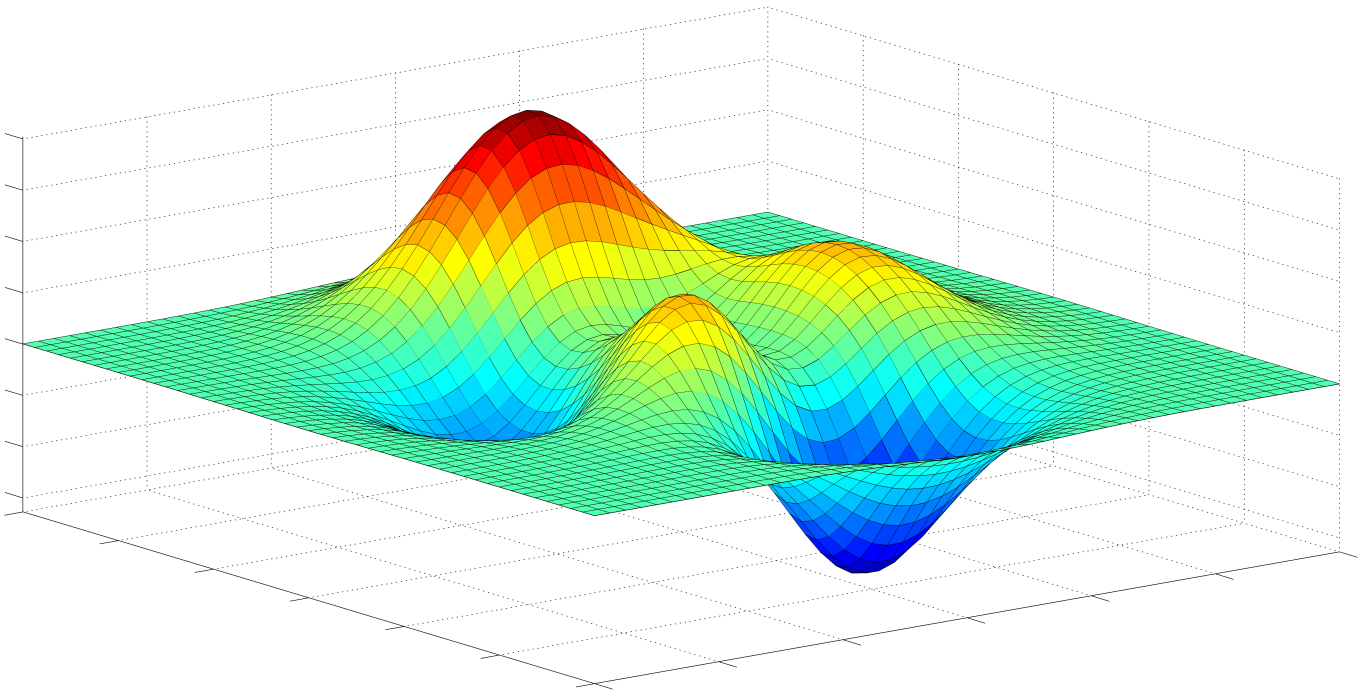
MATLAB
AVANÇADO

Melissa Weber Mendonça



UFSC

Conteúdo

CAPÍTULO 1

Estruturas de dados e MATLAB Básico

1.1 Revisão

Vamos rever alguns conceitos básicos para a programação em MATLAB.

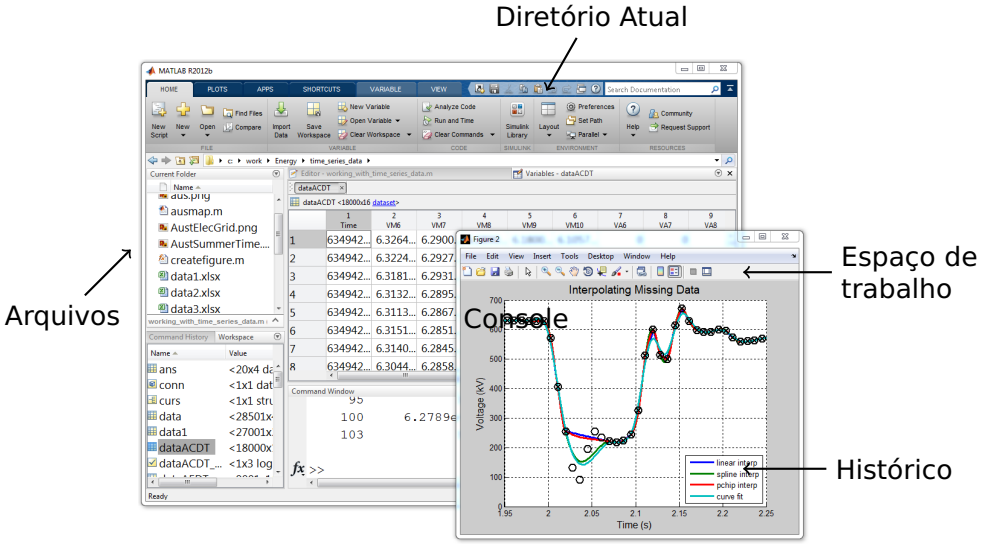


Figura 1.1: Janela de trabalho do MATLAB (varia de acordo com a versão instalada)

1.1.1 Console e scripts

Para realizar tarefas simples no MATLAB, podemos usar diretamente o *console*, digitando os comandos e obtendo as respostas. Se tivermos uma sequência de comandos a serem executados, podemos salvar estes comandos, em ordem, e um arquivo de extensão **`.m`** chamado *script*. Em seguida, podemos digitar o nome deste arquivo no console, sem a extensão **`.m`**, tomando cuidado para que isso seja feito dentro do diretório em que o arquivo foi salvo. Assim, se temos um *script* **`arquivo.m`** salvo no diretório atual, podemos executar os comandos contidos neste script digitando no console

```
» arquivo
```

e em seguida acionando a tecla *Enter*.

1.1.2 Atribuição

Para determinarmos uma variável e atribuir um valor a ela no MATLAB, basta digitarmos

```
» variavel = valor
```

(não é preciso declarar variáveis no MATLAB).

Assim, para criar diferentes tipos de variável, usamos os seguintes comandos:

- Números (inteiros ou reais):

```
» a = 1
» b = 3.14
```

- Vetores: as entradas devem estar separadas por vírgulas, espaços ou ponto-e-vírgula, dependendo da dimensão desejada para o vetor. Por exemplo,

```
» v = [1,2,3]
» v = [1 2 3]
```

  definem um vetor com uma linha e três colunas; já

```
» u = [1;2;3]
```

  define um vetor com três linhas e uma coluna.

