Introdução ao
MATLAB

R. Sampaio, E. Cataldo, R. Riquelme

Rio de Janeiro, 7 de agosto de 1997

Introdução

O MATLAB é uma linguagem de programação e uma ferramenta de cálculo.

Todos os programas são abertos; isto é, o algoritmo usado na solução de um problema é conhecido e pode ser modificado. Este fato é, talvez, a maior virtude do MATLAB pois possibilitou que fossem preparados vários aplicativos, os *toolboxes*, usando a plataforma MATLAB como base. Temos aplicativos em áreas como controle, análise de sinais, equações diferenciais parciais, robótica, análise modal, finanças, estatística e várias outras.

Enquanto que o programa principal do MATLAB depende da plataforma (Unix, Windows, Macintosh) os arquivos .m, que constituem os aplicativos, são independentes da plataforma e podem ser facilmente convertidos de uma plataforma para a outra.

O MATLAB é muito fácil de aprender, fácil de utilizar e, quando bem utilizado, aumenta, em muito, a produtividade. É também uma ferramenta profissional usada mundialmente, principalmente por engenheiros, sobretudo pelos que têm contato com trabalhos experimentais.

A melhor forma de aprender o MATLAB é usando-o. A experiência mostra que uma vez vencido o acanhamento inicial, o medo de enfrentar algo novo, o trabalho se torna agrádavel e, na verdade, vicia.

Manuais como este que agora voce lê são úteis no início da aprendizagem para que se tome consciência dos problemas e se adquira uma bagagem de conhecimentos suficiente para que se prossiga sozinho, aprendendo por tentativas, errando até acertar. O estágio que gostaríamos que todos alcançassem é aquele em que se é capaz de construir aplicativos, fazendo os programas na plataforma MATLAB. O que procuramos fazer neste manual foi mostrar como este processo de tentativas funciona. Tentamos descrever como os comandos podem ser aprendidos. Não apenas a sintaxe, que pode sempre ser buscada com os comandos *help*, *lookfor*, e outros, mas também informação de como as varíaveis são armazenadas e de como fazer para usar o MATLAB acoplado a um outro programa.

Rubens Sampaio
PUC-Rio
rsampaio@mec.puc-rio.br

Edson Cataldo
PUC-Rio, UFF
ecataldo@mec.puc-rio.br

Roberto Riquelme
PUC-Rio, U de Concepción
rriquelm@mec.puc-rio.br

Índice

Capítulo 1

Comandos Básicos do MATLAB

1.1 Entrando no MATLAB

Quando você entra pela primeira vez no MATLAB,o símbolo >> aparece na tela (chamado de *prompt* do MATLAB). Este símbolo indica que você pode escrever um comando. Os comandos em MATLAB podem terminar com ';' ou não . Quando um comando termina com ';' ele é executado mas os conteúdos das variáveis envolvidas não são mostrados na tela. Se você deseja olhar os conteúdos das variáveis à medida que o programa está sendo executado então deve omitir o ';'.

Neste capítulo mostramos os comandos básicos do MATLAB. Discutimos a forma com que o MATLAB armazena os dados e como eles podem ser trabalhados.

1.2 Primeiros passos

- Com o comando

```
>> t=1
```

obtemos o número

```
t =   1
```

- Com o comando

```
>> T=[0 1 2 3 4 5 6 7 8 9 10];
```

obtemos

```
T =    0     1     2     3     4     5     6     7     8     9    10
```

Outra maneira de formar este vetor é utilizando as sentenças

```
>> T=0:1:10;
```

ou

```
>> for n=1:1:11
  T(n)=n-1;
   end
```

Devemos observar que o MATLAB é otimizado para trabalhar com matrizes, por isso a segunda forma não é a ideal.É mais vagarosa que a primeira.

- Com o comando

```
>>whos
```

temos informação sobre o que está armazenado na memória, relativo à seção que estamos trabalhando.

```
   Name        Size        Elements     Bytes    Density   Complex
      T       1 by 11            11        88       Full      No
      t       1 by 1              1         8       Full      No

      Grand total is 12 elements using 96 bytes
```

Cabe ressaltar que o MATLAB faz diferença entre as maiúsculas e minúsculas. Neste caso 't' e 'T' são variáveis distintas para o MATLAB.

- Com o comando

```
>>T(5)
```

obtemos o elemento que está na posição 5 no vetor T.

```
ans = 4
```

É bom notar que neste caso, como não demos um nome para a variável que armazena a resposta, o MATLAB atribui o nome 'ans' a esta variável.

- Com o comando

```
>>x= sin(t);
```

associamos à x o valor de sin(t), com t em radianos.

```
x = 0.8415
```

- Com o comando

```
>>y= sin(T);
```

associamos à y o vetor de comprimento 11 que corresponde ao valor do seno de cada elemento do vetor T.

```
y =  Columns 1 through 7
        0    0.8415    0.9093    0.1411   -0.7568   -0.9589   -0.2794
     Columns 8 through 11
   0.6570    0.9894    0.4121   -0.5440
```

- Com o comando

```
>>a=2*T
```

construimos o vetor 'a' formado por todos os elemento do vetor T multiplicado por 2

```
a =    0     2     4     6     8    10    12    14    16    18    20
```

- Com o comando

```
>> q=T.^2
```

temos

```
q =    0     1     4     9    16    25    36    49    64    81   100
```

Notamos que, como T é um vetor, o ponto (.) antes da operação ($\wedge$) é muito importante. Este tipo de operação (.$\wedge$) chama-se pontual. Como veremos mais adiante, essas operações também existem para matrizes.

- O MATLAB opera com números complexos da mesma forma que com números reais, os sinais de operação são os mesmos. Chamaremos atenção apenas de alguns comandos.
Consideremos os complexos

```
>> a=1+i;
>> b=-5+2i;
```

- Com o comando

  ```
  >> e=a+b
  ```

  obtemos a soma de a com b

  ```
  e = -4.0000 + 3.0000i
  ```

- Com o comando

  ```
  >> g=a*b
  ```

  obtemos o produto de a por b

  ```
  g = -7.0000 - 3.0000i
  ```

- Com o comando

  ```
  >> f=a/b
  ```

  obtemos a divisão de a por b

  ```
  f = -0.1034 - 0.2414i
  ```

- Com o comando

  ```
  >> h=abs(b)
  ```

  obtemos o módulo de b

  ```
  h = 5.3852
  ```

- Com o comando

  ```
  >>  d=angle(a)
  ```

  obtemos o argumento do número complexo a normalizado; isto é, no intervalo $[-\pi, \pi]$

  ```
  d = 0.7854
  ```

- Com o comando

  ```
  >> c=conj(a)
  ```

  obtemos o conjugado de a

  ```
  c = 1.0000 - 1.0000i
  ```

- Mostramos a seguir uma tabela com as funções mais comuns que operam essencialmente sobre escalares

sin	asin	exp	abs
cos	acos	log(natural)	sqrt
tan	atan	rem(resto)	sign

- Mostramos a seguir uma tabela com as funções mais comuns que operam essencialmente sobre vetores (linha ou coluna). Podem, ainda, operar sobre matrizes $m \times n (m \geq 2)$,operando coluna a coluna.

length	número de componentes de um vetor
abs	módulo das componentes de um vetor
max	maior componente de um vetor
min	menor componente de um vetor
sum	a soma das componentes de um vetor
prod	produto das componentes de um vetor
mean	média das componentes de um vetor
std	desvio padrão das componentes de um vetor com respeito à média
norm	norma euclidiana do vetor
sort	ordena o vetor em ordem crescente

  Cabe observar que se o vetor 'a' é complexo, a função *sort* opera sobre *abs(a)*.

- Com o comando

```
>>clear
```

  limpamos a memória do MATLAB. Este comando é tratado com mais detalhe na seção 1.7.

- Com o comando

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

  obtemos

```
A =  1     2     3
     4     5     6
```

- Dadas as matrizes

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

- Com o comando

```
>> A(:,1)
```

listamos a primeira coluna da matriz A

```
ans = 1
      4
```

- Com o comando

```
>> A(2,:)
```

listamos a segunda linha da matriz A

```
ans = 4   5   6
```

- Com o comando

```
>> A(1,2)
```

obtemos o valor na posição (1,2) da matriz A

```
ans = 2
```

- Com o comando

```
>> C=A*B
```

obtemos o produto de A por B

```
C = 18    27
    42    66
```

- Para fazer operações pontuais com matrizes elas devem ter a mesma dimensão. Por exemplo, dadas as matrizes

```
>> A=[1 2 3; 9 8 2; 1 -2 -4];
>> B=[0 1 -2; -1 -2 -5; 1 1 0];
```

- Com o comando

```
>> C=A.^2
```

obtemos a matriz formada pelos quadrados dos elementos.

```
C =  1     4     9
    81    64     4
     1     4    16
```

- Com o comando

```
>> D=A.*B
```

obtemos a matriz formada pelos produtos dos elementos que se correspondem.

```
D =  0     2    -6
    -9   -16   -10
     1    -2     0
```

- Dada a matriz

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

  - Com o comando

```
>>  f=det(A)
```

    obtemos o determinante de A

```
f =  6
```

  - Com o comando

```
>>  r=rank(A)
```

    obtemos o posto de A

```
r =  3
```

  - Com o comando

```
>>   T=A'
```

    obtemos a transposta de A

```
T =  1     4    -1
     2     5     2
     3     6     3
```

  - O comando de transposição ' quando aplicado a uma matriz complexa C fornece a adjunta de C, isto é: a conjugada da transposta, $\bar{C}^t$. Para trocar linhas por colunas de uma matriz complexa C temos que dar o comando

```
>> conj(C')
```

    Por exemplo, com os comandos

```
>> C=[1+i 2; 3-i 4+3i; -1+2i -9];
>> B=conj(C')
```

    obtemos

```
B  = 1.0000 + 1.0000i   3.0000 - 1.0000i  -1.0000 + 2.0000i
     2.0000             4.0000 + 3.0000i  -9.0000
```

Notar que com o comando

```
>> D=C'
```

obtemos

```
D  = 1.0000 - 1.0000i   3.0000 + 1.0000i  -1.0000 - 2.0000i
     2.0000             4.0000 - 3.0000i  -9.0000
```

que não é a matriz transposta de C.

