas de dados suportadas pelo SCILAB, como constantes, vetores, matrizes, hipermatrizes, polinômios e listas. No Capítulo 4 Expressões são mostradas as expressões aritméticas, lógicas e literais, bem como, o modo de executá-las. O Capítulo 5 Gráficos é dedicado à elaboração de gráficos bi e tridimensionais. No Capítulo 6 Linguagem de programação, o SCILAB é visto como uma linguagem de programação, sendo mostrado como escrever programas e funções, estruturas condicionais e de repetição e depuração de programas. O Capítulo 7 Comandos de entrada e saída apresenta os formatos de exibição, gravação e leitura de variáveis do espaço de trabalho, além de leitura de gravação de dados em arquivos. Finalmente, no Capítulo 8 Computação científica são apresentadas funções do SCILAB para resolver problemas de Computação Científica.
Sugestões para aprimorar o presente texto, bem como, para efetuar correções serão bem-vindas pelo *e-mail*: ffcampos@dcc.ufmg.br.

Belo Horizonte, agosto de 2010.

Frederico F. Campos, filho
DCC.ICEx.UFMG

Sumário

1 Lógica de programação **1**

1.1 Estrutura básica de um algoritmo 1

1.2 Comandos de entrada e saída 2

1.3 Estrutura sequencial 3

1.4 Variáveis e comentários 3

1.5 Expressões 4

1.5.1 Comando de atribuição 4

1.5.2 Expressões aritméticas 4

1.5.3 Expressões lógicas 7

1.5.4 Ordem geral de precedência 8

1.5.5 Expressões literais 9

1.6 Estruturas condicionais 10

1.6.1 Estrutura condicional simples 10

1.6.2 Estrutura condicional composta 11

1.7 Estruturas de repetição 12

1.7.1 Número indefinido de repetições 12

1.7.2 Número definido de repetições 14
1.8 Falha no algoritmo 17
1.9 Modularização 17
1.10 Estruturas de dados 18
1.10.1 Vetores 18
1.10.2 Matrizes 21
1.10.3 Hipermatrizes 23
1.11 Exercícios 24

2 Ambiente de programação **27**

2.1 Janela de comando 27
2.1.1 Espaço de trabalho 29
2.1.2 Diretório corrente 30
2.1.3 Comando `unix` 31
2.2 Comandos de auxílio ao usuário 31
2.2.1 Comando `help` 31
2.2.2 Comando `apropos` 32
2.2.3 Menu de barras 33
2.3 Exercícios 33

3 Estruturas de dados **35**

3.1 Constantes 35
3.1.1 Numéricas 35

3.1.2 Lógicas 36
3.1.3 Literais 36
3.2 Variáveis 36
3.2.1 Regras para nomes de variáveis 36
3.2.2 Comando de atribuição 37
3.2.3 Variáveis especiais 37
3.3 Vetores 38
3.4 Matrizes 42
3.4.1 Construção e manipulação de matrizes 42
3.4.2 Funções matriciais básicas 45
3.4.3 Funções para manipulação de matrizes 48
3.4.4 Matrizes elementares 54
3.5 Hipermatrizes 57
3.6 Polinômios 59
3.6.1 Construção 59
3.6.2 Avaliação 60
3.6.3 Adição e subtração 60
3.6.4 Multiplicação 60
3.6.5 Divisão 61
3.6.6 Derivação 62
3.6.7 Cálculo de raízes 62
3.7 Variáveis lógicas 63

3.8 Variáveis literais 64

3.9 Listas 64

3.10 Exercícios 66

4 Expressões **69**

4.1 Expressões aritméticas 69

4.1.1 Ordem de precedência 69

4.1.2 Expressões vetoriais 70

4.1.3 Expressões matriciais 72

4.2 Expressões lógicas 75

4.3 Expressões literais 78

4.3.1 Conversão de caracteres 78

4.3.2 Manipulação de caracteres 80

4.4 Execução de expressões 86

4.5 Exercícios 87

5 Gráficos **91**

5.1 Gráficos bidimensionais 91

5.1.1 Função `plot` 91

5.1.2 Função `fplot2d` 94

5.2 Gráficos tridimensionais 97

5.2.1 Função `meshgrid` 97

5.2.2 Função `plot3d` 98

5.2.3 Função `mesh` . . . 101

5.2.4 Função `surf` . . . 102

5.3 Janela de figura . . . 102

5.4 Exercícios . . . 103

6 Linguagem de programação **105**

6.1 Programação . . . 105

6.1.1 Programa . . . 105

6.1.2 Subprograma `function` . . . 107

6.2 Estruturas condicionais . . . 113

6.2.1 Estrutura `if-end` . . . 113

6.2.2 Estrutura `if-else-end` . . . 113

6.2.3 Estrutura `if-elseif-end` . . . 114

6.2.4 Estrutura `select-case-end` . . . 115

6.3 Estruturas de repetição . . . 116

6.3.1 Estrutura `for-end` . . . 117

6.3.2 Estrutura `while-end` . . . 118

6.3.3 Estrutura com interrupção no interior . . . 119

6.4 Depuração de programa . . . 121

6.5 Exercícios . . . 122

7 Comandos de entrada e saída **125**

7.1 Formato de exibição . . . 125

7.2 Espaço de trabalho 127
7.2.1 Gravar dados 128
7.2.2 Recuperar dados 129
7.2.3 Entrada de dados 129
7.2.4 Janela de mensagem 130
7.3 Diário 131
7.4 Leitura e gravação de dados 132
7.4.1 Abertura de arquivo 132
7.4.2 Fechamento de arquivo 133
7.4.3 Gravação em arquivo 133
7.4.4 Leitura em arquivo 134
7.5 Exercícios 136

8 Computação científica **139**

8.1 Medidas de tempo 139
8.2 Álgebra linear 140
8.2.1 Parâmetros da matriz 140
8.2.2 Decomposições 143
8.2.3 Solução de sistemas 148
8.2.4 Inversa 149
8.2.5 Autovalores e autovetores 150
8.3 Interpolação 151

8.3.1 Cálculo das diferenças finitas ascendentes 151

8.3.2 Interpolação unidimensional 152

8.4 Integração numérica 153

8.5 Exercícios 155

Capítulo 1

Lógica de programação

Segundo Wirth [5], programas de computadores são formulações concretas de algoritmos abstratos baseados em representações e estruturas específicas de dados, sendo um algoritmo[1] a descrição de um conjunto de comandos que resultam em uma sucessão finita de ações.

Para este mesmo autor, a linguagem exerce uma grande influência na expressão do pensamento. Portanto, um modo mais adequado para a elaboração de algoritmos deve considerar apenas a expressão do raciocínio lógico. Além do mais, ao expressar as várias etapas do raciocínio por meio de uma linguagem de programação o programador é induzido a se preocupar com detalhes pouco importantes da linguagem.

O modelo lógico resultante de uma notação algorítmica deve ser codificado com facilidade em qualquer linguagem de programação. Assim, o uso de uma notação algorítmica leva o programador a expressar o seu raciocínio lógico independente da linguagem de programação. Uma notação algorítmica deve conter um número mínimo de estruturas de controle (sequencial, condicional e de repetição) de modo a serem implementadas facilmente nas linguagens de programação disponíveis.

Os algoritmos deste texto são descritos na notação utilizada em Algoritmos Numéricos [2], a qual é baseada naquela proposta por Farrer e outros [3, 4].

1.1 Estrutura básica de um algoritmo

Um algoritmo apresenta a seguinte estrutura básica:

[1] Esta palavra deriva do nome do matemático árabe Mohammed ibu-Musa al-Khowarizmi ($\approx$ 800 d.C.).

Algoritmo <***nome-do-algoritmo***>
{ **Objetivo:** Mostrar a estrutura básica de um algoritmo }
  <*declaração das variáveis*>
  <*comandos_1*>
  <*comandos_2*>
  ...
  <*comandos_n*>
fim algoritmo

onde

Algoritmo <***nome-do-algoritmo***>

indica o início do algoritmo denominado <*nome-do-algoritmo*> e o seu término é definido por

fim algoritmo.

A finalidade do algoritmo é descrita na forma de comentário, sendo

{ **Objetivo:** <*objetivo-do-algoritmo*> }.

As variáveis usadas no algoritmo, bem como seus tipos e estruturas, são declaradas por meio da <*declaração das variáveis*> e os <*comandos_i*> especificam as n ações a serem executadas pelo algoritmo.

1.2 Comandos de entrada e saída

O comando

leia <*lista-de-variáveis*>

é usado para indicar que a <*lista-de-variáveis*> está disponível para armazenar os dados lidos em algum dispositivo externo. Não se faz necessário descrever exatamente como os valores dessas variáveis serão fornecidos ao algoritmo. Compete ao programador decidir durante a codificação do programa se os dados serão fornecidos interativamente pelo teclado, lidos de um arquivo, passados como argumentos de um subprograma ou até mesmo definidos como constantes dentro do próprio programa. Por sua vez, o comando

escreva $<$*lista-de-variáveis*$>$

é utilizado para indicar as variáveis cujos valores devem ser escritos em algum dispositivo externo.

Exemplo 1.1 Apresenta um algoritmo básico, o qual lê as variáveis a, b e c necessárias à sua execução e escreve as variáveis x e y em algum dispositivo externo.

```
Algoritmo Operações_aritméticas
{ Objetivo: Somar e subtrair dois números }
   leia a, b, c
   x ← a + b
   y ← b − c
   escreva x, y
fim algoritmo
```

■

Exemplo 1.2 Exibe um algoritmo para ler uma temperatura em grau Fahrenheit e converter em grau Celsius.

```
Algoritmo Converte_grau
{ Objetivo: Converter grau Fahrenheit em Celsius }
   leia Fahrenheit
   Celsius ← (Fahrenheit − 32) * 5/9
   escreva Fahrenheit, Celsius
fim algoritmo
```

■

1.3 Estrutura sequencial

A mais simples das estruturas de controle de um algoritmo é a estrutura sequencial. Ela indica que os comandos devem ser executados na ordem em que aparecem. No Exemplo 1.1, o comando $x \leftarrow a + b$ é executado seguido pelo comando $y \leftarrow b - c$.

1.4 Variáveis e comentários

Uma variável corresponde a uma posição de memória do computador onde pode ser armazenado um valor. As variáveis são representadas por identificadores que são cadeias de

caracteres alfanuméricos. Os elementos de vetores e matrizes podem ser referenciados ou por subscritos ou por índices, por exemplo, v_i ou $v(i)$ e M_{ij} ou $M(i,j)$.

Um comentário é um texto inserido em qualquer parte do algoritmo para documentá-lo e aumentar a sua clareza. Esse texto é delimitado por chaves { <texto> }, como, por exemplo, { avaliação do polinômio }.

1.5 Expressões

Expressões são combinações de variáveis, constantes e operadores. Existem três tipos de expressões: aritméticas, lógicas e literais, dependendo dos tipos dos operadores e das variáveis envolvidas.

1.5.1 Comando de atribuição

O resultado de uma expressão é armazenado em uma variável por meio do símbolo $\leftarrow$ (recebe)

$$< variável > \leftarrow < expressão >$$

como visto no Exemplo 1.1.

1.5.2 Expressões aritméticas

Expressão aritmética é aquela cujos operadores são aritméticos e cujos operandos são constantes e/ou variáveis aritméticas; elas visam avaliar alguma fórmula matemática. Os operadores aritméticos são mostrados na Tabela 1.1, e a Tabela 1.2 apresenta algumas funções matemáticas elementares.

Tabela 1.1: Operadores aritméticos.

Operação	Operador	Uso
adição	`+`	`a + b`
mais unário	`+`	`+a`
subtração	`-`	`a - b`
menos unário	`-`	`-a`
multiplicação	`*`	`a * b`
divisão	`/`	`a / b`
potenciação	`^`	`a ^ b`

Tabela 1.2: Funções matemáticas.

Função	Descrição	Função	Descrição
Trigonométricas			
sen	seno	cos	co-seno
tan	tangente	sec	secante
Exponenciais			
exp	exponencial	$\log_{10}$	logaritmo decimal
$\log_e$	logaritmo natural	$raiz_2$	raiz quadrada
Numéricas			
abs	valor absoluto	quociente	divisão inteira
arredonda	arredonda em direção ao inteiro mais próximo	sinal	sinal(x) = 1 se $x>0$, = 0 se $x=0$ e = -1 se $x<0$
max	maior valor	resto	resto de divisão
min	menor valor	trunca	arredonda em direção a 0

Exemplo 1.3 A Tabela 1.3 mostra exemplos de uso das funções matemáticas numéricas. É importante observar a diferença entre as funções arredonda e trunca. ■

Tabela 1.3: Resultados de funções matemáticas numéricas.

Função	x [e y]	Valor	x [e y]	Valor
abs(x)	5	5	-3	3
arredonda(x)	0,4	0	0,5	1
quociente(x, y)	5 e 3	1	3 e 5	0
resto(x, y)	5 e 3	2	3 e 5	3
sinal(x)	-2	-1	7	1
trunca(x)	1,1	1	1,9	1

Ordem de precedência

Dada uma expressão matemática, ela deve ser escrita em uma forma linear no algoritmo, por exemplo,

$$t = a + \frac{b}{c}$$

deve ser escrita como $t \leftarrow a + b/c$, enquanto que a notação de

$$u = \frac{a+b}{c}$$

é $u \leftarrow (a + b)/c$. Ao converter a notação matemática para a notação algorítmica tem que se respeitar a ordem de precedência das operações aritméticas, a qual é apresentada na

Tabela 1.4. Quando duas operações têm a mesma prioridade efetua-se primeiro a operação mais à esquerda. No cálculo de u acima foi utilizado parênteses para que a adição fosse efetuada primeiro que a divisão, pois a divisão tem precedência sobre a adição.

Tabela 1.4: Ordem de precedência das operações aritméticas.

Prioridade	Operações
1^a	resolver parênteses
2^a	avaliar função
3^a	potenciação
4^a	menos e mais unário
5^a	multiplicação e divisão
6^a	adição e subtração

Exemplo 1.4 Escrever as expressões aritméticas na notação algorítmica.

$$t = a + \frac{b}{c} + d \longrightarrow t \leftarrow a + b/c + d.$$

$$x = \frac{a+b}{c+d} \longrightarrow x \leftarrow (a+b)/(c+d).$$

$$y = a + \frac{b}{c + \frac{d+1}{2}} \longrightarrow y \leftarrow a + b/(c + (d+1)/2).$$

É importante verificar o balanceamento dos parênteses, ou seja, o número de abre parênteses '(' tem que ser igual ao número de fecha parênteses ')'. ■

Exemplo 1.5 Avaliar as expressões aritméticas do Exemplo 1.4, para $a = 1$, $b = 2$, $c = 3$ e $d = 4$, arredondando o resultado para cinco casas decimais.

$$\begin{aligned}
t &\leftarrow a + b/c + d, \\
&\quad 1 + 2/3 + 4, \\
t &\leftarrow 5{,}66667. \\
x &\leftarrow (a+b)/(c+d), \\
&\quad (1+2)/(3+4) = 3/7, \\
x &\leftarrow 0{,}42857. \\
y &\leftarrow a + b/(c + (d+1)/2), \\
&\quad 1 + 2/(3 + (4+1)/2) = 1 + 2/(3 + 5/2) = 1 + 2/(5{,}5), \\
y &\leftarrow 1{,}36364.
\end{aligned}$$

■

1.5.3 Expressões lógicas

Expressão lógica é aquela cujos operadores são lógicos e cujos operandos são relações e/ou variáveis do tipo lógico. Uma relação é uma comparação realizada entre valores do mesmo tipo. A natureza da comparação é indicada por um operador relacional definido conforme a Tabela 1.5. O resultado de uma relação ou de uma expressão lógica é **verdadeiro** ou **falso**.

Tabela 1.5: Operadores relacionais.

Relação	Operador	Uso
igual a	$=$	$\mathtt{a} = \mathtt{b}$
diferente de	$\neq$	$\mathtt{a} \neq \mathtt{b}$
maior que	$>$	$\mathtt{a} > \mathtt{b}$
maior ou igual a	$\geq$	$\mathtt{a} \geq \mathtt{b}$
menor que	$<$	$\mathtt{a} < \mathtt{b}$
menor ou igual a	$\leq$	$\mathtt{a} \leq \mathtt{b}$

Exemplo 1.6 Avaliar as expressões lógicas para $c = 1$, $d = 3$, $x = 2$, $y = 3$ e $z = 10$.

$c \geq d, \quad 1 \geq 3 \rightarrow$ **falso**.

$d = c + x, \quad 3 = 1 + 2 \rightarrow$ **verdadeiro**.

$x + y < z - c, \quad 2 + 3 < 10 - 1 \rightarrow$ **verdadeiro**. ■

Os operadores lógicos mostrados na Tabela 1.6 permitem a combinação ou negação das relações lógicas.

Tabela 1.6: Operadores lógicos.

Operação	Operador	Uso
conjunção	**e**	$<$*expressão_1*$>$ **e** $<$*expressão_2*$>$
disjunção	**ou**	$<$*expressão_1*$>$ **ou** $<$*expressão_2*$>$
negação	**não**	**não**($<$*expressão*$>$)

A Tabela 1.7 mostra os resultados obtidos com os operadores lógicos, sendo que `V` significa **verdadeiro** e `F` significa **falso**.

Tabela 1.7: Resultados com operadores lógicos.

a **e** *b*		
*a**b*	V	F
V	V	F
F	F	F

a **ou** *b*		
*a**b*	V	F
V	V	V
F	V	F

não *a*		
a	V	F
	F	V

Ordem de precedência

De modo similar às expressões aritméticas, as expressões lógicas também possuem uma ordem de precedência, como mostrado na Tabela 1.8.

Tabela 1.8: Ordem de precedência das operações lógicas.

Prioridade	Operação
1^a	relacional
2^a	negação
3^a	conjunção
4^a	disjunção

Exemplo 1.7 Avaliar as expressões lógicas abaixo para $c = 1$, $d = 3$, $x = 2$, $y = 3$ e $z = 10$.

$d > c$ **e** $x + y + 5 = z$,

$3 > 1$ **e** $2 + 3 + 5 = 10, \rightarrow 3 > 1$ **e** $10 = 10$,

`V` **e** `V` $\rightarrow$ **verdadeiro**.

$x = d - 2$ **ou** $c + y \neq z$,

$2 = 3 - 2$ **ou** $1 + 3 \neq 10, \rightarrow 2 = 1$ **ou** $4 \neq 10$,

`F` **ou** `V` $\rightarrow$ **verdadeiro**.

$d + x \geq z/2$ **e não**$(d = y)$,

$3 + 2 \geq 10/2$ **e não**$(3 = 3) \rightarrow 5 \geq 5$ **e não**$(\texttt{V})$,

`V` **e** `F` $\rightarrow$ **falso**.

1.5.4 Ordem geral de precedência ■

Combinando as ordens de precedência das operações aritméticas e lógicas, tem-se uma ordem geral de precedência das operações matemáticas, como apresentada na Tabela 1.9.

Tabela 1.9: Ordem de precedência das operações matemáticas.

Prioridade	Operações
1^a	( )
2^a	função
3^a	^
4^a	+ e − unário
5^a	$*$ e /
6^a	+ e −
7^a	$>$, $\geq$, $<$, $\leq$, $=$ e $\neq$
8^a	**não**
9^a	**e**
10^a	**ou**

Exemplo 1.8 Avaliar as expressões abaixo para $a = 1$, $b = 2$ e $c = 3$.

$-(a + 5)\hat{}2 > b + 6 * c$ **e** $4 * c = 15$,
$-(1 + 5)\hat{}2 > 2 + 6 * 3$ **e** $4 * 3 = 15$,
$-6\hat{}2 > 20$ **e** $12 = 15$, $\quad -36 > 20$ **e** F,
F **e** F $\rightarrow$ **falso**.

$(a + b)/c \leq b\hat{}c - a$ **ou** $a * (b + c) \geq 5$,
$(1 + 2)/3 \leq 2\hat{}3 - 1$ **ou** $1 * (2 + 3) \geq 5$,
$1 \leq 7$ **ou** $5 \geq 5$,
V **ou** V $\rightarrow$ **verdadeiro**.

$c + a/2 \neq b - a$ **e** $b + c\hat{}(b + 1) < -(b + c)/4$,
$3 + 1/2 \neq 2 - 1$ **e** $2 + 3\hat{}(2 + 1) < -(2 + 3)/4$,
$3{,}5 \neq 1$ **e** $29 < -1{,}25$,
V **e** F $\rightarrow$ **falso**.

■

1.5.5 Expressões literais

Uma expressão literal é formada por operadores literais e operandos, os quais são constantes e/ou variáveis do tipo literal. Operações envolvendo literais, tais como, concatenação, inserção, busca etc, geralmente, são realizadas por meio de funções disponíveis nas bibliotecas das linguagens de programação.

O caso mais simples de uma expressão literal é uma constante literal, a qual é constituída por uma cadeia de caracteres delimitada por apóstrofo ('), por exemplo, *mensagem* $\leftarrow$ 'matriz singular'.

1.6 Estruturas condicionais

Os algoritmos apresentados nos Exemplos 1.1 e 1.2 utilizam apenas a estrutura sequencial. Ela faz com que os comandos sejam executados na ordem em que aparecem. Uma estrutura condicional possibilita a escolha dos comandos a serem executados quando certa condição for ou não satisfeita alterarando, assim, o fluxo natural de comandos. A condição é representada por uma expressão lógica. As estruturas condicionais podem ser simples ou compostas.

1.6.1 Estrutura condicional simples

Esta estrutura apresenta a forma

```
se <condição> então
   <comandos>
fim se
```

Neste caso, a lista de <*comandos*> será executada se, e somente se, a expressão lógica <*condição*> tiver como resultado o valor **verdadeiro**.

Exemplo 1.9 Fazer um algoritmo para calcular o logaritmo decimal de um número positivo.

```
Algoritmo Logaritmo_decimal
{ Objetivo: Calcular logaritmo decimal de número positivo }
   leia x
   se x > 0 então
      Log ← log10(x)
      escreva x, Log
   fim se
fim algoritmo
```

Os comandos $Log \leftarrow \log_{10}(x)$ e **escreva** x, Log só serão executados se a variável x contiver um valor maior que 0. ■

Exemplo 1.10 Escrever um algoritmo para ler dois números e escrevê-los em ordem decrescente.

```
Algoritmo Ordem_decrescente
{ Objetivo: Escrever dois números dados em ordem decrescente }
   leia m, n
   se m < n então
      aux ← m
      m ← n
      n ← aux
   fim se
   escreva m, n
fim algoritmo
```

Haverá a troca entre as variáveis m e n, por meio dos três comandos $aux \leftarrow m$, $m \leftarrow n$ e $n \leftarrow aux$ se, e somente se, a variável m contiver um valor menor que n. ■

1.6.2 Estrutura condicional composta

No caso de haver duas alternativas possíveis, deve ser usada uma estrutura da forma

```
se <condição> então
   <comandos_1>
senão
   <comandos_2>
fim se
```

Se a expressão lógica <*condição*> tiver como resultado o valor **verdadeiro**, então a sequência <*comandos_1*> será executada e a sequência <*comandos_2*> não será executada. Por outro lado, se o resultado de <*condição*> for **falso**, então será a lista <*comandos_2*> a única a ser executada.

Exemplo 1.11 Elaborar um algoritmo para verificar se um número está dentro de um dado intervalo, por exemplo, se $1 \leq a < 5$.

```
Algoritmo Intervalo
{ Objetivo: Verificar se um número pertence ao intervalo [1, 5) }
   leia a
   se a ≥ 1 e a < 5 então
      escreva a ' pertence ao intervalo '[1, 5)
   senão
      escreva a ' não pertence ao intervalo '[1, 5)
   fim se
fim algoritmo
```

O comando **escreva** *a* ' pertence ao intervalo' será executado se, e somente se, a variável *a* contiver um valor maior ou igual a *1* e menor que *5*. Caso contrário, se *a* for menor que *1* ou maior ou igual a *5* então **escreva** *a* ' não pertence ao intervalo' será o único comando executado. ■

Exemplo 1.12 Elaborar um algoritmo para avaliar as funções modulares $f(x) = |2x|$ e $g(x) = |5x|$.

```
Algoritmo Funções_modulares
{ Objetivo: Avaliar duas funções modulares }
   leia x
   se x ≥ 0 então
      fx ← 2 * x
      gx ← 5 * x
   senão
      fx ← −2 * x
      gx ← −5 * x
   fim se
   escreva x, fx, gx
fim algoritmo
```

Se a variável *x* contiver um valor positivo ou nulo, então os dois comandos $fx \leftarrow 2 * x$ e $gx \leftarrow 5 * x$ serão executados, seguindo-se o comando **escreva** *x*, *fx*, *gx*. No entanto, se *x* contiver um valor negativo, os comandos $fx \leftarrow -2 * x$, $gx \leftarrow -5 * x$ e **escreva** *x*, *fx*, *gx* serão os únicos a serem executados. ■

1.7 Estruturas de repetição

Uma estrutura de repetição faz com que uma sequência de comandos seja executada repetidamente até que uma dada condição de interrupção seja satisfeita. Existem, basicamente, dois tipos dessas estruturas, dependendo se o número de repetições for indefinido ou definido.

1.7.1 Número indefinido de repetições

Este tipo de estrutura de repetição apresenta a forma

```
repita
   < comandos_1 >
   se < condição > então
      interrompa
   fim se
   < comandos_2 >
fim repita
< comandos_3 >
```

O comando **interrompa** faz com que o fluxo de execução seja transferido para o comando imediatamente a seguir do **fim repita**. Assim, as listas <*comandos_1*> e <*comandos_2*> serão repetidas até que a expressão lógica <*condição*> resulte no valor **verdadeiro**. Quando isso ocorrer, a repetição será interrompida (<*comandos_2*> não será executada) e a lista <*comandos_3*>, após ao **fim repita**, será executada.

Exemplo 1.13 Elaborar um algoritmo para calcular $\sqrt{a},\ a > 0$, utilizando o processo babilônico baseado na fórmula de recorrência [2]

$$x_{k+1} = \left(x_k + \frac{a}{x_k}\right) \times 0{,}5 \text{ para } x_0 > 0.$$

```
Algoritmo Raiz_quadrada
{ Objetivo: Calcular a raiz quadrada de um número positivo }
   leia a, z   { valor para calcular a raiz quadrada e valor inicial }
   i ← 0
   repita
      i ← i + 1
      x ← (z + a/z) * 0,5
      se abs(x − z) < 10^−10 ou i = 20 então
         interrompa
      fim se
      z ← x
   fim repita
   { teste de convergência }
   se abs(x − z) < 10^−10 então
      escreva x   { raiz quadrada de a }
   senão
      escreva 'processo não convergiu com 20 iterações'
   fim se
fim algoritmo
```

É gerada uma sequência de valores em x que é interrompida quando a diferença entre dois valores consecutivos for menor que a tolerância 10^{-10} ou atingir 20 iterações. O teste de convergência verifica qual das duas condições foi satisfeita, se for a primeira então o processo convergiu. ■

Exemplo 1.14 Escrever um algoritmo para determinar o maior número de ponto flutuante que, somado a 1, seja igual a 1.

```
Algoritmo Epsilon
{ Objetivo: Determinar a precisão da máquina }
   Epsilon ← 1
   repita
      Epsilon ← Epsilon/2
      se Epsilon + 1 = 1 então
         interrompa
      fim se
   fim repita
   escreva Epsilon
fim algoritmo
```

Esta sequência faz com que seja calculada a chamada precisão da máquina ε. Quando a variável *Epsilon* assumir um valor que, adicionado a 1, seja igual a 1, então a estrutura **repita**–**fim repita** é abandonada e o comando **escreva** *Epsilon* será executado. ■

A forma **repita**–**fim repita** é o caso geral de uma estrutura de repetição. Se a lista <*comandos_1*> não existir, ter-se-á uma estrutura de repetição com interrupção no início (estrutura **while**). Similarmente, se não houver a lista <*comandos_2*>, então será uma estrutura com interrupção no final (estrutura **repeat**–**until**).

1.7.2 Número definido de repetições

Quando se souber com antecedência quantas vezes a estrutura deve ser repetida, pode ser usado um comando de forma mais simples

```
para <controle> ← <valor-inicial> até <valor-final> passo <delta> faça
   <comandos>
fim para
```

Nesta estrutura, inicialmente, é atribuído à variável <*controle*> o valor de <*valor-inicial*> e verificado se ele é maior que o <*valor-final*>. Se for maior, a estrutura **para**–**faça** não será executada. Se for menor ou igual, então os <*comandos*> serão executados e a variável

<*controle*> será incrementada com o valor de <*delta*>. Novamente, é verificado se a variável <*controle*> é maior que o <*valor-final*>; se não for maior, então os <*comandos*> serão executados e assim sucessivamente. As repetições se processam até que a variável <*controle*> se torne maior que o <*valor-final*>. Quando o incremento <*delta*> tiver o valor 1, então o **passo** <*delta*> pode ser omitido da estrutura **para**–**faça**.

Exemplo 1.15 Escrever um algoritmo para mostrar que a soma dos n primeiros números ímpares é igual ao quadrado de n

$$\sum_{i=1}^{n}(2i-1) = 1+3+5+\ldots+2n-1 = n^2. \tag{1.1}$$

Por exemplo, para $n = 5$ a variável i assume os valores $i = 1, 2, 3, 4$ e 5. Assim, a expressão do somatório $(2i-1)$ gera os $n = 5$ primeiros ímpares $(2i-1) = 1, 3, 5, 7$ e 9, cuja soma é $1+3+5+7+9 = 25 = 5^2$.

Para implementar um somatório se faz necessário o uso de uma variável auxiliar para acumular o resultado da soma de cada novo termo. Inicialmente, esse acumulador, denominado *Soma*, recebe o valor *0* que é o elemento neutro da adição. Para cada valor de i é incrementado o valor de $2i-1$, como mostrado na tabela

i	$2i-1$	*Soma*
–	–	0
1	1	$0+1=1$
2	3	$1+3=4$
3	5	$4+5=9$
4	7	$9+7=16$
5	9	$16+9=25$

.

Ao final, quando $i = n$ a variável *Soma* conterá o valor do somatório ($n^2 = 5^2 = 25$). A implementação da notação matemática (1.1) é mostrada no algoritmo abaixo.

```
Algoritmo Primeiros_ímpares
{ Objetivo: Verificar propriedade dos números ímpares }
   leia n
   Soma ← 0
   para i ← 1 até n faça
      Soma ← Soma + 2 * i − 1
   fim para
   escreva Soma, n²
fim algoritmo
```

A sequência de números ímpares $(2i - 1)$ é gerada por uma estrutura **para–faça**, na qual a variável de controle *i* começa com o valor *1* e é incrementada de *1* até assumir o valor *n*. Essa mesma sequência de números ímpares pode ser gerada pela estrutura similar

```
para j ← 1 até 2 * n − 1 passo 2 faça
    Soma ← Soma + j
fim para
```

■

Exemplo 1.16 Elaborar um algoritmo para calcular o fatorial de um número inteiro n, sendo

$$n! = \prod_{k=1}^{n} k = 1 \times 2 \times 3 \times \ldots \times n. \tag{1.2}$$

Para $n = 4$, por exemplo, a variável k assume os valores $k = 1, 2, 3$ e 4. A expressão do produtório resulta em $4! = 1 \times 2 \times 3 \times 4 = 24$.

Como no caso do somatório, para implementar um produtório é necessário uma variável auxiliar para acumular o resultado do produto de cada termo. Esse acumulador recebe, inicialmente, o valor *1* que é o elemento neutro da multiplicação. Para cada valor de k o acumulador, denominado *Fatorial*, é multiplicado por k, conforme esquematizado na tabela

k	*Fatorial*
–	1
1	$1 \times 1 = 1$
2	$1 \times 2 = 2$
3	$2 \times 3 = 6$
4	$6 \times 4 = 24$

.

Quando $k = n$ a variável *Fatorial* conterá o valor do produtório que é igual a $n!$. A implementação da notação matemática (1.2) é apresentada no algoritmo abaixo.

```
Algoritmo Fatorial
{ Objetivo: Calcular o fatorial de um número inteiro }
   leia n
   Fatorial ← 1
   para k ← 2 até n faça
      Fatorial ← Fatorial * k
   fim para
   escreva n, Fatorial
fim algoritmo
```

■

Esse algoritmo utiliza a estrutura de repetição **para –faça**. Inicialmente, a variável de controle *k* recebe o valor *2*, para evitar a operação desnecessária 1×1 se $k = 1$. Se $n = 1$ então a estrutura **para–faça** não será executada, mas ter-se-á o resultado correto $1! = 1$.

1.8 Falha no algoritmo

O comando

abandone

é usado para indicar que haverá uma falha evidente na execução do algoritmo, isto é, uma condição de erro. Por exemplo, uma divisão por zero, uma singularidade da matriz ou mesmo o uso inapropriado de parâmetros. Neste caso, a execução deve ser cancelada.

1.9 Modularização

A técnica de modularização consiste em subdividir o algoritmo principal em módulos, formados por um subconjunto de comandos, cada qual com um objetivo bem específico. Uma grande vantagem da modularização é a utilização de um mesmo módulo por outros algoritmos que requeiram aquela funcionalidade.

A interação entre as variáveis dos módulo e as variáveis do algoritmo principal é realizada pelos parâmetros de entrada e de saída. Os dados necessários para a execução de um algoritmo (módulo) são declarados por meio do comando

parâmetros de entrada *<lista-de-variáveis>*,

onde *<lista-de-variáveis>* são os nomes das variáveis, separadas por vírgulas, contendo os valores fornecidos. Por outro lado, os valores de interesse calculados pelo algoritmo (módulo) são definidos pelo comando

parâmetros de saída *<lista-de-variáveis>*.

É de responsabilidade do projetista do módulo definir a *<lista-de-variáveis>* em termos de identificadores, ordem e número.

Exemplo 1.17 Escrever um módulo, baseado no algoritmo de Exemplo 1.16, que receba como parâmetro de entrada o número inteiro *n* e retorne como parâmetro de saída o fatorial desse número na variável *Fatorial*.

```
Algoritmo Calcula_fatorial
{ Objetivo: Calcular o fatorial de um número inteiro }
parâmetro de entrada n    { número inteiro }
parâmetro de saída Fatorial    { fatorial do número inteiro }
   Fatorial ← 1
   para k ← 2 até n faça
      Fatorial ← Fatorial * k
   fim para
fim algoritmo
```

.

A forma de chamada do módulo acima pelo módulo principal depende da linguagem de programação escolhida. Essa forma pode ser, por exemplo,

```
Algoritmo Fatorial_principal
{ Objetivo: Utilizar o módulo Calcula_fatorial }
   leia n    { número inteiro }
   Fatorial ← Calcula_fatorial(n)    { chamada do módulo }
   escreva Fatorial    { fatorial do número inteiro }
fim algoritmo
```

■

1.10 Estruturas de dados

Conforme visto na Seção 1.4 Variáveis e comentários, uma variável corresponde a uma posição de memória do computador onde está armazenado um determinado valor. Ela é representada por um identificador que é uma cadeia de caracteres alfanuméricos. Quando se faz referência a um identificador, na realidade, está tendo acesso ao conteúdo de uma posição de memória.