- Matrizes: para definir uma matriz, entre com as entradas por linhas, separando as linhas (que devem ter todas o mesmo número de colunas) por ponto-e-vírgula. Exemplo:

```
» A = [1 2 3;4 5 6]
```

- Texto: o valor de uma variável de texto deve ser sempre informado entre aspas.

```
» texto = 'Aqui vai meu texto.'
```

Observe que para que o MATLAB execute o comando sem mostrar o resultado no console, basta acrescentarmos ponto-e-vírgula ao final da linha:

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

Além disso, para que o MATLAB exiba o valor de uma variável, basta digitarmos o nome da variável no console e em seguida pressionar *Enter*:

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

Já para mostrarmos mensagens de texto no MATLAB, usamos o comando **disp**:

```
»  disp('Olá!')
```

Para entrarmos com comandos longos, separando-os em várias linhas, podemos usar **...**:

```
    »  soma = 1+2+3+4+5 ...
          +6+7+8+9+10
ans =
       55
```

1.1.3 Matrizes

No MATLAB, *todas as variáveis são matrizes*. Isso significa que todas as variáveis podem ser acessadas através de índices de linha e coluna, informados nesta ordem, entre parênteses e separados por vírgula. É importante lembrar que no MATLAB todas as variáveis são indexadas a partir do índice **1**.

Para acessarmos os elementos de uma variável, usamos a sintaxe do seguinte exemplo:

```
    »  v = [5;6;7]
    »  v(1)
ans =
       5
    »  v(2,1)
ans =
       6
    »  A = [1 0 2;3 6 5;5 3 4]
    »  A(1,2)
ans =
       0
    »  numero = 2
    »  numero(1)
ans =
       2
    »  numero(1,1)
ans =
       2
```

Observação. O MATLAB permite que se acesse os elementos de uma matriz usando um índice único; nesse caso, os elementos são acessados da seguinte maneira:

$$A(i + m(j-1)) = A(i,j),$$

com $1 \leq i \leq m$, $1 \leq j \leq n$, $A \in \mathbb{R}^{m \times n}$.

1.1.4 Comandos básicos

Em seguida, listamos alguns comandos básicos do MATLAB.

- **eye(n,n)** (ou **eye(n)**) cria uma matriz identidade de dimensão $n \times n$.
- **zeros(m,n)** cria uma matriz de zeros de dimensão $m \times n$.
- **ones(m,n)** cria uma matriz de números 1 de dimensão $m \times n$.
- **rand(m,n)** cria uma matriz de dimensão $m \times n$ com entradas aleatórias (com valor entre 0 e 1).
- **size(A)** retorna um vetor contendo as dimensões da variável A (que pode ser um número, um vetor, uma matriz ou um texto).
- **length(A)** retorna o comprimento da maior dimensão da variável A (se A for uma matriz m por n, com $m \geq n$, o resultado será m).
- **inv(A)** calcula a matriz inversa da matriz A.
- **k*a** calcula o produto de k por cada elemento de a. Se a for um vetor, ou uma matriz, o resultado é uma variável de mesma dimensão de a, com seus respectivos elementos multiplicados por k.
- **a*b** retorna o produto de dois escalares a e b, ou multiplica duas matrizes (ou dois vetores), caso tenham dimensão compatível, usando a fórmula de produto de matrizes que já conhecemos.
- **a.*b** multiplica *os elementos* de quaisquer duas variáveis de mesmo tamanho a e b. Exemplo:

```
    »  a = [1;2;3]
    »  b = [4;5;6]
    »  a.*b
ans =
       4
      10
      18
```

- **a/b** divide o número a pelo número b ($b \neq 0$). Se a e b forem matrizes de dimensão compatível, isso é o mesmo que calcular $a * inv(b)$.
- **a./b** divide *os elementos* de quaisquer duas variáveis de mesmo tamanho a e b. Exemplo:

```
    »  a = [1;2;3]
    »  b = [4;5;6]
    »  a./b
ans =
      1/4
      2/5
      3/6
```

- **a^k** calcula a^k, se a for um número ou uma matriz.

- **a.^k** eleva cada um dos elementos da variável *a* (que pode ser de qualquer tipo) à *k*-ésima potência. Exemplo:

```
»  a = [1;2;3]
»  a.^2 = [1;4;9]
```

- **a'** (ou **transpose(A)**) calcula a transposta de *a*. Se *a* for um número, isso é a mesma coisa que *a*; se *a* for um vetor ou matriz, o resultado é o vetor transposto ou a matriz transposta.
- **reshape(A,m,n)** reorganiza os elementos da matriz em uma nova forma, com **m** linhas e **n** colunas.

Observação Um texto, no MATLAB, é tratado como uma matriz (ou um vetor). Pode-se calcular tamanho e acessar elementos por índices assim como fizemos com as matrizes numéricas. Exemplo:

```
    »  texto = 'Palavra'
    »  texto(1)
ans =
       'P'
    »  size(texto)
ans =
       1    7
    »  texto'
ans =
       P
       a
       l
       a
       v
       r
       a
```

1.1.5 Acessando submatrizes: *Slicing*

O MATLAB oferece uma maneira fácil de se acessar subelementos de matrizes, chamada *slicing*. Nesta operação, usamos a sintaxe

A(linhainicial:linhafinal, colunainicial:colunafinal)

para acessar a submatriz determinada entre as linhas **linhainicial** e **linhafinal**, e entre as colunas **colunainicial** e **colunafinal**. Aqui, é preciso tomar cuidado para que as dimensões da matriz resultante sejam consistentes.

Exemplo:

```
    »  A = [1 2 3;4 5 6;7 8 9];
    »  A(2,:)
ans =
       4  5  6
    »  A(:,3)
ans =
       3
       6
       9
    »  A(1:2,3)
ans =
       3
       6
```

Note que **A(:,:)** retorna a matriz original, e que **A(:)** retorna a matriz em formato vetor:

```
» A(:)
 ans =
        1
        4
        7
        2
        5
        8
        3
        6
        9
```

Usando slicing, podemos facilmente apagar elementos de matrizes (ou linhas/colunas inteiras) usando a seguinte sintaxe:

```
»  A(i,:)  = []
»  A(:,j) = []
```

Podemos também acrescentar elementos a qualquer momento:

```
»  lista = [1,3,4,5]
»  lista = [lista 2]
»  lista
```

Observação Como a sintaxe acima também se aplica a texto, podemos concatenar textos facilmente e usar slicing:

```
»  frase = 'Oi, como vai?'
»  frase = [frase(1:length(frase)-1) ' você?']
»  frase(1:2)
```

1.1.6 Controle de fluxo

Para repetirmos um bloco de código várias vezes, usamos a estrutura **for**:

```
for  variável = início:fim
     comando
end
```

Se quisermos determinar um passo diferente de 1 entre os valores *início* e *fim*, usamos a sintaxe

```
for  variável = início:passo:fim
     comando
end
```

Observe que estruturas de controle de fluxo podem ser escritas diretamente no console do MATLAB.