- Com o comando

```
>>    F=inv(A)
```

obtemos a inversa de A

```
F = 0.5000    0.0000   -0.5000
   -3.0000    1.0000    1.0000
    2.1667   -0.6667   -0.5000
```

- O comando

```
>>     [V,D]=eig(A)
```

produz uma matriz diagonal D de autovalores e uma matriz V cujas colunas são os correspondentes autovetores, tal que A*V=V*D

```
V =0.2550 - 0.0640i   0.2550 + 0.0640i   0.3526
  -0.3183 - 0.7047i  -0.3183 + 0.7047i   0.8965
   0.1708 + 0.5512i   0.1708 - 0.5512i   0.2683

D = 0.3156 + 0.7857i        0                  0
         0             0.3156 - 0.7857i        0
         0                  0             8.3687
```

- O comando usado para produzir uma decomposição em valores singulares é

```
>>   [U,S,V] = svd(A)
```

S é uma matriz diagonal, com a mesma dimensão de A, com elementos não negativos na diagonal em ordem decrescente,e U e V são matrizes unitárias, tal que A=U*S*V'.

```
U = 0.3735    0.1631    0.9132
    0.8775   -0.3815   -0.2907
    0.3010    0.9099   -0.2856

S = 9.9425         0         0
         0    2.4672         0
```

```
          0         0    0.2446

V = 0.3603   -0.9212    0.1466
    0.5769    0.0966   -0.8111
    0.7330    0.3768    0.5663
```

– O comando usado para produzir a decomposição LU é

```
>> [L,U]=lu(A)
```

U é uma matriz triangular superior e L é uma matriz "quase triangular inferior"isto é, seria uma matriz triangular inferior se A fosse substituida por PA onde P é uma matriz de permutação . A razão disto tem a ver com a escolha do pivô, e não será discutido aqui. Outra forma deste comando é

```
>> [L,U,P]=lu(A)
```

U é uma matriz triangular superior, L, uma matriz triangular inferior e P uma matriz de permutação , tal que $PA = LU$. Consideremos os seguintes exemplos

* Com os comandos

```
>> A=[1 2; 3 4];
>> [L,U]=lu(A)
```

Obtemos

```
L = 0.3333    1.0000
    1.0000         0

U = 3.0000    4.0000
         0    0.6667
```

Com o comando

```
>> [L,U,P]=lu(A)
```

obtemos

```
L = 1.0000         0
    0.3333    1.0000

U = 3.0000    4.0000
         0    0.6667

P =  0     1
     1     0
```

Notar a diferença entre o primeiro e o segundo comando.

* Com os comandos

```
>>   A=[1 3 4 5; -1 2 0 2; 1 2 5 7; 5 2 1 6];
>> [L,U]=lu(A)
```

Obtemos

```
L = 0.2000    1.0000         0         0
   -0.2000    0.9231    1.0000         0
    0.2000    0.6154   -0.7442    1.0000
    1.0000         0         0         0

U = 5.0000    2.0000    1.0000    6.0000
         0    2.6000    3.8000    3.8000
         0         0   -3.3077   -0.3077
         0         0         0    3.2326
```

com o comando

```
>> [L,U,P]=lu(A)
```

obtemos

```
L = 1.0000         0         0         0
    0.2000    1.0000         0         0
   -0.2000    0.9231    1.0000         0
    0.2000    0.6154   -0.7442    1.0000

U = 5.0000    2.0000    1.0000    6.0000
         0    2.6000    3.8000    3.8000
         0         0   -3.3077   -0.3077
         0         0         0    3.2326

P =  0     0     0     1
     1     0     0     0
     0     1     0     0
     0     0     1     0
```

– O comando usado para obter a matriz inversa generalizada é

```
>> X=pinv(A)
```

X é uma matriz com a mesma dimensão de A^t e tal que:
(i) $A * X * A = A$
(ii)$X * A * X = X$ e
(iii) $A * X$ e $X * A$ são Hermitianas. Com os comandos

```
>> A=[1 2 3; 4 5 6; -1 2 3];
>> X=pinv(A)
```

obtemos

```
X =
    0.5000    0.0000   -0.5000
   -3.0000    1.0000    1.0000
    2.1667   -0.6667   -0.5000
```

Quando a matriz A é inversível, a inversa generalizada coincide com a inversa de A. É bom lembrar que matrizes não -inversíveis ou não -quadradas têm inversa generalizada, por exemplo

```
>> B=[1 2 3; 2 1 0; 3 3 3]
B =
     1     2     3
     2     1     0
     3     3     3
>> XB=pinv(B)
XB =
   -0.1852    0.3148    0.1296
    0.0370    0.0370    0.0741
    0.2593   -0.2407    0.0185
>> C=[1 2; 3 4; 5 6]
C =
     1     2
     3     4
     5     6
>> XC=pinv(C)
XC =
   -1.3333   -0.3333    0.6667
    1.0833    0.3333   -0.4167
```

- Em muitas aplicações é preciso trabalhar com matrizes de matrizes e também obter uma submatriz de uma matriz. Por exemplo, dadas as matrizes $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ -1 & -9 \end{pmatrix}$ e $B = \begin{pmatrix} 1 & 7 & 3 & 6 \\ 2 & 5 & 3 & -1 \\ 0 & 2 & 4 & 2 \end{pmatrix}$ queremos construir as matrizes $C = \left(\begin{array}{c|c} A & \Theta_1 \\ \hline \Theta_2 & B \end{array}\right)$ e $D = \begin{pmatrix} B_{12} & B_{13} \\ B_{22} & B_{23} \end{pmatrix}$ onde Θ_1 é a matriz nula 3×4 e Θ_2 é a matriz nula 3×2. Para a primeira matriz podemos fazer

```
>> A=[1 2; 3 4; -1 -9];
>> B=[1 7 3 6; 2 5 3 -1; 0 2 4 2];
>> C=[A zeros(3,4); zeros(3,2) B]
```

obtemos

```
C =
     1     2     0     0     0     0
     3     4     0     0     0     0
    -1    -9     0     0     0     0
     0     0     1     7     3     6
     0     0     2     5     3    -1
     0     0     0     2     4     2
```

Para a segunda podemos fazer

```
>> D=B(1:2,2:3)
```

e obtemos

```
D =
     7     3
     5     3
```

- As seguintes funções para a construção de matrizes estão disponíveis em MATLAB
  - Matriz identidade
    Com o comando

    ```
    >>eye(3)
    ```

    obtemos

    ```
    ans = 1     0     0
          0     1     0
          0     0     1
    ```

  - Matriz de zeros
    Com o comando

    ```
    >> zeros(3,2)
    ```

    obtemos

    ```
    ans = 0     0
          0     0
          0     0
    ```

  - Matriz em que todos os elementos valem 1.
    Com o comando

    ```
    >> ones(3,5)
    ```

obtemos

```
ans = 1     1     1     1     1
      1     1     1     1     1
      1     1     1     1     1
```

– Matriz gerada aleatoriamente
Com o comando

```
>> rand(2,3)
```

obtemos

```
ans = 0.2190    0.6789    0.9347
      0.0470    0.6793    0.3835
```

- Na operação com matrizes, temos divisão à esquerda e à direita.
Se A é uma matriz inversível e b é um vetor coluna, resp. linha, compatíveis, então

$x = A\backslash b$ é a solução de $A * x = b$ e, respectivamente,
$x = b/A$ é a solução de $x * A = b$.

Na divisão à esquerda, se A for quadrada, x é calculado fazendo eliminação gaussiana de A e retrosubstituição; ou seja, A é fatorada na forma LU. O sistema fica $LUx = b$ e para resolver este sistema calculamos y do sistema $Ly = Pb$ e então x do sistema $Ux = y$. Se a matriz A não for quadrada,o MATLAB utiliza a ortogonalização de Householder com pivotamento de colunas. O valor de x é calculado pelo método dos mínimos quadrados. A divisão à direita é realizada a partir da esquerda; isto é, $b/A = (A'\backslash b')'$. Consideremos os exemplos a seguir.

– Consideremos o sistema

$$\begin{aligned} 2x + 3y + z &= 2 \\ x - y + z &= 3 \\ x - y + 2z &= 0 \end{aligned}$$

Este sistema pode ser colocado na forma $AX = b$. Para resolvê-lo fazemos, no MATLAB,

```
>> A=[2 3 1; 1 -1 1; 1 -1 2];
>> b=[2;3;0];
>> X=A\b
```

A resposta obtida é

```
X = 4.6000
   -1.4000
   -3.0000
```

Neste caso a matriz A é inversível e a solução é única. Para achar a solução podemos também fazer

```
>> X=inv(A)*b
```

ou

```
>> X=pinv(A)*b
```

Cabe ressaltar que se a matriz A não for singular então as operações X=A\b; X=inv(A)*b e X=pinv(A)*b se equivalem.

– Consideremos o sistema

$$\begin{array}{rl} 2x+3y+2z= & 1 \\ x-y+z= & 2 \\ x+y+z= & 0 \end{array}$$

Podemos escrever este sistema na forma $AX = b$. Para resolver este sistema fazemos no MATLAB

```
>> A=[2 3 2; 1 -1 1; 1 1 1];
>> b=[1;2;0];
>> X=A\b
```

A resposta obtida é

```
X = Inf
    Inf
    Inf
```

Neste caso a matriz A é singular. Até agora não sabemos se o sistema tem ou não solução . Para saber se o sistema tem ou não solução usaremos a inversa generalizada e o conceito de erro, i. e., definimos

$$\begin{aligned} e &= b - AX \\ J &= \frac{1}{2}e^t e \end{aligned}$$

Se $J = 0$, para algum X, o sistema tem solução . Para decidir sobre a unicidade da solução é preciso que façamos uma análise, mais detalhada. Um programa geral que faz essa análise é dado no apêndice A. No caso em que $J \neq 0$ o sistema não tem solução . Para achar uma resposta o MATLAB minimiza a forma quadrática J usando mínimos quadrados. O vetor solução deste problema não linear de minimização é tomado como 'solução ' do problema linear $AX = b$. Para que possamos entender esses problemas, façamos :

```
>> X=pinv(A)*b
```

obtemos

```
X = 0.6333
   -0.6000
    0.6333
```

```
>> e=b-A*X;
>> J=e'*e/2
```

obtemos

```
J = 0.2667
```

Daí, observamos que o sistema não tem solução .

– Consideremos o sistema

$$\begin{array}{rl} x-2y+2z= & 2 \\ x+y+3z= & 1 \\ 2x-y+5z= & 3 \end{array}$$

Neste caso

```
>> A=[1 -2 2; 1 1 3;2 -1 5];
>> b=[2;1;3];
>> X=A\b
```

A resposta obtida é

```
X = Inf
    Inf
    Inf
```

Aplicando inversa generalizada

```
>> X=pinv(A)*b
```

obtemos

```
X = 0.2162
   -0.4730
    0.4189
```

```
>> e=b-A*X;
>> J=e'*e/2
```

obtemos

```
J = 4.9304e-031
```

A resposta é practicamente zero. Para termos certeza absoluta de que o sistema tem infinitas soluções é preciso estudar o posto. No exemplo acima o sistema tem infinitas soluções .