Quando se tem várias variáveis com o conteúdo do mesmo tipo elas podem ser agrupadas em uma única variável, sendo cada conteúdo individualizado por índices. Por isso, elas são denominadas variáveis compostas homogêneas e as mais comuns são os vetores, com um único índice, e as matrizes, com dois ou mais índices.

1.10.1 Vetores

Vetores ou arranjos são variáveis homogêneas unidimensionais, ou seja, são variáveis agrupadas em um único nome, cujos elementos são individualizados por um só índice. Por exemplo, sejam as notas de 45 alunos referentes à Prova 1 da disciplina Análise Numérica, mostradas na tabela abaixo, sendo nc o número de chamada de cada aluno,

Notas	
nc	*Prova 1*
1	15
2	19
3	12
4	20
5	18
6	20
7	13
8	19
⋮	⋮
45	17

O vetor *Notas* contém 45 elementos, cada um referenciando a nota da Prova 1 de cada aluno. Deste modo, $Notas(1) = 15$ é a nota do aluno 1 da lista de chamada, $Notas(2) = 19$ é a nota do aluno 2 da lista e assim, sucessivamente, até que $Notas(45) = 17$ é a nota do aluno 45. Por exemplo, para obter a nota média da Prova 1 basta calcular

$$\text{Nota\_média} = \frac{1}{45}\sum_{i=1}^{45} Notas(i) = \frac{1}{45}(15 + 19 + 12 + \ldots + 17).$$

Exemplo 1.18 A Figura 1.1 mostra um algoritmo para determinar o maior elemento de um vetor x de tamanho n.

```
Algoritmo Vetor_maior
{ Objetivo: Determinar o maior elemento de um vetor }
parâmetros de entrada n, x
   { tamanho do vetor e o vetor }
parâmetro de saída Maior
   { Maior elemento do vetor }
   Maior ← x(1)
   para i ← 2 até n faça
      se x(i) > Maior então
         Maior ← x(i)
      fim se
   fim para
fim algoritmo
```

Figura 1.1: Maior elemento de um vetor x de tamanho n.

■

Exemplo 1.19 O algoritmo da Figura 1.2 inverte a ordem dos elementos de um vetor x de tamanho n. ■

```
Algoritmo Vetor_inverte
{ Objetivo: Inverter a ordem dos elementos de um vetor }
parâmetros de entrada n, x
   { tamanho do vetor e o vetor }
parâmetro de saída x
   { vetor com os elementos em ordem inversa }
   para i ← 1 até trunca(n/2) faça
      t ← x(i)
      x(i) ← x(n + 1 − i)
      x(n + 1 − i) ← t
   fim para
fim algoritmo
```

Figura 1.2: Inversão da ordem dos elementos de um vetor x de tamanho n.

(Ver significado da função trunca na Tabela 1.2, na página 5.)

Exemplo 1.20 Dado um vetor x com n componentes, a Figura 1.3 mostra um algoritmo para calcular a média aritmética $\bar{x}$ e o desvio padrão s de seus elementos, sabendo que

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \;\text{ e }\; s^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right).$$

```
Algoritmo Média_desvio
{ Objetivo: Calcular média aritmética e desvio padrão }
parâmetros de entrada n, x
   { tamanho e elementos do vetor }
parâmetros de saída Média, DesvioPadrão
   Soma ← 0
   Soma2 ← 0
   para i ← 1 até n faça
      Soma ← Soma + x(i)
      Soma2 ← Soma2 + x(i)^2
   fim para
   Média ← Soma/n
   DesvioPadrão ← raiz2((Soma2 − Soma^2/n)/(n − 1))
   escreva Média, DesvioPadrão
fim algoritmo
```

Figura 1.3: Algoritmo para cálculo da média aritmética e desvio padrão.

(Ver significado da função raiz_2 na Tabela 1.2, na página 5.)

■

1.10.2 Matrizes

Matrizes são variáveis homogêneas bidimensionais, isto é, são variáveis agrupadas em um único nome com os elementos individualizados por meio de dois índices. Para exemplificar, considere a tabela abaixo com as notas de 45 alunos referentes às Provas 1, 2, 3 e 4 da disciplina Análise Numérica, onde *nc* é o número de ordem na chamada de cada aluno,

Notas				
Análise Numérica				
nc	*Prova 1*	*Prova 2*	*Prova 3*	*Prova 4*
1	15	12	17	10
2	19	20	18	20
3	12	19	17	18
4	20	20	20	20
5	18	13	20	19
6	20	19	19	20
7	13	17	14	18
8	19	20	20	17
⋮	⋮	⋮	⋮	⋮
45	17	12	10	16

.

Para referenciar a nota de cada aluno em cada prova são necessários dois índices. O primeiro índice indica qual aluno entre os quarenta e cinco e o segundo índice especifica qual das quatro provas. Por isso, a matriz *Notas* contém 45 linhas referentes à cada aluno e 4 colunas indicando cada uma das provas. Assim, $Notas(1,1) = 15$ é a nota do aluno 1 na Prova 1, $Nota(2,3) = 18$ é a nota do aluno 2 na Prova 3 e $Notas(45,4) = 16$ é a nota do aluno 45 na Prova 4.

Exemplo 1.21 O algoritmo da Figura 1.4 calcula a soma das linhas da matriz A de dimensão $m \times n$.

■

Exemplo 1.22 A Figura 1.5 apresenta um algoritmo para determinar o maior elemento em cada linha de uma matriz A de dimensão $m \times n$.

■

```
Algoritmo Matriz_soma_linha
{ Objetivo: Calcular a soma de cada linha da matriz }
parâmetros de entrada m, n, A
   { número de linhas, número de colunas e elementos da matriz }
parâmetro de saída SomaLinha
   { vetor contendo a soma de cada linha }
   para i ← 1 até m faça
      SomaLinha(i) ← 0
      para j ← 1 até n faça
         SomaLinha(i) ← SomaLinha(i) + A(i, j)
      fim para
   fim para
fim algoritmo
```

Figura 1.4: Algoritmo para calcular a soma das linhas de uma matriz.

```
Algoritmo Matriz_maior
{ Objetivo: Determinar maior elemento em cada linha da matriz }
parâmetros de entrada m, n, A
   { número de linhas, número de colunas e elementos da matriz }
parâmetro de saída Maior
   { vetor contendo o maior elemento de cada linha }
   para i ← 1 até m faça
      Maior(i) ← A(i, 1)
      para j ← 2 até n faça
         se A(i, j) > Maior(i) então
            Maior(i) ← A(i, j)
         fim se
      fim para
   fim para
fim algoritmo
```

Figura 1.5: Algoritmo para determinar o maior elemento da linha de uma matriz.

Exemplo 1.23 A Figura 1.6 mostra um algoritmo para calcular o vetor x $(n \times 1)$ resultante do produto de uma matriz A $(n \times m)$ por um vetor v $(m \times 1)$

$$x_i = \sum_{j=1}^{m} a_{ij} v_j,\ i = 1, 2, \ldots, n.$$

```
Algoritmo Produto_matriz_vetor
{ Objetivo: Calcular o produto de uma matriz por um vetor }
parâmetros de entrada n, m, A, v
   { número de linhas, número de colunas, }
   { elementos da matriz e elementos do vetor }
parâmetro de saída x
   { vetor resultante do produto matriz-vetor }
   para i ← 1 até n faça
      Soma ← 0
      para j ← 1 até m faça
         Soma ← Soma + A(i,j) * v(j)
      fim para
      x(i) ← Soma
   fim para
fim algoritmo
```

Figura 1.6: Produto matriz-vetor. ■

1.10.3 Hipermatrizes

Hipermatrizes são variáveis compostas homogêneas multidimensionais, sendo o número máximo de dimensões limitado por cada linguagem de programação. Por exemplo, seja a tabela abaixo mostrando as notas de 45 alunos referentes às Provas 1, 2, 3 e 4 da disciplina Algoritmos e Estruturas de Dados III (AEDS III), sendo nc o número de ordem na chamada de cada aluno,

Notas				
AEDS III				
nc	*Prova 1*	*Prova 2*	*Prova 3*	*Prova 4*
1	16	14	18	12
2	17	19	20	18
3	14	20	20	19
4	19	18	19	19
5	20	19	19	20
6	12	20	20	19
7	15	18	11	18
8	18	19	17	20
⋮	⋮	⋮	⋮	⋮
45	18	14	15	17

Nesse caso, para referenciar a nota de cada aluno em cada prova e em cada disciplina na mesma variável *Nota* são necessários agora três índices. O primeiro índice indica qual aluno entre os quarenta e cinco, o segundo índice especifica qual das quatro provas e o terceiro estabelece qual das duas disciplinas entre Análise Numérica e AEDS III. Assim, a hipermatriz tridimensional *Notas* contém duas páginas, uma para cada disciplina (terceiro índice), com cada página contendo 45 linhas referentes à cada aluno (primeiro índice) e com 4 colunas indicando cada uma das provas (segundo índice). Por exemplo, $\mathit{Notas}(3, 2, 1) = 19$ é a nota do aluno 3 na Prova 2 em Análise Numérica enquanto que $\mathit{Nota}(8, 4, 2) = 20$ é a nota do aluno 8 na Prova 4 de AEDS III.

1.11 Exercícios

Seção 1.5 Expressões

Escrever as fórmulas abaixo na notação algorítmica e avaliar a expressão para $a = 2$, $b = 5$, $c = 7$ e $d = 10$.

1.1 $v \leftarrow \dfrac{a+d}{4} - \dfrac{3}{b-c-1}$.

1.2 $w \leftarrow \dfrac{b + \frac{d+2}{6}}{a-1} + c$.

1.3 $x \leftarrow a + \dfrac{b}{4 + \frac{c-2}{d-5}}$.

1.4 $y \leftarrow c \geq d+1$ **e** $ab = d$.

1.5 $z \leftarrow b + \dfrac{d}{2} < ac$ **ou** $\sqrt{a^2+b+c} > \dfrac{d}{a+3}$.

Seção 1.6 Estruturas condicionais

Escrever um algoritmo para avaliar uma função abaixo:

1.6 $e(x) = x^2 + \sqrt{x+1}$, para $x > -1$.

1.7 $f(x) = 3x^2 + 2x + \cos(x^3 - 1)$, para $1 \leq x < 5$.

1.8 $g(x) = \operatorname{sen}^{-1}(x) + e^{2x+1}$, para $|x| \leq 1$.

1.9 $h(x) = \begin{cases} e^x - 1, & -3 \leq x \leq 0, \\ x \operatorname{sen}(5x), & 0 \leq x \leq 5, \end{cases}$.

1.10 Fazer um algoritmo para ler dois números e escrevê-los em ordem crescente.

Seção 1.7 Estruturas de repetição

Fazer um algoritmo para calcular uma expressão abaixo.

1.11 $S = \frac{1}{2} + \frac{3}{4} + \frac{5}{6} + \ldots + \frac{9}{10}$.

1.12 $T = \frac{30 \times 29}{1} + \frac{28 \times 27}{2} + \frac{26 \times 25}{3} + \ldots + \frac{2 \times 1}{15}$.

1.13 $N = \frac{1}{1} - \frac{2}{4} + \frac{3}{9} - \frac{4}{16} + \ldots - \frac{12}{144}$.

1.14 $F = 1! + 2! - 3! + 4! + 5! - 6! + \ldots \pm n!$.

1.15 $P = -\frac{1^3}{1!} + \frac{3^3}{2!} - \frac{5^3}{3!} + \ldots + \frac{19^3}{10!}$.

Elaborar um algoritmo para ler um argumento e calcular uma função elementar abaixo, aproximada por série de potências [1]. A série deve ser truncada quando um termo for menor que 10^{-10} ou atingir 1000 termos. Essas expansões em série, geralmente, são muito ineficientes para aproximar funções, pois requerem um número elevado de termos. No entanto, elas possibilitam ótimos exemplos para uso das estruturas de repetição.

1.16 $\log_e(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \ldots,$ para $0 < x \leq 2$.

1.17 $\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$.

1.18 $\operatorname{sen}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$.

1.19 $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$.

1.20 $\operatorname{arcsen}(x) = x + \frac{1}{2} \times \frac{x^3}{3} + \frac{1 \times 3}{2 \times 4} \times \frac{x^5}{5} + \frac{1 \times 3 \times 5}{2 \times 4 \times 6} \times \frac{x^7}{7} + \ldots,$ para $|x| \leq 1$.

Seção 1.9 Modularização

Converter os algoritmos dos exemplos abaixo, que utilizam os comandos **leia** e **escreva**, em módulos com parâmetros de entrada e de saída.

1.21 Exemplo 1.13.

1.22 Exemplo 1.18.

1.23 Exemplo 1.20.

1.24 Exemplo 1.21.

1.25 Exemplo 1.23.

Seção 1.10 Estruturas de dados

Elaborar um algoritmo para cada item, sendo x, y e v vetores de dimensão n e k um escalar.

1.26 Calcular o produto interno $k = x^T y = x_1y_1 + x_2y_2 + x_3y_3 + \cdots + x_ny_n = \sum_{i=1}^{n} x_iy_i$.

1.27 Avaliar $v = kx + y \rightarrow v_i = k \times x_i + y_i,$ para $i = 1, 2, 3, \ldots, n$.

1.28 Determinar a norma Euclidiana $\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$.

1.29 Calcular a norma de máxima magnitude $\|x\|_\infty = \lim_{p\to\infty} \sqrt[p]{\sum_{i=1}^{n} |x_i|^p} = \max_{1\leq i\leq n} |x_i|$.

1.30 Ordenar os elementos de x em ordem decrescente.

Sejam o vetor x de dimensão n, o vetor y de dimensão m, a matriz A $(m \times n)$ e a matriz B $(n \times p)$. Escrever um algoritmo para cada item abaixo.

1.31 Avaliar o produto externo $M = xy^T$, com $m_{ij} = x_iy_j,\ i = 1, 2, \ldots, n$ e $j = 1, 2, \ldots, m$.

1.32 Avaliar a norma de Frobenius $\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}$.

1.33 Calcular a norma de soma máxima de coluna $\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{m} |a_{ij}|$.

1.34 Determinar a norma de soma máxima de linha $\|A\|_\infty = \max_{1\leq i\leq m} \sum_{j=1}^{n} |a_{ij}|$.

1.35 Calcular o produto $C = AB$, sendo $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj},\ i = 1, 2, \ldots, m$ e $j = 1, 2, \ldots, p$.

Capítulo 2

Ambiente de programação

O SCILAB é executado por meio do *script* `scilab` no diretório `<SCIDIR>/bin`, sendo que `<SCIDIR>` denota o diretório onde o programa foi instalado. A interação entre o SCILAB e o usuário é feita por intermédio de uma janela, na qual um comando é fornecido, interpretado e exibido o resultado.

Neste capítulo será descrita a janela de comando do SCILAB e seus comandos básicos, bem como, os meios que o programa tem de auxiliar o usuário durante a sessão, no caso de dúvida sobre a sintaxe ou mesmo a existência de algum comando.

2.1 Janela de comando

Para executar o SCILAB em ambiente Windows ou Macintosh, o cursor deve ser colocado sobre o seu ícone e pressionado o botão da esquerda do *mouse* ou então, no caso de ambiente Linux, entrar com o comando `scilab` ou `scilab -nw` (no window) em uma janela de comando. Sem a opção `-nw` aparecerá uma nova janela pela qual será feita a interação entre o SCILAB e o usuário (ver Figura 2.1).

Se for utilizada a opção `-nw` a janela do SCILAB aparecerá na própria janela de comando (ver Figura 2.2). O sinal de que o programa está pronto para receber um comando é indicado pelo *prompt* formado pelos três caracteres (`-->`).

Um comando é finalizado acionando-se a tecla **Enter** ou **Return**. Se um comando for muito longo, então três pontos (`...`) seguidos do pressionamento da tecla **Enter** ou **Return** indica que o comando continuará na próxima linha. Vários comandos podem ser colocados em uma mesma linha se eles forem separados por vírgulas ou pontos-e-vírgulas. Além disto, as vírgulas indicam ao SCILAB para mostrar os resultados e os pontos-e-vírgulas para suprimir

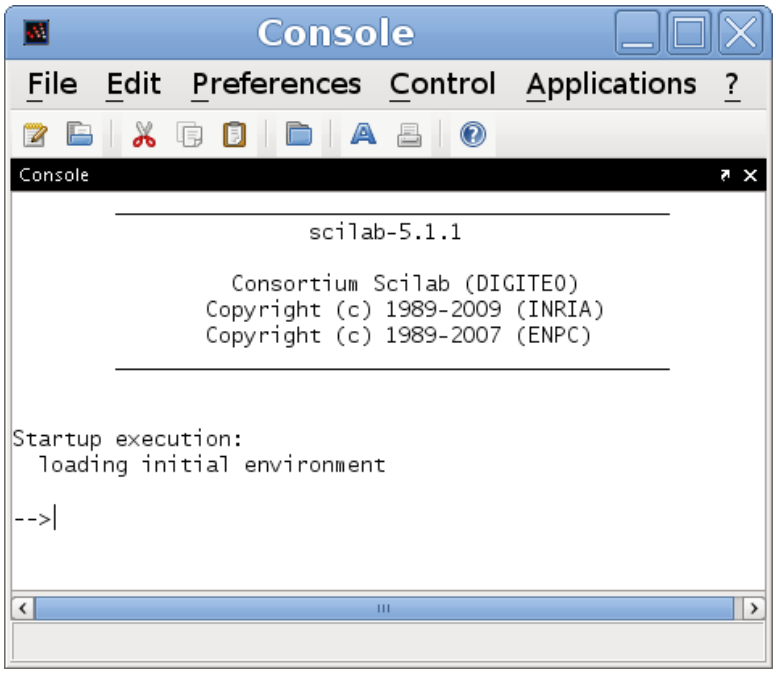


Figura 2.1: Janela de comando da versão 5.1.1 do SCILAB (sem `-nw`).

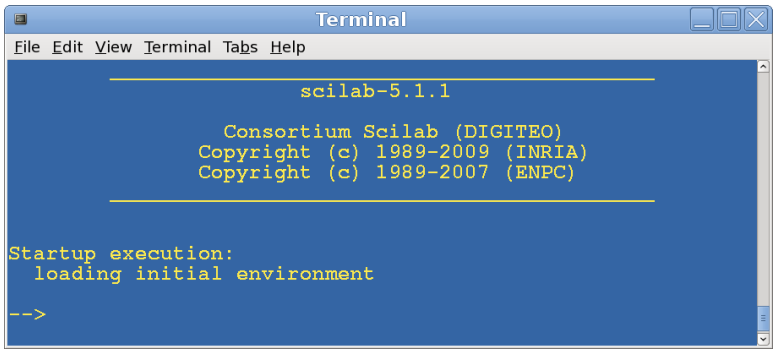


Figura 2.2: Janela de comando da versão de 5.1.1 do SCILAB (com `-nw`).

a exibição.

Um texto após duas barras invertidas (`//`) é ignorado; ele pode ser utilizado como um comentário para a documentação de um programa.

As teclas ↑ e ↓ podem ser usadas para listar os comandos previamente dados e as teclas → e ← movem o cursor na linha de comando facilitando a sua modificação.

O número de linhas e colunas a serem exibidas de cada vez na janela de comando pode ser

redefinida pelo comando `lines`. Sua sintaxe é

```
lines(<número-de-linhas>,<número-de-colunas>)
```

onde `<número-de-linhas>` e `<número-de-colunas>` definem o número de linhas e colunas, respectivamente, a serem exibidas, sendo o segundo argumento opcional. O comando é desativado por `lines(0)`.

O comando `clc` é usado para limpar a janela de comando[1] e `tohome` posiciona o cursor no canto superior esquerdo da janela de comando. O término de execução do SCILAB é feito pelos comandos `quit` ou `exit`.

Quando o SCILAB for ativado, os comandos contidos no arquivo `scilab.ini` são automaticamente executados, caso ele exista, para que sejam atribuídos valores a alguns parâmetros. Deste modo, o usuário pode criar um arquivo contendo, por exemplo, definição de constantes matemáticas e físicas, formatos de exibição ou quaisquer comandos do SCILAB para personalizar a sua janela de comando. Este arquivo deve ser criado em um diretório específico, dependendo do sistema operacional utilizado (use `help startup` para mais informações).

A abrangência e potencialidade do SCILAB está muito além do que será mostrado neste texto, por isso é aconselhável executar a opção `Scilab Demonstrations`, dentro da opção ? no canto superior à direita[1], para visualizar uma demonstração e se ter uma idéia dessa potencialidade.

2.1.1 Espaço de trabalho

As variáveis criadas durante uma sessão ficam armazenadas em uma memória denominada espaço de trabalho. O comando `who` lista o nome das variáveis que estão sendo usadas, ou seja, que estão presentes no espaço de trabalho. Este comando lista as variáveis criadas pelo usuário e as definidas pelo próprio SCILAB. Por exemplo, no início de uma sessão, quando o usuário criou apenas uma variável,

```
-->a = 1   // cria a variavel a com o valor 1
 a  =
    1.
-->who
Your variables are:
                      a                      home                 matiolib            parameterslib
  simulated_annealinglib    genetic_algorithmslib                umfpacklib                      fft
              scicos_pal             %scicos_menu            %scicos_short             %scicos_help
    %scicos_display_mode            modelica_libs          scicos_pal_libs         %scicos_lhb_list
     %CmenuTypeOneVector              %scicos_gif          %scicos_contrib           scicos_menuslib
         scicos_utilslib           scicos_autolib           spreadsheetlib             demo_toolslib
    development_toolslib        scilab2fortranlib       scipadinternalslib                 scipadlib
              soundlib               texmacslib                tclscilib                 m2scilib
        maple2scilablib    metanetgraph_toolslib         metaneteditorlib  compatibility_functilib
          statisticslib                  timelib                stringlib     special_functionslib
              sparselib     signal_processinglib                       %z                       %s
```

[1]Quando o SCILAB for executado sem a opção `-nw`.

```
         polynomialslib           overloadinglib         optimizationlib       linear_algebralib
                  jvmlib                    iolib       interpolationlib               integerlib
        dynamic_linklib                   guilib      data_structureslib                 cacsdlib
      graphic_exportlib              graphicslib              fileiolib              functionslib
  elementary_functionlib differential_equationlib            helptoolslib                   corelib
                    PWD                       %F                    %T                     %nan
                   %inf                 COMPILER                   SCI                  SCIHOME
                 TMPDIR                    MSDOS                 %gui                     %pvm
                    %tk                    %fftw                    $                       %t
                     %f                     %eps                  %io                      %i
                     %e                       %pi
 using      33369 elements out of     5000000.
 and           86 variables out of       9231.
Your global variables are:
           %modalWarning                 demolist                %helps          %helps_modules
             %driverName          %exportFileName              LANGUAGE              %toolboxes
          %toolboxes_dir
 using       2681 elements out of     5000001.
 and            9 variables out of        767.
```

2.1.2 Diretório corrente

Diretório corrente é aquele considerado em uso, sem ser necessário especificá-lo explicitamente durante uma ação. Para saber qual o diretório de trabalho corrente utiliza-se o comando `pwd`,

```
-->pwd   // mostra o diretorio de trabalho corrente
 ans  =
 ffcampos
```

O resultado exibido depende do diretório de onde o SCILAB foi executado. O diretório corrente pode ser alterado por meio do comando `cd`,

```
-->cd scilab   // muda o diretorio de trabalho corrente para 'scilab'
 ans  =
 ffcampos/scilab
```

O SCILAB fornece diversos comandos para gerenciamento de arquivos, os quais são mostrados na Tabela 2.1.

Tabela 2.1: Comandos para gerenciamento de arquivos.

Comando	Descrição
`dir` ou `ls`	lista os arquivos do diretório corrente;
`mdelete('<nome_do_arquivo>')`	remove o arquivo `<nome_do_arquivo>`;
`cd <dir>` ou `chdir <dir>`	muda o diretório para `<dir>`;
`pwd`	mostra o diretório corrente.

2.1.3 Comando `unix`

O comando `unix('<comando_do_unix>')` permite a execução de um comando do sistema operacional Unix dentro do SCILAB. Por exemplo, caso o arquivo `precisao.sci` exista no diretório corrente, o seu conteúdo é exibido pelo comando,

```
-->unix('more precisao.sci');   // lista conteudo do arquivo
// programa precisao
// Objetivo: determinar a precisao de um computador
n = 0;
Epsilon = 1;
while 1 + Epsilon > 1
   n = n + 1;
   Epsilon = Epsilon / 2;
end
n, Epsilon, %eps
```

2.2 Comandos de auxílio ao usuário

O SCILAB possui muito mais comandos do que aqueles apresentados neste texto, o que torna mais difícil lembrá-los. Com o intuito de auxiliar o usuário na procura de comandos, o programa provê assistência por intermédio de suas extensivas capacidades de auxílio direto. Estas capacidades estão disponíveis em três formas: o comando `help`, o comando `apropos` e, interativamente, por meio de um menu de barras.

2.2.1 Comando `help`

O comando `help` é a maneira mais simples de obter auxílio no caso de conhecer o tópico em que se quer assistência. Sua sintaxe é

```
help <tópico>
```

onde `<tópico>` é o nome da função ou de diretório. Por exemplo, quando se usa `help sqrt`, abre-se uma janela como a mostrada na Figura 2.3.

Assim, são dadas informações sobre a função `sqrt` para extrair raiz quadrada. O comando `help` funciona a contento quando se conhece exatamente o tópico sobre o qual quer assistência. Considerando que muitas vezes este não é o caso, o `help` pode ser usado sem `<tópico>` para apresentar uma página de hipertexto contendo a lista de ítens disponíveis.

O comando `help` é mais conveniente se o usuário conhecer exatamente o tópico em que deseja auxílio. No caso de não saber soletrar ou mesmo desconhecer um tópico as outras duas formas de assistência são muitas vezes mais proveitosas.

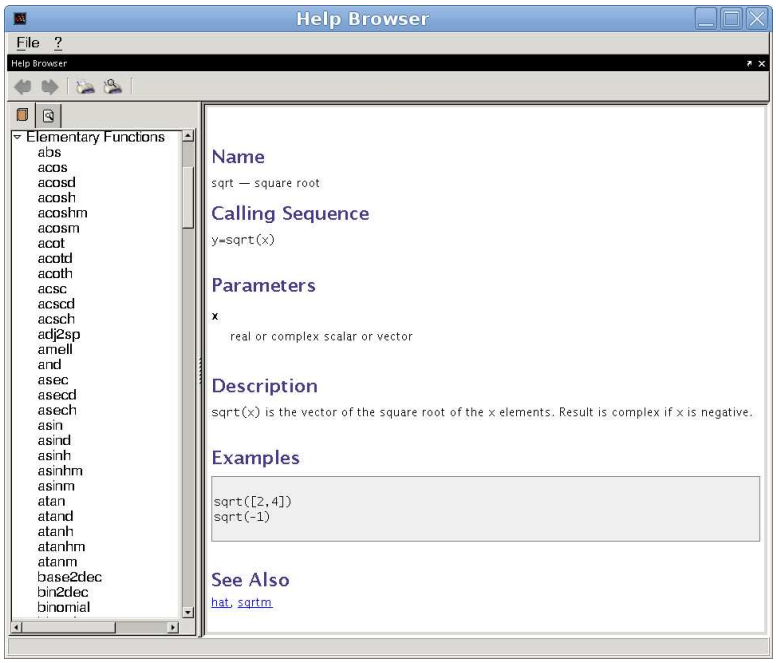


Figura 2.3: Janela do comando `help`.

2.2.2 Comando `apropos`

O comando `apropos` provê assistência pela procura em todas as primeiras linhas dos tópicos de auxílio e retornando aquelas que contenham a palavra-chave especificada. O mais importante é que a palavra-chave não precisa ser um comando do SCILAB. Sua sintaxe é

```
apropos <palavra-chave>
```

onde `<palavra-chave>` é a cadeia de caracteres que será procurada nos comandos do SCILAB. Por exemplo, para informações sobre fatorização, tecla-se `apropos factorization`, resultando em uma tela com página dupla. A página da esquerda apresenta as classes de funções do SCILAB que contém pelo menos uma função com a descrição de `factorization`. A segunda página mostra dentro de uma classe quais as funções que apresentam a palavra `factorization` em sua descrição. A escolha da classe `Linear Algebra`, por exemplo, resulta em uma tela, como mostrada na Figura 2.4.

Apesar de a palavra `factorization` não ser um comando do SCILAB, ela foi encontrada na descrição de várias funções. Um *clique* sobre uma das linhas da página à esquerda escolhe a classe de funções e um *clique* sobre uma função da página à direita descreve aquela função específica, como um comando `help`. Por exemplo, a escolha de `chfact - sparse Cholesky factorization`, é equivalente ao comando `help chfact`.

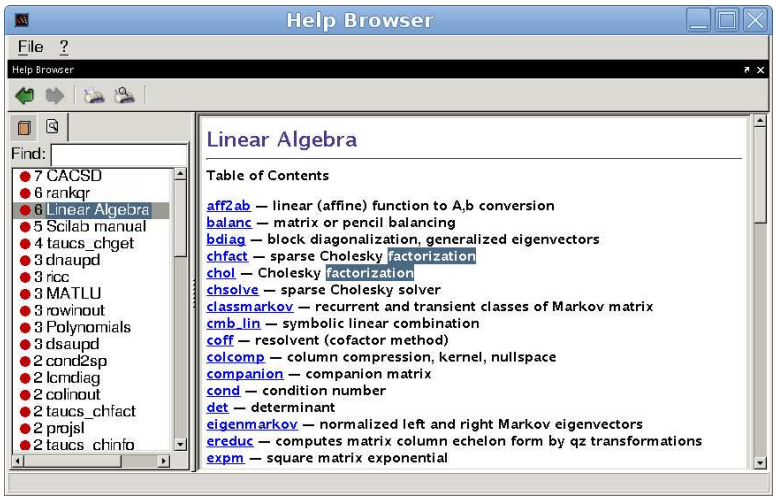


Figura 2.4: Janela do comando `apropos`.

2.2.3 Menu de barras

Quando o `SCILAB` for executado sem a opção `-nw`, um auxílio por intermédio de menu-dirigido é disponível selecionando a opção **Help Browser** indicada por um (**?**) dentro de um círculo (último ícone à direita) no menu de barras, como visto na Figura 2.1. Como o uso desta forma de assistência é bem intuitivo, o melhor a fazer é experimentar!

2.3 Exercícios

Seção 2.1 Janela de comando

2.1 Executar o programa SCILAB.

2.2 Verificar a diferença entre `,` e `;` nos comandos `%pi+1, %pi*10` e `%pi+1; %pi*10;`.

2.3 Testar o funcionamento das teclas ↑ e ↓.

2.4 Ver o funcionamento das teclas → e ←.

2.5 Verificar a diferença entre `lines(20)`, `lines()` e `lines(0)`.

Seção 2.2 Comandos de auxílio ao usuário

2.6 Quantos parâmetros tem a função erro `erf`?

2.7 Quais são os comandos utilizados para interpolação (*interpolation*)?

2.8 Qual o comando usado para calcular o determinante (*determinant*) de uma matriz?

2.9 Qual o comando para achar raízes (*roots*) de uma equação polinomial?

2.10 Comparar a soma dos autovalores (*eigenvalues*) com o traço (*trace*) de uma matriz de ordem qualquer. Fazer a mesma comparação usando uma matriz com elementos aleatórios.

Capítulo 3

Estruturas de dados

Neste capítulo serão apresentados alguns ítens básicos, tais como, constantes, variáveis, vetores, matrizes, hipermatrizes, polinômios e listas, o que tornará possível o uso imediato do SCILAB no modo interativo. Além disto, o conhecimento sobre as estruturas de dados é fundamental para um uso eficiente do programa.

3.1 Constantes

O SCILAB suporta três tipos de constantes: numéricas, lógicas e literais.

3.1.1 Numéricas

Uma constante numérica é formada por uma seqüência de dígitos que pode estar ou não precedida de um sinal positivo (`+`) ou um negativo (`-`) e pode conter um ponto decimal (`.`). Esta seqüência pode terminar ou não por uma das letras `e`, `E`, `d` ou `D` seguida de outra seqüência de dígitos precedida ou não de um sinal positivo (`+`) ou um negativo (`-`). Esta segunda seqüência é a potência de 10 pela qual a primeira seqüência é multiplicada. Por exemplo, `1.23e-1` significa $1{,}23_{\times}10^{-1} = 0{,}123$ e `4.567d2` é $4{,}567_{\times}10^{2} = 456{,}7$

```
-->1.23e-1
 ans  =
    0.123
-->4.567d2
 ans  =
    456.7
```

Algumas linguagens de programação requerem um tratamento especial para números complexos, o que não é o caso do SCILAB. Operações matemáticas com números complexos são escritas do mesmo modo como para números reais. Para indicar a parte imaginária basta

acrescentar os três caracteres (`*%i`), ou seja, multiplicar (`*`) por $\sqrt{-1} = i$ representado por (`%i`),

```
-->3+2*%i
 ans  =
    3. + 2.i
```

As variáveis reais e complexas em SCILAB ocupam 24 e 32 *bytes* de memória, respectivamente.

3.1.2 Lógicas

Uma constante lógica pode assumir apenas dois valores: `%t` ou `%T` *(true)* para *verdadeiro* e `%f` ou `%F` *(false)* para *falso*,

```
-->%t   // valor verdadeiro
 ans  =
  T
-->%f   // valor falso
 ans  =
  F
```

3.1.3 Literais

Uma constante literal é composta por uma cadeia de caracteres em vez de números ou *verdadeiro* e *falso*. A cadeia de caracteres deve estar delimitada por aspas (') ou apóstrofos (").

```
-->'abcde'   // um valor literal
 ans  =
 abcde
```

3.2 Variáveis

Uma variável é uma posição da memória do computador utilizada para armazenar uma informação, sendo representada por um identificador.

3.2.1 Regras para nomes de variáveis

Como qualquer outra linguagem de programação, o SCILAB tem regras a respeito do nome de variáveis, conforme mostrado na Tabela 3.1.

Tabela 3.1: Regras para nomes de variáveis.

Regra	Comentário
conter até 24 caracteres	caracteres além do 24º são ignorados;
começar com uma letra seguida de letras, números ou dos caracteres _, #, $, ! e ?	alguns caracteres de pontuação são permitidos;
tamanho da letra é diferenciador	`raiz`, `Raiz` e `RAIZ` são três variáveis distintas.

3.2.2 Comando de atribuição

O SCILAB é um interpretador de expressões. A expressão fornecida é analisada sintaticamente e se estiver correta então será avaliada. O resultado é atribuído à uma variável por intermédio do comando de atribuição

```
<variável> = <expressão>
```

Por exemplo,

```
-->a=10.2+5.1
 a  =
    15.3
```

Quando o comando de atribuição for finalizado pelo caracter (`;`) então o resultado é atribuído à variável, porém o seu conteúdo não é exibido,

```
-->b=5-2;   // atribuicao sem exibir o resultado
-->b   // conteudo da variavel b
 ans  =
    3.
```

Se o nome da variável e o sinal de atribuição (`=`) forem omitidos então o resultado será dado à variável *default* `ans` (*answer*),

```
-->8/5
 ans  =
    1.6
```

3.2.3 Variáveis especiais

O SCILAB tem diversas variáveis especiais, as quais são consideradas como pré-definidas, não podendo ser alteradas ou removidas (com excessão de `ans`). Elas estão listadas na Tabela 3.2.