Exemplo:

```
  »  for i = 1:3
       i
     end
i =
     1
i =
     2
i =
     3
  »  for i = 3:-1:1
       i
     end
i =
     3
i =
     2
i =
     1
```

Quando é necessário repetir certo comando de código várias vezes *até que* uma certa condição lógica seja satisfeita, usamos a estrutura **`while`**:

```
»  i = 1;
»  while i < 3
     disp('Mais um.')
     i = i + 1;
   end
```

Para executarmos um bloco de comandos apenas se uma determinada condição lógica for satisfeita, usamos a estrutura **`if`**:

```
  if (sentença lógica)
    faça (1)
  else
    faça (2)
  end
```

Em Matlab, uma sentença lógica pode ter dois valores: **`0`** (Falso) ou **`1`** (Verdadeiro).

Observação Para atribuir um valor a uma variável usamos o símbolo **=**; para compararmos se dois valores são iguais, usamos o símbolo **==**.

Exemplo:

```
»  i = 2;
»  if i == 1
     disp('Um.')
   else
     disp('Não é um...')
   end
   Não é um...
»  i = 1;
»  if i == 1
     disp('Um.')
   else
     disp('Não é um...')
   end
   Um.
```

1.2 Estruturas de dados heterogêneas: Células

Muitas vezes, gostaríamos de armazenar dados *heterogêneos*; por exemplo, poderíamos desejar armazenar uma tabela do tipo

Título	Núm. Páginas	Datas de Empréstimo e Devolução
"Álgebra Linear"	205	12/08, 15/08
"Cálculo"	346	10/09, 12/09
"Geometria"	123	04/08, 05/09
"Topologia"	253	01/08, 04/09

Porém, esses dados são de naturezas diferentes: misturamos texto (*string*), números e intervalos. Como armazenar isso em uma só tabela no MATLAB?

A estrutura de dados que possibilita esse armazenamento é chamada *célula* (**cell**), e pode ser utilizada da seguinte maneira:

```
»  tabela = { 'Algebra Linear', 205, [1208, 1508];
              'Calculo', 346, [1009, 1209];
              'Geometria', 123, [0408, 0509];
              'Topologia', 253, [0108, 0409] }
```

As células no MATLAB funcionam como matrizes, mas os índices são dados sempre entre chaves. A diferença entre células e matrizes é que em uma matriz, todas as entradas devem ter o mesmo tipo de dados; em uma célula, podemos misturar tipos diferentes de dados.

Como as células podem conter uma variedade de tipos de dados, nem sempre conseguimos acessar os valores diretamente pelo console. Por exemplo,

```
       »  tabela
tabela =
           'Algebra Linear'  [205]  [1x2 double]
           'Calculo'         [346]  [1x2 double]
           'Geometria'       [123]  [1x2 double]
           'Topologia'       [253]  [1x2 double]
```

Assim, para vermos exatamente o que está armazenado na variável **tabela**, basta usarmos o comando

```
» celldisp(tabela)
```

Outros comandos válidos:

- Para verificar o tamanho de uma célula, usamos o comando

```
» size(tabela)
```

- Para criar uma célula vazia com *m* por *n* elementos, usamos o comando

```
» tabela = cell(m,n)
```

- É possível transpor uma célula usando a mesma notação que usamos para matrizes:

```
» tabela'
» transpose(tabela)
```

1.2.1 Acessando dados dentro de uma célula

Existem duas maneiras de acessar elementos dentro de uma célula:

- Se usamos índices entre parênteses, estamos acessando um subconjunto da célula original.
- Se usamos chaves (**{}**), estamos acessando os valores no interior de cada elemento da célula.

Exemplo:

```
       » sub = tabela(1:2,1:2)
   sub =
          'Algebra Linear'  [205]
          'Calculo'         [346]
       » tabela(2,:)  = {'MATLAB', 300, [1201, 1401]};
       » tabela
tabela =
          'Algebra Linear'  [205]  [1x2 double]
          'MATLAB'          [300]  [1x2 double]
          'Geometria'       [123]  [1x2 double]
          'Topologia'       [253]  [1x2 double]
```

Note que mesmo as células que contêm valores numéricos não estão armazenadas como números. Repare nos colchetes:

```
      » vetor = tabela(:,2)
vetor =
          [205]
          [300]
          [123]
          [253]
      » 3*vetor
         Undefined function 'mtimes' for input
          arguments of type 'cell'.
```

Podemos facilmente converter esses dados para uma variável numérica usando o comano **cell2mat**:

```
      »  vetor = cell2mat(tabela(:,2))
vetor =
          205
          300
          123
          253
      »  3*vetor
  ans =
          615
          900
          369
          759
```

Para acessarmos o conteúdo de uma célula individual, usamos as chaves. Por exemplo, na nossa tabela,

```
»  tabela{1,1}
```

é um texto com valor **'Algebra Linear'**, enquanto que

```
»  tabela{1,2}
```

é um número de valor 205.

O resultado de um acesso simultâneo a várias células, por exemplo usando slicing, é uma *lista* de valores:

```
    »  tabela{1:3,2}
ans =
       205
ans =
       300
ans =
       123
```

Note que se fizermos

```
»  teste = tabela{1:3,2}
```

a variável **teste** conterá apenas o primeiro resultado da operação!

Para armazenarmos todo o resultado do acesso a estes valores da célula, podemos associar o resultado a uma lista com o mesmo número de elementos que o número de resultados do acesso:

```
  »  [a,b,c] = tabela{1:3,2}
a =
     205
b =
     300
c =
     123
```

Se todos os elementos selecionados da célula possuirem o mesmo tipo de dados, podemos atribuir esses elementos selecionados a uma só variável. Por exemplo, neste caso todos os dados selecionados são números, e assim:

```
           »  numpaginas = [tabela{1:3,2}]
numpaginas =
              205   346   123
```

No nosso exemplo, também temos variáveis de texto dentro da primeira coluna da célula. Podemos acessar um subconjunto do texto contido em uma das células associando os respectivos índices. Por exemplo:

```
    »  tabela{1,1}(1:3)
ans =
       'Alg'
```

Podemos construir estruturas de dados multidimensionais e heterogêneas, inclusive incluindo uma célula dentro de outra.