– Consideremos o sistema

$$\begin{array}{rl} x+y= & 1 \\ x-2y= & 2 \\ 2x-y= & 3 \end{array}$$

Neste caso

```
>> A=[1 1; 1 -2;2 -1];
>> b=[1;2;3];
>> X=A\b
```

A resposta fica

```
X = 1.3333
   -0.3333
```

Aplicando inversa generalizada

```
>> X=pinv(A)*b
```

obtemos

```
X = 1.3333
   -0.3333

>> e=b-A*X;
>> J=e'*e/2
```

obtemos

```
J = 5.1769e-031
```

A resposta é practicamente zero. Para termos certeza absoluta de que o sistema tem uma única solução é preciso analisar o posto. No exemplo o sistema tem solução única.

– Consideremos o sistema

$$\begin{array}{rl} x+y= & 2 \\ 4x+4y= & 8 \\ 5x+5y= & 10 \end{array}$$

Neste caso

```
>> A=[1 1; 4 4;5 5];
>> b=[2;8;10];
>> X1=A\b
```

A resposta fica

```
X1 = 2.0000
          0
```

Aplicando inversa generalizada

```
>> X=pinv(A)*b
```

obtemos

```
X = 1.0000
    1.0000
```

e

```
>> e=b-A*X;
>> J=e'*e/2
```

obtemos

```
J = 1.5777e-030
```

Este exemplo mostra que $J = 0$ e o sistema tem infinitas soluções . Devemos notar que, fazendo $e_1 = b - AX1$ e $J_1 = \frac{1}{2}e_1^t e_1$, temos $J_1 = 1.9968e - 030$.

– Consideremos o sistema

$$\begin{array}{|rl} x + y = & 2 \\ 4x + 4y = & 8 \\ 5x + 5y = & 5 \end{array}$$

Neste caso

```
>> A=[1 1; 4 4;5 5];
>> b=[2;8;5];
>> X=A\b
```

A resposta fica

```
Warning: Rank deficient, rank = 1  tol =    4.3170e-15
X = 1.4048
          0
```

Aplicando inversa generalizada

```
>> X=pinv(A)*b
```

obtemos

```
X = 0.7024
    0.7024
```

```
>> e=b-A*X;
>> J=e'*e/2
```

e obtemos

```
J = 5.0595
```

Este resultado mostra claramente que o sistema não tem solução .

Um bom exercício é repetir os cálculos para a divisão à direita.

1.3 O Comando for

A sintaxe do comando **for** é dada por:

for i=vi:in:vf
instruções
end

onde vi, in, vf são valor inicial, incremento e valor final da variável escalar i.

1.3.1 Exemplos

- com os comandos

```
>>x=0;
>> for i=.1:.1:1
   x=[x i^3];
   end
>> x
```

Obtemos

```
x = Columns 1 through 7
         0    0.0010    0.0080    0.0270    0.0640    0.1250    0.2160
    Columns 8 through 11
    0.3430    0.5120    0.7290    1.0000
```

Neste exemplo o valor de i começa em .1, é incrementado de .1 até chegar a 1 .

- Podemos fazer

```
>> for n=10:-.3:-51
  []
   end
```

Neste caso n começa em 10, é decrementado de 0.3 até chegar a -51. Devemos notar que o incremento pode ser negativo (neste caso, decremento) e que o último valor que n assume, neste exemplo, é -50.9.

- Podemos, ainda, fazer

```
>> for w=-34:12
   []
   end
```

Neste caso w começa em -34 é incrementado de 1,até chegar a 12. Quando o incremento é omitido o valor assumido é 1.

1.4 Operadores relacionais e conectivos lógicos

Os operadores relacionais são dados por

$<$	menor que
$>$	maior que
$<=$	menor ou igual a
$>=$	maior ou igual a
$==$	igual a
$\sim=$	diferente de

Os conectivos lógicos são dados por

&	e
|	ou
$\sim$	não
xor	ou excludente

Os operadores relacionais e os conectivos lógicos são usados com os comandos *while* e *if*, que veremos a seguir.

1.5 O comando while

A sintaxe deste comando é

while *relação*
instruções
end

As intruções são executadas enquanto a relação for verdadeira

1.5.1 Exemplo

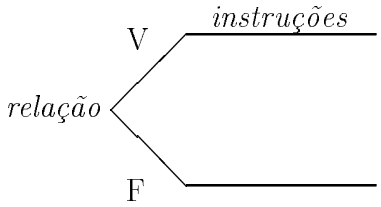
- ```
  >>n=0;
  >>while n<=10 n=n+2;
    end
  ```

  Neste caso o valor final de n é 12.

1.6 O comando if

A sintaxe básica deste comando é

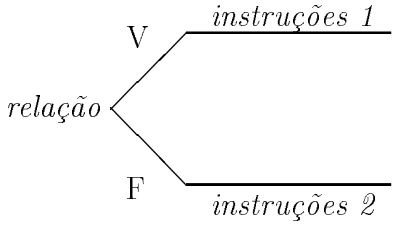
if *relação*
  instruções
end



As instruções são executadas se a relação for verdadeira.
Outras formas deste comando são possíveis. Por exemplo,

if *relação*
  instruções 1
else *instruções 2*
end

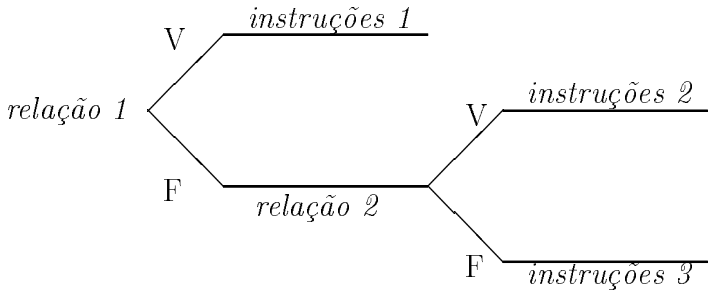
Neste caso o esquema é



Se a *relação* for verdadeira as *instruções 1* são executadas, caso contrário são executadas as *instruções 2*.
Outra forma deste comando é
  ```

if *relação 1*
  instruções 1
elseif *relação 2*
  instruções 2
else *instruções 3*
end

Neste caso o esquema é



Neste caso, se a *relação 1* for falsa, uma nova condição , *relação 2*, é verificada, se for verdadeira as *instruções 2* são executadas e se for falsa as *instruções 3* são executadas.

Notemos que este comando aceita mais de um *elseif*.

1.7 O comando clear

Este comando é usado para apagar variáveis no espaço de trabalho. Isto é feito quando desejamos liberar memória no MATLAB e para poder manter um controle mais efetivo do trabalho que estiver sendo feito. Nos exemplos a seguir mostramos algumas das formas como este comando é usado.

1.7.1 Exemplo 1

Dadas as seguintes variáveis no MATLAB

```
>> a=1;
>> b=3;
>> c=[1 2 3];
>> A=[1 2 3; 4 5 6];
```

Fazendo

```
>> whos
```

obtemos

```
  Name         Size        Elements     Bytes    Density   Complex
    A        2 by 3              6        48       Full      No
    a        1 by 1              1         8       Full      No
    b        1 by 1              1         8       Full      No
    c        1 by 3              3        24       Full      No
 Grand total is 11 elements using 88 bytes
```

Todas estas variáveis ficam na memória do MATLAB. Fazendo

```
>> clear
```

todas são apagadas. Para termos certeza desse fato fazemos

```
>> whos
```

e verificamos que não há nada na memória do MATLAB.

1.7.2 Exemplo 2

Consideremos as mesmas variáveis anteriores, i.e.,

```
>> a=1;
>> b=3
>> c=[1 2 3];
>> A=[1 2 3; 4 5 6];
```

Fazendo

```
>> whos
```

obtemos

```
  Name         Size        Elements     Bytes    Density   Complex
    A        2 by 3              6        48       Full      No
    a        1 by 1              1         8       Full      No
    b        1 by 1              1         8       Full      No
    c        1 by 3              3        24       Full      No
Grand total is 11 elements using 88 bytes
```

Todas estas variáveis ficam na memória do MATLAB. Fazendo

```
>> clear b
```

só apagamos a variável b. Fazendo

```
>> whos
```

obtemos

```
  Name         Size         Elements      Bytes    Density   Complex
    A        2 by 3                6         48       Full        No
    a        1 by 1                1          8       Full        No
    c        1 by 3                3         24       Full        No
Grand total is 10 elements using 80 bytes
```

Para apagar, por exemplo, as variáveis a, b e c é só fazer *clear a b c*.

1.8 O comando plot

Este comando é usado para a construção de gráficos. Para isso devemos utilizar dois vetores do mesmo comprimento. No exemplo a seguir mostramos como usamos este comando.

1.8.1 Exemplo

- Dados os comandos

```
>> t=0:.01:10;
>> x=2*sin(t);
>> plot(t,x)
```

obtemos o gráfico da figura 1.1

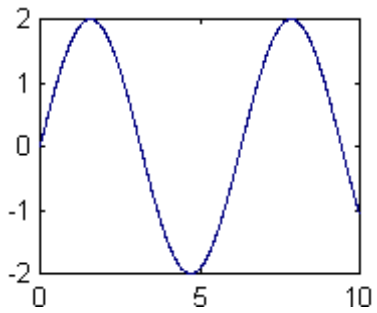


Figura 1.1: Exemplo de um gráfico.

Mostramos a seguir uma tabela contendo alguns comandos relacionados a gráficos. No capítulo seguinte exploraremos mais esses comandos.

Comandos relacionados a gráficos

plot	Gráfico Linear
loglog	Gráfico em escala log-log
semilogx	Gráfico em escala semi-log
semilogy	Gráfico em escala semi-log
polar	Gráfico em coordenadas polares
bar	Gráfico de barras
stairs	Gráfico em forma escada
hist	Gráfico histograma
title	Título do gráfico
xlabel	Título do eixo dos x
ylabel	Título do eixo dos y
subplot	Divide a tela em várias partes
text	Coloca texto num gráfico com ajuda do mouse
grid	Linhas de grelha
plot3	Gráfico em 3-D
mesh	Superfície em 3-D
axes	Cria eixos em posições arbitrárias
axis	Controla a escala dos eixos
hold	Mantém o gráfico corrente

1.9 Valor numérico e zeros de polinômios

Muitas vezes precisamos calcular o valor numérico de um polinômio. Para isto o MATLAB tem o comando *polyval*. Para achar as raízes de um polinômio há o comando *roots*. Como exemplo vamos calcular o valor numérico e descobrir os zeros do polinômio $p(x) = \frac{5}{8}x^5 - 5x^4 - 3x^3 + 2x - 10$. Calcularemos seu valor numérico nos pontos $x = 0, 1, 1.24, 2, 12.56$.
No MATLAB a entrada do polinômio é feita através de um vetor contendo os coeficientes. No nosso caso, $p = [\frac{5}{8} \quad -5 \quad -3 \quad 0 \quad 2 \quad -10]$.Construímos outro vetor com os valores de x. Assim, temos,