Para obter os valores destas variáveis especiais, faz-se

```
-->%e, %eps, %i, %inf, %nan, %pi, %s
```

Tabela 3.2: Variáveis especiais do SCILAB.

Variável	Valor
`ans`	nome de variável *default* usada para resultados;
`%e`	base do logaritmo natural, $e = 2{,}71828\ldots$
`%eps`	menor número de ponto flutuante que adicionado a 1 resulta um número maior que 1. Seu valor é $\epsilon = 2^{-52} \approx 2{,}2204_{\times}10^{-16}$ em computadores com aritmética de ponto flutuante IEEE;
`%i`	$i = \sqrt{-1}$;
`%inf`	infinito, por exemplo, $1/0$;
`%nan`	não é um número (Not-A-Number), por exemplo, $0/0$;
`%pi`	$\pi = 3,14159\ldots$
`%s`	usada como variável de polinômio.

```
%e  =
   2.7182818
%eps  =
   2.220D-16
%i  =
   i
%inf  =
  Inf
%nan  =
  Nan
%pi  =
   3.1415927
%s  =
   s
```

3.3 Vetores

Vetor ou arranjo é um conjunto de variáveis homogêneas (conteúdo de mesmo tipo) identificadas por um mesmo nome e individualizadas por meio de um índice. No SCILAB, um vetor é definido de várias formas, por exemplo, elemento por elemento, separados por espaço em branco ou vírgula e delimitados pelos caracteres (`[`) e (`]`),

`<vetor>` = [e_1, e_2, ... , e_n]

sendo `<vetor>` o nome da variável e e_i o seu i-ésimo elemento,

```
-->a = [5 1.5,-0.3]   // elementos separados por espaço em branco ou vírgula
 a  =
   5.    1.5  - 0.3
```

O vetor a possui os elementos a(1)=5, a(2)=1.5 e a(3)=-0.3. Se os valores do vetor forem igualmente espaçados ele pode ser definido por intermédio da expressão

```
<vetor> = <valor_inicial>:<incremento>:<valor_final>    .
```

Para gerar um vetor com o primeiro elemento igual a 10, o segundo igual a 15, o terceiro igual a 20 e assim, sucessivamente, até o último igual a 40 basta o comando

```
-->b = 10:5:40   // valor inicial = 10, incremento = 5 e valor final = 40
 b  =
    10.    15.    20.    25.    30.    35.    40.
```

Se o incremento desejado for igual a 1 então ele poderá ser omitido,

```
-->u = 5:9   // valor inicial = 5, incremento = 1 e valor final = 9
 u  =
    5.    6.    7.    8.    9.
```

Em vez de usar incremento, um arranjo pode também ser construído definindo o número desejado de elementos na função linspace, cuja sintaxe é

```
  <vetor> = linspace(<valor_inicial>,<valor_final>,<número_de_elementos>)    .
```

Assim, para criar um arranjo com o primeiro elemento igual a 10, o último igual a 40 e possuindo 7 elementos,

```
// valor inicial = 10, valor final = 40 e elementos = 7
-->c = linspace(10,40,7)
 c  =
    10.    15.    20.    25.    30.    35.    40.
```

Isto produz um vetor idêntico a b=10:5:40 definido acima. Se no comando linspace o parâmetro <número_de_elementos> for omitido então serão gerados 100 pontos. No caso de números complexos, os incrementos são feitos separadamente para a parte real e para a parte imaginária,

```
// valor inicial = 1+i, valor final = 2+4i e elementos = 5
-->d = linspace(1+%i,2+4*%i,5)
 d  =
    1. + i      1.25 + 1.75i    1.5 + 2.5i    1.75 + 3.25i    2. + 4.i
```

Os elementos podem ser acessados individualmente; por exemplo, para o vetor c definido acima

```
-->c(2)   // segundo elemento de c
 ans  =
    15.
```

ou em blocos usando os comandos de definição de arranjos,

```
-->c(3:5)   // terceiro ao quinto elementos
 ans  =
    20.    25.    30.
```

Lembrando que `5:-2:1` gera a seqüência `5, 3, 1`, então,

```
-->c(5:-2:1)   // quinto, terceiro e primeiro elementos
 ans  =
    30.    20.    10.
```

O endereçamento indireto é também possível, permitindo referenciar os elementos de um vetor na ordem definida pelos elementos de um outro vetor. Por exemplo, sejam os vetores `c`, definido acima, e `ind`

```
-->ind = 1:2:7   // valor inicial = 1, incremento = 2 e valor final = 7
 ind  =
    1.    3.    5.    7.
```

Assim, `ind([4 2])` produz um vetor contendo os elementos `4` e `2` do vetor `ind`, ou seja, `[7 3]`. Por sua vez,

```
-->c(ind([4 2]))   // vetor c indexado pelo vetor ind
 ans  =
    40.    20.
```

é equivalente a `c([7 3])`, ou seja, o vetor gerado contém os elementos `7` e `3` do vetor `c`.

Nos exemplos acima, os vetores possuem uma linha e várias colunas, por isto são também chamados vetores linha. Do mesmo modo, podem existir vetores coluna, ou seja, vetores com várias linhas e uma única coluna. Para criar um vetor coluna elemento por elemento estes devem estar separados por (`;`)

`<vetor>` = [e_1 ; e_2 ; ... ; e_n]

Deste modo, para gerar um vetor coluna com os elementos `1.5`, `-3.2` e `-8.9`,

```
-->v = [1.5;-3.2;-8.9]   // elementos separados por ponto-e-vírgula
 v  =
    1.5
  - 3.2
  - 8.9
```

Por isto, separando os elementos de um vetor por brancos ou vírgulas são especificados os elementos em diferentes colunas (vetor linha). Por outro lado, separando os elementos por ponto-e-vírgula especifica-se os elementos em diferentes linhas (vetor coluna). Para transformar um vetor linha em vetor coluna e vice-versa, usa-se o operador de transposição (`'`),

```
-->x = v'   // vetor linha obtido pela transposição de vetor coluna
 x  =
    1.5  - 3.2  - 8.9
-->y = (1:3)'   // vetor coluna obtido pelo operador de transposição
 y  =
    1.
    2.
    3.
```

No caso do vetor ser complexo, a transposição é obtida pelo operador (.'), pois o uso do operador (') resultará em um complexo conjugado transposto. Por exemplo, seja o vetor complexo,

```
// valor inicial = 1+0,1i, incremento = 1+0,1i e valor final = 3+0,3i
-->z = (1:3)+(0.1:0.1:0.3)*%i
 z  =
    1. + 0.1i    2. + 0.2i    3. + 0.3i
```

o transposto é

```
-->t = z.'   // vetor transposto
 t  =
    1. + 0.1i
    2. + 0.2i
    3. + 0.3i
```

e o complexo conjugado transposto

```
-->cc = z'   // vetor complexo conjugado transposto
 cc  =
    1. - 0.1i
    2. - 0.2i
    3. - 0.3i
```

No caso do vetor não ser complexo a transposição pode ser feita usando (') ou (.'). A função `length` é utilizada para saber o comprimento de um vetor, ou seja, quantos elementos ele possui

```
-->l = length(z)
 l  =
    3.
```

Por fim, deve-se notar a importância da posição do caracter branco na definição de um vetor

```
-->p=[1+2]
 p  =
    3.
-->p=[1 +2]
```

```
 p  =
    1.    2.
-->p=[1+ 2]
 p  =
    3.
-->p=[1 + 2]
 p  =
    3.
```

3.4 Matrizes

As matrizes são arranjos bidimensionais homogêneos e constituem as estruturas fundamentais do SCILAB e por isto existem várias maneiras de manipulá-las. Uma vez definidas, elas podem ser modificadas de várias formas, como por inserção, extração e rearranjo.

Similarmente aos vetores, para construir uma matriz os elementos de uma mesma linha devem estar separados por branco ou vírgula e as linhas separadas por ponto-e-vírgula ou **Enter** (ou **Return**),

<matriz> = [e_{11} e_{12} ... e_{1n} ; e_{21} e_{22} ... e_{2n} ; ... ; e_{m1} e_{m2} ... e_{mn}]

3.4.1 Construção e manipulação de matrizes

Para criar uma matriz `A` com 2 linhas e 3 colunas,

```
-->A = [3 2 -5; 4 7 9]   // atribui os elementos da matriz A
 A  =
    3.    2.  - 5.
    4.    7.    9.
```

Para modificar um elemento basta atribuir-lhe um novo valor,

```
-->A(1,2)=8   // altera o elemento da linha 1 e coluna 2
 A  =
    3.    8.  - 5.
    4.    7.    9.
```

Cuidado: se for atribuído um valor a um elemento não existente, ou seja, além dos elementos da matriz então o SCILAB aumenta esta matriz, automaticamente, preenchendo-a com valores nulos de forma a matriz permanecer retangular,

```
-->A(3,6)=1   // atribui valor ao elemento da linha 3 e coluna 6
 A  =
    3.    8.  - 5.    0.    0.    0.
    4.    7.    9.    0.    0.    0.
    0.    0.    0.    0.    0.    1.
```

Seja agora a matriz quadrada B de ordem 3,

```
-->B = [1 2 3; 4 5 6; 7 8 9]   // atribui os elementos da matriz B
 B  =
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
```

De modo similar aos vetores, os elementos de uma matriz podem ser referenciados individualmente, tal como,

```
-->B(2,3)   // elemento da linha 2 e coluna 3
 ans  =
    6.
```

ou em conjuntos, neste caso usando a notação de vetor. Por exemplo, os elementos das linhas 1 e 3 e coluna 2,

```
-->B([1 3],2)   // elementos das linhas 1 e 3 da coluna 2
 ans  =
    2.
    8.
```

A notação de vetor, <valor_inicial>:<incremento>:<valor_final>, também pode ser usada ou até mesmo linspace. Lembrando que 3:-1:1 gera a seqüência 3, 2 e 1 e 1:3 produz 1, 2 e 3, então o comando,

```
-->C = B(3:-1:1,1:3)   // obtém a matriz C a partir da inversão das linhas de B
 C  =
    7.    8.    9.
    4.    5.    6.
    1.    2.    3.
```

cria uma matriz C a partir das linhas 3, 2 e 1 e colunas 1, 2 e 3 de B, ou seja, cria uma matriz C a partir das linhas de B em ordem contrária. Considerando que são referenciadas todas as 3 colunas de B, a notação simplificada (:) pode ser igualmente usada em vez de 1:3

```
-->C = B(3:-1:1,:)   // notação simplificada
 C  =
    7.    8.    9.
    4.    5.    6.
    1.    2.    3.
```

Para criar uma matriz D a partir das linhas 1 e 2 e colunas 1 e 3 de B, faz-se

```
-->D = B(1:2,[1 3])
 D  =
    1.    3.
    4.    6.
```

Para construir uma matriz `E` a partir da matriz `B` seguida da coluna 2 de `C` seguida ainda de uma coluna com os elementos iguais a `3`,

```
-->E = [B C(:,2) [3 3 3]']
 E  =
    1.    2.    3.    8.    3.
    4.    5.    6.    5.    3.
    7.    8.    9.    2.    3.
```

Para remover uma linha ou coluna de uma matriz usa-se a matriz vazia (`[]`). Para remover a coluna 3 de `E`,

```
-->E(:,3) = []   // remoção da coluna 3 da matriz E
 E  =
    1.    2.    8.    3.
    4.    5.    5.    3.
    7.    8.    2.    3.
```

E posteriormente para remover a linha 1,

```
-->E(1,:) = []   // remoção da linha 1 da matriz E
 E  =
    4.    5.    5.    3.
    7.    8.    2.    3.
```

Quando o símbolo (`:`) for usado sozinho para referenciar os dois índices de uma matriz então é gerado um vetor constituído pelas colunas da matriz. Seja o vetor `vet` obtido das colunas da matriz `E` acima,

```
-->vet = E(:)   // vetor formado pelas colunas da matriz E
 vet  =
    4.
    7.
    5.
    8.
    5.
    2.
    3.
    3.
```

O símbolo (`$`) é utilizado para indicar o índice da última linha ou coluna da matriz,

```
-->E(1,$-1)   // elemento da primeira linha e penultima coluna da matriz E
 ans  =
    5.
```

3.4.2 Funções matriciais básicas

O SCILAB tem funções que se aplicam aos vetores e às matrizes. Algumas destas funções básicas são mostradas na Tabela 3.3.

Tabela 3.3: Exemplos de funções básicas do SCILAB.

Função	Descrição
`size`	dimensão da matriz;
`length`	número de elementos;
`sum`	soma dos elementos;
`prod`	produto dos elementos;
`max`	maior elemento;
`min`	menor elemento;
`mean`	média aritmética;
`stdev`	desvio padrão.

Função size

A função `size` é usada para fornecer o número de linhas e colunas de uma matriz. Ela pode ser usada de duas formas:

```
-->t = size(E)
 t  =
    2.    4.
```

onde a variável `t` é um vetor linha com duas posições, contendo o número de linhas e colunas de `E`, respectivamente. A outra forma é

```
-->[lin,col] = size(E)
 col  =
    4.
 lin  =
    2.
```

onde as variáveis simples `col` e `lin` contém o número de colunas e linhas de `E`, respectivamente.

Função length

Se a função `length` for usada em uma matriz ela fornecerá o número de elementos da matriz,

```
-->e = length(E)
 e  =
    8.
```

Se for acrescentado o argumento `'r'` (*row*) então a função produz um vetor linha obtido dos elementos das colunas da matriz. Se o argumento for `'c'` (*column*) então será gerado um vetor coluna a partir dos elementos das linhas da matriz. Se não for usado o argumento extra então a função produz um escalar utilizando todos elementos da matriz. Para a matriz `A`,

```
-->A = [1 2 3;4 5 6;7 8 9]   // define os elementos da matriz A
 A  =
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
```

Função sum

```
// vetor linha com i-ésimo elemento = soma dos elementos da coluna i da matriz A
-->sum(A,'r')
 ans  =
    12.    15.    18.
// vetor coluna com i-ésimo elemento = soma dos elementos da linha i da matriz A
-->sum(A,'c')
 ans  =
    6.
    15.
    24.
-->sum(A)   // escalar igual a soma dos elementos da matriz A
 ans  =
    45.
```

Função prod

```
// vetor linha com i-ésimo elemento = produto dos elementos da coluna i de A
-->prod(A,'r')
 ans  =
    28.    80.    162.
// vetor coluna com i-ésimo elemento = produto dos elementos da linha i de A
-->prod(A,'c')
 ans  =
    6.
    120.
    504.
-->prod(A)   // escalar igual ao produto dos elementos da matriz A
 ans  =
    362880.
```

Função max

```
// vetor linha com i-ésimo elemento = maior elemento da coluna i da matriz A
-->max(A,'r')
```

```
 ans  =
    7.    8.    9.
// vetor coluna com i-ésimo elemento = maior elemento da linha i da matriz A
-->max(A,'c')
 ans  =
    3.
    6.
    9.
-->max(A)   // escalar igual ao maior elemento da matriz A
 ans  =
    9.
```

Função min

```
// vetor linha com i-ésimo elemento = menor elemento da coluna i da matriz A
-->min(A,'r')
 ans  =
    1.    2.    3.
// vetor coluna com i-ésimo elemento = menor elemento da linha i da matriz A
-->min(A,'c')
 ans  =
    1.
    4.
    7.
-->min(A)   // escalar igual ao menor elemento da matriz A
 ans  =
    1.
```

Função mean

```
// vetor linha com i-ésimo elemento = média dos elementos da coluna i da matriz A
-->mean(A,'r')
 ans  =
    4.    5.    6.
// vetor coluna com i-ésimo elemento = média dos elementos da linha i da matriz A
-->mean(A,'c')
 ans  =
    2.
    5.
    8.
-->mean(A)   // escalar igual a média dos elementos da matriz A
 ans  =
    5.
```

Função stdev

```
//vetor linha com i-ésimo elemento = desvio padrão dos elementos da coluna i de A
-->stdev(A,'r')
```

```
 ans  =
    3.    3.    3.
//vetor coluna com i-ésimo elemento = desvio padrão dos elementos da linha i de A
-->stdev(A,'c')
 ans  =
    1.
    1.
    1.
-->stdev(A)   // escalar igual ao desvio padrão dos elementos da matriz A
 ans  =
    2.7386128
```

3.4.3 Funções para manipulação de matrizes

O SCILAB possui várias funções para manipulação de matrizes dentre as quais destacam-se aquelas mostradas na Tabela 3.4.

Tabela 3.4: Algumas funções para manipulação de matrizes.

Função	Descrição
`diag`	inclui ou obtém a diagonal de uma matriz;
`tril`	obtém a parte triangular inferior de uma matriz;
`triu`	obtém a parte triangular superior de uma matriz;
`matrix`	altera a forma de uma matriz;
`sparse`	cria uma matriz esparsa.
`full`	cria uma matriz densa.
`gsort`	ordena os elementos de uma matriz.

Função diag

Inclui ou obtém a diagonal de uma matriz, sendo sua sintaxe,

```
<saída> = diag(<argumento> [,<k>])
```

de modo que, se `<argumento>` for um vetor de dimensão `n` então `<saída>` será uma matriz diagonal contendo os elementos de `<argumento>` na diagonal principal; o parâmetro inteiro opcional `<k>` faz com que `<saída>` seja uma matriz de ordem `n + abs(<k>)` com os elementos de `<argumento>` ao longo de sua `<k>`-ésima diagonal: para `<k> = 0` (*default*) é a diagonal principal, `<k> > 0` é a `<k>`-ésima diagonal superior (superdiagonal) e `<k> < 0` é a `<k>`-ésima diagonal inferior (subdiagonal). Por exemplo,

```
-->d = [1 2 3]   // define um vetor com 3 elementos
 d  =
    1.    2.    3.
-->D = diag(d)   // cria matriz diagonal de ordem 3 com elementos do vetor d
 D  =
```

```
    1.    0.    0.
    0.    2.    0.
    0.    0.    3.
-->D5 = diag(d,2) // cria matriz de ordem 5 com elementos de d na superdiagonal 2
 D5  =
    0.    0.    1.    0.    0.
    0.    0.    0.    2.    0.
    0.    0.    0.    0.    3.
    0.    0.    0.    0.    0.
    0.    0.    0.    0.    0.
```

Por outro lado, se o <argumento> for uma matriz então diag cria um vetor coluna com os elementos da <k>-ésima diagonal da matriz,

```
-->A = [1 2 3; 4 5 6; 7 8 9]   // define a matriz A
 A  =
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
-->d1 = diag(A,-1)   // cria vetor com elementos da subdiagonal 1 de A
 d1  =
    4.
    8.
-->d0 = diag(A)   // cria vetor com elementos da diagonal principal de A
 d0  =
    1.
    5.
    9.
// cria matriz diagonal com elementos da superdiagonal 1 de A
-->D = diag(diag(A,1))
 D  =
    2.    0.
    0.    6.
```

Função tril

Obtém a parte triangular inferior de uma matriz, cuja sintaxe é

```
<saída> = tril(<argumento> [,<k>])
```

sendo <argumento> uma matriz e o parâmetro inteiro opcional <k> faz com que <saída> seja uma matriz de mesma dimensão de <argumento> de obtida a partir da <k>-ésima diagonal de <argumento> e os elementos abaixo dela: para <k> = 0 (*default*) diagonal principal, <k> > 0 <k>-ésima diagonal superior e <k> < 0 <k>-ésima diagonal inferior. Por exemplo,

```
-->M = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]   // define matriz M
 M  =
```

```
    1.     2.     3.     4.
    5.     6.     7.     8.
    9.     10.    11.    12.
    13.    14.    15.    16.
// cria matriz triangular inferior a partir da superdiagonal 1 de M
-->L1 = tril(M,1)
 L1  =
    1.     2.     0.     0.
    5.     6.     7.     0.
    9.     10.    11.    12.
    13.    14.    15.    16.
// cria matriz triangular inferior a partir da subdiagonal 2 de M
-->L2 = tril(M,-2)
 L2  =
    0.     0.     0.    0.
    0.     0.     0.    0.
    9.     0.     0.    0.
    13.    14.    0.    0.
```

Função triu

Obtém a parte triangular superior de uma matriz, sendo sua sintaxe

```
<saída> = triu(<argumento> [,<k>])
```

sendo `<argumento>` uma matriz e o parâmetro inteiro opcional `<k>` faz com que `<saída>` seja uma matriz de mesma dimensão de `<argumento>` obtida a partir da `<k>`-ésima diagonal de `<argumento>` e os elementos acima dela: para `<k> = 0` (*default*) diagonal principal, `<k> > 0` `<k>`-ésima diagonal superior e `<k> < 0` `<k>`-ésima diagonal inferior. Para a matriz `M` acima,

```
// cria matriz triangular superior a partir da subdiagonal 1 de M
-->U1 = triu(M,-1)
 U1  =
    1.    2.     3.     4.
    5.    6.     7.     8.
    0.    10.    11.    12.
    0.    0.     15.    16.
// cria matriz triangular superior a partir da superdiagonal2 de M
-->U2 = triu(M,2)
 U2  =
    0.    0.    3.    4.
    0.    0.    0.    8.
    0.    0.    0.    0.
    0.    0.    0.    0.
```

Função matrix

Altera a forma de uma matriz, cuja sintaxe é

```
<saída> = matrix(<matriz>, <linhas>, <colunas>)
```

.

Esta função cria uma matriz `<saída>` com número de linhas dados por `<linhas>` e número de colunas igual a `<colunas>`, a partir de `<matriz>` com (`<linhas>` × `<colunas>`) elementos selecionados no sentido das colunas. Por exemplo, para a matriz `B` de dimensão 2×3,

```
-->B = [1 2 3;4 5 6]   // define matriz B de dimensao 2 x 3
 B  =
    1.    2.    3.
    4.    5.    6.
-->c = matrix(B,1,6)   // cria matriz (vetor linha) de dimensao 1 x 6
 c  =
    1.    4.    2.    5.    3.    6.
-->D = matrix(B,3,2)   // cria matriz de dimensao 3 x 2
 D  =
    1.    5.
    4.    3.
    2.    6.
```

Função sparse

Constrói uma matriz esparsa de dois modos distintos. No primeiro

```
<saída> = sparse(<matriz>)
```

cria uma matriz esparsa `<saída>` armazenando somente os elementos não nulos de `<matriz>` na forma densa. Para a matriz `E`,

```
-->E = [2 0 -1; 0 5 6; 7 0 9]   // define matriz com alguns elementos nulos
 E  =
    2.    0.  - 1.
    0.    5.    6.
    7.    0.    9.
-->S = sparse(E)   // cria matriz esparsa a partir da matriz densa E
 S  =
(    3,    3) sparse matrix
(    1,    1)        2.
(    1,    3)      - 1.
(    2,    2)        5.
(    2,    3)        6.
(    3,    1)        7.
(    3,    3)        9.
```

Os tripletos acima indicam (linha, coluna, valor), ou seja, linha = 1, coluna = 1, valor = 2; linha = 1, coluna = 3, valor = −1 e assim sucessivamente. No segundo modo,

```
<saída> = sparse(<linha_coluna>, <valor>)
```

constrói uma matriz esparsa `<saída>` a partir de uma matriz `<linha_coluna>` com duas colunas contendo os índices dos elementos não nulos e do vetor `<valor>` com os valores desses elementos. Por exemplo,

```
// define matriz de dimensao 7 x 2
-->lincol = [1 1; 1 3; 2 2; 2 4; 3 3; 4 1; 5 5]
 lincol  =
    1.    1.
    1.    3.
    2.    2.
    2.    4.
    3.    3.
    4.    1.
    5.    5.
-->vet = [2 -1 5 4 3 8 -9]   // define vetor com 7 elementos
 vet  =
    2.  - 1.    5.    4.    3.    8.  - 9.
-->F = sparse(lincol,vet)   // cria matriz esparsa a partir de lincol e vet
 F  =
(    5,    5) sparse matrix
(    1,    1)        2.
(    1,    3)      - 1.
(    2,    2)        5.
(    2,    4)        4.
(    3,    3)        3.
(    4,    1)        8.
(    5,    5)      - 9.
```

Função full

O comando `<matriz> = full(<esparsa>)` converte a matriz `<esparsa>` para a sua representação densa `<matriz>`. Para a matriz `F` acima,

```
-->FF = full(F)   // cria matriz densa a partir de esparsa
 FF  =
    2.    0.  - 1.    0.    0.
    0.    5.    0.    4.    0.
    0.    0.    3.    0.    0.
    8.    0.    0.    0.    0.
    0.    0.    0.    0.  - 9.
```

Função gsort

Uma função, particularmente, útil é a `gsort` que permite a ordenação dos elementos de um vetor ou de uma matriz. Sua sintaxe é

```
[<matriz_saída>[,<índice>]] = gsort(<matriz_entrada>[,<opção>][,<direção>])
```

sendo `<matriz_entrada>` um vetor ou uma matriz contendo elementos numéricos ou literais; `[,<opção>]` (opcional) é uma cadeia de caracteres que especifica qual o tipo de ordenação a ser realizada, `'r'`: por linha, `'c'`: por coluna, `'g'` (*default*): todos os elementos da matriz são ordenados; `[,direção]` (opcional) indica a direção de ordenação, `'i'`: ordem crescente e `'d'`(*default*): ordem decrescente. Os parâmetros de saída são `<matriz_saída>`, com o mesmo tipo e dimensão de `<matriz_entrada>`, contém o resultado da ordenação e o vetor opcional `<índice>` contém os índices originais, tal que
`<matriz_saída>` = `<matriz_entrada>(<índice>)`.
Para o vetor `v` e a matriz `M`,

```
-->v = [3 9 5 6 2]
 v  =
    3.    9.    5.    6.    2.
-->M = [2 5 1; 3 8 9; 7 6 4]
 M  =
    2.    5.    1.
    3.    8.    9.
    7.    6.    4.
```

tem-se os resultados,

```
-->gsort(v,'g','i')   // ordena os elementos do vetor v em ordem crescente
 ans  =
    2.    3.    5.    6.    9.
// ordena os elementos do vetor v em ordem decrescente
-->[w,ind] = gsort(v,'g','d')
 ind  =
    2.    4.    3.    1.    5.
 w  =
    9.    6.    5.    3.    2.
```

Para a ordenação decrescente requerida, `w = v(ind)`, isto é, `w(1) = v(ind(1))`, `w(2) = v(ind(2))`, ... , `w(5) = v(ind(5))`.

```
-->B = gsort(M,'r','i')   // ordena as linhas da matriz M em ordem crescente
 B  =
    2.    5.    1.
    3.    6.    4.
    7.    8.    9.
```

```
-->D = gsort(M,'c','d')   // ordena as colunas da matriz M em ordem decrescente
 D  =
    5.    2.    1.
    9.    8.    3.
    7.    6.    4.
```

3.4.4 Matrizes elementares

O SCILAB fornece várias matrizes elementares de grande utilidade, como as mostradas na Tabela 3.5. O número de parâmetros providos fornecem as dimensões da matriz. Se o parâmetro for uma matriz então será criada uma matriz de igual dimensão.

Tabela 3.5: Algumas matrizes elementares do SCILAB.

Função	Descrição da matriz
`zeros`	nula;
`ones`	elementos iguais a `1`;
`eye`	identidade ou parte dela;
`grand`	elementos aleatórios com dada distribuição.

Seja a matriz,

```
-->P = [1 2 3; 4 5 6]   // define uma matriz de dimensao 2 x 3
 P  =
    1.    2.    3.
    4.    5.    6.
```

Função zeros

O comando

`<matriz> = zeros(<linhas>, <colunas>)`

gera uma matriz de dimensão (`<linhas>` × `<colunas>`), com elementos nulos e `<matriz> = zeros(<argumento>)` cria uma matriz com elementos nulos de mesma dimensão da matriz `<argumento>`,

```
-->z = zeros(1,4)   // cria matriz nula de dimensao 1 x 4
 z  =
    0.    0.    0.    0.
```

Função ones

O comando

`<matriz> = ones(<linhas>, <colunas>)`

cria uma matriz de dimensão (`<linhas>` × `<colunas>`), com elementos iguais a `1` e `<matriz> = ones(<argumento>)` gera uma matriz com elementos iguais a `1` com a mesma dimensão da matriz `<argumento>`,

```
-->U = ones(P)   // cria matriz de 1's com mesma dimensao da matriz P
 U  =
    1.    1.    1.
    1.    1.    1.
```

Função eye

O comando

```
<matriz> = eye(<linhas>, <colunas>)
```

gera uma matriz identidade (com `1` na diagonal principal e `0` fora da diagonal) de dimensão (`<linhas>` × `<colunas>`). Por sua vez, `<matriz> = eye(<argumento>)` cria uma matriz identidade de mesma dimensão da matriz `<argumento>`,

```
-->I = eye(P'*P)   // cria matriz identidade com a mesma dimensao de P'P
 I  =
    1.    0.    0.
    0.    1.    0.
    0.    0.    1.
```

Função grand

O comando

```
<alea> = grand(<linhas>, <colunas>, <tipo_dist> [,<param_1>,...,<param_n>])
```

gera a matriz `<alea>` com elementos aleatórios de dimensão (`<linhas>` × `<colunas>`) com o tipo de distribuição dado pela cadeia de caracteres `<tipo_dist>`, sendo `<param_i>` um conjunto de parâmetros opcionais necessários para definir uma distribuição específica. Por exemplo,

Distribuição uniforme no intervalo `[0,1)`

```
// cria matriz de dimensao 3 x 5 com elementos aleatórios com
// distribuição uniforme no intervalo [0,1)
-->U = grand(3,5,'def')
 U  =
    0.8147237    0.8350086    0.9133759    0.3081671    0.2784982
    0.135477     0.1269868    0.2210340    0.0975404    0.1883820
    0.9057919    0.9688678    0.6323592    0.5472206    0.5468815
```

Distribuição uniforme no intervalo `[Inf,Sup)`

```
// cria matriz de dimensao 3 x 4 com elementos aleatórios com
// distribuição uniforme no intervalo [10,12)
-->R = grand(3,4,'unf',10,12)
 R  =
    11.985763    11.929777    11.451678    11.914334
    11.915014    11.93539     11.941186    10.219724
    11.992923    10.315226    11.962219    10.970751
```

Distribuição normal com média `Med` *e desvio padrão* `Despad`

```
// cria matriz de dimensao 2 x 4 com elementos aleatórios com
// distribuição normal com média = 1,5 e desvio padrão = 2,6
-->N = grand(2,4,'nor',1.5,2.6)
 N  =
    2.761686     0.2303274  - 3.5420607  - 1.4501598
    2.7455511  - 1.1118334    1.0909685    1.94921
```

Distribuição normal multivariada com média `Med` *e covariância* `Cov`
sendo `Med` uma matriz `(m x 1)` e `Cov` uma matriz `(m x m)` simétrica positiva definida.

```
-->M = grand(5,'mn',[0 0]',[1 0.5;0.5 1])
 M  =
    1.0745727    0.9669574  - 0.7460049    0.8510105  - 0.2161951
    2.2602797    0.7828132  - 0.2634706    1.1711612    0.4133844
```

Distribuição de Poisson com média `Med`

```
// cria matriz de dimensao 3 x 5 com elementos aleatórios com
// distribuição de Poisson com média = 3,5
-->P = grand(3,5,'poi',3.5)
 P  =
    5.    3.    2.    4.    2.
    5.    3.    2.    5.    2.
    3.    4.    2.    1.    6.
```

Gera `n` *permutações aleatórias de um vetor coluna de tamanho* `m`

```
// cria matriz de dimensão (length(vet) x 8) com 8 permutações no vetor vet de
// tamanho 5
-->vet = [1 2 3 4 5]'   // define vetor coluna de tamanho 5
 vet  =
    1.
    2.
    3.
    4.
    5.
-->V = grand(8,'prm',vet)
 V  =
    3.    1.    2.    4.    5.    3.    5.    3.
    2.    4.    4.    2.    2.    5.    2.    2.
    1.    3.    5.    5.    1.    1.    4.    4.
    5.    5.    1.    3.    3.    2.    1.    5.
    4.    2.    3.    1.    4.    4.    3.    1.
```

A função `grand` produz números pseudo-aleatórios e a seqüência gerada de qualquer distribuição é determinada pelo estado do gerador. Quando o SCILAB for ativado, é atribuído um valor inicial ao estado, o qual é alterado à cada chamada da função `grand`. Para atribuir um valor ao estado do gerador quando esta função já tiver sido executada usa-se o comando `grand('setsd',<semente>)`, sendo `<semente>` o valor inicial desejado do estado. Por exemplo,

```
-->grand('setsd',1)   // atribui 1 ao estado
-->a = grand(1,5,'def')   // gera uma sequência aleatória uniforme
 a  =
    0.417022    0.9971848    0.7203245    0.9325574    0.0001144
-->b = grand(1,5,'def')   // gera outra sequência aleatória uniforme
 b  =
    0.1281244    0.3023326    0.9990405    0.1467559    0.2360890
-->grand('setsd',1)   // atribui 1 novamente ao estado
-->c = grand(1,5,'def')   // gera outra sequência aleatória uniforme
 c  =
    0.417022    0.9971848    0.7203245    0.9325574    0.0001144
-->c - a   // mostra que as sequências c e a são idênticas
 ans  =
    0.    0.    0.    0.    0.
```

A função `grand` permite a geração de vinte diferentes tipos de distribuição de números pseudo-aleatórios, utilizando seis diferentes geradores. Para maiores informações use o comando de auxílio `help grand`.