Exemplo:

```
»  v = { { 1, 'teste', [1;2] };
         { [0,3], 12, 'nome', rand(4,4) } }
```

Neste caso, os elementos devem ser referenciados da seguinte forma:

```
»  v{2}{1}
```

Podemos, analogamente ao que fizemos com vetores, concatenar células:

```
»  C1 = {'Joao', 16};
»  C2 = {'Maria', 18; 'Ricardo', 13};
»  cola = {C1 C2}
»  uniao = [C1; C2]
```

1.3 Estruturas de dados heterogêneas: *Structs*

Outra maneira de armazenar dados heterogêneos é usar *structs*: cada struct é composta de campos que podem conter quaisquer tipos de dados (assim como as células), e que são referenciados por *nome*. Para criarmos uma struct chamada **dados** com o campo chamado **Nome**, podemos usar diretamente a sintaxe

```
»  dados.Nome = 'Melissa'
»  dados.Sobrenome = 'Mendonça'
```

ou

```
»  dados = struct('Nome', 'Melissa', ...
      'Sobrenome', 'Mendonça')
```

Nestes casos, a struct **dados** conterá um *campo* chamado **Nome** e outro chamado **Sobrenome**; podemos armazenar em cada elemento da struct **dados** valores para estes campos; assim, isso significa que

```
»  dados(1).Nome
```

retorna o valor **'Melissa'**. Observe os exemplos:

```
      »  dados = struct('Nome', 'Melissa', ...
            'Sobrenome', 'Mendonça')
      »  dados(1)
  ans =
               Nome:  'Melissa'
          Sobrenome:  'Mendonça'
      »  dados(1).Nome
  ans =
         Melissa
      »  dados(2) = struct('Nome', 'Fulano', ...
            'Sobrenome', 'Beltrano')
dados =
         1x2 struct array with fields:
           Nome
           Sobrenome
      »  dados(1)
  ans =
               Nome:  'Melissa'
          Sobrenome:  'Mendonça'
      »  dados(2)
  ans =
               Nome:  'Fulano'
          Sobrenome:  'Beltrano'
      »  dados.Nome
  ans =
         Melissa
  ans =
         Fulano
      »  [nome1, nome2] = dados.Nome
nome1 =
         Melissa
nome2 =
         Fulano
```

Para criarmos uma struct vazia, podemos usar o comando

```
»  vazia = struct([])
```

1.3.1 Como lidar com os nomes dos campos

As structs possuem campos nomeados, o que pode tornar mais fácil acessar os dados armazenados nesse tipo de variável. Alguns comandos do MATLAB permitem fazer isso.

- O comando **fieldnames(s)** permite recuperar em uma célula a lista dos nomes dos campos da struct **s**.
- O comando **s = orderfields(s1)** ordena os campos da struct **s1** de modo que a nova struct **s** tem os campos em ordem alfabética.

- O comando **`s = orderfields(s1, s2)`** ordena os campos da struct **s1** de forma que a nova struct **s** tenha os nomes dos campos na mesma ordem em que aparecem na struct **s2** (as structs **s1** e **s2** devem ter os mesmos campos).
- O comando **`s = orderfields(s1, c)`** ordena os campos em **s1** de forma que a nova struct **s** tenha campos na mesma ordem em que aparecem na célula **c** (a célula **c** deve conter apenas os nomes dos campos de **s1**, em qualquer ordem).

1.3.2 Structs e células

Podemos preencher uma struct usando um comando só, determinando os valores possíveis para cada campo através de células. Por exemplo, no caso anterior, poderíamos ter entrado o comando

```
»  dados = struct('Nome', {'Melissa', 'Fulano'}, ...
      'Sobrenome', {'Mendonça', 'Beltrano'})
```

para criar a mesma struct.

Se quisermos preencher vários campos com o mesmo valor, não precisamos nos repetir. Por exemplo,

```
»  dados = struct('Nome', {'Melissa', 'Fulano'}, ...
      'Sobrenome', {'Mendonça', 'Beltrano'}, ...
      'Presentes', {'sim'})
```

gera uma struct com os dados

```
  dados(1) =
        Nome:  'Melissa'
   Sobrenome:  'Mendonça'
   Presentes:  'sim'
  dados(2) =
        Nome:  'Fulano'
   Sobrenome:  'Beltrano'
   Presentes:  'sim'
```

1.3.3 Convertendo dados

Existem duas funções do MATLAB que nos permitem converter dados de células para structs ou de structs para células.

O comando

```
»  s = cell2struct(célula, campos, dimensão)
```

cria uma struct **s** a partir de uma célula **c**, de forma que os dados dispostos ao variarmos a dimensão escolhida possam ser distribuídos nos campos correspondentes informados no comando.

Exemplo:

```
      »  tabela = {'Melissa', 'Mendonça', 'sim';
            'Fulano', 'Beltrano', 'sim'}
tabela =
           'Melissa'  'Mendonca'  'sim'
           'Fulano'   'Beltrano'  'sim'
      »  campos = {'Nome', 'Sobrenome', 'Presente'};
      »  s = cell2struct(tabela, campos, 2);
    s =
           2x1 struct array with fields:
            Nome
            Sobrenome
            Presente
      »  s(1)
   ans =
                Nome:  'Melissa'
            Sobrenome:  'Mendonça'
             Presente:  'sim'
      »  s(2)
   ans =
                Nome:  'Fulano'
            Sobrenome:  'Beltrano'
             Presente:  'sim'
```

Por outro lado, o comando

```
»  célula = struct2cell(struct)
```

cria uma célula a partir da estrutura *struct*.

Exemplo:

```
      »  celula = struct2cell(s)
celula =
           'Melissa'   'Fulano'
           'Mendonça'  'Beltrano'
           'sim'       'sim'
```

1.4 Funções

No MATLAB, uma função é um arquivo **`minhafuncao.m`** com a sintaxe

minhafuncao.m

```
function  [y] = minhafuncao(x)
          % Descricao da funcao
          comandos;
```

Uma vez construida a função, podemos chamá-la no console, usando

```
»  y = minhafuncao(x)
```

Observação. Uma função deve sempre ter o mesmo nome que o arquivo no qual ela está salva.

Exemplo: Construir uma função que calcule o módulo de um número real x.

modulo.m

```
function  [y] = modulo(x)
          if x < 0
            y = -x;
          else
            y = x;
          end
```

Para retornar mais de um argumento, ou para que a função tenha mais de um argumento de entrada, basta separar os valores por vírgulas.

Exemplo: O comando **size** é uma função que recebe como argumento de entrada uma variável e retorna um vetor com as dimensões desta variável:

```
»  [m,n] = size(A)
```

Se quisermos aplicar a mesma função a um conjunto de valores, basta colocarmos os valores em um vetor:

```
»  m = f([-2 1 3])
```

Por fim, se uma função está definida com vários argumentos de saída mas não precisamos de todos estes argumentos, podemos usar o símbolo ~.

Exemplo:

```
»  [~,n] = size(rand(10,5));
```

1.4.1 Funções com número variável de argumentos

O MATLAB possui alguns comandos que permitem verificarmos a quantidade de argumentos de entrada e de saída de uma função:

- O comando **nargin**, executado dentro do corpo de uma função, retorna o número de argumentos de entrada para o qual a função está definida.
- O comando **nargin(f)**, em que **f** é uma função, retorna o número de argumentos de entrada da função **f**, e pode ser executado fora da função (inclusive no console).
- O comando **nargout**, executado dentro do corpo de uma função, retorna o número de argumentos de saída para o qual a função está definida.
- O comando **nargin(f)**, em que **f** é uma função, retorna o número de argumentos de saída da função **f**, e pode ser executado fora da função (inclusive no console).