```
>> p=[5/8 -5 -3 0 2 -10];
>> x=[0 1 1.24 2  12.56];
>> y=polyval(p,x)
```

e obtemos

```
y = 1.0e+004 *
  -0.0010   -0.0015   -0.0023   -0.0090    6.4997
```

Este último vetor é interpretado da seguinte forma : o valor de p para o primeiro elemento de x é o primeiro elemento do vetor y e assim por diante.
Para achar as raízes de p fazemos

```
>> yy=roots(p);
```

e obtemos

```
yy =
  8.5587
 -0.9890 + 0.8316i
 -0.9890 - 0.8316i
  0.7097 + 0.7848i
  0.7097 - 0.7848i
```

Como podemos notar, o MATLAB fornece todas as raízes do polinômio. Usando o próprio MATLAB podemos comprovar o resultado calculando o valor numérico do polinômio nas suas raízes.

```
>> polyval(p,yy)
```

e obtemos

```
ans = 1.0e-011 *
     0.9297
    -0.0005 + 0.0007i
    -0.0005 - 0.0007i
    -0.0011 + 0.0002i
    -0.0011 - 0.0002i
```

A notação $1.0e-011$ significa 10^{-11}.

Para achar os coeficentes do polinômio que tem as raízes r=-1, 0, 1+i, 3 usamos o comando *poly*,

```
>> r=[-1 0 1+i 3];
>> p=poly(r)
```

obtemos

```
p =
  Columns 1 through 4
   1.0000  -3.0000-1.0000i  -1.0000+2.0000i   3.0000+3.0000i

  Column 5
        0
```

da resposta temos que o polinômio procurado é :

$$p(x) = x^4 - (3+i)x^3 + (-1+2i)x^2 + (3+3i)x.$$

Capítulo 2

Programando em MATLAB

2.1 Primeiros programas

Em MATLAB existem dois tipos de programa: *função (function)* e *roteiro (script)*. Mostraremos, nesta seção, como fazer estes programas e discutiremos a diferença que existe entre os dois tipos na seção 2.5.

Podemos fazer programas com MATLAB usando qualquer editor de texto dando um nome ao programa com a extensão .m .

Por exemplo

1. Construímos o programa ap1.m com as instruções

   ```
   clear
   t=0:.01:10;
   x=sin(t).*cos(t);
   plot(t,x)
   ```

   e colocamos em um diretório da raiz $c:\backslash$, por exemplo, mat . Para irmos ao diretório mat, devemos escrever na tela do MATLAB

   ```
   >> cd c:\mat
   ```

   para executar o programa, devemos escrever na tela o nome do programa

   ```
   >> ap1
   ```

   e obtemos o gráfico da figura 2.1

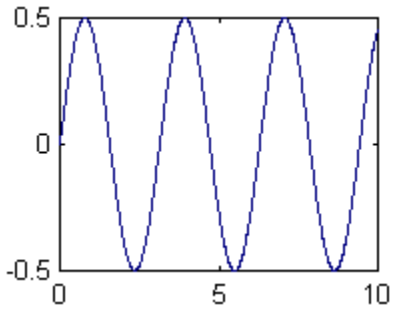


Figura 2.1: Gráfico feito usando MATLAB.

Este tipo de programa é o que chamamos de *roteiro*. Em geral, para este tipo de programa, é conveniente começar com o comando *clear*, para que a memória fique limpa.

2. Construímos a função

```
function y=ap2(x)
y=5*x*sin(x);
```

com o nome ap2.m . Neste programa você escolhe um valor para x na tela do MATLAB e obtém o valor de y

```
>> ap2(3)
```

e temos

```
y = 2.1168
```

Este tipo de programa é o que chamamos de *function*. Neste tipo de programa a primeira linha sempre tem a seguinte estrutura :

function saída=<nome do programa>(entrada).

Se não quisermos a saída, podemos omití-la junto com o sinal =. Devemos ressaltar que o < nome do programa > não leva a extensão '.m'. Na *(entrada)* colocamos as variáveis que servirão como entrada para o programa. O programa deve ser gravado com a extensão '.m'. No decorrer deste capítulo discutiremos mais exemplos.

3. Queremos fazer um programa que nos permita calcular valores para a função

$$f(x) = \begin{cases} 1 & , \text{ se } x < -1 \\ x^2 & , \text{ se } -1 \leq x \leq 1 \\ -x+2 & , \text{ se } x > 1 \end{cases}$$

O programa que cria esta função chamaremos de p11.m e é mostrado a seguir.

```
 function a=p11(x)
 if x<-1
a=1;
 elseif x>=-1 & x<=1
a=x^2;
 else
a=-x+2;
 end
```

Devemos notar que o valor retornado por esta função é um escalar. Vamos, agora, fazer o gráfico da função . Para isso escrevemos o seguinte programa, chamado p12.m

```
 clear
 n=0;
 for t=-2:.01:2
n=n+1;
x(n)=t;
y(n)=p11(t);%Aqui chamamos o programa p11.m, com entrada  t e  saida  y(n)
 end
 plot(x,y)%Aqui fazemos o grafico de x versus y
```

Devemos observar que neste programa foram introduzidos comentários. Para inserir comentários em qualquer dos dois tipos de programas basta preceder o comentário pelo símbolo %.
Para executar este programa devemos escrever, na tela do MATLAB, p12 e obtemos o gráfico da figura 2.2

Um bom exercício é tentar fazer um programa similar a p11.m, mas onde x é um vetor.

2.2 Programas interativos

Muitas vezes precisamos fazer um programa interativo de modo que as entradas possam ser dadas via teclado. Para isso, usamos os comandos *input* e *disp*. Por exemplo:

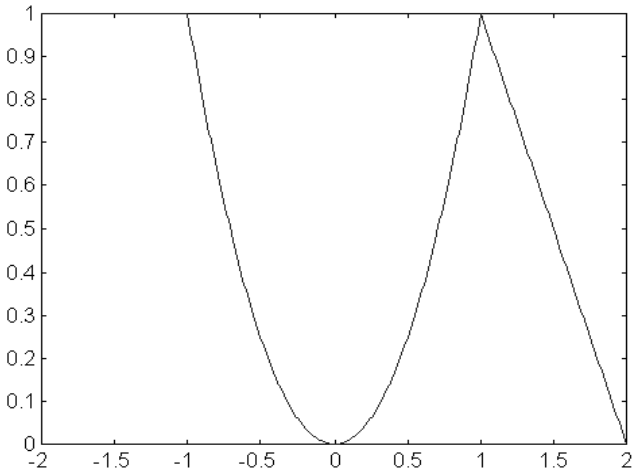


Figura 2.2: Gráfico feito usando o programa p12.m.

1. Queremos calcular o valor da função $f(x) = ae^{-2x+b}$ para valores de a, b e x.

```
clear
disp('Este comando faz com que o texto apareca na tela do MATLAB')
a=input('Entre com o valor de a=');
b=input('Entre com o valor de b=');
x0=input('Entre com o valor inicial de x=');
xf=input('Entre com o valor final de x=');
x=x0:.01:xf;
y=a*exp(-2*x+b);
plot(x,y)
```

Ao executar esse programa serão pedidos os valores das variáveis, uma a uma, i. e., primeiramente aparecerá na tela

```
Este comando faz com que o texto apareca na tela do MATLAB
Entre com  o valor de a=
```

associamos um valor para a, por exemplo 2, e logo aparecerá na tela

```
Entre com  o valor de b=
```

e associamos outro valor para b, por exemplo 3 e assim por diante.
Executando o programa, temos

```
Este comando faz que o texto apareca na tela do MATLAB
Entre com  o valor de a=2
Entre com  o valor de b=3
Entre com  o valor inicial de x=0
Entre com  o valor final de x=2
```

O gráfico que obtemos com este programa é:

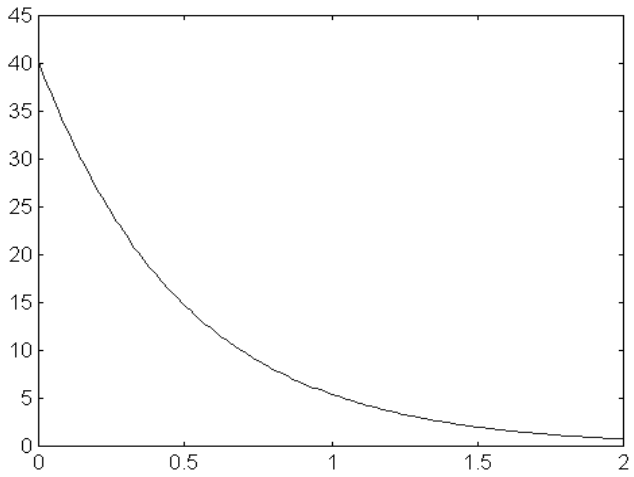


Figura 2.3: Gráfico obtido com o programa interativo

2.3 Zeros de funções

O MATLAB acha zeros de funções usando o comando *fzero*. Para isto é necessário escrever um programa auxiliar que contenha a função , da qual desejamos encontrar os zeros.

1. Queremos achar um zero da função $f(x) = \sin(x) - \cos(x)$.
   Programa auxiliar

   ```
   function f=apo2(x) x=sin(x)-cos(x);
   ```

   Para encontrarmos um zero da função devemos fazer

   ```
   >> fzero('apo2',1)
   ```

   O nome apo2 é o nome do programa auxiliar e o valor 1 é uma aproximação da raiz. A resposta do MATLAB é

```
ans = 0.7854
```

2. Encontremos um zero da função $f(x) = x^2 - e^{5x} + \sin(x)$.
   O programa auxiliar é

```
function f=apo3(x)
f=x^2-exp(-5*x)+sin(x);
```

   Fazendo

```
fzero('apo3',0)
```

   Obtemos

```
ans = 0.2420
```

2.4 Calculando Integrais

Podemos aproximar numericamente integrais definidas, da forma :

$$I = \int_a^b f(x)dx.$$

Podemos calcular a integral usando a regra de Simpson com o comando *quad* ou usando a regra trapezoidal com o comando *trapz*. Discutiremos, primeiro, o cálculo com o uso da regra trapezoidal. A sintaxe do comando é *trapz(x,y)*.

2.4.1 Regra trapezoidal

Para usar *trapz* construímos dois vetores: x que é formado por pontos de uma partição do intervalo $[a, b]$ e y com os valores da função nos pontos da partição do intervalo $[a, b]$. Os vetores x e y têm o mesmo comprimento e (x_i, y_i) representa um ponto na curva. Como exemplo, desejamos calcular a integral :

$$\int_0^1 e^{-x^2}dx.$$

A seguir mostramos a sequência de comandos para realizar esse cálculo e também a resposta obtida.