3.5 Hipermatrizes

Hipermatrizes são variáveis homogêneas com mais de duas dimensões. Elas podem ser criadas pela função `hypermat(<dimensão>, <valores>)`, onde o vetor `<dimensão>` define as dimensões da hipermatriz e a matriz `<valores>` define os elementos. Por exemplo, para criar a hipermatriz `H` com dimensões $2 \times 3 \times 4$, ou seja, composta por 4 submatrizes de dimensões 2×3, faz-se

```
-->mat1 = [1.1 1.2 1.3; 1.4 1.5 1.6]   // define submatriz 1
 mat1  =
    1.1    1.2    1.3
    1.4    1.5    1.6
-->mat2 = [2.1 2.2 2.3; 2.4 2.5 2.6]   // define submatriz 2
 mat2  =
    2.1    2.2    2.3
    2.4    2.5    2.6
-->mat3 = [3.1 3.2 3.3; 3.4 3.5 3.6]   // define submatriz 3
 mat3  =
```

```
    3.1    3.2    3.3
    3.4    3.5    3.6
-->mat4 = [4.1 4.2 4.3; 4.4 4.5 4.6]   // define submatriz 4
 mat4  =
    4.1    4.2    4.3
    4.4    4.5    4.6
// cria hipermatriz H de dimensao 2 x 3 x 4 a partir de 4 submatrizes
-->H = hypermat([2 3 4],[mat1 mat2 mat3 mat4])
 H  =
(:,:,1)
    1.1    1.2    1.3
    1.4    1.5    1.6
(:,:,2)
    2.1    2.2    2.3
    2.4    2.5    2.6
(:,:,3)
    3.1    3.2    3.3
    3.4    3.5    3.6
(:,:,4)
    4.1    4.2    4.3
    4.4    4.5    4.6
```

As hipermatrizes podem ser manipuladas como as matrizes bidimensionais. Assim, para alterar para `12.3` os elementos da posição `(2,1)` das submatrizes `1` e `4` da hipermatriz `H` acima,

```
-->H(2,1,[1 4]) = 12.3   / atribui o valor 12,3 aos elementos H(2,1,1) e H(2,1,4)
 H  =
(:,:,1)
    1.1     1.2    1.3
    12.3    1.5    1.6
(:,:,2)
    2.1    2.2    2.3
    2.4    2.5    2.6
(:,:,3)
    3.1    3.2    3.3
    3.4    3.5    3.6
(:,:,4)
    4.1     4.2    4.3
    12.3    4.5    4.6
```

Outros comandos,

```
-->size(H)   // dimensoes da hipermatriz
 ans  =
    2.    3.    4.
```

```
// vetor linha contendo a soma dos elementos das colunas da submatriz 3
-->sum(H(:,:,3),'r')
 ans  =
    6.5    6.7    6.9
// vetor coluna contendo os menores elementos das linhas da submatriz 2
-->min(H(:,:,2),'c')
 ans  =
    2.1
    2.4
```

3.6 Polinômios

O SCILAB fornece várias funções que permitem que as operações envolvendo polinômios sejam feitas de um modo bem simples.

3.6.1 Construção

Os polinômios podem ser construídos ou a partir de seus coeficientes ou de seus zeros.

Construção a partir dos coeficientes

Um polinômio na forma $P(x) = c_1 + c_2x + c_3x^2 + \ldots + c_{n-1}x^{n-2} + c_nx^{n-1}$ pode ser construído a partir dos coeficientes c_i, usando a função `poly(<parâmetros>, <variável>, <modo>)`, onde o vetor `<parâmetros>` contém os coeficientes c_i, `<variável>` é uma cadeia de caracteres que determina a variável do polinômio e a outra cadeia de caracteres `<modo>` especifica como o polinômio será construído. Se `<modo> = 'coeff'` então ele será construído a partir de seus coeficientes. Por exemplo, para construir o polinômio $P(x) = 24 + 14x - 13x^2 - 2x^3 + x^4$ basta,

```
-->P = poly([24 14 -13 -2 1],'x','coeff')   // P(x) a partir dos coeficientes
 P  =
                  2    3   4
    24 + 14x - 13x - 2x + x
```

Construção a partir das raízes

Utilizando a mesma função `poly`, mas com `<parâmetros>` contendo os zeros do polinômio e sendo `<modo> = 'roots'` (ou não sendo especificado) então ele será construído a partir de seus zeros. Para construir um polinômio com zeros $\xi_1 = -1$, $\xi_2 = -3$, $\xi_3 = 2$ e $\xi_4 = 4$, faz-se

```
-->R = poly([-1 -3 2 4],'s','roots')   //  R(s) a partir dos zeros
 R  =
                  2    3   4
    24 + 14s - 13s - 2s + s
```

3.6.2 Avaliação

Um polinômio é avaliado por meio da função `horner(<polinômio>, <abscissas>)`, onde `<polinômio>` contém o polinômio e `<abscissas>` contém os pontos nos quais ele deve ser avaliado. Por exemplo, para avaliar $T(x) = 1 - 3x + 7x^2 + 5x^3 - 2x^4 + 3x^5$ nos pontos $x = -1, 0, 1, 2$ e 3,

```
-->T = poly([1 -3 7 5 -2 3],'x','coeff')   // T(x) a partir dos coeficientes
 T  =
                2    3    4    5
    1 - 3x + 7x + 5x - 2x + 3x
-->y = horner(T,(-1:3))   // avalia T(x) em x = -1, 0, 1, 2 e 3
 y  =
    1.    1.    11.    127.    757.
```

3.6.3 Adição e subtração

Para somar e subtrair polinômios, mesmo de graus diferentes, basta usar os operadores de adição (`+`) e subtração (`-`). Assim, para somar os polinômios $a(x) = 1 - 4x^2 + 5x^3 + 8x^4$ e $b(x) = -4 - x + 5x^2 + 2x^3$,

```
-->a = poly([1 0 -4 5 8],'x','coeff')   // constroi a(x)
 a  =
          2    3    4
    1 - 4x + 5x + 8x
-->b = poly([-4 -1 5 2],'x','coeff')   // constroi b(x)
 b  =
              2    3
  - 4 - x + 5x + 2x
-->c = a + b   // soma polinomios
 c  =
             2    3    4
  - 3 - x + x + 7x + 8x
```

resultando $c(x) = -3 - x + x^2 + 7x^3 + 8x^4$; e para subtrair

```
-->d = a - b   // subtrai polinomios
 d  =
              2    3    4
    5 + x - 9x + 3x + 8x
```

3.6.4 Multiplicação

A multiplicação de dois polinômios é feita utilizando o operador de multiplicação (`*`). Sejam os polinômios $e(v) = 4 - 5v + 3v^2$ e $f(v) = -1 + 2v$, a multiplicação resulta em,

```
-->e = poly([4 -5 3],'v','coeff')   // constroi e(v)
```

```
 e  =
                2
    4 - 5v + 3v
-->f = poly([-1 2],'v','coeff')   // constroi f(v)
 f  =
  - 1 + 2v
-->g = e * f   // multiplica e(v) por f(v)
 g  =
                  2    3
  - 4 + 13v - 13v + 6v
```

resultando no polinômio $g(v) = -4 + 13v - 13v^2 + 6v^3$.

3.6.5 Divisão

O comando `[<resto>, <quociente>] = pdiv(<polinômio_1>, <polinômio_2>)` faz a divisão entre o `<polinômio_1>` e o `<polinômio_2>`, retornando o polinômio quociente em `<quociente>` e o polinômio do resto da divisão em `<resto>`, ou seja, `<polinômio_1> = <polinômio_2> * <quociente> + <resto>`. Por exemplo, a divisão de $h(x) = 6 - 5x + 4x^2 - 3x^3 + 2x^4$ por $i(x) = 1 - 3x + x^2$ é efetuada por,

```
-->h = poly([6 -5 4 -3 2],'x','coeff')   // constroi h(x)
 h  =
                2    3    4
    6 - 5x + 4x - 3x + 2x
-->i = poly([1 -3 1],'x','coeff')   // constroi i(x)
 i  =
              2
    1 - 3x + x
-->[r,q] = pdiv(h,i)   // divide h(x) por i(x)
 q  =
               2
    11 + 3x + 2x
 r  =
  - 5 + 25x
```

resultado no quociente $q(x) = 11 + 3x + 2x^2$ com resto $r(x) = -5 + 25x$. No caso do uso do operador de divisão (`/`) em vez da função `pdiv`, é criado um polinômio racional. Para os polinômios $h(x)$ e $i(x)$ definidos acima,

```
-->j = h / i   // polinomio racional obtido da divisao de h(x) por i(x)
 j  =
                2    3    4
    6 - 5x + 4x - 3x + 2x
    ---------------------
                   2
         1 - 3x + x
```

3.6.6 Derivação

A função `derivat` efetua a derivação polinomial. Deste modo, para obter a primeira e segunda derivada de $P(x) = -5 + 14x - 12x^2 + 2x^3 + x^4$,

```
-->p = poly([-5 14 -12 2 1],'x','coeff')   // constroi p(x)
 p  =
                 2    3   4
  - 5 + 14x - 12x + 2x + x
-->p1 = derivat(p)   // derivada primeira de p(x)
 p1  =
                 2    3
    14 - 24x + 6x + 4x
-->p2 = derivat(p1)   // derivada segunda de p(x)
 p2  =
                  2
  - 24 + 12x + 12x
```

resultando $P'(x) = 14 - 24x + 6x^2 + 4x^3$ e $P''(x) = -24 + 12x + 12x^2$.

3.6.7 Cálculo de raízes

A função `roots(<polinômio>)` calcula os zeros de `<polinômio>`. Por exemplo, para calcular as quatro raízes de $T(x) = 24 - 14x - 13x^2 + 2x^3 + x^4 = 0$,

```
-->T = poly([24 -14 -13 2 1],'x','coeff')   // constroi T(x)
 T  =
                  2    3   4
    24 - 14x - 13x + 2x + x
-->raizes = roots(T)   // calcula as raizes de T(x) = 0
 raizes  =
    1.
  - 2.
    3.
  - 4.
```

Para calcular as raízes de um polinômio dado em termos de seus coeficientes, sem construir previamente o polinômio, os coeficientes devem ser fornecidos do maior para o menor grau. Por exemplo, para calcular as quatro raízes de $T(x) = 24 - 14x - 13x^2 + 2x^3 + x^4 = 0$,

```
-->a = roots([1 2 -13 -14 24])   // calcula raizes a partir dos coeficientes
 a  =
    1.
  - 2.
    3.
  - 4.
```

Quando o polinômio for real e possuir grau menor ou igual a 100 a função `roots` calcula os zeros por um algoritmo rápido, caso contrário, os zeros são calculados por meio dos autovalores da matriz companheira. Se for acrescentado o segundo argumento da função `roots`, dado por `'e'`, então os zeros serão calculados utilizando os autovalores, independente do grau do polinômio.

Pode ser mostrado que as raízes de $P(x) = c_1 + c_2x + c_3x^2 + \ldots + c_{n-1}x^{n-2} + c_nx^{n-1} = 0$ são os autovalores da matriz companheira,

$$C = \begin{pmatrix} -c_{n-1}/c_n & -c_{n-2}/c_n & \ldots & -c_2/c_n & -c_1/c_n \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \end{pmatrix}.$$

Deste modo, `r = roots(T,'e')` é equivalente à

```
-->A = companion(T)   // matriz companheira do polinomio T(x)
 A  =
  - 2.    13.    14.  - 24.
    1.    0.     0.     0.
    0.    1.     0.     0.
    0.    0.     1.     0.
-->r = spec(A)   // autovalores da matriz companheira A
 r  =
  - 4.
    3.
  - 2.
    1.
```

3.7 Variáveis lógicas

Conforme visto, uma constante lógica tem os valores `%t` (ou `%T`) para *verdadeiro* e `%f` (ou `%F`) para *falso*, os quais podem ser atribuídos à uma variável lógica, com estrutura de vetor ou matriz. Por exemplo,

```
-->a = %t   // variavel logica simples
 a  =
  T
-->b = [%f %F %t %T]   // vetor logico
 b  =
  F F T T
-->C = [%T %f %t;%f %T %F]   // matriz logica
 C  =
  T F T
  F T F
```

As variáveis lógicas podem ser utilizadas para acessar os elementos de uma matriz (ou vetor). Quando o valor for *verdadeiro* o índice é acessado e no caso de ser *falso* o índice não será acessado. Por exemplo,

```
-->v = [11 12 13 14 15 16 17 18 ]   // define o vetor v com 8 elementos
 v  =
    11.    12.    13.    14.    15.    16.    17.    18.
-->ind = [%t %t %f %t %f]   // define vetor logico ind com 5 elementos
 ind  =
  T T F T F
-->v(ind)   // elementos 1, 2 e 4 do vetor
 ans  =
    11.    12.    14.
```

O número de elementos do vetor lógico deve ser menor ou igual ao número de elementos do vetor que ele referencia. O uso de variáveis lógicas é fundamental nas expressões lógicas que serão vistas na Seção 4.2 Expressões lógicas.

3.8 Variáveis literais

Uma variável literal contém uma constante literal formada por uma cadeia de caracteres delimitada por aspas (') ou apóstrofos ("). Por exemplo,

```
-->a = 'SCILAB', b = "cadeia de caracteres"
 a  =
 SCILAB
 b  =
 cadeia de caracteres
```

Os caracteres podem ser concatenados usando o operador (+) ou a notação vetorial,

```
-->d = a+' manipula '+b
 d  =
 SCILAB manipula cadeia de caracteres
-->e = [a 'utiliza' b]
 e  =
!SCILAB  utiliza  cadeia de caracteres  !
```

A utilização de variáveis literais será abordada com mais detalhes na Seção 4.3 Expressões literais.

3.9 Listas

Uma lista é um conjunto de dados não necessariamente do mesmo tipo, podendo conter matrizes ou mesmo outras listas. As listas são úteis para definirem dados estruturados. Uma lista é construída a partir do comando `list`,

```
-->Rol = list('tipo',%f,[1 2 3; 4 5 6])   // gera a lista Rol com 3 objetos
 Rol  =
       Rol(1)
 tipo
       Rol(2)
  F
       Rol(3)
    1.    2.    3.
    4.    5.    6.
-->Rol(2)   // objeto 2 de Rol
 ans  =
  F
-->Rol(3)(2,1)   // elemento (2,1) do objeto 3 de Rol
 ans  =
    4.
```

Uma lista pode ser um objeto de outra lista,

```
-->Rol(1) = list('tamanho',[10 20 30])  // objeto 1 de Rol torna-se uma lista
 Rol  =
       Rol(1)
        Rol(1)(1)
 tamanho
        Rol(1)(2)
    10.    20.    30.
       Rol(2)
  F
       Rol(3)
    1.    2.    3.
    4.    5.    6.
```

Para obter e atribuir valor a objeto da lista, faz-se

```
-->Rol(1)(2)(1,3)   // elemento (1,3) do objeto 2 de Rol(1)
 ans  =
    30.
-->Rol(3)(2,1) = 55.8   // atribuindo novo valor ao elemento (2,1) de Rol(3)
 Rol  =
       Rol(1)
        Rol(1)(1)
 tamanho
        Rol(1)(2)
    10.    20.    30.
       Rol(2)
  F
       Rol(3)
    1.      2.    3.
    55.8    5.    6.
```

3.10 Exercícios

Seção 3.1 Constantes

Observar atentamente e anotar o resultado dos comandos do SCILAB. Apesar de os comandos estarem separados por vírgula, entrar com um de cada vez.

3.1 Atribuir o valor $-1{,}23\times10^3$ à variável `a` e $4{,}17\times10^{-2}$ à variável `b`.

3.2

3.3 Criar um vetor lógico com cinco elementos sendo o primeiro, segundo e quarto com o valor *verdadeiro* e os outros com o valor *falso*.

3.4 Criar uma matriz de dimensão 2×3 com elementos lógicos, com a terceira coluna com elementos com o valor *falso* e as demais com o valor *verdadeiro*.

3.5

Seção 3.2 Variáveis

3.6 Conferir os nomes permitidos de variáveis `arco-seno=0, limite:sup=3, Area=3` e `area=1`.

3.7 Atribuir a cadeia de caracteres `'abc'` à variável literal `letras` e `'01234'` à variável literal `numeros`.

3.8

3.9

3.10

Seção 3.3 Vetores

Construir os vetores e analisar as operações:

3.11 `al = [1 3.5 -4.2 7.5], ac = (2:3:11)'`.

3.12 `b = 10:5:30, c = 5:-2:-6, d = 5:10`.

3.13 `e = linspace(0,10,5), e(3), e(2:4)`.

3.14 `f = linspace(10,20,6), f(3:-1:1), f([6 5 1 3])`.

3.15 `g = linspace(2,11,4) + linspace(0.2,1.1,4)*%i , h = g', i = g.'`

Seção 3.4 Matrizes

Construir as matrizes e verificar o resultado das operações e funções

3.16 `A = [6 -1 4; 0 2 -3; 5 7 8], A(2,1) = 1,`
`sum(A,'r'), sum(A,'c'), prod(A,'r'), prod(A,'c')`.

3.17 `B = A(2:3,[2 3 1]), min(B,'r'), max(B,'c')`.

3.18 `C = [A B'], C(:,[2 4])=[], C(2,:) = [], size(C)`.

3.19 `zeros(3,5), ones(2,4), eye(5,3)`.

3.20 `D = grand(3,3,'def'), diag(diag(D)), triu(D), tril(D)`.

Seção 3.5 Hipermatrizes

3.21

3.22

3.23

3.24

3.25

Seção 3.6 Polinômios

3.26

3.27

3.28

3.29

3.30

Seção 3.7 Variáveis lógicas

3.31

3.32

3.33

3.34

3.35

Seção 3.8 Variáveis literais

3.36

3.37

3.38

3.39

3.40

Seção 3.9 Listas

3.41

3.42

3.43

3.44

3.45

Capítulo 4

Expressões

Uma expressão é uma combinação de constantes, variáveis e operadores, cuja avaliação resulta em um valor único. Neste capítulo serão mostrados os três tipos de expressões do SCILAB: aritméticas, lógicas e literais.

4.1 Expressões aritméticas

É disponível no SCILAB as operações aritméticas básicas mostradas na Tabela 4.1.

Tabela 4.1: Operações aritméticas básicas do SCILAB.

Operação	Expressão	Operador	Exemplo
adição	$a + b$	`+`	`1+2`
subtração	$a - b$	`-`	`5.1-4.7`
multiplicação	$a \times b$	`*`	`6*9.98`
divisão	$a \div b$	`/` ou `\`	`6/7  5\3`
potenciação	a^b	`^`	`2^10`

4.1.1 Ordem de precedência

O SCILAB obedece a ordem de precedência das operações aritméticas apresentadas na Seção 1.5 Expressões. Para o Exemplo 1.5, na página 6,

```
-->a = 1; b = 2; c = 3; d = 4;
-->t = a + b / c + d
 t  =
    5.6666667
```

```
-->x = (a + b) / (c + d)
 x  =
    0.4285714
-->y = a + b / (c + (d + 1) / 2)
 y  =
    1.3636364
-->w = ((a+b*c)/(a+1)-(b+4)/(c+5))/(-c^2+(-b)^(d+1))
 w  =
  - 0.0670732
-->z = (a^b+3-c^(d+3))/(a*b/c+sqrt(1+d/(a+b)))
 z  =
  - 994.89931
```

4.1.2 Expressões vetoriais

As operações básicas entre vetores só são definidas quando estes tiverem o mesmo tamanho e orientação (linha ou coluna). Estas operações básicas são apresentadas na Tabela 4.2. As operações de multiplicação, divisão e potenciação envolvendo vetores quando antecedidas pelo caracter (.) significa que estas operações são efetuadas entre os correspondentes elementos dos vetores.

Tabela 4.2: Operações vetoriais básicas.

Sejam a = [a_1 a_2 ...a_n], b = [b_1 b_2 ...b_n] e c um escalar		
Operação	Expressão	Resultado
adição escalar	a+c	[a_1+c a_2+c ... a_n+c]
adição vetorial	a+b	[a_1+b_1 a_2+b_2 ... a_n+b_n]
multiplicação escalar	a*c	[a_1*c a_2*c ... a_n*c]
multiplicação vetorial	a.*b	[a_1*b_1 a_2*b_2 ... a_n*b_n]
divisão à direita	a./b	[a_1/b_1 a_2/b_2 ... a_n/b_n]
divisão à esquerda	a.\b	[b_1/a_1 b_2/a_2 ... b_n/a_n]
potenciação	a.^c	[a_1^c a_2^c ... a_n^c]
	c.^a	[c^a_1 c^a_2 ... c^a_n]
	a.^b	[a_1^b_1 a_2^b_2 ... a_n^b_n]

Considere as variáveis,

```
-->a = 1:5, b = 10:10:50, c = 2
 a  =
    1.    2.    3.    4.    5.
 b  =
    10.    20.    30.    40.    50.
```

```
 c  =
    2.
```

Adição escalar,

```
-->a + c
 ans  =
    3.    4.    5.    6.    7.
```

Adição vetorial,

```
-->a + b
 ans  =
    11.    22.    33.    44.    55.
```

Multiplicação escalar,

```
-->a * c
 ans  =
    2.    4.    6.    8.    10.
```

Multiplicação vetorial entre elementos correspondentes,

```
-->a .* b
 ans  =
    10.    40.    90.    160.    250.
```

Divisão vetorial à direita entre elementos correspondentes,

```
-->a ./ b
 ans  =
    0.1    0.1    0.1    0.1    0.1
```

Divisão vetorial à esquerda entre elementos correspondentes,

```
-->a .\ b
 ans  =
    10.    10.    10.    10.    10.
```

Potenciação,

```
-->a .^ c
 ans  =
    1.    4.    9.    16.    25.
-->c .^ a
 ans  =
    2.    4.    8.    16.    32.
-->a .^ b
 ans  =
    1.    1048576.    2.059D+14    1.209D+24    8.882D+34
```

4.1.3 Expressões matriciais

De modo similar às operações vetorias, existem as operações matriciais básicas, as quais estão compiladas na Tabela 4.3. O operador (\) envolvendo matrizes e vetores está relacionado com solução de sistemas lineares, conforme será visto na Seção 8.2.3 Solução de sistemas.

Tabela 4.3: Operações matriciais básicas.

Sejam `c` um escalar e
`A = [` $a_{11}\ a_{12}\ \ldots\ a_{1p}$ `;` $a_{21}\ a_{22}\ \ldots\ a_{2p}$ `; ... ;` $a_{m1}\ a_{m2}\ \ldots\ a_{mp}$ `]`
`B = [` $b_{11}\ b_{12}\ \ldots\ b_{1n}$ `;` $b_{21}\ b_{22}\ \ldots\ b_{2n}$ `; ... ;` $b_{p1}\ b_{p2}\ \ldots\ b_{pn}$ `]`		

Operação	Expressão	Resultado
adição escalar	`A+c`	$a_{ij} + c$
adição matricial	`A+B`	$a_{ij} + b_{ij}$
multiplicação escalar	`A*c`	$a_{ij} * c$
multiplicação matricial	`A*B`	AB
multiplicação entre elementos correspondentes	`A.*B`	$a_{ij} * b_{ij}$
divisão à direita entre elementos correspondentes	`A./B`	a_{ij}/b_{ij}
divisão à esquerda entre elementos correspondentes	`A.\B`	b_{ij}/a_{ij}
potenciação	`A^c`	A^c
	`A.^c`	a_{ij}^c
	`c.^A`	$c^{a_{ij}}$
	`A.^B`	$a_{ij}^{b_{ij}}$

Sejam as matrizes `A` e `B` de ordem 3 e o escalar `c`,

```
-->A = [1 2 3; 4 5 6; 7 8 9], B = [10 20 30; 40 50 60; 70 80 90], c = 3
 A  =
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
 B  =
    10.    20.    30.
    40.    50.    60.
    70.    80.    90.
 c  =
    3.
```

Adição escalar,

```
-->A + c
```

```
 ans  =
    4.     5.     6.
    7.     8.     9.
    10.    11.    12.
```

Adição matricial,

```
-->A + B
 ans  =
    11.    22.    33.
    44.    55.    66.
    77.    88.    99.
```

Multiplicação escalar,

```
-->A * c
 ans  =
    3.     6.     9.
    12.    15.    18.
    21.    24.    27.
```

A diferença no resultado das expressões quando os operadores contém o caracter (.) deve ser observada. Na multiplicação matricial,

```
-->A * B
 ans  =
    300.     360.     420.
    660.     810.     960.
    1020.    1260.    1500.
```

Multiplicação entre elementos correspondentes,

```
-->A .* B
 ans  =
    10.     40.     90.
    160.    250.    360.
    490.    640.    810.
```

Divisão à direita entre elementos correspondentes,

```
-->A ./ B
 ans  =
    0.1    0.1    0.1
    0.1    0.1    0.1
    0.1    0.1    0.1
```

Divisão à esquerda entre elementos correspondentes,

```
-->A .\ B
 ans  =
    10.    10.    10.
    10.    10.    10.
    10.    10.    10.
```

E nos diferentes tipos de potenciação,

```
-->A .^ c   // elemento da matriz elevado a uma constante
 ans  =
    1.      8.      27.
    64.     125.    216.
    343.    512.    729.
-->A ^ c   // matriz elevada a uma constante
 ans  =
    468.     576.     684.
    1062.    1305.    1548.
    1656.    2034.    2412.
-->c .^ A   // constante elevada a elemento da matriz
 ans  =
    3.       9.       27.
    81.      243.     729.
    2187.    6561.    19683.
-->A .^ B   // elemento de matriz elevado a elemento de matriz
 ans  =
    1.           1048576.     2.059D+14
    1.209D+24    8.882D+34    4.887D+46
    1.435D+59    1.767D+72    7.618D+85
```

É importante observar que no SCILAB as operações `c.^A` e `c^A` são equivalentes.

Como pode ser esperado de uma linguagem para aplicações nas áreas técnicas e científicas, o SCILAB oferece várias funções importantes. A Tabela 4.4 apresenta algumas funções matemáticas elementares do SCILAB.

Se a variável for um vetor ou uma matriz a avaliação de uma função se dá para cada elemento da variável,

```
-->a = 1:5
 a  =
    1.    2.    3.    4.    5.
-->b = sqrt(a)   // raiz quadrada
 b  =
    1.    1.4142136    1.7320508    2.    2.236068
```

Os resultados acima podem ser apresentados na forma de uma tabela por intermédio do comando

Tabela 4.4: Funções matemáticas elementares do SCILAB.

Função	Descrição	Função	Descrição
trigonométricas			
`acos`	arco co-seno	`cotg`	co-tangente
`asin`	arco seno	`sin`	seno
`atan`	arco tangente	`tan`	tangente
`cos`	co-seno		
exponenciais			
`acosh`	arco co-seno hiperbólico	`log`	logaritmo natural
`asinh`	arco seno hiperbólico	`log10`	logaritmo decimal
`atanh`	arco tangente hiperbólica	`sinh`	seno hiperbólico
`cosh`	co-seno hiperbólico	`sqrt`	raiz quadrada
`coth`	co-tangente hiperbólica	`tanh`	tangente hiperbólica
`exp`	exponencial		
complexas			
`abs`	valor absoluto	`imag`	parte imaginária do complexo
`conj`	complexo conjugado	`real`	parte real do complexo
numéricas			
`ceil`	arredonda em direção a $+\infty$	`lcm`	mínimo múltiplo comum
`fix`	arredonda em direção a 0	`modulo`	resto de divisão
`floor`	arredonda em direção a $-\infty$	`round`	arredonda em direção ao inteiro mais próximo
`gcd`	máximo divisor comum	`sign`	sinal

```
-->[a;b]'
 ans  =
    1.    1.
    2.    1.4142136
    3.    1.7320508
    4.    2.
    5.    2.236068
```

4.2 Expressões lógicas

Uma expressão se diz lógica quando os operadores forem lógicos e os operandos forem relações e/ou variáveis do tipo lógico. Uma relação é uma comparação realizada entre valores do mesmo tipo. A natureza da comparação é indicada por um operador relacional conforme a Tabela 4.5.

O resultado de uma relação ou de uma expressão lógica é *verdadeiro* ou *falso*. No SCILAB o resultado é literal, sendo que `T` significa *verdadeiro* e `F` significa *falso*. Note que o caracter (=) é usado para atribuição de um valor à uma variável enquanto que os caracteres (==) são usados para comparação de igualdade. Os operadores relacionais são usados para comparar

Tabela 4.5: Operadores relacionais do SCILAB.

Operador relacional	Descrição
==	igual a
>	maior que
>=	maior ou igual a
<	menor que
<=	menor ou igual a
~= ou <>	diferente de

vetores ou matrizes do mesmo tamanho ou comparar um escalar com os elementos de um vetor ou de matriz. Sejam os vetores `a` e `b` e o escalar `c`,

```
-->a = 1:10, b = 9:-1:0, c = 5
 a  =
    1.    2.    3.    4.    5.    6.    7.    8.    9.    10.
 b  =
    9.    8.    7.    6.    5.    4.    3.    2.    1.    0.
 c  =
    5.
```

Assim,

```
-->d = a >= c
 d  =
  F F F F T T T T T T
```

produz um vetor que contém o valor `T` (*verdadeiro*) quando o elemento correspondente do vetor `a` for maior ou igual a `5`, caso contrário contém o valor `F` (*falso*). Apesar de o resultado da relação lógica ser um valor literal, ele pode fazer parte de uma expressão aritmética, sendo que `F` vale `0` e `T` tem o valor `1`,

```
-->e = a + (b <= 3)
 e  =
    1.    2.    3.    4.    5.    6.    8.    9.    10.    11.
```

Quando um elemento de `b` for menor ou igual a `3`, o valor `1` (resultado da relação lógica) é adicionado ao correspondente valor de `a`. Os operadores lógicos permitem a combinação ou negação das relações lógicas. Os operadores lógicos do SCILAB são listados na Tabela 4.6.

Para os vetores `a` e `b` definidos anteriormente,

```
-->f = (a>3) & (a<=8)
 f  =
  F F F T T T T T F F
```

Tabela 4.6: Operadores lógicos do SCILAB.

Operador lógico	Descrição	Uso
&	**e**	conjunção
|	**ou**	disjunção
~	**não**	negação

Os elementos de `f` são iguais a `T` quando os correspondentes elementos de `a` forem maiores do que `3` **e** menores ou iguais a `8`, caso contrário são iguais a `F`. Para fazer a negação, ou seja, onde for `T` será `F` e vice-versa basta,

```
-->g = ~f
 g  =
  T T T F F F F F T T
```

A Tabela 4.7 apresenta a tabela verdade para os operadores lógicos do SCILAB.

Tabela 4.7: Tabela verdade.

a	b	a & b	a | b	~a
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

A ordem de precedência no SCILAB para expressões envolvendo operadores aritméticos e lógicos são indicados na Tabela 4.8.

Tabela 4.8: Ordem de precedência das operações aritméticas e lógicas.

Ordem de precedência	Operadores
1^a	( )
2^a	função
3^a	^ .^ ' .'
4^a	+ (unário) e - (unário)
5^a	* / \ .* ./ .\
6^a	+ e -
7^a	> >= < <= == ~=
8^a	~
9^a	&
10^a	|

Os parênteses podem ser usados para alterar a ordem de precedência. Para os vetores `a` e `b` definidos acima, a avaliação da expressão,

```
-->h = -(a-5).^2 > -9 & 3*b <= 15
 h  =
  F F F F T T T F F F
```

é equivalente às várias etapas,

```
-->h1 = a-5   // parenteses
 h1  =
  - 4.  - 3.  - 2.  - 1.    0.    1.    2.    3.    4.    5.
-->h2 = h1.^2   // potenciacao
 h2  =
    16.    9.    4.    1.    0.    1.    4.    9.    16.    25.
-->h3 = -h2   // - unario
 h3  =
  - 16.  - 9.  - 4.  - 1.    0.  - 1.  - 4.  - 9.  - 16.  - 25.
-->h4 = 3*b   // multiplicacao
 h4  =
    27.    24.    21.    18.    15.    12.    9.    6.    3.    0.
-->h5 = h3 > -9   // operador relacional (primeiro a esquerda)
 h5  =
  F F T T T T T F F F
-->h6 = h4 <= 15   // operador relacional (segundo a esquerda)
 h6  =
  F F F F T T T T T T
-->h = h5 & h6   // operador logico
 h  =
  F F F F T T T F F F
```

4.3 Expressões literais

O SCILAB dispõe de várias funções para manipulação de cadeia de caracteres, sendo algumas delas mostradas a seguir.

4.3.1 Conversão de caracteres

Serão apresentadas funções para alterar o tamanho da caixa das letras e fazer a conversão de caracteres para número e vice-versa.

Caixas altas e baixas

A função `convstr(<texto>,<tipo>)` converte a cadeia de caracteres `<texto>` de acordo com o `<tipo>`. Se ele for `'u'` (*upper*) então converte para caixa alta (maiúsculo) e se for `'l'` (*lower*) ou omitido converte para caixa baixa (minúsculo),

```
-->texto = 'Caixas Altas e Baixas'   // define o texto
 texto  =
```

```
 Caixas Altas e Baixas
-->convstr(texto,'u')   // converte para caixa alta
 ans  =
 CAIXAS ALTAS E BAIXAS
-->convstr(texto)   // converte para caixa baixa
 ans  =
 caixas altas e baixas
```

Código ASCII

Os comandos `<número> = ascii(<texto>)` e `<texto> = ascii(<número>)` fazem a conversão dos caracteres ASCII (*American Standard Code for Information Interchange*) contidos no vetor de caracteres `<texto>` para valores numéricos e atribuem ao vetor `<número>` e vice-versa, de acordo com a Tabela 4.9.

Tabela 4.9: Caracteres em código ASCII.

Representação decimal dos caracteres															
33	!	45	-	57	9	69	E	81	Q	93	]	105	i	117	u
34	"	46	.	58	:	70	F	82	R	94	^	106	j	118	v
35	#	47	/	59	;	71	G	83	S	95	_	107	k	119	w
36	$	48	0	60	<	72	H	84	T	96	‘	108	l	120	x
37	%	49	1	61	=	73	I	85	U	97	a	109	m	121	y
38	&	50	2	62	>	74	J	86	V	98	b	110	n	122	z
39	’	51	3	63	?	75	K	87	W	99	c	111	o	123	{
40	(	52	4	64	@	76	L	88	X	100	d	112	p	124	|
41	)	53	5	65	A	77	M	89	Y	101	e	113	q	125	}
42	*	54	6	66	B	78	N	90	Z	102	f	114	r	126	~
43	+	55	7	67	C	79	O	91	[	103	g	115	s		
44	,	56	8	68	D	80	P	92	\	104	h	116	t		

Por exemplo, para o vetor `s` de tamanho 5,

```
-->s = '5+3*i'   // vetor de caracteres
 s  =
 5+3*i
-->n = ascii(s)   // converte caracteres para numeros
 n  =
    53.    43.    51.    42.    105.
-->t = ascii(n)   // converte numeros para caracteres
 t  =
 5+3*i
```

Código do SCILAB

Os comandos `<texto> = code2str(<número>)` e `<número> = str2code(<texto>)` fazem a conversão de um vetor de números inteiros `<número>` para a cadeia de caracteres `<texto>` e vice-versa, segundo o código do SCILAB,

```
-->texto = 'Caracteres'   // define o texto
 texto  =
 Caracteres
-->numero = str2code(texto)'   // converte texto para numero (vetor transposto)
 numero  =
  - 12.    10.    27.    10.    12.    29.    14.    27.    14.    28.
-->palavra = code2str(numero)   // converte numero para texto
 palavra  =
 Caracteres
```

É importante observar que os códigos de conversão produzidos pelas funções `code2str` e `str2code` são diferentes daqueles produzidos pela função `ascii`,

```
-->car = '01AZaz'   // define caracteres
 car  =
 01AZaz
-->cod = str2code(car)'   // converte para codigo do SCILAB (vetor transposto)
 cod  =
    0.    1.  - 10.  - 35.    10.    35.
-->asc = ascii(car)   // converte para codigo ASCII
 asc  =
    48.    49.    65.    90.    97.    122.
```

Números

O comando `<texto> = string(<número>)` converte a constante numérica `<número>` para a sua representação em caracteres e atribui o resultado a `<texto>`. Este comando é de grande utilidade quando da escrita de rótulos e títulos em gráficos, conforme será visto no Capítulo 5,

```
-->v = 4.25
 v  =
    4.25
-->titulo = 'velocidade = ' + string(v) + ' m/s'
 titulo  =
 velocidade = 4.25 m/s
```

4.3.2 Manipulação de caracteres

Serão descritas, a seguir, funções para obter o tamanho, criar, operar e procurar caracteres.