Além disso, podemos definir funções com número variável de argumentos de entrada ou de saída, usando respectivamente os comandos **varargin** e **varargout**.

O comando **varargin** é uma variável de entrada utilizada na definição de uma função que permite que a função receba qualquer número de argumentos de entrada. Quando a função for executada, **varargin** (que deve ser o último argumento de entrada) será uma célula de tamanho 1 por *n*, onde *n* é o número de argumentos de entrada que a função recebeu além daqueles explicitamente declarados na função.

Exemplo:

somas.m

```
function  [y] = somas(x,varargin)
          if nargin == 1
            disp('Nada a calcular.')
          elseif nargin == 2
            y = x + varargin{1};
          elseif nargin == 3
            y = x + varargin{1} + varargin{2};
          else
            disp('Argumentos demais!')
          end
```

O comando **varargout** é uma variável de saída utilizada na definição de uma função que permite que a função devolva qualquer número de argumentos de saída. Analogamente ao comando **varargin**, o comando **varargout** deve aparecer como o último argumento de saída na declaração da função, e é uma célula de tamanho 1 por *n*, em que *n* é o número de argumentos de saída requisitados na execução da função além daqueles explicitamente declarados na função.

Exemplo:

valores.m

```
function [f,varargout] = valores(x)
   f = x^2;
   if nargout == 2
     varargout{1} = 2*x;
   elseif nargout == 3
     varargout{1} = 2*x;
     varargout{2} = 2;
   elseif nargout > 3
     disp(['Esta funcao calcula apenas f, derivada ' ...
             'de f e segunda derivada de f.'])
   end
```

1.4.2 Funções anônimas

Suponha que queremos declarar uma função no console, sem ter que guardá-la em um arquivo. Por exemplo, queremos avaliar a função $f(x) = \sin x$ em alguns pontos. Para isto, podemos usar o conceito de *função anônima*.

Para definirmos uma *função anônima*, usamos a seguinte sintaxe:

```
»  f = @(x) sin(x)
```

e, em seguida, podemos avaliar a função, usando o comando

```
»  x = pi;
»  f(pi)
```

Esta definição é bastante útil quando queremos passar uma função como argumento para outra função: por exemplo, para calcularmos o valor mínimo de uma função avaliada em 3 pontos distintos, podemos definir a função como uma função anônima, e em seguida passar o seu *handle* como argumento para a função que calcula o mínimo entre os valores:

```
»  f = @(x) x-1;
»  min(f([-2 1 0]))
```

Se quisermos criar uma função anônima com mais de uma variável, usamos

```
»  f = @(x,y,z,t) x+y+z+t
```

Para retornarmos mais de um valor de uma função anônima, usamos o comando *deal*:

```
»  f = @(t,u,v) deal(t+u+v,t-u+2*v)
»  [a,b] = f(1,2,3)
```

CAPÍTULO 2

Gráficos em 2D e 3D

2.1 Gráficos em 2D

Cada ponto no gráfico é dado por uma coordenada (x, y), onde x é um número real e y é um número real associado a x (como $y = f(x)$). Mas, não podemos representar a reta real (*contínua*) no MATLAB. Por isso, precisamos definir um *vetor* de pontos em um determinado intervalo (em $\mathbb{R}$ ou $\mathbb{R}^2$):

$$x = (x_1, x_2, \ldots, x_n)$$

e calcular o valor de f apenas nestes pontos; o MATLAB ligará os pontos para desenhar o gráfico.

2.1.1 Comando **`plot`**

Para fazer gráficos simples em 2D (no plano), inicialmente precisamos definir o intervalo a ser utilizado para desenhar o gráfico. Para isso, precisamos de 3 argumentos:

```
» x = a:delta:b
```

Aqui, **a** é a extremidade esquerda do intervalo; **b** é a extremidade direita do intervalo; e **`delta`** é o espaçamento desejado entre cada subintervalo, e determina assim o número total de pontos neste intervalo. O resultado, **x**, é um *vetor* de pontos (que pode ser um vetor linha ou um vetor coluna).

Observação Analogamente, podemos usar o comando **`linspace`** para termos controle direto sobre o número de pontos desejados no intervalo:

```
» y = linspace(a,b,n)
```

gera um vetor de pontos y com extremidade esquerda **a**, extremidade direita **b** e **n** pontos no total.

Em seguida, chamamos o comando

```
» plot(x,y)
```

onde **x** é o vetor de pontos no intervalo desejado, e **y** é um vetor dos valores da função a ser desenhada nos pontos do intervalo.

Exemplo: Para fazermos o gráfico da função $f(x) = x^2$ no intervalo $[-4, 4]$, usamos os seguintes comandos:

```
» x = -4:0.5:4
» y = x.^2
» plot(x,y)
```

Desta forma,

$$x = (-4, -3.5, -3, -2.5, -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4)$$

$$y = (16, 12.25, 9, 6.25, 4, 2.25, 1, 0.25, 0, 0.25, 1, 2.25, 4, 6.25, 9, 12.25, 16)$$

Algumas opções do comando `plot` Outras opções, como espessura da linha ou cor do desenho, podem ser especificadas no comando plot.

Exemplo:

```
» plot(x,y,'r*')
» plot(x,y,'m^')
```

A lista de opções pra o comando **`plot`** pode ser obtida no MATLAB ao digitarmos **`help plot`**. Um resumo se encontra na Tabelas 2.1 e 2.2.

Símbolos	Cores
'k'	Preto
'r'	Vermelho
'g'	Verde
'b'	Azul
'm'	Rosa
'c'	Azul claro
'y'	Amarelo
'w'	Branco

Tabela 2.1: Cores.

Observação Note que se usarmos o comando **`plot`** sem especificar o estilo de linha, ou se usarmos o estilo padrão (**'-'**), o resultado será uma curva, enquanto que se utilizarmos as outras opções, teremos um desenho do conjunto dos pontos x e y desejados. Assim, podemos desenhar também conjuntos discretos sem maiores complicações.

Para modificar a espessura da linha, usamos a propriedade **'Linewidth'**:

```
» plot(x,y,'Linewidth',2)
```

Para mais opções e propriedades dos gráficos, consulte o *help* do MATLAB.

Observação Todos os comandos modificadores do gráfico que veremos adiante devem ser executados *após* a criação do gráfico. Portanto, o primeiro passo para se fazer uma figura é sempre criar o gráfico básico (eixo), para depois acrescentar títulos, descrições, etc.

Observação Para fecharmos a janela de gráficos atual, usamos o comando **`close`**.

Símbolo	Estilo de linha
'-'	Linha contínua
'-'	Linha tracejada
':'	Pontilhado
'-.'	Traço e ponto
'.'	Pontos
'+'	Cruzes
'*'	Asteriscos
'o'	Círculos
'x'	X
'^'	Triângulos
'v'	Triângulos
'>'	Triângulos
'<'	Triângulos
's'	Quadrados
'd'	Losangos
'p'	Pentágonos
'h'	Hexágonos

Tabela 2.2: Estilos de linha.