```
>> x=0:0.1:1;
>> y=exp(-x.^2);
>> trapz(x,y)
ans =
   0.7462
```

Esses comandos podem ser agrupados num programa.

2.4.2 Regra de Simpson

Discutiremos, agora, o uso do comando *quad*. Há várias formas de usar esse comando, daremos um exemplo usando a forma :

```
quad('fc',a,b,tol)
```

que retorna uma aproximação para a integral de f no intervalo $[a, b]$. *fc* é o nome de uma função pré-definida ou um arquivo .m correspondente à f. O parâmetro *tol* refere-se à tolerância desejada.

Como exemplo, calculemos

$$\int_0^1 2xe^{-x^2}dx$$

para diferentes tolerâncias.

Primeiro, criamos o programa *integr.m* que contém a função desejada. Assim,

```
function y=integr(x)
y=2*x.*exp(-x.^2)
```

Calculemos a integral usando a tolerância usual que é 10^{-3}

```
>> quad('integr',0,1)
```

Obtemos como resultado

```
ans = 6.321326608555348e-001
```

Calculando agora com uma tolerância de 10^{-5}

```
>> quad('integr',0,1,1e-5)
```

Obtemos como resultado

```
ans =  6.321205736524541e-001
```

Usamos o formato *long e* que será discutido mais adiante na seção 3.1.

O valor exato desta integral é $1 - e^{-1}$, que calculando no MATLAB

```
>> 1-exp(-1)
```

Obtemos como resultado

```
ans =  6.321205588285577e-001
```

Notemos que quanto menor a tolerância melhor a aproximação para a solução .

2.5 Diferença entre *função* e *roteiro*

A principal diferença entre estes tipos de programas está no aproveitamento de memória do MATLAB.

Na *função* as variáveis usadas na execução do programa não são guardadas na memória do MATLAB após a execução do programa, enquanto que no caso do *roteiro*, as variáveis são guardadas na memória do MATLAB e ficam, portanto, disponíveis para serem usadas por outros programas. Consideremos os exemplos a seguir para mostrar esta diferença:

1. Função (*function*)

```
function x=ex1(t)
a=exp(t);
b=cos(t);
x=a*b;
```

o nome deste programa é ex1.m.

Fazendo

```
>> ex1(2)
```

obtemos

```
ans = -3.0749
```

ao executar o comando

```
>> whos
```

obtemos

```
    Name         Size        Elements     Bytes    Density   Complex
    ans        1 by 1               1         8       Full      No
Grand total is 1 elements using 8 bytes
```

Neste caso as variáveis auxiliares 'a' e 'b' não ficam na memória do MATLAB.

2. Consideremos o mesmo programa anterior escrito como roteiro

```
clear
t=input('Entre com o valor de t =');
a=exp(t);
b=cos(t);
x=a*b
```

Chamemos este programa por ex2.m. Fazendo

```
>> ex2
 Entre com o valor de t =2
```

obtemos

```
 x =  -3.0749
```

Fazendo

```
>> whos
```

obtemos

```
    Name         Size       Elements     Bytes    Density   Complex
      a       1 by 1              1         8       Full      No
      b       1 by 1              1         8       Full      No
      t       1 by 1              1         8       Full      No
      x       1 by 1              1         8       Full      No
 Grand total is 4 elements using 32 bytes
```

Este exemplo mostra neste caso que as variáveis auxiliares usadas no programa ficam na memória do MATLAB.

2.6 O comando global

Este comando é muito importante quando temos vários programas interagindo. Consideremos o exemplo

```
function g1(n)
global a
a=0:.01:n;
b=5:.01:n;
A=[a b];
C=A.^2;
```

Este exemplo de programa *função* mostra a diferença entre variáveis locais e globais. O comando *global a* permite, neste caso, que a variável 'a' seja usada em outros programas, *função* ou *roteiro*. Chamamos essas variáveis de *globais*. As outras, que não são globais, chamamos de *locais*. Estas variáveis só existem enquanto o programa que as contêm está sendo executado. Para que outro programa possa usar uma variável local, ela deve ser globalizada; isto é, temos que usar o comando *global <nome das variáveis>*.

Cabe ressaltar que num programa *roteiro* todas as variáveis são globais, já que ficam na memória do MATLAB.

Outro programa que usa variáveis globais está na seção 2.7.2.

2.7 Resolvendo equações diferenciais

Na plataforma básica do MATLAB encontramos as fórmulas de Runge-Kutta para resolver sistemas de equações diferenciais ordinárias de primeira ordem. São elas *ode23* e *ode45*. O programa *ode23* resolve sistemas de equações diferenciais ordinárias de primeira ordem usando as fórmulas de Runge-Kutta de segunda e terceira ordens, enquanto que *ode45* resolve sistemas de equações diferenciais ordinárias de primeira ordem usando as fórmulas de Runge-Kutta de quarta e quinta ordens.

Há também, na plataforma básica do MATLAB, a função *ode23p* que além de fazer o que a função *ode23* faz, também calcula o gráfico da evolução no tempo das primeiras duas ou três componentes da solução . O MATLAB dispõe, ainda, de outros métodos para resolver equações diferenciais que não estão na plataforma básica. Um destaque especial deve ser dado ao conjunto de programas *ODEsuite* que contém várias funções para resolver equações diferenciais.

2.7.1 Pêndulo simples linearizado

A equação da dinâmica do pêndulo simples linearizado (pequenos valores do ângulo).

$$\ddot{\theta} + \frac{g}{l}\theta = 0.$$

Para resolver esta equação precisamos de um programa principal que explicite o método de solução e de um programa auxiliar que defina a equação a resolver. Usaremos um método Runge-Kutta escrito para resolver sistemas de primeira ordem, conveniente em problemas de dinâmica.Se a equação não estiver nesta forma (primeira ordem) devemos primeiro transformá-la.No nosso exemplo, a equação é de segunda ordem. Devemos, então, fazer uma mudança de variáveis para transforma-la num sistema de duas equações de primeira ordem . A mudança é feita da seguinte forma

$$\begin{aligned} y_1 = \theta &\Rightarrow \dot{y}_1 = \dot{\theta} \\ y_2 = \dot{y}_1 = \dot{\theta} &\Rightarrow \dot{y}_2 = \ddot{\theta} = -\frac{g}{l}\theta = -\frac{g}{l}y_1. \end{aligned}$$

Em resumo, temos duas equações de primeira ordem, dadas por

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= -\frac{g}{l}y_1. \end{aligned}$$

O programa principal que chamaremos de ap3.m é

```
clear
y0=[1 0];
t0=0; tf=10;
[t Y]=ode45('ap4',t0,tf,y0);
```

```
plot(t,Y(:,1));
title('Pendulo');
xlabel('Tempo');
ylabel('Theta')
```

onde

- y0=[1 0]; é o vetor de condições iniciais
- t0=0; é o tempo inicial
- tf=10; é o tempo final
- [t Y]=ode45('ap4',t0,tf,y0); é o comando que chama o programa auxiliar ap4.m onde definimos o sistema a resolver e aplicamos *ode45*. Este comando fornece como saída um vetor [t Y], onde t é o tempo e Y (neste caso) é uma matriz 118 por 2. Na primeira coluna está o valor de $y_1 = \theta$, e na segunda, o valor de $y_2 = \dot{\theta}$.
- plot(t,Y(:,1)); construímos o gráfico $t \times y_1 = \theta$ (podemos construir também o gráfico $t \times y_2$, ou seja, $t \times \dot{\theta}$).

O programa auxiliar, que é chamado de ap4.m, é dado por

```
function A=ap4(t,y)
l=2;
g=9.8;
A=[y(2) -g*y(1)/l];
```

No vetor A=[y(2) -g*y(1)/l] colocamos o valor da derivada de y_1 na primeira posição e na segunda posição o valor da derivada de y_2.

Para executar este programa escrevemos na tela do MATLAB

```
>> ap3
```

e obtemos o gráfico dado na figura 2.4 (a)

Um bom exercicio é refazer o que foi feito acima usando *ode23* e *ode23p*.

2.7.2 Sistema massa-mola-amortecedor

Consideremos o sistema massa-mola-amortecedor

$$m\ddot{x} + c\dot{x} + kx = asin(t)$$

Faremos um programa geral para resolver esta equação,i.e., um programa que resolva esta equação para quaisquer m, c, k, a e condições iniciais dadas. A troca de variáveis

para transformar esta equação num sistema de duas equações de primeira ordem é feita da seguinte forma

$$\begin{aligned} y_1 = x &\Rightarrow \dot{y}_1 = \dot{x} \\ y_2 = \dot{y}_1 = \dot{x} &\Rightarrow \dot{y}_2 = \ddot{x} = \frac{a}{m}sin(t) - \frac{c}{m}\dot{x} - \frac{k}{m}x = \frac{a}{m}sin(t) - \frac{c}{m}y_2 - \frac{k}{m}y_1. \end{aligned}$$

Em resumo, temos duas equações de primeira ordem

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= \frac{a}{m}sin(t) - \frac{c}{m}y_2 - \frac{k}{m}y_1. \end{aligned}$$

Consideremos ap5 o programa principal

```
function ap5(m1,c1,k1,a1,p,q,t0,tf)
global m c k a
m=m1;
c=c1;
k=k1;
a=a1;
y0=[p q];
[t Y]=ode45('ap6',t0,tf,y0);
plot(t,Y(:,1));
title('Sistema MASSA-MOLA-AMORTECEDOR');
xlabel('TEMPO');
ylabel('DESLOCAMENTO')
```

onde

- O comando *global* permite que essas variáveis sejam usadas em outros programas sem ter que definí-las novamente. Este comando já foi tratado na seção 2.6.
- y0=[p q] é o vetor de condições iniciais
- t0=0 é o tempo inicial
- tf=10 é o tempo final
- [t Y]=ode45('ap4',t0,tf,y0) é o comando que chama a programa auxiliar ap6.m onde definimos o sistema a resolver e aplicamos *ode45*.
- plot(t,Y(:,1)); construímos o gráfico $t \times x$.

O programa auxiliar que é chamado ap6.m é dado por