Tamanho

A função `length(<variável>)` fornece o número de caracteres presentes na cadeia de caracteres `<variável>`; se ela for uma matriz então é informado o número de caracteres de cada elemento,

```
-->length('abcde')   // numero de caracteres
 ans  =
    5.
-->length(['ab' '123'; '*' 'abcd'])   // numero de caracteres de matriz
 ans  =
    2.    3.
    1.    4.
```

Criação

O comando `<variável> = emptystr(<linhas>,<colunas>)` gera uma matriz de caracteres `<variável>` vazia de dimensão `<linhas>` $\times$ `<colunas>`. Por sua vez, `<variável> = emptystr(<matriz>)` cria a matriz de caracteres `<variável>` vazia com as mesmas dimensões de `<matriz>`,

```
-->mat = emptystr(2,3)   // matriz vazia de dimensao 2 x 3
 mat  =
!      !
!      !
!      !
-->length(mat)   // numero de caracteres de mat
 ans  =
    0.    0.    0.
    0.    0.    0.
-->matnum = [1 2; 3 4]   // define matriz numerica
 matnum  =
    1.    2.
    3.    4.
-->S = emptystr(matnum)   // matriz vazia com mesma dimensao de matnum
 S  =
!    !
!    !
!    !
-->length(S)   // numero de caracteres de S
 ans  =
    0.    0.
    0.    0.
```

A função `strcat(<vetor>,<caracteres>)` insere a cadeia `<caracteres>` entre cada elemento da cadeia de caracteres `<vetor>`,

```
-->vet = ['a' 'b' 'c' 'd' 'e']   // define vetor de caracteres
 vet  =
!a  b  c  d  e  !
-->vv = strcat(vet,',')   // insere ',' entre cada elemento
 vv  =
 a,b,c,d,e
-->ve = strcat(vet,' >= ')   // insere ' >= ' entre cada elemento
 ve  =
 a >= b >= c >= d >= e
```

Operação

O comando `<matriz_pos> = justify(<matriz>,<posição>)` cria `<matriz_pos>`, uma matriz de caracteres, alterando a posição das colunas da matriz de caracteres `<matriz>` de acordo com `<posição>`. Se for igual a `'l'` (*left*) as colunas serão movidas para a esquerda, se for `'c'` (*center*) elas serão centradas e se `<posição> = 'r'` (*right*) as colunas serão movidas para a direita,

```
-->M = ['abcdefg' 'h' 'ijklmnopq'; '1' '1234567' '7890'; 'abc' '01' 'def']
 M  =
!abcdefg  h        ijklmnopq  !
!                             !
!1        1234567  7890       !
!                             !
!abc      01       def        !
-->C = justify(M,'c')   // cria matriz C a partir de M com colunas centradas
 C  =
!abcdefg     h     ijklmnopq  !
!                             !
!   1     1234567    7890     !
!                             !
!  abc      01        def     !
```

A função `stripblanks(<texto>)` elimina os caracteres brancos do início e do final da cadeia de caracteres `<texto>`,

```
-->v = '   a b c   '   // vetor v com 3 caracteres brancos no inicio e no final
 v  =
    a b c
-->length(v)   // numero de caracteres de v
 ans  =
    11.
-->s = stripblanks(v)   // remove caracteres brancos do inicio e do final de v
 s  =
 a b c
-->length(s)   // numero de caracteres de s
 ans  =
    5.
```

A função `strsplit(<texto>,<índice>)` quebra a cadeia de caracteres `<texto>` nas posições dadas no vetor numérico `<índice>`, com valores estritamente crescentes, gerando um vetor de caracteres,

```
-->s = '1234567890abcdefgh'   // define vetor de caracteres
 s  =
 1234567890abcdefgh
-->v = strsplit(s,[3 8 12])   // quebra o vetor s nas posicoes 3, 8 e 12
 v  =
!123     !
!        !
!45678   !
!        !
!90ab    !
!        !
!cdefgh  !
```

A função `strsubst(<matriz>,<texto_orig>,<texto_subs>)` substitui todas as ocorrências da cadeia de caracteres `<texto_orig>` na matriz de caracteres `<matriz>` pelos caracteres contidos em `<texto_subs>`,

```
-->frase = 'determinante = produto dos autovalores'   // define o texto
 frase  =
 determinante = produto dos autovalores
-->strsubst(frase,'e','E')   // substitui as letras 'e' por 'E'
 ans  =
 dEtErminantE = produto dos autovalorEs
-->strsubst(frase,' = ',' e'' igual ao ')   // substitui simbolo por texto
 ans  =
 determinante e' igual ao produto dos autovalores
-->strsubst(frase,' ','')   // remove todos os caracteres brancos
 ans  =
 determinante=produtodosautovalores
```

Procura

Seja o comando `[<linhas>,<qual>] = grep(<matriz>,<texto>)`. Para cada entrada da `<matriz>` de caracteres, a função `grep` pesquisa se pelo menos uma cadeia de caracteres no vetor `<texto>` é igual a uma subcadeia de `<matriz>`. O vetor numérico `<linhas>` informa os índices das entradas de `<matriz>` onde, pelo menos, uma igualdade foi encontrada. O argumento opcional `<qual>` fornece índices informando qual das cadeias de caracteres de `<texto>` foi encontrada. Por exemplo,

```
-->matriz = ['o scilab e'' um software livre';
-->          'para aplicacoes cientificas';
```

```
-->         'ver www.scilab.org']   // define a matriz de caracteres
 matriz  =
!o scilab e' um software livre  !
!                               !
!para aplicacoes cientificas    !
!                               !
!ver www.scilab.org             !
-->[linha,qual] = grep(matriz,['ver','scilab'])   // procura por 'ver' e 'scilab'
 qual  =
    2.    1.    2.
 linha  =
    1.    3.    3.
```

As cadeias de caracteres `ver` (posição 1) e `scilab` (posição 2) foram encontradas nas linhas `1`, `3` e `3`, sendo que na linha `1` foi encontrada a cadeia de índice `2 (scilab)` e na linha `3` foram as de índice `1 (ver)` e `2 (scilab)`.

Considere o comando `[<índices>,<qual>] = strindex(<texto>,<vetor_i>)`. A função `strindex` procura os índices de onde o `i`-ésimo elemento do vetor de caracteres `<vetor_i>` foi encontrado na cadeia de caracteres `<texto>`. Estes índices são retornados no vetor numérico `<índices>`. O argumento opcional `<qual>` fornece índices informando qual das cadeias de caracteres de `<vetor_i>` foram encontradas. Por exemplo,

```
-->ind = strindex('abcd dabc','a')   // procura os indices de ocorrencia de 'a'
 ind  =
    1.    7.
-->ind = strindex('baad daaa','aa')   // indices de ocorrencia de 'aa'
 ind  =
    2.    7.    8.
-->[ind,qual] = strindex('baad daaa bdac',['aa' 'da'])
 qual  =
    1.    2.    1.    1.    2.
 ind  =
    2.    6.    7.    8.    12.
```

Os caracteres `aa` e `da` foram encontrados nas posições `2`, `6`, `7`, `8`, e `12` e os elementos que apareceram foram, respectivamente, `1 (aa)`, `2 (da)`, `1 (aa)`, `1 (aa)` e `2 (da)`.

O comando `<elementos> = tokens(<texto>,<delimitador>)` cria um vetor coluna de caracteres `<elementos>` particionando a cadeia de caracteres `<texto>` onde ocorrer a presença de um dos componentes do vetor `<delimitador>`. Cada componente do segundo argumento é formado por um único caracter. Se este argumento for omitido então é assumido o vetor `[' ',<Tab>]` (caracteres branco e `ascii(9)`). Por exemplo,

```
-->tokens('Exemplo de cadeia de caracteres')   // particiona com delimitador ' '
```

```
 ans  =
!Exemplo     !
!            !
!de          !
!            !
!cadeia      !
!            !
!de          !
!            !
!caracteres  !
-->tokens('Exemplo de cadeia de caracteres','c')   // delimitador 'c'
 ans  =
!Exemplo de   !
!             !
!adeia de     !
!             !
!ara          !
!             !
!teres        !
-->tokens('Exemplo de cadeia de caracteres',['d' 't']) // delimitadores 'd' e 't'
 ans  =
!Exemplo   !
!          !
!e ca      !
!          !
!eia       !
!          !
!e carac   !
!          !
!eres      !
```

Seja o comando `<matriz> = tokenpos(<texto>,<delimitador>)`. A função `tokenpos` particiona a cadeia de caracteres `<texto>` onde ocorrer a presença de um dos componentes do vetor `<delimitador>`. Os índices inicial e final de cada elemento resultante da partição são atribuídos à matriz numérica de duas colunas `<matriz>`. Cada componente do segundo argumento é formado por um único caracter. Se este argumento for omitido então é assumido o vetor `[' ',<Tab>]` (caracteres branco e `ascii(9)`). Por exemplo,

```
-->tokenpos('Exemplo de cadeia de caracteres') // particiona com delimitador ' '
 ans  =
    1.     7.
    9.     10.
    12.    17.
    19.    20.
    22.    31.
```

```
-->tokenpos('Exemplo de cadeia de caracteres',['d' 't'])// delimitadores 'd', 't'
 ans  =
    1.     8.
    10.    13.
    15.    18.
    20.    26.
    28.    31.
```

Seja o comando <elementos> = part(<texto>,<índices>), onde <elementos> e <texto> são matrizes de caracteres e <índices> é um vetor numérico de índices. A função part obtém <elementos> a partir dos elementos de <texto> nas posições especificadas em <índice>. Por exemplo,

```
-->c = part(['123','4567','890'],[1,3])   // obtendo com indices 1 e 3
 c  =
!13  46  80  !
```

Observar que cada valor de <elementos> é separado por dois caracteres branco. Se um elemento de <índice> for maior que o comprimento de um componente de <texto> então <elementos> receberá um caractere branco,

```
-->vet = part(['a','bcd','efghi'],[1,2,4,1]) // obtendo com indices 1, 2, 4 e 1
 vet  =
!a  a  bc b  efhe  !
```

O comprimento do primeiro componente 'a' é 1 que é menor que 2 e 4, portanto, o primeiro componente de vet recebe dois caracteres brancos nas posições 2 e 3. O segundo componente 'bcd' tem comprimento 3 que é menor que 4 por isso recebe um branco na posição 3.

4.4 Execução de expressões

O SCILAB possui comandos com o propósito de executar expressões, um dos quais é o eval, cuja sintaxe é

```
<resultado> = eval(<expressão>)
```

A cadeia de caracteres <expressão> contém a expressão matemática que deve ser interpretada e atribuída à variável <resultado>. Por exemplo,

```
-->r = eval('sin(%pi/2)')
 r  =
    1.
-->x = 1; y = 2; eval('2*x+exp(y-2)')
 ans  =
    3.
```

Note que para avaliar uma expressão que possui as variáveis x e y, estas têm que ser previamente definidas. A função eval é também capaz de executar tarefas mais complexas,

```
-->Tabela = ['sqrt(x)'; 'exp(x)';'1/x+5*x^2']   // define matriz com 3 expressoes
 Tabela  =
!sqrt(x)    !
!           !
!exp(x)     !
!           !
!1/x+5*x^2  !
-->x = 16; eval(Tabela(1,:))   // avalia a expressao da linha 1 com x = 16
 ans  =
    4.
-->x = 2; eval(Tabela(3,:))   // avalia a expressao da linha 3 com x = 2
 ans  =
    20.5
```

A função `evstr(<expressão>)` converte a representação do valor numérico da cadeia de caracteres `<expressão>` escrita em código ASCII para a representação numérica. O argumento `<expressão>` deve estar escrito de acordo com as regras de definição de constantes numéricas vistas na Seção 3.1 Constantes,

```
-->x = evstr('1.23d2+5.678e-1*%i')
 x  =
    123. + 0.5678i
```

com este valor numérico é possível, por exemplo, calcular a raiz quadrada,

```
-->y = sqrt(x)
 y  =
    11.090566 + 0.0255983i
```

4.5 Exercícios

Observar atentamente e anotar o resultado dos comandos do SCILAB. Apesar de os comandos estarem separados por vírgula, entrar com um de cada vez.

Seção 4.1 Expressões aritméticas

4.1 Avaliar as expressões escalares
`5+3, 4-1, 6*3, 10/2, 2\ 5, 3^2, 1+2*3-4/5, 6*2^3`.

4.2 Sejam $a = 1, b = 2, c = 3, d = 4, e = 5$. Avaliar

$$x = \sqrt[3]{e(e-a)(e-b)+c^d},$$

$$y = a - \frac{b^2 - c}{d+e} + \left(\frac{\cos(a)}{d + \operatorname{sen}(b+c)}\right)^3,$$

$$z = \log_{10}\left(\frac{b}{d+e^2}\right) + \exp\left(\frac{b+a}{b-a} + \frac{c^2}{1+b^3}\sqrt{1+b^2}\right).$$

Avaliar as expressões vetoriais para `u = 2:6, v = linspace(10,14,5), k = 5`,

4.3 `u-k, u+v, u*k, u.*v, u./v, u.\v`.

4.4 `u.^k, k.^u, v.^u`.

4.5 `w = sqrt(u+v) - cos(u-k*v)`.

Avaliar as expressões matriciais para
`M = [2 -3 1; 4 6 -1; -5 2 1], N = [1 1 2; 3 1 -1; 3 2 1], x = (1:3)', z = 2`.

4.6 `M-z, M+N, M*z`.

4.7 `M.*N, M*N, M*x, M./N, M.\N`.

4.8 `M.^z, M^z, z.^M, z^M, M.^N`.

4.9 Verificar a diferença entre as funções usando $a = \pi$ e $a = -5{,}6$,
`ceil(a), fix(a), floor(a), round(a), sign(a)`.

4.10 Para $c = 3 + 4i$ e $c = 5 - 3i$, observar os resultados das funções complexas,
`abs(c), conj(c), real(c), imag(c)`.

Seção 4.2 Expressões lógicas

4.11 Com relação às variáveis lógicas, dados `a = %t, b = %f, c = sqrt(2) > 1, d = exp(0) == 0`, completar a tabela

a	b	c	d	a&b	a&c	b&d	a|b	a|c	b|d	~a	~b

Avaliar as expressões lógicas para `x = -2:2` e `v = 1:5`,

4.12 `x > -1`.

4.13 `abs(x) == 1`.

4.14 `x <= -1 & v > 1`.

4.15 `x > 1 | abs(v-3) < 1`.

Seção 4.3 Expressões literais

4.16 Definir os comandos para gerar as cadeias de caracteres `Título`, `variável` e `equações`.

4.17 Dada a cadeia de caracteres `['1' '2' '3' '4']`, inserir `' menor que '` entre eles.

4.18 Quebrar a cadeia de caracteres `'123456789012345'` nas posições `5` e `10`.

4.19 Dado `'abcde edabc eacdb'`, substituir toda ocorrência do caracter `'d'` pelo caracter `'X'`.

4.20 Dada a seqüência de caracteres `'abcdeedabceacdbdbcae'` determinar os índices de onde ocorre `'bc'`.

Seção 4.4 Execução de expressões

4.21 Avaliar o resultado da expressão dada pela cadeia de caracteres `'sqrt(4)+cos(%pi)'`.

Para `x = 1:5` e `y = 3`, avaliar as expressões abaixo,

4.22 `'exp(x-y)'`.

4.23 `'modulo(x,y) .* sinh(x*y/10)'`.

4.24 Converter para número as cadeias de caracteres `'12.3d-1'` e `'0.0456789e2'`.

4.25 Qual a diferença entre as funções `evstr` e `string`?

Capítulo 5

Gráficos

Uma das grandes virtudes do SCILAB é a facilidade que ele oferece para produzir gráficos de boa qualidade. Neste capítulo serão vistos como gerar gráficos bi e tridimensionais e os modos de imprimi-los ou gravá-los em arquivos para que possam ser incluídos em textos.

5.1 Gráficos bidimensionais

Para gerar gráficos bidimensionais são usadas as versáteis funções `plot` e `fplot2d`.

5.1.1 Função `plot`

A sintaxe da função `plot` é

```
plot(<x_1>,<y_1>,<tipo_de_linha_1>,...,<x_n>,<y_n>,<tipo_de_linha_n>)
```

onde `<x_i>` e `<y_i>` são vetores contendo as abscissas e ordenadas dos pontos a serem exibidos, respectivamente e `<tipo_de_linha_i>` é uma cadeia de 1 a 4 caracteres que especifica a cor, o estilo da linha e o marcador dos pontos dados. Os tipos de linha são mostrados na Tabela 5.1. Use o comando `help plot` para mais detalhes.

Considere os vetores

```
-->x = linspace(-8,8,50);   // define as abscissas
-->y = sin(x);   // vetor com elementos da funcao seno
-->z = cos(x);   // vetor com elementos da funcao co-seno
```

Para gerar um gráfico de $\text{sen}(x)$ em função de x basta,

```
-->plot(x,y)
```

Tabela 5.1: Tipos de linha da função `plot`.

Símbolo	Cor	Símbolo	Estilo de linha	Símbolo	Marcador
`r`	vermelho	`-`	linha sólida (*default*)	`+`	mais
`g`	verde	`--`	linha tracejada	`o`	círculo
`b`	azul	`:`	linha pontilhada	`*`	asterisco
`c`	turquesa	`-.`	linha de traço e ponto	`.`	ponto
`m`	lilás			`x`	cruz
`y`	amarelo			`'s'`	quadrado
`k`	preto			`'d'`	diamante
`w`	branco			`^`	triang. cima
				`v`	triang. baixo
				`>`	triang. direita
				`<`	triang. esquerda
				`'pentagram'`	pentagrama
				`'none'`	sem marca (*default*)

O gráfico produzido é mostrado na Figura 5.1(a). Também pode ser gerado um gráfico um pouco mais complexo, $\text{sen}(x)$ e $\cos(x)$ em função de x com os valores de $\text{sen}(x)$ em linha sólida e os pontos dados sendo destacados com (`*`) e os de $\cos(x)$ em linha tracejada e com (`o`). Para produzir o grafico da Figura 5.1(b) basta o comando

```
-->plot(x,y,'-*',x,z,'--o')
```

O SCILAB oferece algumas funções para identificação dos gráficos.

Títulos e rótulos

A função

```
xtitle(<título>[,<rótulo_x>[,<rótulo_y>[,<rótulo_z>]]])
```

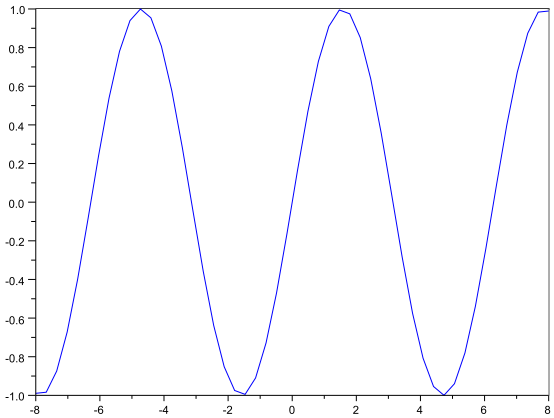
escreve a cadeia de caracteres `<título>` no cabeçalho e a cadeia de caracteres `<rótulo_i>` no `i`-ésimo eixo do gráfico. Os três argumentos `<rótulo_i>` são opcionais.

Texto

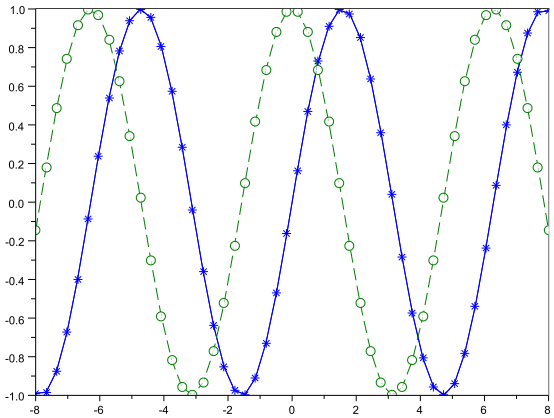
A função

```
xstring(x,y,<texto>[,<ângulo>][,<caixa>])
```

escreve a cadeia de caracteres `<texto>` na posição de coordenadas x,y com inclinação `<ângulo>`, em graus, no sentido horário. O escalar inteiro `<caixa>` informa se será desenhada uma caixa em torno do `<texto>`. Se `<ângulo> = 0` e `<caixa> = 1` então uma caixa será desenhada em torno do `<texto>`. Os argumentos `<ângulo>` e `<caixa>` são opcionais. Assim os comandos,



(a) $f(x) = \text{sen}(x)$.



(b) $f(x) = \text{sen}(x)$ e $g(x) = \cos(x)$.

Figura 5.1: Gráficos produzidos pela função `plot`.

```
-->xtitle('funcoes seno e co-seno','eixo x','eixo y')
-->xstring(4.1,0.7,'cos(x)')
-->xstring(6.1,-0.6,'sen(x)',0,1)
```

produzirão os títulos e rótulos no gráfico da Figura 5.1(b), como mostrado na Figura 5.2(a).

Legenda

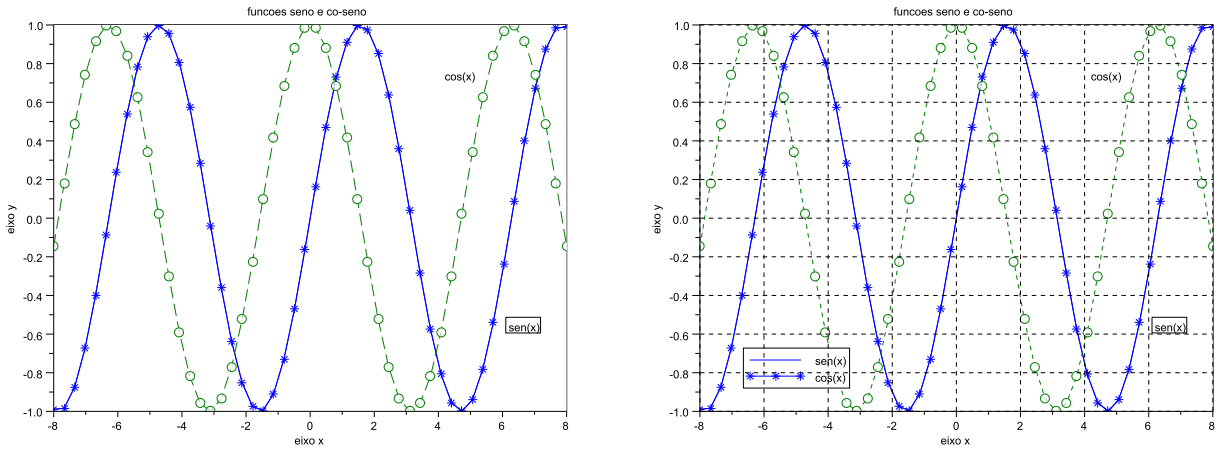
A função

`legend([<texto_1>,...,<texto_n>][,<posição>][,<caixa>])`

adiciona legendas ao esboço da figura, usando as cadeias de caracteres `<texto_i>` como rótulos. O argumento `<posição>` especifica onde as legendas serão colocadas, `1`: canto superior direito (*default*), `2`: canto superior esquerdo, `3`: canto inferior esquerdo, `4`: canto inferior direito e `5`: a legenda é colocada usando o *mouse*. A caixa de legendas move-se com o *mouse* e assim que estiver na posição desejada pressiona-se um botão. Valores negativos de `<posição>` permitem colocar as legendas fora do quadro do gráfico (usar `help legend` para mais detalhes). A variável lógica `<caixa>` indica se uma caixa será ou não desenhada em torno das legendas, sendo o valor *default* igual a `%T`. Os argumentos `<posição>` e `<caixa>` são opcionais.

Grade

A função `xgrid(<estilo>)` faz com que apareça uma grade no gráfico produzido de acordo com a constante inteira `<estilo>`, que define a forma e a cor da grade. A Figura 5.2(b) mostra o efeito dos comandos.



(a) Títulos e rótulos. Legendas e grade.

Figura 5.2: Documentação de gráficos.

```
-->legend(['sen(x)','cos(x)'],5)
-->xgrid(1)
```

5.1.2 Função `fplot2d`

A função `fplot2d` é utilizada para esboçar gráfico de função no $\mathbb{R}^2$, sendo sua sintaxe

```
fplot2d(<abscissas>,<função>,<argumentos>)
```

onde `<abscissas>` é um vetor numérico contendo as abscissas, `<função>` é o nome de uma `function` externa e `<argumentos>` é um conjunto opcional que define o estilo do gráfico na forma `<opção_1> = <valor_1>,..., <opção_n> = <valor_n>`. O argumento `<opção_i>` pode ser `style`, `rect`, `logflag`, `frameflag`, `axesflag`, `nax` e `leg`.

`style`

Esta opção é utilizada para definir como as curvas serão desenhadas, com seus valores sendo dados por um vetor com elementos inteiros de tamanho igual ao número de curvas. Se `style(i)` for positivo a linha então é desenhada como uma linha plana e o índice define qual cor será usada. Se `style(i)` for negativo ou nulo então a curva é esboçada usando marcadores e `abs(style(i))` define o tipo de marcador.

`rect`

Define os limites das abscissas e ordenadas com os valores definidos por um vetor de quatro elementos (`rect = [Xmin, Ymin, Xmax, Ymax]`). As abscissas estarão entre os limites `Xmin` e `Xmax` e as ordenadas entre `Ymin` e `Ymax`.

`logflag`

Especifica a escala (linear ou logarítmica) ao longo dos dois eixos de coordenadas com valores associados podendo ser `'nn'`, `'nl'`, `'ln'`, `'ll'`, sendo que cada letra representa um eixo com `n` significando escala normal ou linear e `l` para escala logarítmica.

`frameflag`

Esta opção controla o cálculo dos limites das coordenadas atuais a partir dos valores mínimos requeridos. Este argumento pode assumir um valor inteiro, tal que, `0`: escala *default* (sem cálculos); `1`: limites dados pela opção `rect`; `2`: calculados pelos máximos e mínimos dos vetores de abscissas e ordenadas; `3`: dados pela opção `rect`, porém aumentados para obter uma escala isométrica; `4`: calculados pelos máximos e mínimos dos vetores de abscissas e ordenadas, mas aumentados para obter uma escala isométrica; `5`: dados pela opção `rect`, porém aumentados para produzir melhores rótulos dos eixos; `6`: calculados pelos máximos e mínimos dos vetores de abscissas e ordenadas, mas aumentados para produzir melhores rótulos dos eixos; `7`: como `frameflag = 1`, todavia os gráficos anteriores são redesenhados para usar a nova escala; `8`: como `frameflag = 2`, contudo os gráficos anteriores são redesenhados para usar a nova escala e `9`: como `frameflag = 8`, porém aumentados para produzir melhores rótulos dos eixos (*default*).

`axesflag`

Especifica como os eixos serão desenhados podendo ter um valor inteiro, tal que, `0`: nada é desenhado em torno do gráfico; `1`: eixos desenhados com o eixo de ordenadas mostrado à esquerda; `2`: gráfico contornado por uma caixa sem marcadores; `3`: eixos desenhados com o eixo de ordenadas mostrado à direita; `4`: eixos desenhados centrados no meio do contorno da caixa e `5`: eixos desenhados de modo a cruzar o ponto `(0,0)` e caso este ponto não esteja dentro da área exibida então os eixos não aparecerão.

`nax`

Atribui os rótulos e define as marcas nos eixos quando a opção `axesflag = 1` for usada. Os valores são definidos por um vetor de quatro elementos inteiros (`[<nx>, <Nx>, <ny>, <Ny>]`), tal que, `<Nx>`: número de marcas principais a ser usado no eixo das abscissas; `<nx>`: número de submarcas a serem desenhadas entre marcas no eixo das abscissas; `<Ny>` e `<ny>`: fornecem informações similares, mas para o eixo das ordenadas. Se a opção `axesflag` não for usada então a opção `nax` supõe que `axesflag = 1`.

`leg`

Esta opção define as legendas referentes à cada curva devendo ser uma cadeia de caracteres na forma `<legenda_1>@<legenda_2>@...@<legenda_n>`, sendo `<legenda_i>` o título correspondente à `i`-ésima curva. Se essa opção não for definida é assumido o valor `' '`.

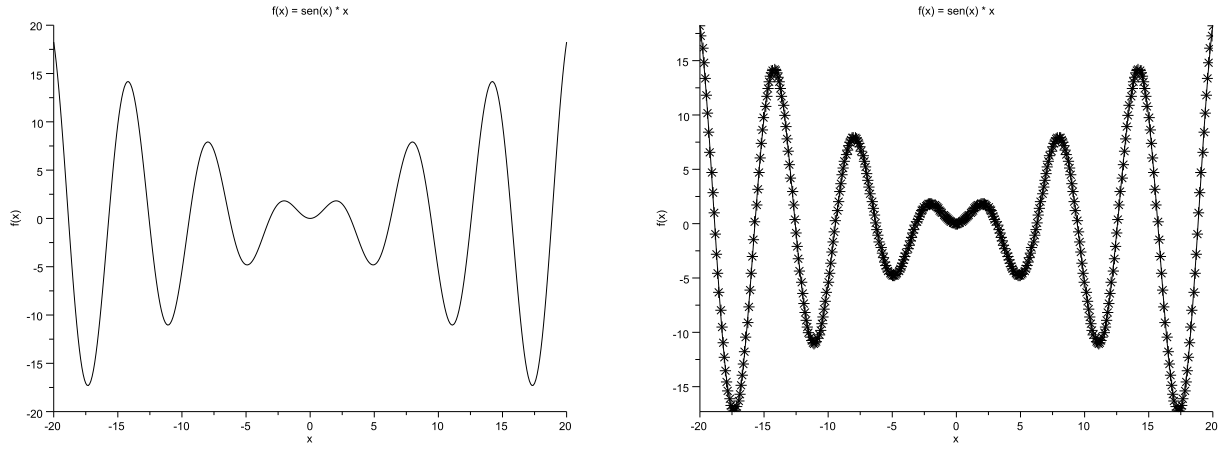
Exemplos

A seguir serão mostrados alguns gráficos gerados pela função `fplot2d`. A lista de comandos

```
-->x = -20:0.1:20;   // define valores de x
-->deff('[y]=f(x)','y=sin(x)*x')   // define funcao
-->fplot2d(x,f)   // esboca grafico
-->xtitle('f(x) = sen(x) * x','x','f(x)')   // titulos e rotulos
```

produzem o gráfico apresentado na Figura 5.3(a) apenas com o título e os rótulos dos eixos. Os comandos abaixo geram o gráfico da Figura 5.3(b), com o uso das opções `style` e `rect`,

```
-->fplot2d(x,f,style=-10,rect=[-15 -10 15 10])   // grafico com style e rect
-->xtitle('f(x) = sen(x) * x','x','f(x)')   // titulos e rotulos
```



(a) Título e rótulos.

(b) Opções `style` e `rect`.

Figura 5.3: Esboços de $f(x) = \text{sen}(x)x$ pela função `fplot2d`.

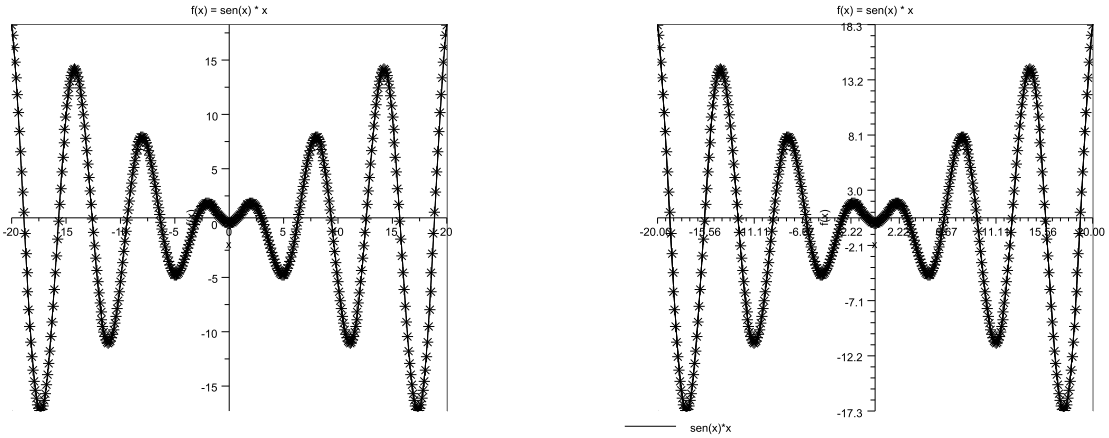
O comando abaixo apresenta as opções `frameflag` e `axesflag`, produzindo o gráfico da Figura 5.4(a),

```
-->fplot2d(x,f,frameflag=4,axesflag=5)   // grafico com frameflag e axesflag
```

As opções `nax` e `leg` podem ser vistas no comando abaixo, que gera o gráfico da Figura 5.4(b),

```
-->fplot2d(x,f,nax=[5 10 4 8],leg='sen(x)*x')   // grafico com nax e leg
```

Para mais informações sobre os comandos para manipulação de gráficos bidimensionais use `help fplot2d`.



(a) Opções `frameflag` e `axesflag`.

(b) Opções `nax` e `leg`.

Figura 5.4: Esboços de $f(x) = \text{sen}(x)x$ pela função `fplot2d` com opções.

5.2 Gráficos tridimensionais

De uma maneira similar aos gráficos bidimensionais, o SCILAB possui vários comandos para que gráficos tridimensionais sejam também facilmente esboçados. Serão descritas a seguir, de modo sucinto, algumas funções com esta finalidade. Para mais informações sobre essas funções deve ser utilizado o comando `help graphics`.