Vários gráficos no mesmo eixo

Para fazer mais de um gráfico no mesmo eixo, usamos os comandos **`hold`**. A utilização básica é a seguinte:

```
» plot(x,y)
» hold on
» plot(x,z)
» hold off
```

O comando **`hold on`** sinaliza ao MATLAB que ainda vamos fazer mais gráficos no mesmo eixo, enquanto que o comando **`hold off`** desliga essa opção. Portanto, qualquer gráfico feito após o comando **`hold off`** vai apagar o gráfico atual e substitui-lo em um novo eixo.

Exemplo: Representar graficamente o sistema linear abaixo:

$$\begin{cases} x_1 + x_2 = 8 \\ 4x_1 + 2x_2 = 26 \end{cases} \Leftrightarrow \begin{cases} x_2 = 8 - x_1 \\ x_2 = 13 - 2x_1 \end{cases} \Leftrightarrow \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 3 \end{pmatrix}$$

```
» x1 = 0:0.1:10
» plot(x1,8-x1)
» hold on
» plot(x1,13-2*x1)
» hold off
```

Título: `title`

Para incluirmos um título em um gráfico produzido no MATLAB, basta usarmos o comando **`title`** *após* a criação do gráfico:

```
» title('y=f(x)')
```

Exemplo:

```
» x = -pi:0.1:pi
» plot(x,sin(x),'r')
» title('Gráfico da função y=sin(x)')
```

Legendas nos eixos: `label`

Para incluirmos legendas nos eixos do gráfico, usamos os comandos **`xlabel`** e **`ylabel`** para os eixos horizontal e vertical, respectivamente. Assim como o comando **`title`**, estes comandos também devem ser utilizados *após* a criação do gráfico.

Exemplo:

```
» t = 0:0.1:10
» p = exp(t)
» plot(t,p)
» title('Gráfico da pressão em função do tempo.')
» xlabel('tempo')
» ylabel('pressão')
```

Legendas dos gráficos: `legend`

Para incluirmos nos gráficos legendas sobre cada linha que está sendo desenhada, usamos o comando **`legend`**, da seguinte forma:

```
» x = -pi:0.1:pi
» plot(x,sin(x))
» title('Gráfico 1')
» legend('y=sin(x)')
```

Se temos mais de um gráfico no mesmo eixo, usamos o comando **`legend`** com os textos desejados nas legendas, *na ordem em que os gráficos foram desenhados*:

```
» legend('legenda 1','legenda 2','legenda 3')
```

Exemplo:

```
» x = -2*pi:0.1:2*pi;
» plot(x,sin(x))
» hold on
» plot(x,cos(x),'r')
» legend('y=sin(x)','y=cos(x)')
» title('Funções seno e cosseno.')
```

Eixos coordenados: `axis`

Às vezes, precisamos fixar ou modificar os eixos coordenados nos quais fazemos os gráficos. Para isso, podemos usar o comando **`axis`** de várias formas.

- **`axis`**: cria um "eixo coordenado" padrão.
- **`axis([x0 x1 y0 y1])`**: modifica os eixos coordenados da figura atual de maneira que o eixo horizontal começa em $x0$ e termina em $x1$, e o eixo vertical começa em $y0$ e termina em $y1$.

- **`axis auto`**: deixa o MATLAB tentar encontrar um eixo coordenado ideal para a imagem desenhada.
- **`axis equal`**: toma eixos coordenados na mesma escala.
- **`axis tight`**: toma eixos coordenados "justos" aos dados.
- **`axis off`**: não mostrar o eixo coordenado (**`axis on`** reverte esta ação)

Malha: `grid`

Podemos mostrar uma malha no nosso gráfico usando o comando **`grid on`**. Para revertermos este comando, basta usar **`grid off`**. Além disso, podemos também escolher uma malha mais refinada usando **`grid minor`**.

Exemplo:

```
» t = 0:pi/20:2*pi
» plot(sin(t),2*cos(t))
» grid on
» grid minor
```

2.1.2 Retas e segmentos: `line`

Para desenharmos retas e segmentos de reta no MATLAB, podemos fazer o gráfico da reta usando sua equação na forma $y = ax+b$, no comando **`plot`**, ou podemos usar o comando **`line`**. Este comando toma como entrada as coordenadas dos pontos $p = (x0, y0)$ e $q = (x1, y1)$ que desejamos ligar pelo segmento de reta a ser desenhado:

```
» line([x0 x1],[y0 y1])
```

Exemplo: Para desenharmos o segmento que vai do ponto $p = (1, 2)$ até o ponto $q = (3, 4)$, usamos o comando

```
» line([1 3],[2 4])
```

Se nos argumentos tivermos mais de dois pontos, o MATLAB gerará um conjunto de segmentos (ou um polígono) ligando os pontos em ordem.

Exemplo: Para desenharmos um triângulo ligando os pontos $(0, 0)$, $(1, 0)$ e $(0, 1)$ no MATLAB, usamos o seguinte comando:

```
» line([0 1 0 0],[0 0 1 0])
» axis([-1 2 -1 2])
```

(acrescentamos o comando **`axis`** no exemplo acima para podermos visualizar corretamente o triângulo.)

2.1.3 Regiões preenchidas: `fill`

Para desenharmos regiões preenchidas no MATLAB, usamos o comando **`fill(X,Y,C)`**, onde **`X`** e **`Y`** são um conjunto de coordenadas para os pontos $p1 = (X1, Y1)$, $p2 = (X2, Y2)$, etc e **`C`** é a cor desejada para o preenchimento (mesmas cores usadas no comando **`plot`**).

Exemplo:

```
» X = [0 1 0 0]
» Y = [0 0 1 0]
» fill(X,Y,'m')
```

2.1.4 Subgráficos: **subplot**

Se quisermos fazer vários gráficos na mesma janela, podemos usar o comando **subplot**, da seguinte forma: » **subplot(m,n,i), plot(x,y)**
desenha o gráfico determinado pelas coordenadas **x** e **y** na posição *i* de uma matriz de gráficos com *m* linhas e *n* colunas. O índice dos gráficos é determinado da seguinte maneira: se temos, por exemplo, 8 gráficos organizados em uma matriz com 2 linhas e 4 colunas, os gráficos serão desenhados na ordem abaixo:

```
1  2  3  4
5  6  7  8
```

Exemplo: Fazer em uma mesma janela os gráficos das funções seno, cosseno, tangente e cotangente entre -2π e 2π.

```
» x = -2*pi:0.01:2*pi;
» subplot(2,2,1), plot(x,sin(x),'r')
» subplot(2,2,2), plot(x,cos(x),'c')
» subplot(2,2,3), plot(x,tan(x),'g')
» subplot(2,2,4), plot(x,cot(x))
```

2.1.5 Curvas no espaço: **plot3**

Se quisermos desenhar um conjunto de pontos ou uma curva parametrizada no espaço, podemos usar o comando **plot3**. O comando

```
» plot3(X,Y,Z)
```

desenha a curva determinada pelos pontos (X, Y, Z) no espaço.
Exemplo:

```
» t = 0:pi/50:10*pi;
» plot3(sin(t),cos(t),t)
» xlabel('sin(t)')
» ylabel('cos(t)')
» zlabel('t')
» axis square
```

Também podemos usar a função plot3 para observar a distribuição de pontos no espaço.
Exemplo:

```
» x = rand(100,1);
» y = rand(100,1);
» z = rand(100,1);
» plot3(x,y,z)
```

2.2 Gráficos em 3D

O procedimento de criação de gráficos em 3D no MATLAB é bastante parecido com o procedimento para a criação de gráficos no plano. A maior diferença é a geração dos intervalos de pontos a serem desenhados e a variedade de tipos de gráfico que podemos fazer.