```
function A=ap6(t,y)
global m c k a
A=[y(2) (a*sin(t)-c*y(2)-k*y(1))/m];
```

Para executar este programa escrevemos na tela do MATLAB

```
>>  ap5(1,.5,2,0,1,0,0,10)
```

onde o primeiro valor corresponde ao m,o segundo a c,e os outros a k, a, $x(0)$, $\dot{x}(0)$, $t0$ e tf respectivamente.
Ao executarmos o programa temos o gráfico da figura 2.4(b).

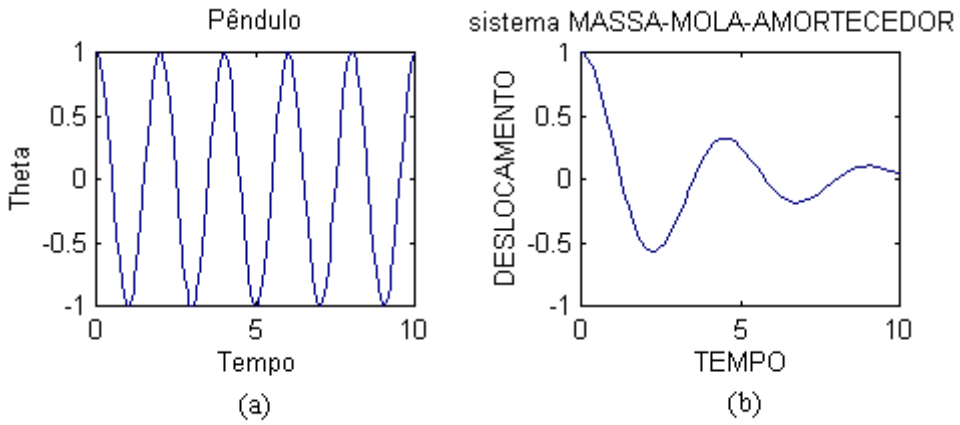


Figura 2.4: (a) Pêndulo; (b) Massa-Mola-Amortecedor.

2.8 Fazendo mais de um gráfico na mesma tela.

Muitas vezes precisamos de mais de um gráfico na mesma tela. Para fazer isto dividimos a tela gráfica com o comando "subplot(m,n,k)".Este comando divide a tela em m linhas e n colunas, que serão ocupadas pelos m x n gráficos, e k é a referência do gráfico. Este comando indica que
(i) a tela foi dividida em **m** linhas e **n** colunas;
(ii) estamos no gráfico **k**, onde $1 \leq k \leq m * n$, e
(iii) os gráficos serão sequencialmente dispostos em linhas.

Como exemplo, consideremos o comando subplot(3,2,1). Com este comando o gráfico será colocado na primeira posição da primeira linha. Com subplot(3,2,4) o gráfico será colocado na segunda posição da segunda linha.

O programa a seguir (p7.m) mostra como fazer isto

```
clear
t=0:.01:10;
x=3*sin(10*t);
y=t.*cos(t);
```

```
z=exp(-.2*t).*x;
w=x.*y;
subplot(2,2,1)
plot(t,x)
title('GRAFICO 1-X')
xlabel('t')
ylabel('x')
subplot(2,2,2)
plot(t,y)
title('GRAFICO 2-Y')
xlabel('t')
ylabel('y')
subplot(2,2,3)
plot(t,z)
title('GRAFICO 3-Z')
xlabel('t')
ylabel('z')
subplot(2,2,4)
plot(t,w)
title('GRAFICO 4-W')
xlabel('t')
ylabel('w')
```

Fazendo

```
>>  p7
```

Temos os gráficos da figura 2.5.

Podemos também colocar vários gráficos na mesma tela, usando o mesmo eixo de abscissas.O programa a seguir mostra como fazer.

```
clear
t=0:.01:10;
x=3*sin(10*t);
y=t.*cos(t);
plot(t,x,t,y)
```

Executando, obtemos o gráfico da figura 2.6 (a)

2.9 Gráficos em 3-D

Há vários comandos para serem utilizados com gráficos em 3-D.

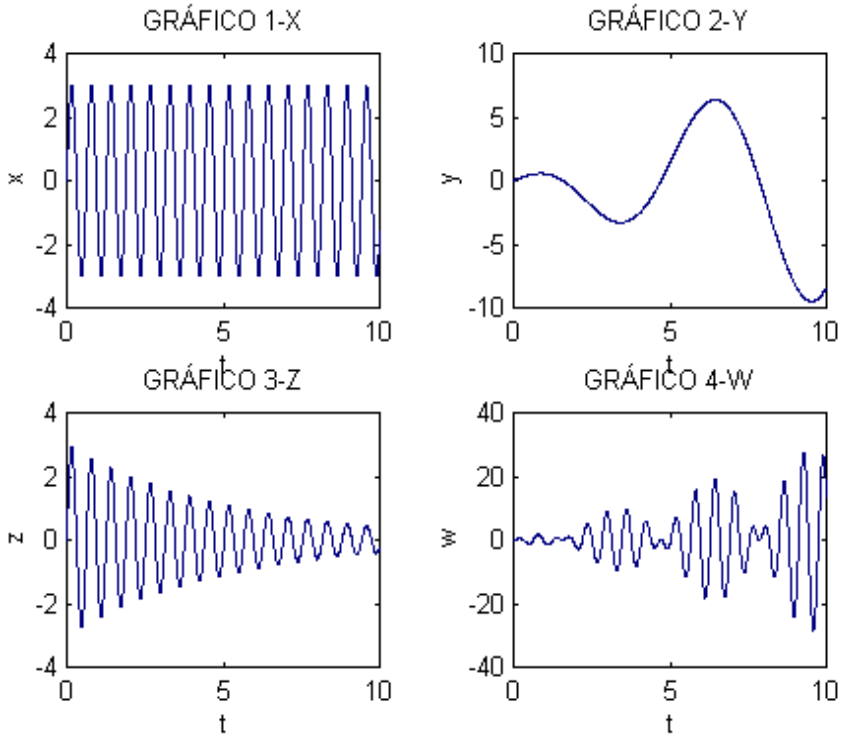


Figura 2.5: Vários gráficos na mesma tela.

1. O comando *plot3*.
   O programa a seguir (ap8.m) mostra como usar o comando.

```
%Programa ap8.m
 clear
 x=0:.01:10;
 y=sin(5*t);
 z=cos(5*t);
 plot3(x,y,z);
 xlabel('eixo X');
 ylabel('eixo Y');
 zlabel('eixo Z');
```

   Devemos notar que no comando plot3(x,y,z), x,y e z devem ser vetores de mesmo comprimento.
   Com este programa obtemos o gráfico da figura 2.6 (b)

2. Os comandos *mesh*, *surf* e *contour*.
   Esses comandos são usados para a construção de gráficos de superfícies (*mesh* e *surf*) e mapas de contorno (*contour*).

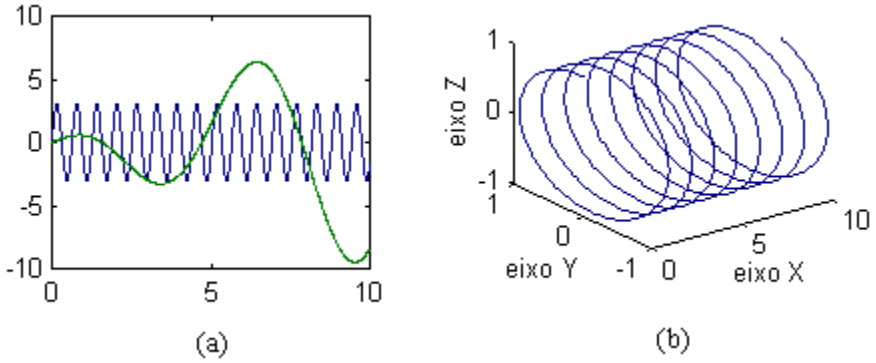


Figura 2.6: (a) Gráficos na mesma tela; (b) Gráfico em 3-D.

```
 clear
 x=-2:.1:2;
 y=-5:.1:5;
 n=length(x);
 m=length(y);
 for j=1:n
for i=1:m
  Z(i,j)=sin(x(j))*cos(y(i));
end
 end
 subplot(221);
 mesh(x,y,Z)
 title('Usando MESH')
 xlabel('X')
 ylabel('Y')
 zlabel('Z')
 subplot(222);
 surf(x,y,Z)
 title('Usando SURF')
 xlabel('X')
 ylabel('Y')
 zlabel('Z')
 subplot(223);
 contour(x,y,Z,10)
 title('Usando CONTOUR com 10 linhas')
 xlabel('X')
 ylabel('Y')
 subplot(224);
 contour(x,y,Z,30)
```