5.2.1 Função `meshgrid`

A função `[<matriz_X>,<matriz_Y>] = meshgrid(<vetor_x>,<vetor_y>)` transforma o domínio especificado por `<vetor_x>` e `<vetor_y>` em matrizes `<matriz_X>` e `<matriz_Y>` que possam ser usadas para avaliar funções de duas variáveis e fazer esboços tridimensionais de malhas e superfícies. As linhas de `<matriz_X>` são cópias de `<vetor_x>` e as colunas de `<matriz_Y>` são cópias de `<vetor_y>`,

```
-->x = -1:0.5:2   // define vetor x
 x  =
  - 1.  - 0.5    0.    0.5    1.    1.5    2.
-->y = -1:0.5:1   // define vetor y
 y  =
  - 1.  - 0.5    0.    0.5    1.
-->[X,Y] = meshgrid(x,y)   // cria matrizes X e Y
 Y  =
  - 1.   - 1.   - 1.   - 1.   - 1.   - 1.   - 1.
  - 0.5  - 0.5  - 0.5  - 0.5  - 0.5  - 0.5  - 0.5
    0.     0.     0.     0.     0.     0.     0.
    0.5    0.5    0.5    0.5    0.5    0.5    0.5
```

```
    1.     1.     1.     1.     1.     1.     1.
 X  =
  - 1.  - 0.5    0.    0.5    1.    1.5    2.
  - 1.  - 0.5    0.    0.5    1.    1.5    2.
  - 1.  - 0.5    0.    0.5    1.    1.5    2.
  - 1.  - 0.5    0.    0.5    1.    1.5    2.
  - 1.  - 0.5    0.    0.5    1.    1.5    2.
-->Z = X + Y   // operacao com X e Y
 Z  =

  - 2.   - 1.5  - 1.   - 0.5    0.     0.5    1.
  - 1.5  - 1.   - 0.5    0.     0.5    1.     1.5
  - 1.   - 0.5    0.     0.5    1.     1.5    2.
  - 0.5    0.     0.5    1.     1.5    2.     2.5
    0.     0.5    1.     1.5    2.     2.5    3.
```

5.2.2 Função `plot3d`

Esta função é uma generalização da função `plot` vista na Seção 5.1 para esboços tridimensionais. Sua sintaxe é

`plot3d(<matriz_X>,<matriz_Y>,<matriz_Z>,<argumentos>)`

onde `<matriz_X>`, `<matriz_Y>` e `<matriz_Z>` são matrizes de mesmas dimensões contendo as coordenadas tridimensionais dos pontos a serem exibidos e `<argumentos>` é um conjunto opcional que define o estilo do gráfico na forma `<opção_1> = <valor_1>,..., <opção_n> = <valor_n>`. O argumento `<opção_i>` pode ser `alpha`, `theta`, `leg`, `flag` e `ebox`.

`alpha` e `theta`

São ângulos, em graus, que definem as coordenadas esféricas do ponto do observador. O argumento `theta` descreve o ângulo no plano xy e `alpha` em torno do eixo z. Se não forem especificados são assumidos os valores pré-definidos `alpha = 35` e `theta = 45`.

`leg`

Define os rótulos referentes à cada um dos três eixos devendo ser uma cadeia de caracteres na forma `<legenda_x>@<legenda_y>@<legenda_z>`, sendo `<legenda_i>` correpondente ao `i`-ésimo eixo. Se essa opção não for fornecida então é assumido o valor `' '`.

`flag`

Essa opção é um vetor numérico com três elementos `flag = [<modo>,<tipo>,<caixa>]`. O elemento inteiro `<modo>` define a cor da superfície. Se positivo então a superfície é desenhada com a cor `<modo>` e o contorno de cada faceta é desenhado com estilo de linha e cor definidos.

Se ele for nulo então somente a malha da superfície é desenhada e se ele for negativo então a superfície é desenhada com a cor `-<modo>` e o contorno da faceta não é desenhado.

O segundo elemento inteiro `<tipo>` define a escala, tal que, `0`: o gráfico é feito usando a escala 3D atual; `1`: reescala automaticamente a caixa 3D com razão de aspecto extremo e os contornos são definidos pelo valor do argumento opcional `ebox`; `2`: reescala automaticamente a caixa 3D com razão de aspecto extremo e os contornos são computados usando os dados fornecidos; `3`: 3D isométrico com limites da caixa fornecido por `ebox`; `4`: 3D isométrico com limites derivados dos dados fornecidos; `5`: limites isométricos expandidos 3D com os limites da caixa fornecidos por `ebox` e `6`: limites isométricos expandidos 3D com limites derivados dos dados fornecidos.

O terceiro elemento inteiro `<caixa>` define a moldura em torno do gráfico, tal que, `0`: nada é desenhado em torno do gráfico; `1`: o mesmo que o anterior porque ainda não está implementado (!); `2`: somente os eixos atrás da superfície são desenhados; `3`: uma caixa em torno da superfície é desenhada e os títulos são adicionados e `4`: uma caixa em torno da superfície é desenhada e títulos e eixos são adicionados.

`ebox`

Esta opção define os limites das três coordenadas do gráfico com os valores dados em um vetor de seis elementos (`ebox = [Xmin, Xmax, Ymin, Ymax, Zmin, Zmax]`). Este argumento é usado junto com `<tipo>` da opção `flag`, caso `<tipo>` tenha o valor `1`, `3` ou `5`. Se a opção `flag` não for fornecida então `ebox` não é considerado.

Exemplos

Para produzir o esboço da função $z = \text{sen}(x)\cos(y)^2$ no intervalo $-3 \leq x \leq 3$ e $-2 \leq y \leq 2$, mostrado na Figura 5.5(a), utiliza-se os comandos

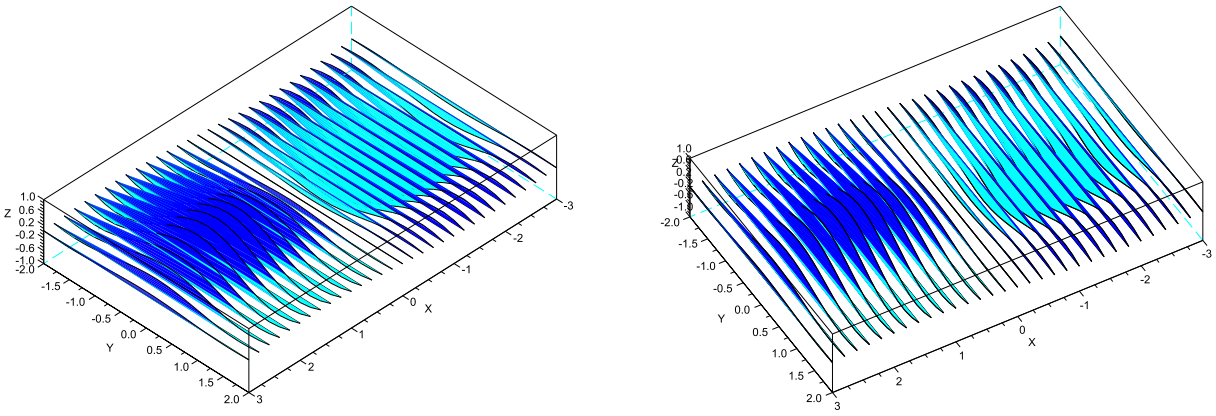
```
-->[X,Y] = meshgrid(-3:0.2:3,-2:0.2:2);// cria matrizes X e Y a partir de vetores
-->Z = sin(X).*cos(Y).^2;   // calcula pontos da funcao
-->plot3d(X,Y,Z)   // desenha grafico 3D
```

Nesta figura foram utilizados os valores pré-definidos `alpha = 35` e `theta = 45` graus. A Figura 5.5(b) apresenta o gráfico com os valores alterados para `alpha = 30` e `theta = 60`, usando o comando

```
-->plot3d(X,Y,Z,alpha=30,theta=60)
```

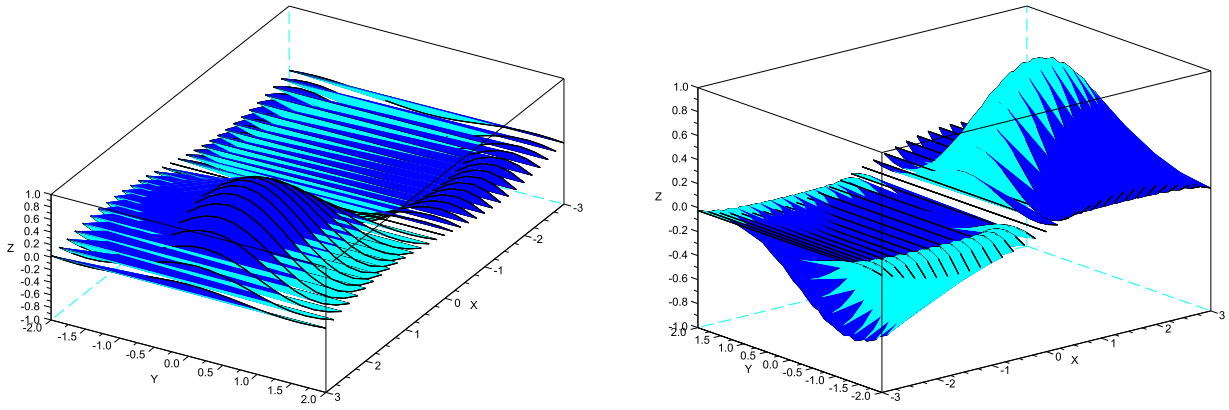
Quando os argumentos têm os valores alterados para `alpha = 60` e `theta = 30`, pelo comando abaixo, o gráfico é rotacionado do modo mostrado na Figura 5.6(a),

```
-->plot3d(X,Y,Z,alpha=60,theta=30)
```



(a) Opções `alpha = 35` e `theta = 45`. (b) Opções `alpha = 30` e `theta = 60`.

Figura 5.5: Esboços de $z = \text{sen}(x)\cos(y)^2$ usando a função `plot3d`.



(a) Opções `alpha = 60` e `theta = 30`. (b) Opções `alpha = 80` e `theta = 230`.

Figura 5.6: Esboços de $z = \text{sen}(x)\cos(y)^2$ usando `plot3d` com variação de `alpha` e `theta`.

Um gráfico com a orientação dos eixos mais natural é apresentado na Figura 5.6(b), sendo obtido pelo comando com `alpha = 80` e `theta = 230`,

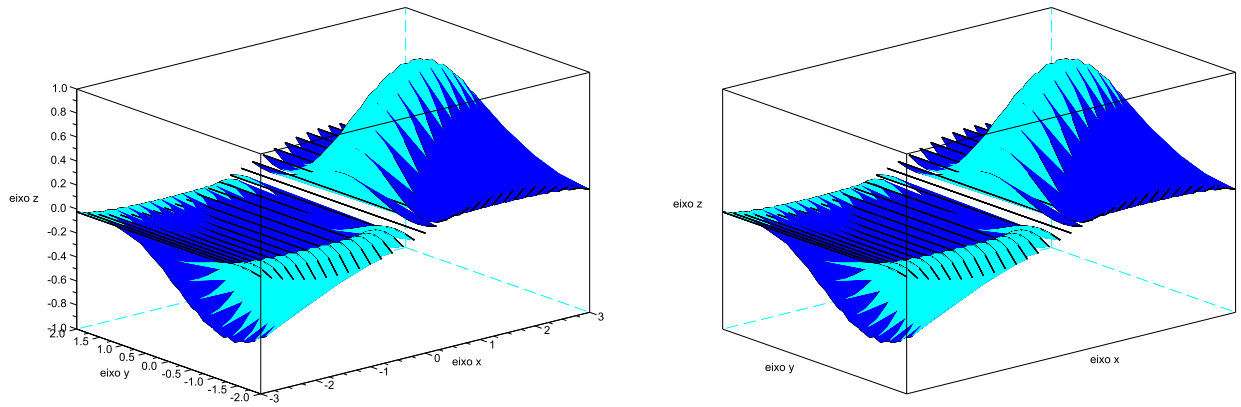
```
-->plot3d(X,Y,Z,alpha=80,theta=230)
```

Um exemplo de uso da opção `leg` pode ser visto na Figura 5.7(a), obtida pelos comandos

```
-->legenda = 'eixo x@eixo y@eixo z'   // define legenda
 legenda  =
 eixo x@eixo y@eixo z
-->plot3d(X,Y,Z,alpha=80,theta=230,leg=legenda)
```

Acrescentando a opção `flag` com os argumentos `<modo> = 0`, `<tipo> = 2` e `<caixa> = 3` pelo comando abaixo, tem-se a Figura 5.7(b),

```
-->plot3d(X,Y,Z,alpha=80,theta=230,leg=legenda,flag=[0 2 3])
```



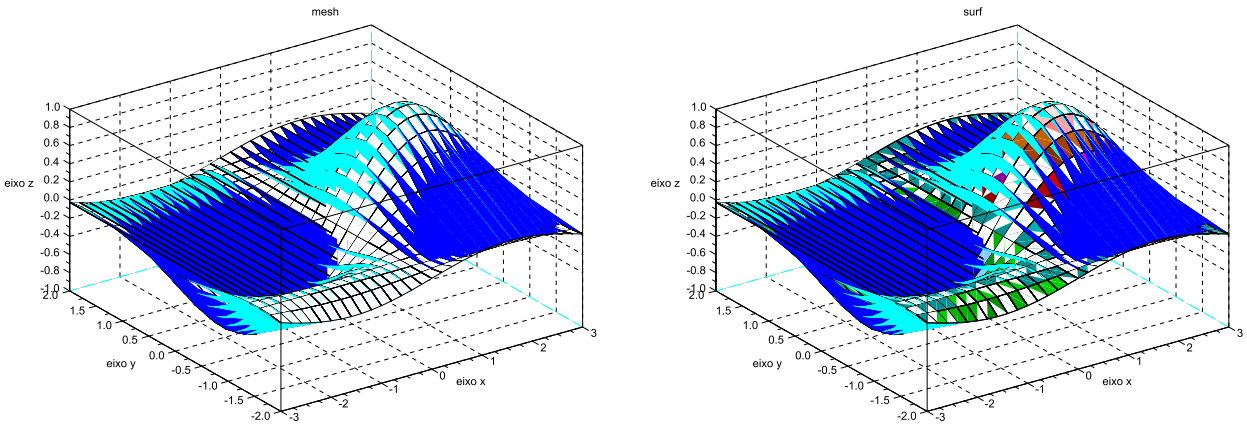
(a) Opção `leg`. (b) Opções `leg` e `flag`.

Figura 5.7: Esboços de $z = \text{sen}(x)\cos(y)^2$ usando `plot3d` com as opções `leg` e `flag`.

5.2.3 Função `mesh`

A função `mesh(<matriz_X>,<matriz_Y>,<matriz_Z>)` produz o esboço de uma malha na superfície especificada pelas matrizes `<matriz_X>`, `<matriz_Y>` e `<matriz_Z>`. Os comandos abaixo geram o esboço mostrado na Figura 5.8(a),

```
-->[X,Y] = meshgrid(-3:0.2:3,-2:0.2:2);   // define as matrizes X e Y
-->Z = sin(X).*cos(Y).^2;   // define a matriz Z com a funcao
-->mesh(X,Y,Z);   // desenha a malha
-->xgrid(1)   // coloca a grade
-->xtitle('mesh','eixo x','eixo y','eixo z')   // titulos e rotulos
```



(a) Função `mesh`. (b) Função `surf`.

Figura 5.8: Esboços de $z = \text{sen}(x)\cos(y)^2$ com diferentes funções.

5.2.4 Função `surf`

A função `surf(<matriz_X>,<matriz_Y>,<matriz_Z>)` produz uma superfície sombreada tridimensional especificada pelas matrizes `<matriz_X>`, `<matriz_Y>` e `<matriz_Z>`. Deste modo, os comandos abaixo produzem a Figura 5.8(b),

```
-->[X,Y] = meshgrid(-3:0.2:3,-2:0.2:2);   // define as matrizes X e Y
-->Z = sin(X).*cos(Y).^2;   // define a matriz Z com a funcao
-->surf(X,Y,Z);   // desenha a superficie
-->xtitle('surf','eixo x','eixo y','eixo z')   // titulos e rotulos
-->xgrid(1)   // coloca a grade
```

5.3 Janela de figura

As figuras geradas pelo SCILAB podem ser facilmente modificadas, impressas ou gravadas em um arquivo para posterior impressão ou inclusão em um texto.

Uma figura é gerada na janela **Scilab Graphic(0)** que apresenta cinco botões: File, Zoom, UnZoom, 3D Rot. e Edit.

Escolhendo File aparecem várias opções entre as quais Export, com a qual surge a janela **Xscilab**, mostrada na Figura 5.9, com várias opções para produzir um arquivo com o gráfico.

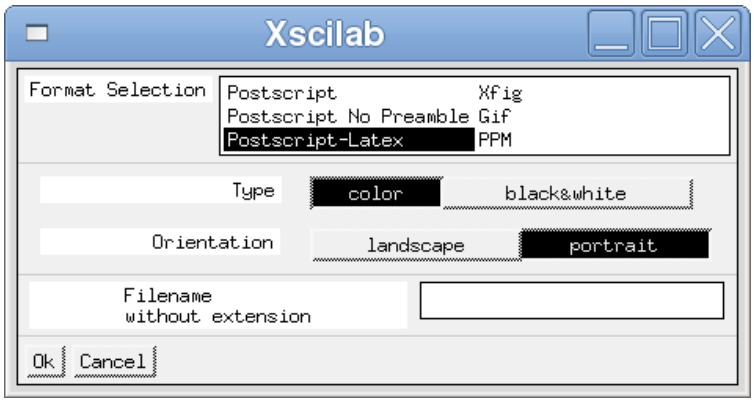


Figura 5.9: Janela para gerar arquivo com figura.

5.4 Exercícios

Seção 5.1 Gráficos bidimensionais

5.1 Seja a função

$$y = e^{1+x/10} + \cos(x)x.$$

Gerar uma tabela com 20 pontos para $-5 \leq x \leq 5$ e exibir o gráfico da função usando o comando `plot` colocando rótulos e grade.

5.2 Repetir a operação acima usando 50 pontos e sem usar grade.

5.3 Utilizar a função `fplot2d` no Exercício 5.1 variando os argumentos opcionais.

5.4 Gerar uma tabela de 40 pontos para

$$y = \text{sen}(x)x, \ -10 \leq x \leq 10$$

e exibir o seu gráfico usando o comando `fplot2d` com rótulos e grade.

5.5 Usar a função `plot` no Exercício 5.4.

Seção 5.2 Gráficos tridimensionais

5.6 Seja a função

$$z = \text{sen}(x)y^2.$$

Gerar uma malha com $-5 \leq x \leq 5$ e $-3 \leq y \leq 3$ usando a função `meshgrid`.

5.7 Gerar um gráfico usando a função `plot3d`.

5.8 Repetir o Exercício 5.7 variando os argumentos opcionais.

5.9 Usando o comando `surf` exibir a superfície de

$$z = x\cos(x)\,\mathrm{sen}(y)$$

com $-\pi \leq x \leq \pi$ e $-\pi \leq y \leq \pi$.

5.10

Seção 5.3 Janela de figura

5.11 Gravar a figura do Exercício 5.9 no arquivo `figura.eps`, utilizando a janela **Xscilab**.

5.12

5.13

5.14

5.15

Capítulo 6

Linguagem de programação

Nos capítulos anteriores foram descritos os elementos fundamentais do SCILAB, os quais possibilitam sua utilização em um modo interativo. Neste capítulo serão abordadas algumas estruturas de programação que tornarão possível desenvolver programas e funções escritos em SCILAB, tais como, estruturas condicionais e estruturas de repetição.

6.1 Programação

Além de poder entrar com cada comando de uma vez, o SCILAB permite, como qualquer outra linguagem de programação, que seja executada uma seqüência de comandos escrita em um arquivo. Por esta razão este tipo de arquivo é chamado de roteiro (*script*). Serão apresentados dois tipos de roteiros: programa e função.

6.1.1 Programa

Um arquivo contendo um programa é criado usando um editor de texto qualquer e para executar esse programa utiliza-se o comando `exec`, cuja sintaxe é

```
exec(<arquivo>,<modo>)
```

onde a cadeia de caracteres `<arquivo>` determina o nome do arquivo (incluido o caminho) com o programa a ser executado e o escalar opcional `<modo>` especifica como será a execução, de acordo com a Tabela 6.1.

Seja o programa escrito no arquivo `decsomat.sci` para gerar uma matriz com elementos aleatórios entre -100 e 100 e decompô-la na soma de três matrizes: uma triangular inferior, uma diagonal e outra triangular superior,

Tabela 6.1: Modos de execução do comando `exec`.

<modo>	Descrição
0	exibe resultados, não ecoa linha de comando e não exibe o *prompt* `-->` (valor *default*);
-1	nada é exibido;
1	ecoa cada linha de comando;
2	o *prompt* `-->` é exibido;
3	ecoa cada linha de comando e exibe o *prompt* `-->`;
4	pára antes de cada *prompt* e continua após um **Enter**;
7	modos 3 e 4 juntos.

```
// editado no arquivo decsomat.sci
// programa decomposicao_matriz
// Objetivo: decompor uma matriz aleatoria na soma de tres matrizes:
//           uma triangular inferior, uma diagonal e outra triangular superior
n = input('Ordem da matriz: ');
// gera matriz n x n com elementos aleatorios com distribuicao uniforme
// no intervalo [0,1)
A = fix(200*(grand(n,n,'def')-0.5*ones(n,n)));
D = diag(diag(A));   // obtem matriz diagonal
L = tril(A) - D;   // matriz triangular inferior com diagonal nula
U = triu(A) - D;   // matriz triangular superior com diagonal nula
A, L, D, U
```

Executando com `<modo> = 1` para ecoar cada linha de comando,

```
-->exec('decsomat.sci',1)
```

produz os resultados,

```
-->// editado no arquivo decsomat.sci
-->// programa decomposicao_matriz
-->// Objetivo: decompor uma matriz aleatoria na soma de tres matrizes:
-->//           uma triangular inferior, uma diagonal e outra triangular superior
-->n = input('Ordem da matriz: ');
Ordem da matriz: 3
-->// gera matriz n x n com elementos aleatorios com distribuicao uniforme
// no intervalo [0,1)
-->A = fix(200*(grand(n,n,'def')-0.5*ones(n,n)));
-->D = diag(diag(A));   // obtem matriz diagonal
-->L = tril(A) - D;   // matriz triangular inferior com diagonal nula
-->U = triu(A) - D;   // matriz triangular superior com diagonal nula
-->A, L, D, U
```

```
 A  =
    62.    67.    82.
  - 72.  - 74.  - 55.
    81.    93.    26.
 L  =
    0.     0.     0.
  - 72.    0.     0.
    81.    93.    0.
 D  =
    62.    0.     0.
    0.   - 74.    0.
    0.     0.     26.
 U  =
    0.    67.    82.
    0.    0.   - 55.
    0.    0.     0.
```

Um programa tem acesso às variáveis no espaço de trabalho e as variáveis criadas por ele farão parte do espaço de trabalho. No exemplo acima, as matrizes `A`, `D`, `L` e `U` foram criadas no espaço de trabalho.

O comando `halt` interrompe a execução do SCILAB até que se tecle **Enter**. Qualquer comando digitado entre o *prompt* e o **Enter** será ignorado,

```
-->a = 1   // comando qualquer
 a  =
    1.
-->halt   // causa a interrupcao no SCILAB
halt-->b = 2   // atribui o valor 2 a variavel b
-->b   // a atribuicao acima nao foi efetuada
  !--error 4
Undefined variable: b
```

6.1.2 Subprograma `function`

Um outro tipo de arquivo de roteiro é usado para o próprio usuário criar novas funções para o SCILAB. Sua sintaxe é

```
function [<parâmetros_saída>] = <nome_função> (<parâmetros_entrada>)
   <comandos>
endfunction
```

sendo `function` uma palavra-chave que determina o início da função, `<parâmetros_saída>` é um conjunto de variáveis contendo os argumentos gerados pela função (separados por vírgula e delimitados por `[` e `]`), `<nome_função>` é uma cadeia de caracteres que especifica o nome da função e `<parâmetros_entrada>` é um conjunto de variáveis com os argumentos fornecidos

à função (separados por vírgula e delimitados por (e)). Os <parâmetros_entrada> são opcionais e mesmo no caso de sua omissão os parênteses devem ser mantidos. Os <comandos> especificam o conteúdo da função e a palavra-chave endfunction o seu final.

Por exemplo, seja o arquivo parabola.sci contendo uma função para calcular as duas raízes de uma parábola,

```
// editado no arquivo parabola.sci
function [raiz1,raiz2] = parabola(a,b,c)
// Objetivo: calcular as duas raizes de uma parabola
//           parabola(a,b,c) calcula as duas raizes da parabola
//           P(X) = a*x^2 + b*x + c = 0
//           retornando-as em raiz1 e raiz2.
delta = sqrt(b^2-4*a*c);
raiz1 = (-b + delta ) / (2*a);
raiz2 = (-b - delta ) / (2*a);
endfunction
```

Os argumentos que devem ser fornecidos à função (parâmetros de entrada) são a, b e c e os valores gerados pela função (parâmetros de saída) são raiz1 e raiz2. O comando exec, apresentado na Seção 6.1.1 Programa, também é utilizado para que as function's editadas em um dado arquivo passem a fazer parte da biblioteca do SCILAB para uso posterior. Por exemplo,

```
-->exists('parabola')   // verifica que a funcao parabola nao existe
 ans  =
    0.
-->exec('parabola.sci',-1)   // funcao carregada para o espaco de trabalho
-->exists('parabola')   // verifica que parabola agora existe
 ans  =
    1.
```

Para calcular as raízes de $p(x) = 16x^2 - 8x + 5$ faz-se,

```
-->[r1,r2] = parabola(16,-8,5)   // executa a funcao parabola
 r2  =
    0.25 - 0.5i
 r1  =
    0.25 + 0.5i
```

Caso o arquivo contendo uma function seja reeditado e salvo, o comando exec deve ser novamente executado para atualizá-la no espaço de trabalho, fato que gera uma mensagem de aviso.

Ao contrário do programa no qual as variáveis são globais, em uma function a lista dos <parâmetros_entrada> e as variáveis internas são locais, ou seja, elas não tem acesso às

variáveis do espaço de trabalho. Por sua vez, os <parâmetros_saída> são criados no espaço de trabalho. No caso da `function parabola` acima, a variável `raizes` é criada no espaço de trabalho do SCILAB.

É possivel chamar uma `function` com números diferentes de parâmetros de entrada ou de saída do que aqueles definidos no arquivo que a contém. Isto é feito pela função `argn`, cuja sintaxe é

```
[<número_parâmetros_saída>,<número_parâmetros_entrada>] = argn()    .
```

Ao evocar `argn` dentro da `function` são fornecidos o <número_parâmetros_saída> e o <número_parâmetros_entrada>. O segundo argumento é opcional.

A `function pegaso` escrita no arquivo `pegaso.sci` calcula o zero pertence ao intervalo `[a b]` da função definida na cadeia de caracteres `funcao`, utilizando o robusto e eficiente método pégaso [2]. Nesta `function` é utilizado o comando `eval` para avaliação de expressão, o qual será descrito na Seção 4.4 Execução de expressões.

```
// editado no arquivo pegaso.sci
function [Raiz,CondErro,Iter] = pegaso(funcao,a,b,Toler,IterMax,Exibe)
// Objetivo: Calcular raiz de equacao pelo metodo pegaso.
//
//           PEGASO(FUNCAO,A,B,TOLER,ITERMAX,EXIBE) calcula a raiz de uma
//           equacao F(X)=0 contida no intervalo [A B] com tolerancia TOLER
//           e com no maximo ITERMAX iteracoes, usando o metodo pegaso,
//           sendo a funcao F(X) dada na cadeia de caracteres FUNCAO.
//           EXIBE especifica se os resultados intermediarios serao
//           mostrados, 0: nao exibe e 1: exibe.
//           FUNCAO, A e B sao necessarios enquanto que TOLER, ITERMAX e
//           EXIBE terao valores pre-definidos caso nao sejam fornecidos.
//
[nargsai,nargent] = argn() // numero de argumentos de saida e de entrada
if nargent < 3, error('Numero de argumentos insuficientes'); end
if nargent < 4, Toler = 1e-10; end
if nargent < 5, IterMax = 100; end
if nargent < 6, Exibe = 0; end
x = a; Fa = eval(funcao); x = b; Fb = eval(funcao);
if Exibe ~= 0
   disp('        Calculo de raiz de equacao pelo metodo pegaso')
   disp('iter     a          b          x         Fx            delta_x')
end
k = 0; x = b; Fx = Fb;
while 1
   k = k + 1; DeltaX = -Fx / (Fb - Fa) * (b - a);
   x = x + DeltaX; Fx = eval(funcao);
```

```
   if Exibe ~= 0
      mprintf('%3i%11.5f%11.5f%11.5f%14.5e%14.5e\n',k,a,b,x,Fx,DeltaX);
   end
   if ((abs(DeltaX) < Toler & abs(Fx) < Toler) | k >= IterMax), break, end
   if Fx*Fb < 0 a = b; Fa = Fb; else Fa = Fa * Fb / (Fb + Fx); end
   b = x; Fb = Fx;
end
Raiz = x;
if nargsai > 1, CondErro = abs(DeltaX) >= Toler | abs(Fx) >= Toler; end
if nargsai > 2, Iter = k; end
endfunction
```

O comando `error(<mensagem>)` exibe a cadeia de caracteres `<mensagem>` e interrompe a execução de um programa ou função.

Os argumentos `funcao`, `a` e `b` devem ser fornecidos senão uma mensagem de erro será exibida e a execução da `function` interrompida. No entanto, os argumentos `Toler` (tolerância da raiz), `IterMax` (número máximo de iterações) e `Exibe` (exibe resultados intermediários) são opcionais; caso não sejam incluídos na lista de argumentos de entrada serão atribuídos valores pré-definidos. Se forem especificados mais de seis argumentos de saída haverá a exibição de uma mensagem de erro e a interrupção da `function`.

Se nenhum ou apenas um argumento de saída for especificado então será retornado a raiz da equação na variável `Raiz`. Se forem dois argumentos então além da `Raiz` será retornado a condição de erro na variável `CondErro`. Se forem três argumentos de saída então serão retornados `Raiz`, `CondErro` e o número de iterações `Iter`. Mais de três argumentos de saída causam a exibição de uma mensagem de erro e a não execução da `function pegaso`.

Para calcular a raiz de $f(x) = \cos(x^2 - 1)\sqrt{x+1} = 0$ pertencente ao intervalo [0 2], com tolerância $\epsilon = 10^{-2}$, com no máximo 10 iterações, listando os resultados intermediários e retornado a raiz, a condição de erro e o número de iterações,

```
-->exec('pegaso.sci',0)   // carrega a funcao pegaso no espaco de trabalho
-->[r,e,i] = pegaso('cos(x^2-1)*sqrt(x+1)',0,2,1e-2,10,1)   // executa pegaso
         Calculo de raiz de equacao pelo metodo pegaso
 iter     a          b          x         Fx            delta_x
  1    0.00000    2.00000    0.47920   8.72828e-01  -1.52080e+00
  2    2.00000    0.47920    0.99219   1.41128e+00   5.12995e-01
  3    2.00000    0.99219    1.68045  -4.09987e-01   6.88254e-01
  4    0.99219    1.68045    1.52552   3.83307e-01  -1.54933e-01
  5    1.68045    1.52552    1.60038   1.54647e-02   7.48614e-02
  6    1.68045    1.60038    1.60340  -1.55627e-04   3.02353e-03
 i  =
    6.
```

```
 e  =
  F
 r  =
    1.6034004
```

Por sua vez, calculando a mesma raiz com os argumentos de entrada opcionais previamente atribuídos, ou seja, tolerância $\epsilon = 10^{-10}$, máximo de 100 iterações, não listando os resultados intermediários e além disto retornado somente a raiz e a condição de erro, faz-se

```
// executa pegaso com tres argumentos
-->[r,e] = pegaso('cos(x^2-1)*sqrt(x+1)',0,2)
 e  =
  F
 r  =
    1.6033703
```

Conforme já mencionado, as variáveis de uma `function` são locais, ou seja, só podem ser referenciadas internamente, não sendo reconhecidas pelo espaço de trabalho e outras `function`'s. No entanto, além do uso de argumentos, um outro modo de trocar informações entre o espaço de trabalho e as `functions`'s é pelo uso de variáveis globais. A declaração

```
global <lista_de_variáveis>
```

faz com que as variáveis especificadas na `<lista_de_variáveis>`, separadas por branco, tornem-se globais e portanto esta declaração deve aparecer no programa e nas `function`'s de interesse. Por exemplo, seja a função no arquivo `soma_diagonal.sci` para calcular a soma da diagonal da matriz resultante do produto das matrizes `A` e `B` definidas no espaço de trabalho,

```
// editado no arquivo soma_diagonal.sci
function somadiag = soma_diagonal
// Objetivo:
// calcular a soma da diagonal do produto das matrizes A*B do espaco de trabalho
global A B
somadiag = sum(diag(A*B));
endfunction
```

A sua execução fornece,

```
-->exec('soma_diagonal.sci',0)   // carrega soma_diagonal no espaco de trabalho
-->A = [1 2; 3 4]; B = [5 6; 7 8];   // define as matrizes A e B
-->somdg = soma_diagonal()   // executa a funcao soma_diagonal
 somdg  =
    69.
```

O uso de variáveis globais dificulta o entedimento e a modificação das `function`'s, além de tornar os módulos do programa menos independentes. Por estas razões, a utilização de variáveis globais deve ser evitada.

O comando `return` colocado dentro de uma `function` causa um retorno normal para o comando seguinte àquele que chamou a `function`. Já o comando

```
[<varesp_1>,...,<varesp_n>] = return (<varloc_1>,...,<varloc_n>)
```

faz com que as variáveis locais `<varloc_i>` sejam copiadas no espaço de trabalho com os correspondentes nomes `<varesp_i>`. Seja a função descrita no arquivo `consistencia.sci`,

```
// editado no arquivo consistencia.sci
function y = consiste(a,b)
// Objetivo: exemplo de consistencia de dados
if a <= 0 | b > 1
   y = %i
   [e,f] = return(a,b)
end
y = log10(a) + sqrt(1-b);
endfunction
```

O seu uso resulta em,

```
-->exec('consistencia.sci',0)   // carrega a function do arquivo
-->x = consiste(100,-3)   // uso da function com argumentos validos
 x  =
    4.
-->x = consiste(100,3)   // uso com argumentos invalidos
 x  =
    i
-->e,f   // verifica que as variaveis estao no espaco de trabalho
 e  =
    100.
 f  =
    3.
```

Uma função pode também ser definida na janela de comando pela função `deff`,

```
deff('[<param_saída>] = <nome_função> (<param_entrada>)', [<comandos>])
```

onde `<param_saída>` é uma lista contendo os argumentos gerados pela função (separados por vírgula e delimitados por `[` e `]`), `<nome_função>` é uma cadeia de caracteres que especifica o nome da função e `<param_entrada>` é uma lista com os argumentos fornecidos à função (separados por vírgula e delimitados por `(` e `)`). A matriz de caracteres `<comandos>` especifica o corpo da função. Por exemplo, seja a função para multiplicar e dividir dois números,

```
-->deff('[mul,div] = operacao(a,b)',['mul = a * b';'div = a / b'])//define funcao
-->[multiplica, divide] = operacao(3,2)   // uso da funcao operacao
 divide  =
    1.5
 multiplica  =
    6.
```

6.2 Estruturas condicionais

Uma estrutura condicional permite a escolha do grupo de comandos a serem executados quando uma dada condição for satisfeita ou não, possibilitando desta forma alterar o fluxo natural de comandos. Esta condição é representada por uma expressão lógica.

6.2.1 Estrutura `if-end`

A estrutura condicional mais simples do SCILAB é

```
if <condição>
   <comandos>
end
```

Se o resultado da expressão lógica `<condição>` for `T` (*verdadeiro*) então a lista `<comandos>` será executada. Se o resultado for `F` (*falso*) então `<comandos>` não serão executados. Considere o programa `logaritmo_decimal.sci`,

```
// programa logaritmo_decimal
// Objetivo: calcular logaritmo decimal
x = input('Entre com x: ');
if x > 0
   LogDec = log10(x);
   disp([x LogDec]);
end
```

e a execução para `x = 0.001`,

```
-->exec('logaritmo_decimal.sci',-1)
Entre com x: 0.001
    0.001  - 3.
```

Neste exemplo, o logaritmo decimal de `x` será atribuído a `LogDec` se, e somente se, o valor de `x` for maior que `0`.