Neste caso, queremos o gráfico de uma função de duas variáveis, ou seja, uma superfície dada por

$$z = f(x, y)$$

2.2.1 Criador de malha: **meshgrid**

Aqui, ao invés de criarmos um intervalo de pontos onde o gráfico será desenhado, precisamos de uma *malha*, ou seja, da intersecção de dois intervalos, um definido no eixo x e outro definido no eixo y, para desenharmos uma função $z = f(x, y)$. Para isto, usamos a função **meshgrid**.

```
»  [X,Y] = meshgrid(x,y)
```

cria matrizes **X** e **Y** prontas para serem usadas pelos comandos de criação de gráficos em 3D.

As matrizes resultantes da função **meshgrid** são construidas da seguinte maneira. Se x tem n elementos e y tem m elementos, as matrizes **X** e **Y** tem ordem $m \times n$, e contém os elementos dos vetores **x** e **y** repetidos. Assim, se **x = [x1,x2,x3]** e **y = [y1,y2,y3,y4]**, então

$$X = \begin{pmatrix} x1 & x2 & x3 \\ x1 & x2 & x3 \\ x1 & x2 & x3 \\ x1 & x2 & x3 \end{pmatrix}$$

e

$$Y = \begin{pmatrix} y1 & y1 & y1 \\ y2 & y2 & y2 \\ y3 & y3 & y3 \\ y4 & y4 & y4 \end{pmatrix}$$

Desta forma, o MATLAB lê os dados das duas matrizes em pares, resultando em um conjunto de pontos da forma

$$\begin{pmatrix} (x1, y1) & (x2, y1) & (x3, y1) \\ (x1, y2) & (x2, y2) & (x3, y2) \\ (x1, y3) & (x2, y3) & (x3, y3) \\ (x1, y4) & (x2, y4) & (x3, y4) \end{pmatrix}$$

e assim calcula todas as combinações dos pontos da malha.

Se queremos usar uma malha igualmente espaçada e definida no mesmo intervalo nos eixos x e y, podemos usar o comando **meshgrid** com somente um argumento de entrada:

```
»  x = -1:0.1:1
»  [X,Y] = meshgrid(x)
```

cria uma malha em $[-1, 1] \times [-1, 1]$ com espaçamento de 0.1 unidades entre os pontos.

2.2.2 Superfícies

Nesta seção, vamos ver os comandos que nos permitem desenhar superfícies no MATLAB.

Superfícies sólidas: `surf`

Para desenharmos uma superfície dada por

$$z = f(x, y)$$

com $(x, y) \in \Omega$, onde Ω é uma região do plano definida por $[x_0, y_0] \times [x_1, y_1]$, usamos o comando **`surf(X,Y,Z)`**.

```
»  x = x0:deltax:x1;
»  y = y0:deltay:y1;
»  [X,Y] = meshgrid(x,y);
»  Z = f(X,Y)
»  surf(X,Y,Z)
```

Exemplo:

```
»  x = -1:0.1:1;
»  [X,Y] = meshgrid(x);
»  Z = X.^2+Y.^2
»  surf(X,Y,Z)
```

Exemplo:

```
»  x = -3:0.1:3;
»  [x,y] = meshgrid(x);
»  f = @(x,y) x.^2+3*y-x.*y.^2
»  surf(x,y,f(x,y));
```

Superfícies vazadas: `mesh`

Para desenharmos a mesma superfície do caso anterior, mas com estilo vazado (como uma tela), usamos o comando **`mesh(X,Y,Z)`**.

Exemplo:

```
»  x = -1:0.1:1;
»  [X,Y] = meshgrid(x);
»  Z = X.^2+Y.^2
»  mesh(X,Y,Z)
```

meshz

O comando **`meshz(X,Y,Z)`** é análogo ao comando **`mesh`**, mas cria uma *cortina* em torno da superfície desenhada.

Exemplo:

```
»  x = -1:0.1:1;
»  [X,Y] = meshgrid(x);
»  Z = X.^2+Y.^2
»  meshz(X,Y,Z)
```

Superfícies Parametrizadas

Para fazermos o gráfico de superfícies parametrizadas, devemos garantir que os parâmetros discretizados tenham todos o mesmo tamanho. Por isso, neste caso, a indicação é usar o comando **`linspace(x1,x2,n)`**.

Exemplo: Fazer o gráfico da superfície parametrizada por (t, u) dada por

$$(x, y, z) = \left(\frac{t}{3}, u\cos t, u\sin t\right)$$

com $t \in [0, 10]$ e $u \in [-1, 1]$.

```
» t = linspace(0,10,40);
» u = linspace(-1,1,40);
» [t,u] = meshgrid(t,u);
» x = t/3;
» y = u.*cos(t);
» z = u.*sin(t);
» mesh(x,y,z)
```

2.2.3 Curvas de nível

Curvas de nível (contornos)

Para fazermos o gráficos de curvas de nível no MATLAB, podemos usar o comando **`contour`** para criar um gráfico plano com as curvas de nível desejadas, ou os comandos **`surfc`** ou **`meshc`** para desenhar as curvas de nível diretamente abaixo das superfícies.

Curvas de nível no plano: `contour`

Para desenharmos as curvas de nível de uma superfície dada por $Z = f(X, Y)$, onde X e Y foram criados com a função **`meshgrid`**, usamos o comando

```
» contour(X,Y,Z)
```

Se quisermos especificar quantas curvas serão desenhadas (no nosso caso, chamaremos esse número de n), usamos o comando **`contour`** na forma

```
» contour(X,Y,Z,n)
```

Observação Podemos omitir os argumentos **`X`** e **`Y`** e usar o comando **`contour(Z)`** ou **`contour(Z,n)`** se isso não criar ambiguidade no código.