```
title('Usando CONTOUR com 30 linhas')
xlabel('axe X')
ylabel('axe Y')
```

Obtemos os seguintes gráficos (olhar fig 2.7).

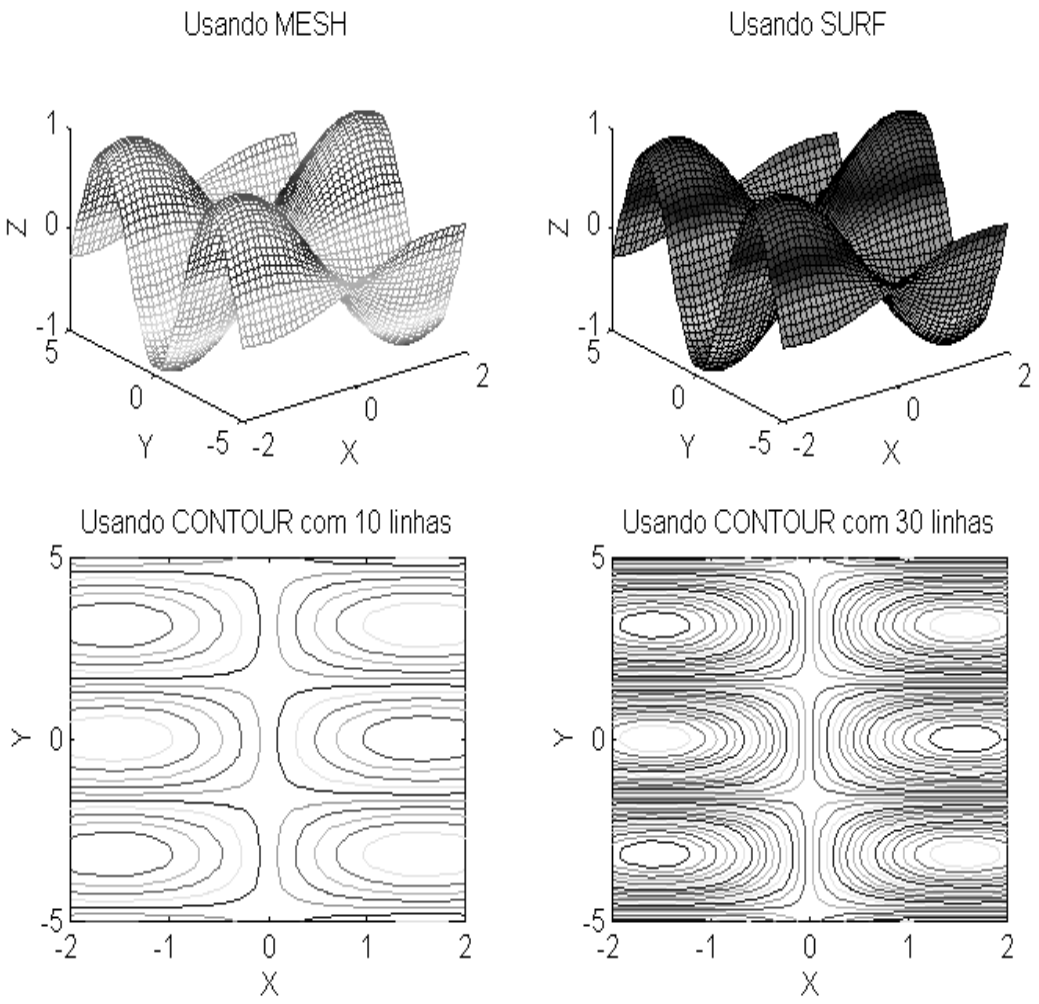


Figura 2.7: Gráficos em 3-D

Capítulo 3

Outros comandos

3.1 Formato de saída

Ainda que todos os cálculos em MATLAB sejam feitos com precisão dupla, o formato da saída na tela pode ser modificado com as seguintes instruções :

format short	dá quatro decimais (o usual)
format long	dá quatorze decimais
format short e	notação científica com 4 decimais
format long e	notação científica com quinze decimais.

Quando um formato é prescrito este é mantido até que seja prescrito outro formato.

Como exemplo escreveremos o número π com os quatro formatos discutidos:

3.1416	*short*
3.14159265358979	*long*
3.1416e+000	*short e*
3.14159265358793e+000	*long e.*

3.2 Comando diary

O comando *diary <nome de arquivo>* faz com que tudo o que aparecer na tela, daquele momento em diante, (exceto gráficos), seja guardado no arquivo *<nome de arquivo>* até que ordenemos *diary off*. Para continuar guardando os dados escrevemos *diary on*. Por exemplo

```
>> clear
>> diary ss
>> a=[1 2 3 4 5 6];
>> b=a.^2
```

```
 b =
     1     4     9    16    25    36

 >>  c=[1 2 3; 4 5 6]

c =
     1     2     3
     4     5     6

>> d=c'

d =
     1     4
     2     5
     3     6

 >> diary off
```

No arquivo *ascii ss* ficam guardados todos estes comandos. Abrindo este arquivo com um editor de texto temos

```
 a=[1 2 3 4 5 6];

b=a.^2


b =
     1     4     9    16    25    36

c=[1 2 3; 4 5 6]


c =
     1     2     3
     4     5     6

d=c'


d =
     1     4
     2     5
     3     6
```

```
diary off
```

Fazendo na tela do MATLAB

```
 >> diary on
 >> d=[1 2; 3 4];
 >> e=inv(d)

e =
   -2.0000    1.0000
    1.5000   -0.5000

 >> diary off
```

o arquivo fica

```
 a=[1 2 3 4 5 6];

b=a.^2


b =
     1     4     9    16    25    36

c=[1 2 3; 4 5 6]


c =
     1     2     3
     4     5     6

d=c'


d =
     1     4
     2     5
     3     6

diary off

d=[1 2; 3 4];
```

```
e=inv(d)


e =
   -2.0000    1.0000
    1.5000   -0.5000

diary off
```

3.3 Comandos save e load

Muitas vezes quando rodamos um programa, obtemos como resposta um arquivo de dados e é de nosso interesse guardar estes dados. Os dados do MATLAB podem ser guardados em formato *ascii* ou no formato *mat*, que é um formato próprio do MATLAB. Os comandos são usados da seguinte forma:

save <arquivo.dat> A B -ascii	guarda as variáveis A B em arquivo.dat em formato ascii
save <arquivo> A B	guarda as variáveis A B em arquivo.mat em formato mat
save <arquivo>	guarda todas as variáveis em arquivo.mat em formato mat
load <arquivo>	lê o arquivo para ser usado no MATLAB

Com os exemplos a seguir mostraremos como são usados estes comandos

1. Para formato *.mat*

```
>>t=1:.5:10;

>>s=[1 2; 3 4];

>>w=[1 2 3; 4 5 6; 6 -7 8];

>>whos

     Name        Size       Elements     Bytes    Density   Complex
        s       2 by 2              4        32       Full      No
        t       1 by 19            19       152       Full      No
        w       3 by 3              9        72       Full      No
Grand total is 32 elements using 256 bytes
```

Até agora criamos três variáveis.Guardemos estas variáveis da seguinte forma

```
>>save ex1 w

>>save ex2
```

No primeiro comando guardamos a variável w em formato mat no arquivo ex1.mat (próprio do MATLAB). No segundo comando guardamos todas as variáveis em formato mat no arquivo ex2.mat.

Vamos ler os arquivos criados.
Para ler o arquivo ex1.mat fazemos

```
>>clear
>>load ex1

>>whos

     Name        Size       Elements     Bytes    Density   Complex
       w       3 by 3             9        72       Full      No
Grand total is 9 elements using 72 bytes
```

Neste exemplo o nome da variável guardada é mantido no arquivo ex1.mat.
Para ler o arquivo ex2.dat fazemos

```
>>clear

>>load ex2

>>whos

     Name        Size       Elements     Bytes    Density   Complex
        s       2 by 2             4        32       Full      No
        t       1 by 19           19       152       Full      No
        w       3 by 3             9        72       Full      No
Grand total is 32 elements using 256 bytes
```

Neste exemplo ainda mantemos os nomes das variáveis. O formato *.mat* é uma boa alternativa quando pretendemos só trabalhar no MATLAB. Muitas vezes necessitamos alternar com outras linguagens, como por exemplo Fortran. Neste caso surge a necessidade de guardarmos as saídas do MATLAB em formato *ascii*. Os exemplos a seguir mostram como fazemos isto.

2. O formato *ascii*.
   Consideremos as seguintes variáveis

```
>> a=1:-2:-5
a =   1    -1    -3    -5
>>  b=4
b = 4
>> c=8
c = 8
>> A=[1 2 3; 4 5 6; 2 9 0]
A = 1     2     3
    4     5     6
    2     9     0
>> f=[2; 4; 5]
f =  2
     4
     5
>> B=[1 2 3; 9 8 2]
B =  1     2     3
     9     8     2
```

Criemos os seguintes arquivos *ascii*

```
>> save ex3a.dat b -ascii
>> save ex3b.dat b c -ascii
>> save ex3.dat a -ascii
>> save ex4.dat A -ascii
>> save ex5.dat A B -ascii
>> save ex6.dat B f -ascii
```

No primeiro comando guardamos o conteúdo da variável b em formato ascii no arquivo ex3a.dat. No segundo comando guardamos os conteúdos de b e c em formato ascii no arquivo ex3b.dat. No terceiro guardamos o contúdo de a em formato ascii no arquivo ex3.dat. No quarto comando guardamos o conteúdo de A em formato ascii no arquivo ex4.dat. No quinto comando guardamos os conteúdos de A e B em formato ascii no arquivo ex5.dat. Finalmente com o último comando guardamos os conteúdos de B e f em formato ascii no arquivo ex6.dat.

Desejamos, agora, ler esses arquivos.

```
>> clear
>> load ex3a.dat
>> whos
   Name        Size       Elements     Bytes    Density   Complex
```

```
   ex3a       1 by 1              1         8       Full      No
Grand total is 1 elements using 8 bytes
```

Notemos que nos arquivos *ascii*, a variável perde seu nome e assume o nome do arquivo. Vamos abrir o arquivo 'ex3a.dat' usando um editor de texto.

```
 4.0000000e+000
```

Para ex3b.dat

```
>> clear
>> load ex3b.dat
>> whos
   Name        Size       Elements     Bytes    Density   Complex
   ex3b       2 by 1              2        16       Full      No
Grand total is 2 elements using 16 bytes
```

Neste caso temos só uma variavel chamada ex3b. Vamos abrir o arquivo 'ex3b.dat' usando um editor de texto.

```
 4.0000000e+000
 8.0000000e+000
```

As variáveis são armazenadas em colunas e seguem a ordem estabelecida quando gravadas, neste caso *b=ex3b(1)* e *c=ex3b(2)*.
Para ex3.dat

```
>> clear
>> load ex3.dat
>> whos
  Name        Size       Elements     Bytes    Density   Complex
   ex3       1 by 6              6        48       Full      No
Grand total is 6 elements using 48 bytes
```

Abrindo o arquivo 'ex3.dat' usando um editor de texto, temos

```
 1.0000000e+000 -1.0000000e+000 -3.0000000e+000 -5.0000000e+000
```

Para ex4.dat

```
>> clear
>> load ex4.dat
>> whos
      Name        Size       Elements     Bytes    Density   Complex
       ex4       3 by 3             9        72       Full      No
Grand total is 9 elements using 72 bytes
```

O arquivo ex4.dat

```
 1.0000000e+000  2.0000000e+000  3.0000000e+000
 4.0000000e+000  5.0000000e+000  6.0000000e+000
 2.0000000e+000  9.0000000e+000  0.0000000e+000
```

Notemos que a matriz é guardada da forma original, só o formato é trocado.
Para o arquivo ex5.dat

```
>> clear
>> load ex5.dat
>> whos
   Name        Size       Elements     Bytes    Density   Complex
   ex5        5 by 3            15       120       Full      No
Grand total is 15 elements using 120 bytes
```

O arquivo ex5.dat

```
 1.0000000e+000  2.0000000e+000  3.0000000e+000
 4.0000000e+000  5.0000000e+000  6.0000000e+000
 2.0000000e+000  9.0000000e+000  0.0000000e+000
 1.0000000e+000  2.0000000e+000  3.0000000e+000
 9.0000000e+000  8.0000000e+000  2.0000000e+000
```

Como nos casos anteriores os nomes das matrizes são perdidos e agora temos uma só matriz chamada ex5.dat. As matrizes originais foram armazenadas por colunas. Para recuperar as matrizes originais devemos observar que nas três primeiras linhas está a matriz A e nas linhas 4 e 5 está a matriz B. Para recuperarmos as matrizes fazemos:

```
>> A=ex6(1:3,:)
A =
    1     2     3
    4     5     6
    2     9     0
```

```
>> B=ex6(4:5,:)
B =
    1     2     3
    9     8     2
```

Neste exemplo devemos tomar cuidado, pois as matrizes armazenadas devem ter o mesmo número de colunas.
Pelo que foi dito aqui, observamos que o arquivo ex6.dat não foi criado corretamente.