6.2.2 Estrutura `if-else-end`

No caso de haver duas alternativas, uma outra estrutura condicional deve ser usada

```
if <condição>
   <comandos_1>
else
   <comandos_2>
end
```

Se o resultado da expressão lógica `<condição>` for `T` (*verdadeiro*) então somente a lista contendo `<comandos_1>` será executada. Se `<condição>` for `F` (*falso*) então será a lista

<comando_2> a única a ser executada. O programa funcao_modular.sci utiliza a função modular $f(x) = |2x|$. Se x for positivo ou nulo então será atribuído à variável fx o resultado de 2*x, todavia, se x for negativo então fx será igual a -2*x,

```
// programa funcao_modular
// Objetivo: avaliar uma funcao modular
x = input('Entre com x: ');
if x >= 0
   fx = 2 * x;
else
   fx = -2 * x;
end
disp([x fx]);
```

Executando com x = -3 produz,

```
-->exec('funcao_modular.sci',-1)
Entre com x: -3
  - 3.    6.
```

6.2.3 Estrutura if-elseif-end

Quando houver mais de duas alternativas, a estrutura if-else-end do SCILAB torna-se

```
if <condição_1>
   <comandos_1>
elseif <condição_2>
   <comandos_2>
elseif <condição_3>
   <comandos_3>
     .  .  .
else
   <comandos_n>
end
```

A lista <comandos_1> será executada se <condição_1> for igual a T (*verdadeiro*); já a lista <comandos_2> será executada se <condição_2> for T e assim para as outras condições. Se nenhuma das condições for T então <comandos_n> será executada. Quando a <condição_i> for satisfeita e os <comandos_i> executados, a estrutura if-elseif-end será abandonada, ou seja, o controle do processamento será transferido para o comando imediatamente após o end.

Seja o programa modulo.sci para calcular o valor absoluto de um número real ou complexo,

```
// programa modulo
// Objetivo: calcular o valor absoluto de um numero real ou complexo
```

```
a = input('Entre com a: ');
if imag(a) ~= 0
   b = sqrt(real(a).^2+imag(a).^2);
elseif a < 0
   b = -a;
else
   b = a;
end
disp([a b]);
```

Para `a = 3 - 4*%i`,

```
--->exec('modulo.sci',-1)
Entre com a: 3 - 4 * %i
    3. - 4.i    5.
```

Deste modo foi executado o primeiro comando para o qual a condição `imag(a) ~= 0` foi satisfeita. Assim, na estrutura `if-elseif-end` uma única lista de comandos é executada.

6.2.4 Estrutura `select-case-end`

Esta estrutura é similar a `if-elseif-end` e sua sixtaxe é

```
select <expressão>
   case <expressão_1> then <comandos_1>
   case <expressão_2> then <comandos_2>
   case <expressão_3> then <comandos_3>
     .   .   .
   case <expressão_n> then <comandos_n>
   else <comandos_e>
end
```

A lista `<comandos_1>` será executada se `<expressão>` for igual a `<expressão_1>`. A lista `<comandos_2>` será executada se `<expressão>` for igual a `<expressão_2>` e assim para as outras expressões. Se `<expressão>` não for igual a nenhuma das expressões anteriores então `<comandos_e>` será executada. Cabe ressaltar que somente uma lista de comandos será executada.

Considere o programa `posicao_poltrona.sci` para determinar a posição de uma poltrona para quem *entra* em um ônibus,

```
// programa posicao_poltrona
// Objetivo: determinar a posicao de uma poltrona em onibus
poltrona = input('Entre com o numero da poltrona: ');
fila = fix(poltrona/4) + 1;
select modulo(poltrona,4)
```

```
   case 0 then
      if poltrona > 0
         fila = fila - 1;
         posicao = 'esquerda / corredor'
         mprintf('poltrona%3i na fila%3i ''a %s\n',poltrona,fila,posicao)
      else
         posicao = 'nao existe'
         mprintf('poltrona%3i %s\n',poltrona,posicao)
      end
   case 1 then
      posicao = 'direita / janela'
      mprintf('poltrona%3i na fila%3i ''a %s\n',poltrona,fila,posicao)
   case 2 then
      posicao = 'direita / corredor'
      mprintf('poltrona%3i na fila%3i ''a %s\n',poltrona,fila,posicao)
   case 3 then
      posicao = 'esquerda / janela'
      mprintf('poltrona%3i na fila%3i ''a %s\n',poltrona,fila,posicao)
   else
      posicao = 'nao existe'
      mprintf('poltrona%3i %s\n',poltrona,posicao)
end
```

Por exemplo, as posições das poltronas de números 15, 42 e -5 são

```
-->exec('posicao_poltrona.sci',-1)
Entre com o numero da poltrona: 15
poltrona 15 na fila  4 'a esquerda / janela

-->exec('posicao_poltrona.sci',-1)
Entre com o numero da poltrona: 42
poltrona 42 na fila 11 'a direita / corredor

-->exec('posicao_poltrona.sci',-1)
Entre com o numero da poltrona: -5
poltrona -5 nao existe
```

6.3 Estruturas de repetição

As estruturas de repetição fazem com que uma seqüência de comandos seja executada repetidamente até que uma dada condição de interrupção seja satisfeita. O SCILAB possui duas estruturas de repetição, as estruturas `for-end` e a `while-end`, com as quais é possível construir uma terceira estrutura com interrupção no interior.

6.3.1 Estrutura `for-end`

A estrutura `for-end` permite que um grupo de comandos seja repetido um número determinado de vezes. Sua sintaxe é

```
for <variável>=<arranjo>
   <comandos>
end
```

onde `<variável>` é a variável-de-controle que assume todos os valores contidos no vetor linha `<arranjo>`. Assim, o número de repetições da lista `<comandos>` é igual ao número de elementos no vetor `<arranjo>`. A variável-de-controle não pode ser redefinida dentro da estrutura `for-end`.

O programa `primeiros_impares.sci` mostra que a soma dos `n` primeiros números ímpares é igual ao quadrado de `n`,

```
// programa primeiros_impares
// Objetivo: verificar propriedade dos numeros impares
n = input('Entre com n: ');
Soma = 0;
for i = 1:2:2*n-1
   Soma = Soma + i;
end
disp([Soma n^2])
```

Quando executado para `n = 5`,

```
-->exec('primeiros_impares.sci',-1)
Entre com n: 5
    25.    25.
```

Para `n = 5` a variável-de-controle `i` assume os valores `1 3 5 7 9`, cuja soma é igual a 25. Para mostrar que as estruturas `for-end` podem estar encadeadas, considere o programa `soma_matriz.sci` para calcular a soma dos elementos das linhas, colunas e diagonal de uma matriz,

```
// programa soma_matriz
// Objetivo:
// calcular a soma dos elmentos das linhas, colunas e diagonal de matriz
A = input('Entre com a matriz: ');
[nlin,ncol] = size(A);
Soma_Linhas = zeros(nlin,1);
Soma_Colunas = zeros(1,ncol);
Soma_Diagonal = 0;
for i = 1:nlin
   Soma_Diagonal = Soma_Diagonal + A(i,i);
```

```
   for j = 1:ncol
      Soma_Linhas(i) = Soma_Linhas(i) + A(i,j);
      Soma_Colunas(j) = Soma_Colunas(j) + A(i,j);
   end
end
A, Soma_Linhas, Soma_Colunas, Soma_Diagonal
```

Um quadrado mágico de ordem n é uma matriz com elementos não repetidos e com valores entre 1 e n^2, tal que a soma dos elementos das linhas, das colunas e da diagonal sejam iguais. Para o quadrado mágico de ordem 4,

```
-->exec('soma_matriz.sci',0)
Entre com a matriz: [16 2 3 13; 5 11 10 8; 9 7 6 12; 4 14 15 1]
 A  =
    16.    2.     3.     13.
    5.     11.    10.    8.
    9.     7.     6.     12.
    4.     14.    15.    1.
 Soma_Linhas  =
    34.
    34.
    34.
    34.
 Soma_Colunas  =
    34.    34.    34.    34.
 Soma_Diagonal  =
    34.
```

Cumpre observar que o SCILAB possui comandos para determinar estes somatórios de um modo bem simples, pelo uso da função `sum` que fornece a soma dos elementos de uma matriz (ver Tabela 3.3 na página 45).

6.3.2 Estrutura `while-end`

A estrutura `while-end`, ao contrário da `for-end`, repete um grupo de comandos um número indeterminado de vezes. Sua sintaxe é

```
while <condição>
   <comandos_1>
   [else <comandos_2>]
end
```

Enquanto a expressão lógica `<condição>` for `T` (*verdadeiro*) a lista `<comandos_1>` será repetida. Quando ela for `F` (*falsa*) então a lista `<comandos_2>` será executada. O comando `else` é opcional.

Por exemplo, seja o programa `precisao.sci` para determinar a precisão de um computador,

```
// programa precisao
// Objetivo: determinar a precisao de um computador
n = 0;
Epsilon = 1;
while 1 + Epsilon > 1
   n = n + 1;
   Epsilon = Epsilon / 2;
end
n, Epsilon, %eps
```

Quando executado fornece

```
-->exec('precisao.sci',0)
 n  =
    53.
 Epsilon  =
    1.110D-16
 %eps  =
    2.220D-16
```

`Epsilon` é a chamada precisão da máquina ϵ, ou seja, o maior número que somado a 1 é igual a 1. Para computadores com aritmética IEEE $\epsilon = 2^{-53}$. Comparada com a variável especial `%eps` $= 2^{-52}$ do SCILAB,

```
-->1 + %eps - 1
 ans  =
    2.220D-16
-->1 + Epsilon - 1
 ans  =
    0.
```

Note que quando `%eps` for somado a 1 resulta um número maior que 1. O mesmo não ocorre com `Epsilon`, porque qualquer valor igual ou menor do que ele somado a 1 será simplesmente 1, ou seja, o computador já não consegue mais representar $1 + \epsilon$.

6.3.3 Estrutura com interrupção no interior

A estrutura `while-end` permite que um grupo de comandos seja repetido um número indeterminado de vezes, no entanto, a condição de interrupção é testada no início da estrutura. Em várias situações em programação se faz necessário interromper a execução da repetição verificando a condição no interior ou final da estrutura e não somente no seu início.

O comando `break` interrompe a execução de uma estrutura `while-end` ou `for-end` e transfere a execução para o comando imediatamente seguinte ao `end`. Em repetições aninhadas, o `break` interrompe a execução apenas da estrutura mais interna. Assim, uma repetição com

condição de interrupção no interior pode ter a forma

```
while %T
   <comandos_1>
   if <condição>
      break
   end
   <comandos_2>
end
```

A estrutura `while-end` é executada indefinidamente a princípio, pois a condição do `while` é sempre `T` (*verdadeiro*). Contudo, quando a `<condição>` do `if` for satisfeita o comando `break` será executado causando a interrupção da repetição `while-end`.

Seja o programa `racional.sci` para fornecer a aproximação racional de um número positivo com uma dada tolerância,

```
// programa racional
// Objetivo: fornecer a aproximacao racional de numero positivo com dada precisao
while %T
   numero = input('Entre com um numero > 0: ');
   if numero <= 0
      break
   end
   tol = input('Entre com tolerancia: ');
   [num,den] = rat(numero,tol);
   disp([numero num den numero-num/den])
end
```

Por exemplo, para $\sqrt{2}$ e π,

```
-->exec('racional.sci',0)

Entre com um numero > 0: sqrt(2)
Entre com tolerancia: 1e-5
    1.4142136    239.    169.    0.0000124

Entre com um numero > 0: %pi
Entre com tolerancia: 1e-10
    3.1415927    208341.    66317.    1.224D-10

Entre com um numero > 0: 0
```

Ele lista continuamente a representação racional de um número fornecido enquanto este for positivo.

6.4 Depuração de programa

O comando `pause` interrompe a execução de um programa e transfere o controle para o teclado; quando em uma função ele interrompe a sua execução. No modo de pausa aparece o símbolo do *prompt* indicando o nível da pausa, por exemplo, `-1->`. É disponibilizado, então, um novo espaço de trabalho no qual todas as variáveis de nível mais baixo estão acessíveis, mesmo as variáveis de uma função.

Para voltar ao espaço de trabalho que evocou a pausa usa-se o comando `return`. Conforme mencionado na Seção 6.1.2 Subprograma `function` o comando

```
[<varesp_1>,...,<varesp_n>] = return (<varloc_1>,...,<varloc_n>)
```

faz com que as variáveis locais `<varloc_i>` sejam copiadas no espaço de trabalho que evocou a pausa com os correspondentes nomes `<varesp_i>`. Se o comando não for usado então as variáveis de nível mais baixo estão protegidas e não podem ser modificadas. Seja a função no arquivo `avalia.sci`,

```
// editado no arquivo avalia.sci
function expressao = avalia(a)
divide = a/10;
raiz = sqrt(a);
logdec = log10(a)
expressao = divide + raiz + logdec;
endfunction
```

e sua execução,

```
-->exec('avalia.sci',-1)   // carrega a funcao para o espaco de trabalho
-->valor = avalia(100)   // avalia a funcao
 valor  =
    22.
-->exists('raiz')   // verifica a inexistencia da variavel local raiz
 ans  =
    0.
```

A variável `raiz` não existe no espaço de trabalho por se tratar de uma variável local. Se for colocado o comando `pause`, todas as variáveis definidas antes dele estarão disponíveis. Por exemplo, se for colocado depois de `raiz = sqrt(a);`

```
-->exec('avalia.sci',-1)   // carrega a funcao modificada
Warning : redefining function: avalia. Use funcprot(0) to avoid this message
-->valor = avalia(100)   // avalia a funcao entrando no modo de pausa
-1->raiz   // existencia da variavel local raiz no novo espaco de trabalho
 ans  =
    10.
```

```
-1->exists('logdec')   // inexistencia da variavel logdec
 ans  =
    0.
-1->return
 valor  =
    22.
```

Neste caso a variável local `raiz` está disponível no espaço de trabalho e `logdec` não está porque ela foi definida depois do `pause`.

6.5 Exercícios

Seção 6.1 Programação

6.1 Criar o arquivo `almaxmin.sci`

```
n = input('Ordem da matriz: ');
A = rand(n,n);
maior = max(A)
menor = min(A)
```

Executar o programa acima para `n` = 5, 10 e 30.

6.2 Escrever uma `function` para calcular a norma-2 de um vetor.

6.3 Escrever uma `function` para calcular a norma-∞ de um vetor.

6.4 Dada a matriz triangular inferior $L = \{l_{ij}\}$ e o vetor c, escrever uma `function` para calcular y tal que $Ly = c$:

$$\begin{pmatrix} l_{11} & & & & \\ l_{21} & l_{22} & & & \\ l_{31} & l_{32} & l_{33} & & \\ \vdots & \vdots & \vdots & & \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{pmatrix}.$$

6.5 Dada a matriz triangular superior $U = \{u_{ij}\}$ e o vetor d, escrever uma `function` para calcular x tal que $Ux = d$:

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ & u_{22} & u_{23} & \cdots & u_{2n} \\ & & u_{33} & \cdots & u_{3n} \\ & & & & \vdots \\ & & & & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{pmatrix}.$$

Seção 6.2 Estruturas condicionais

6.6 Qual o valor de `LogDec` no exemplo sobre a estrutura `if-end` mostrada na Seção 6.2.1 para `x = 2`?

6.7 Encontrar o valor de `fx` no exemplo sobre `if-else-end` da Seção 6.2.2 para `x = 10`.

6.8 Achar o valor de `b` no exemplo de `if-elseif-end` da Seção 6.2.3 para `a = -3`.

6.9 Qual o valor de `b` no exemplo sobre a estrutura `if-elseif-end` da Seção 6.2.3 para `a = -3+4*%i`?

6.10 Encontrar as posições das poltronas números 4, 25 e 40 usando o exemplo da estrutura `select-case-end` da Seção 6.2.4.

Seção 6.3 Estruturas de repetição

6.11 Determinar o valor de `Soma` no exemplo sobre a estrutura `for-end` mostrada na Seção 6.3.1 para `n = 7`.

6.12 Calcular a soma das linhas, das colunas e da diagonal de uma matriz de Toeplitz de ordem 5.

6.13 Explicar como é feita a determinação da precisão de um computador.

6.14 Calcular a norma-2 do vetor `x = [1e200 2e200 3e200]` usando a `function` escrita no Exercício 6.2. Qual a alteração a ser feita para que a função calcule corretamente?

6.15 Calcular a raiz de $f(x) = \text{sen}(x)x + 4 = 0$ que está no intervalo [1 5], utilizando a `function pegaso` mostrada na Seção 6.1.2.

Seção 6.4 Depuração de programa

6.16

6.17

6.18

6.19

6.20

Capítulo 7

Comandos de entrada e saída

O SCILAB fornece algumas facilidades para especificar o formato de saída dos resultados, gravação e leitura das variáveis de uma sessão de trabalho e de arquivos.

7.1 Formato de exibição

Para saber o valor de uma variável basta entrar com o seu nome. O resultado é exibido usando um formato pré-definido,

```
-->%pi
 %pi  =
    3.1415927
```

O comando `disp(<variável>)` é usado para exibir o conteúdo de `<variável>` sem mostrar o seu nome ou para exibir uma cadeia de caracteres contida na `<variável>`.

```
-->disp('numeros aleatorios entre 0 e 1'), disp(grand(2,6,'def'))
 numeros aleatorios entre 0 e 1
    0.8147237    0.9057919    0.1269868    0.9133759    0.6323592    0.0975404
    0.135477     0.8350086    0.9688678    0.2210340    0.3081671    0.5472206
```

Ao exibir um resultado numérico o SCILAB segue diversas regras. Se o resultado for um número real (ponto flutuante), ele é mostrado com dez caracteres, incluindo o ponto decimal e o sinal. Quando o sinal for positivo é exibido um caracter branco em seu lugar. Se os dígitos significativos do resultado estiverem fora desta faixa então o resultado será exibido em notação científica,

```
-->disp([%e*1e-5 %e*1e5 %e*1e-10 %e*1e10])
    0.0000272    271828.18    2.718D-10    2.718D+10
```

A função `format(<tipo>,<dígitos>)` é usada para alterar o formato numérico de exibição, onde `<tipo>` é um dos caracteres `'v'` ou `'e'` para indicar formato variável ou em notação científica, respectivamente. O parâmetro `<dígitos>` indica o número máximo de caracteres a serem exibidos, incluindo sinal, ponto decimal e no caso de formato `'e'` os quatro caracteres referentes à potência de 10. Caso esse parâmetro não seja fornecido é assumido o valor 10. Uma vez definido o formato de exibição ele é mantido até que seja modificado por um outro comando `format`,

```
-->format('v',8), %pi   // pi com formato variavel com 8 caracteres
 ans  =
    3.14159
-->format('v',12), %pi   // pi com formato variavel com 12 caracteres
 ans  =
    3.141592654
-->format('e',8), %eps   // eps em notacao cientifica com 8 caracteres
 ans  =
    2.2D-16
-->format('e',12), %eps   // eps em notacao cientifica com 12 caracteres
 ans  =
    2.22045D-16
-->sqrt(%eps)   // verificar que o ultimo formato e' mantido
 ans  =
    1.49012D-08
```

A função `mprintf` exibe dados formatados pertencentes ao espaço de trabalho na tela principal do SCILAB. Sua sintaxe é

```
mprintf(<formato>,<variáveis>)
```

onde a cadeia de caracteres `<formato>` contém caracteres alfanuméricos e/ou especificações de conversão para exibir na tela a lista `<variáveis>`. Estas especificações de conversão são delimitadas pelo caracter `%` e uma das letras `i`, `e`, `f`, `g` ou `s`, de acordo com a Tabela 7.1.

Tabela 7.1: Formatos de exibição.

Formato	Especificação
%n`i`	usado para valores inteiros, sendo n o tamanho do campo de exibição;
%n.d`f`	notação na forma $[-]888.888$, sendo n o tamanho do campo (número total de caracteres exibidos) e d o número de dígitos decimais;
%n.d`e`	notação na forma $[-]8.888 \pm 88$, sendo n o tamanho do campo (número total de caracteres exibidos) e d o número de dígitos decimais;
%n.d`g`	equivalente a %n.d`e` ou %n.d`f`, dependendo de qual formato for mais curto, além disso os zeros insignificantes não são exibidos;
%n`s`	exibe caracteres em um campo de tamanho n.

Deste modo,

```
-->mprintf('a precisao deste computador =%12.5e\n',%eps/2)
a precisao deste computador = 1.11022e-16
```

onde `\n` é usado para começar uma nova linha. Quando for necessário ter o caracter (') exibido basta usá-lo duas vezes,

```
-->mprintf('o valor de pi e'' aproximadamente%13.10f\n',%pi)
o valor de pi e' aproximadamente 3.1415926536
```

Assim, a exibição pode ser feita pelos comandos `disp` e `mprintf`,

```
-->x = 1:0.5:3;
-->M = [x;sqrt(x)]';
-->format('v',10)
-->disp(M)
    1.     1.
    1.5    1.2247449
    2.     1.4142136
    2.5    1.5811388
    3.     1.7320508
-->mprintf('%5.3f%10.5f\n',M)
1.000   1.00000
1.500   1.22474
2.000   1.41421
2.500   1.58114
3.000   1.73205
```

7.2 Espaço de trabalho

Durante uma sessão as variáveis utilizadas residem no espaço de trabalho do SCILAB e podem ser armazenadas em um arquivo quando desejado. Conforme visto, o comando `who` lista o nome das variáveis que estão sendo usadas, ou seja, que estão presentes no espaço de trabalho. Por sua vez, o comando `whos` fornece informações mais detalhadas sobre essas variáveis. Estes dois comandos listam as variáveis criadas pelo usuário e as definidas pelo próprio SCILAB. Por exemplo, no início de uma sessão, quando o usuário criou apenas as variáveis `a`, `b`, `C` e `d`, tem-se,

```
-->a = 2.5, b = [1.2 3.2 -5.4], C = [2.1 3.4; 6.1 -9.3], d = 'caracteres'
 a  =
    2.5
 b  =
    1.2    3.2  - 5.4
 C  =
    2.1    3.4
```

```
    6.1  - 9.3
 d  =
 caracteres
-->whos
Name                    Type          Size          Bytes
whos                    function                    9000
d                       string        1 by 1        64
C                       constant      2 by 2        48
b                       constant      1 by 3        40
a                       constant      1 by 1        24
M                       constant      5 by 2        96
.
.   algumas variaveis foram removidas da lista
.
%t                      boolean       1 by 1        24
%f                      boolean       1 by 1        24
%eps                    constant      1 by 1        24
%io                     constant      1 by 2        32
%i                      constant      1 by 1        32
%e                      constant      1 by 1        24
%pi                     constant      1 by 1        24
```

As variáveis no espaço de trabalho criadas pelo usuário podem ser removidas, incondicionalmente, usando o comando `clear`

`--> clear tempo raiz`	remove as variáveis `tempo` e `raiz`,
`--> clear`	remove **todas** as variáveis do espaço de trabalho. **Atenção:** não será solicitada a confirmação. **Todas** as variáveis estarão, irremediavelmente, removidas.

7.2.1 Gravar dados

O comando `save` é usado para gravar as variáveis do espaço de trabalho em um arquivo. Sua sintaxe é

```
save(<nome_do_arquivo>,<variáveis>)
```

onde `<nome_do_arquivo>` é uma cadeia de caracteres que especifica o nome do arquivo binário onde variáveis do espaço de trabalho serão gravadas e `<variáveis>` é uma lista de nomes que define quais as variáveis do espaço de trabalho criadas pelo usuário serão gravadas no arquivo `<nome_do_arquivo>`. As variáveis devem estar separadas por vírgula. Se os nomes `<variáveis>` não forem especificados, então todas as variáveis do espaçco de trabalho criadas pelo usuário serão salvas. Por exemplo, considerando as variáveis `a`, `b`, `C` e `d` criadas acima,

```
-->save('quatro.dat')   // salva as quatro variaveis no arquivo 'quatro.dat'
-->save('duas.dat',b,d)   // salva as variaveis b e d no arquivo 'duas.dat'
```

7.2.2 Recuperar dados

O comando `load` é usado para recuperar os dados gravados em um arquivo pelo comando `save` e colocá-los de volta no espaço de trabalho. Sua sintaxe é

```
load(<nome_do_arquivo>,<variáveis>)
```

onde `<nome_do_arquivo>` e `<variáveis>` são os mesmos definidos para o comando `save`. Se o arquivo contiver uma variável com o mesmo nome de uma já existente no espaço de trabalho então o comando `load` faz com que a variável do espaço de trabalho seja substituída pela variável existente no arquivo,

```
-->clear   // remove todas as variaveis do espaco de trabalho
-->d = 'novos'   // atribui novo valor a variavel d
 d  =
 novos
-->load('duas.dat','b','d') // recupera as variaveis b e d do arquivo 'duas.dat'
-->d   // verifica o valor de d
 ans  =
 caracteres
```

7.2.3 Entrada de dados

A leitura de dados pelo teclado é feita pelo comando `input`, cuja sintaxe é

```
<variável> = input(<mensagem>,'string')
```

O comando acima exibe a cadeia de caracteres `<mensagem>` e espera até que o usuário forneça o valor de `<variável>` pelo teclado. O segundo argumento `'string'` ou simplesmente `'s'` é opcional e informa que `<variável>` é uma cadeia de caracteres,

```
-->Indice = input('entre com o indice: ')
entre com o indice: 13
 Indice  =
    13.
// cadeia de caracteres com parametro 's'
-->texto = input('fornecer o texto: ','s')
fornecer o texto: programa SCILAB
 texto  =
 programa SCILAB
// caracteres sem parametro 's', mas com ' '
-->texto = input('fornecer o texto: ')
fornecer o texto: 'programa SCILAB'
 texto  =
 programa SCILAB
```

7.2.4 Janela de mensagem

A interação SCILAB / usuário pode também ser feita por meio de janelas de mensagens utilizando comandos, tais como, `x_mdialog` e `x_message`. O comando

```
<resultado> = x_mdialog(<título>,<rótulos_i>,<valores_i>)
```

exibe uma janela contendo a cadeia de caracteres `<título>` com uma mensagem, vários vetores de caracteres `<rótulos_i>` com o nome do `i`-ésimo valor requerido e os vetores de caracteres `<valores_i>` com os valores iniciais sugeridos, correspondentes a `<rótulos_i>`. Se for acionado o botão **Ok** então `<resultado>` receberá os valores sugeridos ou os outros valores digitados; caso seja acionado o botão **Cancel** então `<resultado> = []`. Por exemplo,

```
-->nomes = ['Comprimento:';'Largura:';'Altura:'] // define os nomes das variaveis
 nomes  =
!Comprimento:  !
!              !
!Largura:      !
!              !
!Altura:       !
-->valor_ini = ['';'';'']   // define os valores iniciais (nenhum no caso)
 valor_ini  =
!  !
!  !
!  !
!  !
!  !
-->dimensao = x_mdialog('Entre com as dimensoes', nomes, valor_ini)// abre janela
```

apresenta a janela mostrada na Figura 7.1. Digita-se os valores desejados e pressiona-se **Ok**. O vetor de caracteres `dimensao` é apresentado. Em seguida converte os caracteres para números, utilizando a função `eval` (ver Seção 4.4 Execução de expressões),

```
 dimensao  =
!1.23  !
!      !
!4.56  !
!      !
!7.89  !
-->com = eval(dimensao(1))   // valor numerico do comprimento
 com  =
    1.23
-->lar = eval(dimensao(2))   // valor numerico da largura
 lar  =
    4.56
-->alt = eval(dimensao(3))   // valor numerico da altura
 alt  =
    7.89
```

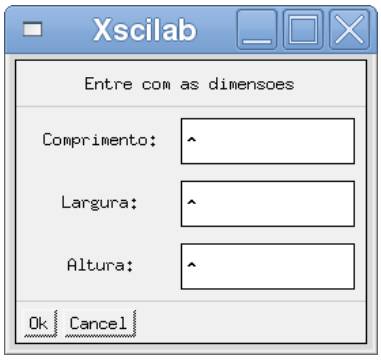


Figura 7.1: Janela do comando `x_mdialog`.

Por sua vez,

```
<resultado> = x_message(<título>,<botões_i>)
```

exibe uma janela contendo a cadeia de caracteres `<título>` com uma mensagem e vários vetores de caracteres `<botões_i>` com as opções. Se for acionado o `i`-ésimo botão então `<resultado> = i`. Se `<botões_i>` não for especificado assume-se o valor **Ok**. Por exemplo, o comando

```
-->resp = x_message(['A matriz e'' simetrica.'; 'Usar Cholesky?'],['Sim' 'Nao'])
```

apresenta a janela mostrada na Figura 7.2. Se for escolhida a opção **Sim** então a variável `resp` receberá o valor `1`,

```
 resp  =
    1.
```

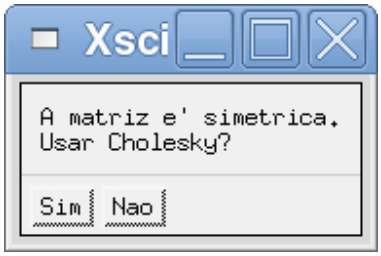


Figura 7.2: Janela do comando `x_message`.

7.3 Diário

Todos os comandos solicitados pelo usuário e as respostas fornecidas pelo SCILAB (com excessão de gráficos) podem ser gravados em um arquivo ASCII para que possam ser impressos ou mesmo incluídos em algum relatório, como foi feito neste texto!

Para esta gravação basta usar o comando `diary(<nome_do_arquivo>)`, a partir do qual a interação SCILAB / usuário será registrada no arquivo, cujo nome é dado pela cadeia de caracteres `<nome_do_arquivo>`. A finalização do registro é feita por `diary(0)`. Por exemplo,

```
-->diary('diario')   // cria o arquivo diario
-->a = 1, b = sqrt(%eps)
 a  =
    1.
 b  =
    1.490D-08
-->diary(0)   // fecha o arquivo diario
```

O conteúdo do arquivo `diario` é,

```
-->unix('more diario');   // executa o comando 'more' do Linux
-->a = 1, b = sqrt(%eps)
 a  =
    1.
 b  =
    1.490D-08
-->diary(0)   // fecha o arquivo diario
```

7.4 Leitura e gravação de dados

A transferência de dados entre o espaço de trabalho e algum dispositivo de entrada e saída (arquivo em disco, impressora etc) aumenta a utilização do SCILAB visto tornar possível, por exemplo, até a troca de informações com um outro programa.

7.4.1 Abertura de arquivo

A função `mopen` abre um arquivo, sendo sua sintaxe

```
[<idenarq>,<erro>] = mopen(<nome_do_arquivo>,<permissão>)
```

Deste modo, a função `mopen` associa o nome externo do arquivo dado pela cadeia de caracteres `<nome_do_arquivo>` à unidade `<idenarq>` que será utilizada nos comandos de entrada e saída no modo especificado pela `<permissão>`. O escalar `<erro>` indica a ocorrência de algum erro. Os caracteres permitidos para `<permissão>` estão listados na Tabela 7.2. O caractere `b` indica que o arquivo é binário.

Se `<permissão>` for omitida então será assumido o valor `'r'`. Caso o comando `mopen` tenha sucesso ao abrir o arquivo, ele retornará o identificador de arquivo `<idenarq>` contendo um número inteiro positivo e o valor de `<erro>` será `0`. Em caso de algum erro o valor de `<erro>` será negativo. O `<idenarq>` é usado com outras rotinas de entrada e saída para identificar o arquivo no qual as operações serão realizadas. Por exemplo,

Tabela 7.2: Atributos de arquivo.

<permissão>	Especificação
'r' ou 'rb'	Abre o arquivo para leitura.
'r+' ou 'r+b'	Abre o arquivo para atualização (leitura e escrita), mas não cria o arquivo.
'w' ou 'wb'	Abre o arquivo para escrita e caso necessário cria o arquivo. Porém, remove o conteúdo do arquivo existente.
'w+' ou 'w+b'	Abre o arquivo para atualização (leitura e escrita) e se necessário cria o arquivo. Todavia, remove o conteúdo do arquivo existente.
'a' ou 'ab'	Cria e abre um arquivo novo ou abre um arquivo já existente para escrita, anexando ao final do arquivo.
'a+' ou 'a+b'	Cria e abre um arquivo novo ou abre um arquivo já existente para atualização (leitura e escrita), anexando ao final do arquivo.

```
-->[fid,erro] = mopen('dados.dat','w')   // abre o arquivo dados.dat para escrita
 erro  =
    0.
 fid  =
    1.
```

7.4.2 Fechamento de arquivo

A função `mclose(<idenarq>)` fecha o arquivo previamente aberto pela função `mopen`, cujo identificador associado a este arquivo seja `<idenarq>`. Quando um arquivo é fechado, a associação entre o identificador `<idenarq>` e o arquivo físico `<nome_do_arquivo>` é desfeita,

```
-->mclose(1)   // fecha o arquivo dados.dat
 ans  =
    0.
```

7.4.3 Gravação em arquivo

A função `mfprintf` grava dados formatados em um arquivo e sua sintaxe é

```
mfprintf(<idenarq>,<formato>,<variáveis>)
```

onde a cadeia de caracteres `<formato>` contém caracteres alfanuméricos e/ou especificações de conversão para gravar no arquivo com identificador associado `<idenarq>` a lista contendo `<variáveis>`. Estas especificações de conversão são delimitadas pelo caracter `%` e uma das letras `i`, `e`, `f`, `g` ou `s`, de acordo com a Tabela 7.1.