```
>> clear
>> load ex6.dat
??? Error using ==> load
Number of columns on line 4 of ASCII file ex5
must be the same as previous lines.
```

Olhando o arquivo temos

```
1.0000000e+000  2.0000000e+000  3.0000000e+000
4.0000000e+000  5.0000000e+000  6.0000000e+000
2.0000000e+000  9.0000000e+000  0.0000000e+000
2.0000000e+000
4.0000000e+000
5.0000000e+000
```

Neste caso o arquivo foi criado, mas o MATLAB não pode lê-lo.

3.4 Comando cputime

O comando *cputime* dá o tempo de CPU, em segundos, dos processos do MATLAB desde o começo da seção .

Consideremos o programa a seguir

```
t=cputime;
   operações
cputime-t
```

Desta maneira obtemos o tempo de CPU gasto para executar as operações especificadas.

Este comando serve para testar a eficiência de nossos programas.
Como aplicação, consideremos o programa

1. ```
   clear
   t=cputime;
   a1=0:5000;
   a2=0:2:10000;
   a3=10000:15000;
   A=[a1 ; a2; a3];
   for i=1:3
   for j=1:5001
   C(i,j)=A(i,j)^3;
   end
   end
   cputime-t
   ```

   Ao executar este programa obtemos o tempo de CPU gasto, que foi de 5.0600 seg.

2. ```
   clear
   t=cputime;
   a1=0:5000;
   a2=0:2:10000;
   a3=10000:15000;
   A=[a1 ; a2; a3];
   C=A.^3;
   cputime-t
   ```

   A executar este programa obtemos o tempo de CPU gasto, que foi de 0.1100 seg.

Notemos que os tempos dependem da configuração de seu computador, mas as magnitudes são proporcionais. Os dois programas fazem as mesmas contas, mas de maneiras diferentes. Observamos, porém, que a maneira utilizada no segundo programa é mais eficiente que no primeiro, pois utiliza a forma matricial de tratar os dados.

3.5 O comando fprintf

Este comando serve para mostrar a saída de um programa. A sintaxe do comando é: *fprintf('formato de saída',variáveis)*. O *formato de saída* contém o texto e o formato desejado para as variáveis especificadas em *variáveis*. Usamos %e, %f e %g para formatar a saída . Se %e é usado, os conteúdos das variáveis a serem impressas estarão em notação exponencial e com %f os conteúdos das variáveis serão mostrados na notação comum de números reais. Com %g é usada a notação %e ou %f dependendo do caso. Se o caráter \n aparecer no formato de saída, a impresão continuará na linha seguinte. O formato de saída usual deve terminar com um \n. Para esclarecer melhor este comando consideremos o seguinte exemplo.

3.5.1 Exemplo

Consideremos A uma matriz quadrada. Desejamos encontrar seus autovalores e seus autovetores.

```
function pp1(A)
n=length(A(:,1));
[V,D]=eig(A);
for i=1:n
fprintf('Valor proprio Lambda(%g) = %f \n',i,D(i,i))
fprintf('Vetor proprio associado ao valor Lambda(%g),\n v(%g) = \n ',i,i);
disp(V(:,i))
pause(2)
end
```

Neste programa *disp(V(:,i))* é usado para evitar na saída do vetor (coluna) a palavra *ans*. O comando *pause(2)* faz com que o programa espere dois segundos para executar o comando seguinte. Se usamos só *pause*, a execução é interrompida até que seja pressionada uma tecla. A execução e a saída deste programa são mostradas a seguir.

```
>> A=[1 2 3 1 0; 3 -10 4 5 6; 0 1 0 1 2; 3 -10 4 5 6; 1 2 3 4 5 ];
>> pp1(A)
Valor proprio Lambda(1) = 9.644410
Vetor proprio associado a valor Lambda(1),
 v(1) =
     0.2271
     0.4149
     0.2394
     0.4149
     0.7395
Valor proprio Lambda(2) = -8.689421
Vetor proprio associado a valor Lambda(2),
 v(2) =
     0.1777
    -0.6680
     0.0941
    -0.6680
     0.2592
Valor proprio Lambda(3) = 0.511543
Vetor proprio associado a valor Lambda(3),
 v(3) =
     0.9323
     0.1148
    -0.2666
```

```
     0.1148
    -0.1830
Valor proprio Lambda(4) = -0.466532
Vetor proprio associado a valor Lambda(4),
 v(4) =
    -0.8392
    -0.1105
     0.5207
    -0.1105
    -0.0110
Valor proprio Lambda(5) = 0.000000
Vetor proprio associado a valor Lambda(5),
 v(5) =
     0.6968
     0.0942
    -0.4519
     0.4708
    -0.2825
```

Apêndice A

Um programa para sistemas de equações lineares

Dado um sistema de equações lineares $AX = b$ queremos saber se o sistema tem ou não solução . Se tiver solução desejamos saber se ela é única ou não . Se a solução for única desejamos conhecer a solução. Se o sistema tiver infinitas soluções desejamos conhecer pelo menos uma solução .

O programa a seguir discute, para um sistema dado, se a solução existe ou não e quando existir se é única ou não, fornecendo também a solução (quando é única) e forncendo ao menos uma solução (quando não é única). Quando o sistema não tiver solução desejamos que o programa encontre um vetor X que minimize o erro pelo método dos mínimos quadrados.

Para saber se o sistema tem ou não solução o programa compara o posto da matriz A com o posto da matriz $[A : b]$. O sistema terá solução quando ambos os postos forem iguais. Para a unicidade precisamos ainda que o posto da matriz A seja igual ao número de incógnitas do sistema, caso contrário o sistema terá infinitas soluções . O programa encontra, neste caso, pelo menos uma solução . No caso do sistema não ter solução o programa encontra um vetor X que minimiza o erro do sistema $AX = b$ pelo método dos mínimos quadrados.

```
function sist1(A,b)
n=rank(A);%Da o posto da matriz A
fprintf('Posto da matriz A= %g \n',n);
m=rank([A b]);%Da o posto de [A:b]
fprintf('Posto da matriz [A:b]= %g \n',m);
if n==m & n==length(A(1,:))
disp('O sistema tem sol unica')
X2=pinv(A)*b;
disp(X2);
```

```
elseif n==m & n<length(A(1,:))
disp('O sistema tem infinitas sols')
disp('Usando divisao a esquerda uma solu\c c\~ao e')
X1=A\b;
disp(X1);
disp('Usando inversa generalizada uma solu\c c\~ao e')
X2=pinv(A)*b;
disp(X2);
else
disp('O sistema nao tem sol')
disp('Usando divisao a esquerda uma aproximacao e')
X1=A\b;
disp(X1);
e1=b-A*X1;
J1=e1'*e1/2;
fprintf('Erro da divisao a esquerda = %f \n',J1)
disp('Usando inversa generalizada uma aproximacao e')
X2=pinv(A)*b;
disp(X2);
e2=b-A*X2;
J2=e2'*e2/2;
fprintf('Erro da inversa generalizada = %f \n',J2)
end
```

Para olhar o funcionamento do programa consideremos os seguintes exemplos

A.1 Sistema com solução única

Consideremos o sistema

$$\begin{array}{rl} x+2y+3z= & 2 \\ 2x-y+z= & 3 \\ 3x-y+2z= & 0 \end{array}$$

Para achar a solução do sistema fazemos

```
>> A=[1 -2 3;2 -1 1;3 -1 2];
>> b=[2;3;0];
>> sist1(A,b)
\end{verbatim]
de onde temos
\begin{verbatim}
Posto da matriz A= 3
Posto da matriz [A:b]= 3
```

```
O sistema tem sol unica
    0.2500
   -5.7500
   -3.2500
```

A.2 Sistema com infinitas soluções

Consideremos o sistema

$$\begin{array}{rl} x+2y+3z+w= & 1 \\ 2x-y+z-w= & 0 \\ 3x-y+2z+2w= & 0 \end{array}$$

Para achar a solução do sistema fazemos

```
>> A=[1 -2 3 1;2 -1 1 -1;3 -1 2 2];
>> b=[1;0;0];
>> sist1(A,b)
```

de onde temos

```
Posto da matriz A= 3
Posto da matriz [A:b]= 3
O sistema tem infinitas sols
Usando divisao a esquerda uma solu\c cao e
   -0.2500
         0
    0.4375
   -0.0625

Usando inversa generalizada uma solu\c cao e
   -0.2500
   -0.2115
    0.2788
   -0.0096
```

A.3 Sistema sem solução

Consideremos o sistema

$$\begin{array}{rl} x+2y+3z+w= & 1 \\ 2x-y+z-w= & 0 \\ 3x-y+2z+2w= & 0 \\ x+y+10z+w= & 2 \\ 2x-y+z-w= & 2 \end{array}$$

Para achar a solução do sistema fazemos

```
>> A=[1 2 3 1;2 -1 1 -1;3 -1 2 2; 1 1 10 1;2 -1 1 -1];
>> b=[1;0;0;2;2];
>> sist1(A,b)
```

de onde temos

```
Posto da matriz A= 4
Posto da matriz [A:b]= 5
O sistema nao tem sol
Usando divisao a esquerda uma aproximacao e
    0.3125
    0.3125
    0.1875
   -0.5000

Erro da divisao a esquerda = 1.000000
Usando inversa generalizada uma aproximacao e
    0.3125
    0.3125
    0.1875
   -0.5000

Erro da inversa generalizada = 1.000000
```