No exemplo abaixo, uma tabela contendo x, $\sqrt{x}$ e e^{-x} para $1 \leq x \leq 2$ é gerada e gravada no arquivo `sqrtexp.dat`,

```
-->x = 1:0.2:2;   // define o vetor x
-->tab = [x; sqrt(x); exp(-x)]'   // gera a tabela tab
 tab  =
    1.     1.           0.3678794
    1.2    1.0954451    0.3011942
    1.4    1.183216     0.2465970
    1.6    1.2649111    0.2018965
    1.8    1.3416408    0.1652989
    2.     1.4142136    0.1353353
-->[fid,erro] = mopen('sqrtexp.dat','w') // abre arquivo sqrtexp.dat para escrita
 erro  =
    0.
 fid  =
    1.
-->mfprintf(fid,'%5.2f%15.10f%15.10f\n',tab)   // escreve tabela no arquivo
-->mclose(fid)   // fecha o arquivo
 ans  =
    0.
```

O conteúdo do arquivo `sqrtexp.dat` é,

```
-->unix('more sqrtexp.dat');   // executa o comando 'more' do Linux
 1.00   1.0000000000   0.3678794412
 1.20   1.0954451150   0.3011942119
 1.40   1.1832159566   0.2465969639
 1.60   1.2649110641   0.2018965180
 1.80   1.3416407865   0.1652988882
 2.00   1.4142135624   0.1353352832
```

7.4.4 Leitura em arquivo

A função `mfscanf` efetua a leitura de dados formatados em um arquivo, sendo sua sintaxe

```
[<tamanho>,<variáveis>] = mfscanf(<num>,<idenarq>,<formato>)
```

onde `<idenarq>` é o identificador associado ao arquivo no qual está sendo feita a leitura dos dados escritos no formato especificado na cadeia de caracteres `<formato>` e o parâmetro opcional `<num>` especifica o número de vezes que o formato é usado. Os dados são convertidos segundo a cadeia de caracteres `<formato>` e atribuídos à lista `<variáveis>`. As especificações de conversão são mostradas na Tabela 7.1. A variável `<tamanho>` retorna o número de elementos que foram lidos do arquivo com sucesso,

```
 -->fid = mopen('sqrtexp.dat','r')   // abre arquivo para leitura
 fid  =
    1.
-->[n,a,b,c] = mfscanf(1,fid,'%5f%15f%15f') // leitura de 1 linha em 3 variaveis
 c  =
```

```
    0.3678795
 b  =
    1.
 a  =
    1.
 n  =
    3.
-->vetor = mfscanf(1,fid,'%5f%15f%15f')   // leitura de 1 linha em 1 vetor
 vetor  =
    1.2    1.0954452    0.3011942
-->matriz = mfscanf(2,fid,'%5f%15f%15f')   // leitura de 2 linhas em 1 matriz
 matriz  =
    1.4    1.183216     0.2465970
    1.6    1.2649111    0.2018965
```

A função `mseek(<número>,<idenarq>,<posição>)` é usada para posicionar o acesso a um registro do arquivo, cujo identificador seja `<idenarq>`. A nova posição é especificada a partir da distância dada pelo `<número>` de bytes do início (se `<posição> = 'set'`), ou da posição atual (se `<posição> = 'cur'`) ou do fim de arquivo (se `<posição> = 'end'`). Se o parâmetro `<posição>` não for especificado é assumido o valor `'set'`,

```
-->mseek(0,fid)   // posiciona leitura para o inicio do arquivo
-->matriz = mfscanf(6,fid,'%5f%15f%15f')   // leitura das 6 linhas em 1 matriz
 matriz  =
    1.     1.           0.3678795
    1.2    1.0954452    0.3011942
    1.4    1.183216     0.2465970
    1.6    1.2649111    0.2018965
    1.8    1.3416408    0.1652989
    2.     1.4142135    0.1353353
```

Durante o processo de leitura é importante verificar se o último registro do arquivo já foi lido. A função `meof(<idenarq>)` faz esta verificação no arquivo de identificador `<idenarq>`. Se o último registro já foi lido então será retornado o valor `1`, caso contrário `0` será retornado. Continuando o exemplo acima,

```
-->meof(fid)   // verifica se fim de arquivo
 ans  =
    0.
-->[n,m] = mfscanf(1,fid,'%5f%15f%15f')   // leitura de 1 linha em 1 variavel
 m  =
     []
 n  =
  - 1.
-->meof(fid)   // verifica se fim de arquivo
 ans  =
```

```
    1.
-->mclose(fid)   // fecha o arquivo
 ans  =

    0.
```

7.5 Exercícios

Seção 7.1 Formato de exibição

Verificar as diferenças entre os formatos de exibição para as variáveis `a = sqrt(2)`, `e = exp(10)`, `x = 1:10`, `y = x'` e `M = rand(3,3)`,

7.1 `a, M, disp(a), disp(M)`.

7.2 `format(15,'e'), a, e, x`.

7.3 `mprintf('%10.5f %12.3e\n',a,e)`.

7.4 `mprintf('%5.1f\n',x)` e `mprintf('%5.1f\n',y)`.

7.5 Explicar a diferença de dígitos entre a variável `tab` e o conteúdo do arquivo `sqrtexp.dat` na mostrado Seção 7.4.3.

Seção 7.2 Espaço de trabalho

Observar os resultados dos comandos para controle do espaço de trabalho utilizando as variáveis dos Exercícios 7.1–7.5,

7.6 `who, whos`.

7.7 `save ('dados'), clear, who`.

7.8 `load ('dados'), who`.

7.9 `save ('esptrab',x,y), clear, who`.

7.10 `load ('esptrab'), who, x, y`.

Seção 7.3 Diário

Entre com os comandos abaixo,

7.11 `diary('meudiario')`.

7.12 `a = 4`.

7.13 `b = log10(a)`.

7.14 `diary(0)`.

7.15 Verificar o conteúdo do arquivo `meudiario`.

Seção 7.4 Leitura e gravação de dados

7.16 Gerar uma tabela com 10 linhas de $x, \text{sen}(x),\ 0 \leq x \leq \pi/2$ e gravá-la no arquivo `seno.dat`.

7.17 Fechar o arquivo `seno.dat` gravado no Exercício 7.16 e verificar o seu conteúdo.

7.18 Acrescentar o valor $\pi + 1, \text{sen}(\pi + 1)$ na última linha do arquivo `seno.dat` e verificar o seu conteúdo.

7.19 Abrir o arquivo `seno.dat` para leitura e ler a primeira linha.

7.20 Gravar os valores `-1` e `-2` na última linha de `seno.dat` e observar o resultado.

Capítulo 8

Computação científica

8.1 Medidas de tempo

O SCILAB provê duas maneiras de medir o tempo gasto para executar um conjunto de comandos: o tempo de execução e o tempo de CPU.

Usando o `tic-toc` é possível saber o tempo gasto para a execução de um grupo de comandos. A sua sintaxe é

```
tic()
   <comandos>
<variável> = toc()
```

O `tic` inicia a contagem do tempo e o `toc` fornece para `<variável>` o tempo, em segundos, passado desde o último `tic`. A atribuição do tempo gasto a `<variável>` é opcional. Se o computador estiver executando várias tarefas simultaneamente, `tic-toc` pode não ser uma medida muito confiável. Uma outra medida é obtida por `timer` que fornece o tempo de CPU (unidade central de processamento) que é o tempo gasto para execução de operações aritméticas e lógicas. Pode ser usado na forma

```
<variável_1> = timer()
   <comandos>
<variável_2> = timer() - <variável_1>
```

Por exemplo, considere a execução do programa no arquivo `medidas.sci`,

```
// define duas matrizes aleatorias
A = grand(2500,300,'unf',0,10); B = grand(300,1000,'unf',0,100);
tic()   // inicia contagem do tempo de execucao
t0 = timer();   // inicia contagem do tempo de CPU
C = A * B - 5;
```

```
maximo = max(abs(diag(C)));
tempo_cpu = timer() - t0   // tempo de CPU
tempo_exec = toc()   // tempo de execucao
```

Os resultados produzidos foram,

```
-->exec('medidas.sci',0)
 tempo_cpu  =
    7.63
 tempo_exec  =
    7.815
```

A diferença de tempo produzida por `tic-toc` e `timer` pode ser bem significativa se o computador estiver executando outros programas ao mesmo tempo. Além disso, esses tempos medidos dependerão do computador utilizado.

8.2 Álgebra linear

Nesta seção serão mostrados alguns comandos do SCILAB relativos aos tópicos usualmente abordados em textos de Cálculo Numérico. Para um conhecimento mais amplo dos comandos do SCILAB com respeito a vetores e matrizes usar os comandos `apropos vector` e `apropos matrix`.

8.2.1 Parâmetros da matriz

O SCILAB disponibiliza diversas funções para se obter informações sobre matrizes, tais como, normas, número de condição, determinante, posto e traço.

Normas

A função `norm(<variável>,<tipo>)` fornece a norma `<tipo>` de um vetor ou matriz contido em `<variável>`, de acordo com a Tabela 8.1. No caso de um vetor, `<tipo>` pode ser qualquer número (inteiro ou real, positivo ou negativo).

Exemplos de normas vetoriais,

```
-->x = [1 2 3 4 5];   // define vetor x
-->norm(x,1)   // norma de soma de magnitudes
 ans  =
    15.
-->norm(x), norm(x,2)   // norma Euclidiana
 ans  =
    7.4161985
 ans  =
    7.4161985
```

Tabela 8.1: Normas vetoriais e matriciais.

Sejam `v = [` $v_1\ v_2 \ldots v_m$ `]` e `M = [` $m_{11}\ m_{12} \ldots m_{1p}$`;` $m_{21}\ m_{22} \ldots m_{2p}$`;` $\ldots$ `;` $m_{n1}\ m_{n2} \ldots m_{np}$ `]`		
`<tipo>`	Descrição	Nome
`norm(v,p)`	$\|v\|_p = \sqrt[p]{\sum_{i=1}^{m} \|v_i\|^p}$	norma-p
`norm(v,2)` ou `norm(v)`	$\|v\|_2 = \sqrt{\sum_{i=1}^{m} \|v_i\|^2}$	norma Euclidiana
`norm(v,'inf')`	$\|v\|_\infty = \max_{1\le i\le m} \|v_i\|$	norma de máxima magnitude
`norm(M,1)`	$\|M\|_1 = \max_{1\le j\le p} \sum_{i=1}^{n} \|m_{ij}\|$	norma de soma máxima de coluna
`norm(M,'inf')`	$\|M\|_\infty = \max_{1\le i\le n} \sum_{j=1}^{p} \|m_{ij}\|$	norma de soma máxima de linha
`norm(M,'fro')`	$\|M\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{p} \|m_{ij}\|^2}$	norma de Frobenius
`norm(M,2)` ou `norm(M)`	$\max \sigma_i$ (valores singulares)	norma espectral

```
-->norm(x,'inf')   // norma de maxima magnitude
 ans  =
    5.
-->norm(x,-%pi)   // norma -pi
 ans  =
    0.9527515
```

e normas matriciais,

```
-->A = [1 2 3; 4 5 6; 7 8 9];   // define matriz A
-->norm(A,1)   // norma de soma maxima de coluna
 ans  =
    18.
-->norm(A,'inf')   // norma de soma maxima de linha
 ans  =
    24.
-->norm(A,'fro')   // norma de Frobenius
 ans  =
```

```
    16.881943
-->norm(A,2), norm(A)   // norma espectral
 ans  =
    16.848103
 ans  =
    16.848103
```

Número de condição

A função `cond(<matriz>)` calcula o número de condição de `<matriz>` quadrada definido em termos da norma-2, ou seja, é a razão entre o maior e o menor valor singular,

$$\texttt{cond(M)} = \kappa_2(M) = \|M\|_2\|M^{-1}\|_2 = \frac{\sigma_{\max}}{\sigma_{\min}}.$$

Para a matrix,

```
-->B = [5 3 0; -1 8 6; 4 2 9];
-->cond(B)   // numero de condicao
 ans  =
    2.6641005
```

A função `rcond(<matriz>)` fornece uma estimativa do recíproco do número de condição de `<matriz>` definido em termos da norma-1. Se `<matriz>` for bem-condicionada então `rcond` é próximo de 1 e se ela for malcondicionada `rcond` será próximo de 0. Para a matriz `B` acima,

```
-->r = rcond(B)
 r  =
    0.2094488
-->1/r
 ans  =
    4.7744361
```

Determinante

A função `det(<matriz>)` calcula o determinante de `<matriz>`, sendo ela quadrada. Na forma `[<mantissa>,<expoente>] = det(<matriz>)` o valor do determinante é apresentado em notação científica: `<mantissa>`$\times 10^{\texttt{<expoente>}}$. Para `A` e `B` definidas previamente,

```
-->det(B)   // determinante de B
 ans  =
    399.
-->[mantissa,expoente] = det(A)   // determinante de A em notacao cientifica
 expoente  =
    6.6613381
 mantissa  =
  - 16.
-->det(A)
 ans  =
    6.661D-16
```

Posto

O posto de uma matriz A $(m \times n)$ é o número máximo de vetores linhas ou de vetores colunas de A que são linearmente independentes, sendo posto$(A) \leq \min(m, n)$. No SCILAB o posto de `<matriz>` é obtido por `rank(<matriz>,<tolerância>)`, sendo igual ao número de valores singulares de `<matriz>` maiores que `<tolerância>`. Se `<tolerância>` não for dada então é assumido o valor `<tolerância> = max(m,n) * norm(<matriz>) * %eps`. Para as matrizes `A` e `B`,

```
-->rank(A)   // posto de A
 ans  =
    2.
-->rank(A,1e-20)   // posto de A com tolerancia 1e-20
 ans  =
    3.
-->rank(B)   // posto de B
 ans  =
    3.
```

Traço

A função `trace(<matriz>)` determina o traço de `<matriz>`, isto é, a soma dos elementos da sua diagonal principal. Para a matriz `A` definida anteriormente,

```
-->trace(A)   // traco de A
 ans  =
    15.
```

8.2.2 Decomposições

O SCILAB disponibiliza vários tipos de decomposições de matrizes entre as quais, LU, Cholesky, QR e SVD.

Decomposição LU

A função `lu(<matriz>)` faz a decomposição LU de `<matriz>`, de dimensão $m \times n$, usando o método de eliminação de Gauss com pivotação parcial.

O uso do comando `[<fatorL>,<fatorU>,<matrizP>] = lu(<matriz>)` gera uma matriz triangular inferior unitária `<fatorL>` (dimensão $m \times \min(m, n)$), uma matriz triangular superior `<fatorU>` (dimensão $\min(m, n) \times n$) e uma matriz de permutações `<matrizP>` (dimensão $n \times n$), tal que `<matrizP> * <matriz> = <fatorL> * <fatorU>`. Por exemplo,

```
-->M = [2 -3 5; 4 1 -1; 1 8 6]   // define a matriz M
 M  =
    2.  - 3.    5.
    4.    1.  - 1.
```

```
    1.    8.    6.
-->[L,U,P] = lu(M)   // decomposicao LU de M
 P  =
    0.    1.    0.
    0.    0.    1.
    1.    0.    0.
 U  =
    4.    1.    - 1.
    0.    7.75    6.25
    0.    0.      8.3225806
 L  =
    1.      0.           0.
    0.25    1.           0.
    0.5   - 0.4516129    1.
-->R = P * M - L * U   // verificando que P * M = L * U
 R  =
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
```

Contudo, o uso de dois parâmetros de saída `[<matrizInf>,<matrizSup>] = lu(<matriz>)` gera uma matriz triangular superior `<matrizSup> = <fatorU>` e uma matriz triangular inferior `<matrizInf>` tal que `<matrizP> * <matrizInf> = <fatorL>` de modo que `<matriz> = <matrizInf> * <matrizSup>`. Para as matrizes acima,

```
-->[Tinf,Tsup] = lu(M)   // decomposicao LU de M com dois parametros
 Tsup  =
    4.    1.    - 1.
    0.    7.75    6.25
    0.    0.      8.3225806
 Tinf  =
    0.5   - 0.4516129    1.
    1.      0.           0.
    0.25    1.           0.
-->M - Tinf * Tsup   // verificando que M = Tinf * Tsup
 ans  =
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
-->Tsup - U   // verificando que Tsup = U
 ans  =
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
-->P * Tinf - L   // verificando que P * Tinf = L
```

```
 ans  =
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
```

Decomposição de Cholesky

A função `<fator> = chol(<matriz>)` fatora `<matriz>` simétrica definida positiva pelo método de Cholesky produzindo uma matriz `<fator>` triangular superior tal que `<fator>' * <fator> = <matriz>`. Por exemplo,

```
-->A = [4 -2 2; -2 10 -7; 2 -7 30]   // define a matriz A
 A  =
    4.  - 2.     2.
  - 2.    10.  - 7.
    2.  - 7.     30.
-->U = chol(A)   // calcula o fator U
 U  =
    2.  - 1.    1.
    0.    3.  - 2.
    0.    0.    5.
-->A - U' * U   // verificando que A = U'*U
 ans  =
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
-->B = [1 -2 4; -2 5 3; 4 3 8]   // define a matriz B
 B  =
    1.  - 2.    4.
  - 2.    5.    3.
    4.    3.    8.
-->F = chol(B)   // calcula o fator F
            !--error 29
Matrix is not positive definite.
```

Houve um erro porque a matriz `B` não é definida positiva.

Decomposição QR

A função `qr` faz a decomposição QR de uma matriz de dimensão $m \times n$. O uso na forma `[<matrizQ>,<matrizR>]=qr(<matriz>)` produz uma matriz ortonormal `<matrizQ>` de ordem m (`<matrizQ>'*<matrizQ>=eye(`m`,`m`)`) e uma matriz triangular superior `<matrizR>` com a mesma dimensão $m \times n$ de `<matriz>`, de modo que `<matriz>=<matrizQ>*<matrizR>`. Sejam,

```
-->A = [2 -4; 3 5; 1 -7; 8 6]   // define a matriz A
```

```
 A  =
    2.  - 4.
    3.    5.
    1.  - 7.
    8.    6.
-->[Q,R] = qr(A)   // calcula fatores Q e R
 R  =
  - 8.8317609  - 5.4349298
    0.         - 9.8214835
    0.           0.
    0.           0.
 Q  =
  - 0.2264554    0.5325844  - 0.5445505  - 0.6070721
  - 0.3396831  - 0.3211171    0.5753910  - 0.6711366
  - 0.1132277    0.7753803    0.5906262    0.1926801
  - 0.9058216  - 0.1096497  - 0.1534623    0.3793593
-->round(Q'*Q)   // verificando que Q e' ortonormal
 ans  =
    1.    0.    0.    0.
    0.    1.    0.    0.
    0.    0.    1.    0.
    0.    0.    0.    1.
-->round( A - Q * R )   // verificando que A = Q*R
 ans  =
    0.    0.
    0.    0.
    0.    0.
    0.    0.
```

Quando os quatro caracteres (`,'e'`) forem colocados após o nome de uma matriz de dimensão $(m \times n,\ m > n)$ o SCILAB produz uma decomposição econômica, de modo que `<matrizQ>` terá dimensão $(m \times n)$ e `<matrizR>` será $(n \times n)$. Para a matriz `A` acima,

```
-->[Qe,Re] = qr(A,'e')   // calcula fatores Q e R na forma economica
 Re  =
  - 8.8317609  - 5.4349298
    0.         - 9.8214835
 Qe  =
  - 0.2264554    0.5325844
  - 0.3396831  - 0.3211171
  - 0.1132277    0.7753803
  - 0.9058216  - 0.1096497
-->round(A - Qe * Re)   // verificando que A = Qe * Re
 ans  =
    0.    0.
    0.    0.
```

```
    0.    0.
    0.    0.
```

A função `qr` possui diversas formas de ser utilizada; para maiores informações use `help qr`.

Decomposição em valores singulares

A função `svd` faz a decomposição em valores singulares de uma matriz de dimensão $m \times n$. O comando `[<matrizU>,<matrizS>,<matrizV>] = svd(<matriz>)` produz uma matriz ortonormal `<matrizU>` de ordem m, uma matriz diagonal `<matrizS>` de dimensão $m \times n$ contendo os valores singulares de `<matriz>` em ordem decrescente e uma matriz ortonormal `<matrizV>` de ordem n, de modo que `<matriz> = <matrizU> * <matrizS> * <matrizV>'`. Por exemplo,

```
-->A = [2 -4; 3 5; 1 -7; 8 6]   // define a matriz A
 A  =
    2.  - 4.
    3.    5.
    1.  - 7.
    8.    6.
-->[U,S,V] = svd(A)   // calcula os fatores U, S e V de A
 V  =
  - 0.5257311    0.8506508
  - 0.8506508  - 0.5257311
 S  =
    12.476603    0.
    0.           6.9522923
    0.           0.
    0.           0.
 U  =
    0.188444     0.5471902  - 0.5445505  - 0.6070721
  - 0.4673105  - 0.0110328    0.5753910  - 0.6711366
    0.4351204    0.6516942    0.5906262    0.1926801
  - 0.7461769    0.5251246  - 0.1534623    0.3793593
-->round( U'*U )   // verificando que U e' ortonormal
 ans  =
    1.    0.    0.    0.
    0.    1.    0.    0.
    0.    0.    1.    0.
    0.    0.    0.    1.
-->round( V'*V )   // verificando que V e' ortonormal
 ans  =
    1.    0.
    0.    1.
-->round( A - U * S * V')   // verificando que A = U*S*V'
```

```
 ans  =
    0.    0.
    0.    0.
    0.    0.
    0.    0.
```

Se os quatro caracteres (`,'e'`) forem colocados após o nome de uma matriz de dimensão ($m \times n,\ m > n$) será computada uma decomposição econômica, de modo que `<matrizU>` terá dimensão ($m \times n$), `<matrizS>` e `<matrizR>` serão ($n \times n$). Utilizando a mesma matriz `A`,

```
-->[Ue,Se,Ve] = svd(A,'e')   // calcula os fatores U, S e V na forma economica
 Ve  =
  - 0.5257311    0.8506508
  - 0.8506508  - 0.5257311
 Se  =
    12.476603    0.
    0.           6.9522923
 Ue  =
    0.188444     0.5471902
  - 0.4673105  - 0.0110328
    0.4351204    0.6516942
  - 0.7461769    0.5251246
-->round( A - Ue * Se * Ve')   // verificando que A = Ue*Se*Ve'
 ans  =
    0.    0.
    0.    0.
    0.    0.
    0.    0.
```

Para se obter apenas os valores singulares de `<matriz>` em `<vetor>` basta `<vetor> = svd(<matriz>)`,

```
-->s = svd(A)   // calcula apenas os valores singulares de A
 s  =
    12.476603
    6.9522923
```

A função `svd` possui outras formas de ser utilizada; para maiores informações use `help svd`.

8.2.3 Solução de sistemas

Dado um sistema $Ax = b$, a solução x pode ser facilmente calculada pelo SCILAB pelo operador (\), por exemplo, para um sistema com matriz simétrica,

```
-->A = [4 -2 2; -2 10 -7; 2 -7 30]   // matriz dos coeficientes
```

```
 A  =
    4.  - 2.     2.
  - 2.    10.  - 7.
    2.  - 7.     30.
-->b = [8 11 -31]'   // vetor dos termos independentes
 b  =
    8.
    11.
  - 31.
-->x = A \ b   // solucao de Ax = b
 x  =
    3.
    1.
  - 1.
-->r = ( b - A * x )'   // verificando a exatidao da solucao
 r  =
    0.    0.    0.
```

No caso de uma matriz não simétrica,

```
-->B = [1 -3 2; -2 8 -1; 4 -6 5]   // matriz dos coeficientes
 B  =
    1.  - 3.    2.
  - 2.    8.  - 1.
    4.  - 6.    5.
-->c = [11 -15 29]'   // vetor dos termos independentes
 c  =
    11.
  - 15.
    29.
-->y = B \ c   // solucao de By = c
 y  =
    2.
  - 1.
    3.
-->r = ( c - B * y )'   // verificando a exatidao da solucao
 r  =
    0.    0.    0.
```

8.2.4 Inversa

A função `inv(<matriz>)` calcula a inversa de `<matriz>` de ordem n, tal que `<matriz> * inv(<matriz>) = eye(`n`,`n`)`. Para a matriz `M` acima,

```
-->E = inv(B)   // inversa de B
 E  =
```

```
  - 1.4166667  - 0.125    0.5416667
  - 0.25         0.125    0.125
    0.8333333    0.25   - 0.0833333
-->round( B * E - eye(3,3) )   // verificando que B * E = I
 ans  =
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
```

8.2.5 Autovalores e autovetores

Os autovalores são os zeros do polinômio característico da matriz. A função `poly` constrói o polinômio característico de uma matriz. Por exemplo,

```
-->A = [4 -1 3; -1 5 2; 3 2 8]   // define a matriz A
 A  =
    4.  - 1.    3.
  - 1.    5.    2.
    3.    2.    8.
-->p = poly(A,'x')   // constroi o polinomio caracteristico de A
 p  =
                  2   3
  - 79 + 78x - 17x + x
-->raizes = roots(p)   // zeros do polinomio caracteristico
 raizes  =
    1.410402
    5.6161546
    9.9734434
```

A função `spec` permite o cálculo do autosistema (autovalores e autovetores) de uma matriz de ordem n por um método mais eficiente do que construindo o polinômio característico. O uso de `[<AutoVetor>,<AutoValor>]=spec(<matriz>)` produz uma matriz diagonal `<AutoValor>` contendo os autovalores e uma matriz `<AutoVetor>` com os correspondentes autovetores. Se a matriz for simétrica então os autovetores são mutuamente ortogonais (`<AutoVetor>' * <AutoVetor> = eye(`n`,`n`)`). Para a matriz `A` acima,

```
-->[Vetor,Valor] = spec(A)   // calcula os autovalores e autovetores
 Valor  =
    1.410402    0.          0.
    0.          5.6161546   0.
    0.          0.          9.9734434
 Vetor  =
    0.7387367    0.5462817    0.3947713
    0.4732062  - 0.8374647    0.2733656
  - 0.4799416    0.0151370    0.8771698
-->round(Vetor'*Vetor-eye(3,3)) // autovetores ortogonais de uma matriz simetrica
```

```
 ans  =
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
```

Para obter apenas os autovalores em `<vetor>` basta `<vetor> = spec(<matriz>)`. Para a matriz simétrica `A` definida acima,

```
-->lambda = spec(A)   // calcula apenas os autovalores de A
 lambda  =
    1.410402
    5.6161546
    9.9734434
```

A função `spec` possui diversas formas de ser utilizada; para maiores informações use `help spec`.

8.3 Interpolação

O SCILAB possui funções para cálculo das diferenças finitas ascendentes e interpolação unidimensional.

8.3.1 Cálculo das diferenças finitas ascendentes

A função `diff(<variável>,<ordem>,<dimensão>)` calcula a diferença finita ascendente de ordem dada por `<ordem>` entre dois elementos de `<variável>` ao longo de `<dimensão>`, que pode assumir os valores `'r'` para linha, `'c'` para coluna ou `'*'` para as colunas da matriz `<variável>` colocadas em um vetor. Os argumentos `<ordem>` e `<dimensão>` são opcionais e seus valores pré-definidos são `1` e `'*'`, respectivamente. Para um vetor,

```
-->x = [1 2 4 4 7]   // define o vetor x
 x  =
    1.    2.    4.    4.    7.
-->d1 = diff(x)   // diferenca de ordem 1 de x
 d1  =
    1.    2.    0.    3.
-->d2 = diff(d1)   // diferenca de ordem 1 de d1
 d2  =
    1.  - 2.    3.
-->xd2 = diff(x,2)   // diferenca de ordem 2 de x
 xd2  =
    1.  - 2.    3.
```

No caso de matriz,

```
-->A = [1:5;(1:5)^2;(1:5)^3]   // define a matriz A
 A  =
    1.    2.    3.     4.     5.
    1.    4.    9.     16.    25.
    1.    8.    27.    64.    125.
-->L1 = diff(A,1,'r')   // diferenca de ordem 1 ao longo das linhas
 L1  =
    0.    2.    6.     12.    20.
    0.    4.    18.    48.    100.
-->C1 = diff(A,1,'c')   // diferenca de ordem 1 ao longo das colunas
 C1  =
    1.    1.     1.     1.
    3.    5.     7.     9.
    7.    19.    37.    61.
-->A(:)'   // colunas da matriz A em um vetor (transposto)
 ans  =
         column  1 to 12
    1.    1.    1.    2.    4.    8.    3.    9.    27.    4.    16.    64.
         column 13 to 15
    5.    25.    125.
-->V1 = diff(A,1,'*')'//diferenca de ordem 1 assumindo A em um vetor (transposto)
 V1  =
         column  1 to 12
    0.    0.    1.    2.    4.  - 5.    6.    18.  - 23.    12.    48.  - 59.
         column 13 to 14
    20.    100.
```

8.3.2 Interpolação unidimensional

Dada uma tabela com pares (x_i, y_i) especificados nos vetores `<abscissas>` e `<ordenadas>`, respectivamente, então os valores de z contidos no vetor `<interpolar>` podem ser interpolados usando a função `interp1`, cuja sintaxe é

```
<resultado> = interp1(<abscissas>,<ordenadas>,<interpolar>,<método>)
```

onde a cadeia de caracteres `<método>` especifica o método a ser utilizado na interpolação, sendo

parâmetro	método
`'linear'`	interpolação linear.
`'spline'`	interpolação por *splines* cúbicos.
`'nearest'`	interpolação usando o vizinho mais próximo.

Quando o método não for especificado será assumida uma interpolação linear. Por exemplo, sejam cinco pontos da função $y = x^4$, para $-1 \leq x \leq 1$ definidos por,

```
-->x = linspace(-1,1,5);   // abscissas dos pontos
-->y = x.^4;   // ordenadas
```

Interpolando os valores $z = -0{,}6$ e $z = 0{,}7$, usando os três métodos,

```
-->interp1(x,y,[-0.6 0.7],'linear')   // interpolacao linear
 ans  =
    0.25    0.4375
-->interp1(x,y,[-0.6 0.7],'spline')   // interpolacao com splines cubicos
 ans  =
    0.144    0.2695
-->interp1(x,y,[-0.6 0.7],'nearest')   // interpolacao com vizinho mais proximo
 ans  =
    0.0625    0.0625
```

Considerando que os valores exatos são $(-0{,}6)^4 \approx 0{,}1296$ e $0{,}7^4 \approx 0{,}2401$, o método de interpolação com *splines* cúbicos produziu os melhores resultados para este caso.

As aproximações da função $y = x^4$ por três funções interpoladoras de `interp1` podem ser visualizadas pelos comandos,

```
-->x1 = linspace(-1,1,50);   // abscissas para a interpolacao
-->y1 = x1.^4;   // ordenadas da curva real
-->nea = interp1(x,y,x1,'nearest');   // interpolacao vizinho mais proximo
-->spl = interp1(x,y,x1,'spline');   // interpolacao por splines
-->plot(x,y,'o',x,y,'-.',x1,y1,'-',x1,nea,'--',x1,spl,':'); xgrid(1)
-->xtitle('M'+ascii(233)+'todos de interp1','x','y');   // gera graficos e grade
-->legend(['pontos','linear','y=x^4','nearest','splines'],3)   // coloca legenda
```

A partir dos 50 pontos em `x1` foram criados os vetores `nea` e `spl` que contêm as ordenadas obtidas por interpolação usando os argumentos `'nearest'` e `'splines'`, respectivamente. A Figura 8.1 mostra graficamente os resultados da função `interp1`. Os cinco pontos iniciais estão representados por `o`. Uma interpolação linear é feita implicitamente desde que o comando `plot` interliga os pontos por retas. O gráfico da função $y = x^4$ está representado por uma linha sólida, a aproximação por polinômios com `'nearest'` por uma linha tracejada e os *splines* por uma linha pontilhada. Usualmente os *splines* produzem uma aproximação mais suave da função como pode ser observado neste caso.

8.4 Integração numérica

A função `integrate` calcula numericamente uma integral e sua sintaxe é

```
integrate(<função>,<variável>,<lim_inf>,<lim_sup>,<erro_abs>,<erro_rel>)
```

onde `<função>` é uma cadeia de caracteres que define a função a ser integrada, `<variável>` é uma cadeia de caracteres que especifica a variável de integração, `<lim_inf>` e `<lim_sup>`

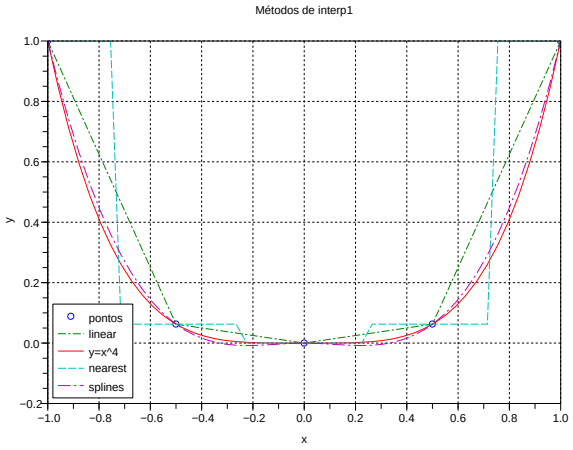


Figura 8.1: Aproximação de $y = x^4$ pela função `interp1`.

são os limites inferior e superior de integração. Os argumentos `<erro_abs>` e `<erro_rel>` são os erros absoluto e relativo e são opcionais. No caso de não serem fornecidos, são assumidos os valores 0 e 10^{-8}, respectivamente.

Por exemplo, seja a função $f(x) = \cos(3x+1)x^2+x^{1,5}$, cujo esboço é mostrado na Figura 8.2.

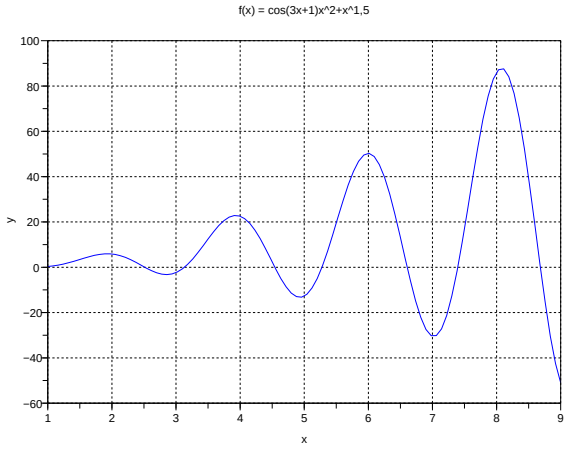


Figura 8.2: Integração de $f(x) = \cos(3x + 1)x^2 + x^{1,5}$.

Para calcular

$$\int_1^9 \cos(3x + 1)x^2 + x^{1,5}\, dx$$

faz-se

```
-->Integral = integrate('cos(3*x+1)*x^2+x^1.5','x',1,9)   // calcula a integral
 Integral  =
    102.51064
```

A função a ser integrada pode também estar definida em uma `function`. Seja o arquivo `func.sci` com o conteúdo,

```
function y = g(x)
y = cos(3*x+1).*x.^2+x.^1.5;
endfunction
```

Assim, para calcular a integral acima usando a `function g` definida no arquivo `func.sci`,

```
-->exec('func.sci',-1)   // carrega a function g do arquivo func.sci
-->Integral = integrate('g','x',1,9)   // calcula a integral
 Integral  =
    102.51064
```

8.5 Exercícios

Seção 8.1 Medidas de tempo

8.1

8.2

8.3

8.4

8.5

Seção 8.2 Álgebra linear

8.6

8.7

8.8

8.9

8.10

Seção 8.3 Interpolação

8.11

8.12

8.13

8.14

8.15

Seção 8.4 Integração numérica

8.16

8.17

8.18

8.19

8.20

Referências Bibliográficas

[1] M. Abramowitz e I. A. Stegun. *Handbook of Mathematical Functions*. Dover, Nova Iorque, 1972.

[2] F. F. Campos, filho. *Algoritmos Numéricos*. LTC Editora, Rio de Janeiro, 2ª edição, 2007.

[3] H. Farrer, C. G. Becker, E. C. Faria, F. F. Campos, filho, H. F. de Matos, M. A. dos Santos, e M. L. Maia. *Pascal Estruturado*. LTC Editora, Rio de Janeiro, 3ª edição, 1999.

[4] H. Farrer, C. G. Becker, E. C. Faria, H. F. de Matos, M. A. dos Santos, e M. L. Maia. *Algoritmos Estruturados*. LTC Editora, Rio de Janeiro, 3ª edição, 1999.

[5] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.