Para especificar em quais pontos queremos mostrar as curvas de nível, usamos um vetor v com os valores em Z onde queremos ver as curvas, e o comando

```
» contour(Z,v)
```

Curvas de nível preenchidas no plano: `contourf`

Podemos criar curvas de nível preenchidas com um comando análogo ao comando **`contour`**,

```
» contourf(X,Y,Z,n)
```

ou

```
» contourf(X,Y,Z,v)
```

Exemplo:

```
» surf(peaks(100))
» contourf(peaks(100))
» colorbar
» contourf(peaks(100),[-6 0 6])
```

Curvas de nível no espaço: `contour3`

Podemos usar o comando

```
» contour3(X,Y,Z)
```

para desenhar as curvas de nível de uma superfície no espaço.

Também podemos usar as variantes **`contour3(X,Y,Z,n)`** e **`contour3(X,Y,Z,v)`**, análogas às correspondentes para o **`contour`**.

Exemplo:

```
» [X,Y] = meshgrid([-2:.25:2]);
» Z = X.*exp(-X.^2-Y.^2);
» contour3(X,Y,Z,30)
```

Superfícies com curvas de nível: `surfc` e `meshc`

Para fazermos um gráfico de uma superfície com suas curvas de nível desenhadas no plano definido por X e Y, usamos os comandos **`surfc`** ou **`meshc`**, dependendo do tipo de superfície desejada.

```
» surfc(X,Y,Z)
» meshc(X,Y,Z)
```

2.2.4 Opções

Assim como fizemos com os gráficos em 2D, algumas opções básicas podem ser escolhidas na hora de desenhar gráficos em 3D.

Cores: `colormap`

Aqui, não é mais possível escolhermos a cor da *linha* desenhada, já que temos uma malha ao invés de uma linha. Ainda assim, podemos selecionar a paleta de cores a ser utilizada no gráfico. Para isto, usamos o comando **`colormap`**:

```
» colormap('mapa')
```

Os mapas de cores disponíveis no MATLAB são:

jet	hsv	hot
cool	spring	summer
autumn	winter	gray
bone	copper	pink
lines		

Observação Podemos usar o comando **`colormap(mapa)`**, onde **`mapa`** é um mapa de cores. Um mapa de cores é uma matriz $m \times 3$ de números reais entre 0 e 1. Cada linha desta matriz é um vetor de valores RGB (vermelho, verde e azul) que define uma cor. A k-ésima linha do mapa de cores define a k-ésima cor, de forma que **`mapa(k,:) = [r(k) g(k) b(k)])`** especifica a intensidade de vermelho (*red*), verde (*green*) e azul (*blue*).

Exemplo:

```
» [x,y,z] = peaks;
» surf(x,y,z)
» colormap(rand(100,3))
```

Observação Podemos voltar ao mapa de cores original usando o comando

```
» colormap('default')
```

Cores: `shading`

Além do **`colormap`**, temos outra opção para mudar a aparência da superfície usando o comando **`shading`**:

```
» shading faceted
» shading flat
» shading interp
```

Seleção de eixos: `axis`

O comando **`axis`** continua funcionando para gráficos 3D; a diferença é que agora devemos entrar os valores para os 3 eixos coordenados, ou seja, devemos usar o formato

```
» axis([x0 x1 y0 y1 z0 z1])
```

2.2.5 Outros comandos

Esfera: `sphere`

Para desenharmos uma esfera de raio 1 centrada no ponto $(0, 0, 0)$ no MATLAB, usamos o comando

```
» sphere
```

Podemos ainda usar

```
» [X,Y,Z] = sphere
```

para prepararmos as matrizes de coordenadas **`X`**, **`Y`** e **`Z`** a serem usadas pelas funções **`surf`** ou **`mesh`** para desenharmos uma esfera posteriormente. Além disso, podemos usar o comando na forma

```
» [X,Y,Z] = sphere(n)
```

para especificar quantos pontos desejamos utilizar na hora de fazer o desenho.

Exemplo: O código

```
» [X,Y,Z] = sphere
» surf(X+3,Y-2,Z-1)
```

desenha uma esfera centrada no ponto $(3, -2, -1)$.

Cilindros: `cylinder`

Assim como no comando **`sphere`**, podemos desenhar um cilindro com o comando

```
» cylinder
```

Podemos gerar matrizes de coordenadas **`X`**,**`Y`** e **`Z`** para desenhar um cilindro posteriormente usando o comando

```
» [X,Y,Z] = cylinder
```

Podemos ainda especificar a quantidade de pontos utilizada neste desenho com o comando

```
» [X,Y,Z] = cylinder(n)
```

Além disso, podemos desenhar um sólido de revolução dado pela curva $c = f(t)$ com o comando

```
» [X,Y,Z] = cylinder(c)
```

Exemplo:

```
» t = 0:pi/10:2*pi;
» [X,Y,Z] = cylinder(2+cos(t));
» surf(X,Y,Z)
» axis square
```

`trimesh`, `trisurf`

Às vezes, é mais conveniente mostrarmos os gráficos usando uma *triangulação do domínio*. Para isto, usamos os comandos **`trimesh`** ou **`trisurf`** junto com o comando **`delaunay`**, que cria a triangulação do domínio:

Exemplo:

```
» [x,y] = meshgrid(-3:0.5:3);
» tri = delaunay(x,y)
» z = x.^2+3*y-x.*y.^2;
» trimesh(tri,x,y,z)
```

Volumes: `slice`

Quando temos um gráfico de volume, por exemplo, de um sólido preenchido, podemos visualizar seções bidimensionais deste volume usando o comando

```
» slice(x,y,z,v,xslice,yslice,zslice)
```

para visualizarmos o volume definido por

$$v = f(x, y, z)$$

na superfície definida por **`xslice`**, **`yslice`**, **`zslice`**.

Exemplo: Queremos visualizar o volume definido por

$$v = xe^{(-x^2-y^2-z^2)}$$

com $-2 \leq x \leq 2$, $-2 \leq y \leq 2$ e $-2 \leq z \leq 2$.

```
» [x,y,z] = meshgrid(-2:.2:2);
» v = x.*exp(-x.^2-y.^2-z.^2);
» xslice = [-1.2,.8,2];
» yslice = 0;
» zslice = [];
» slice(x,y,z,v,xslice,yslice,zslice)
```

2.2.6 Gráficos como objetos

É importante entendermos que no MATLAB, cada figura é um *objeto*, no sentido que tem *propriedades* que determinamos no momento do desenho. Por exemplo, uma figura gerada pelo comando

```
» surf(X,Y,Z)
```

tem diversas propriedades, como por exemplo o mapa de cores utilizado, o tamanho dos eixos, as legendas dos eixos, etc. Vamos ver rapidamente como podemos obter e modificar estas propriedades de uma figura criada pelo MATLAB.

Os comandos que utilizam esta filosofia são, em geral, úteis quando temos mais de uma janela de gráfico aberta e precisamos fazer modificações nestas janelas separadamente, por exemplo. Neste caso, precisamos de um identificador de cada janela. Isto é o que o MATLAB chama de *figure handle*.

Para criarmos um objeto de *janela gráfica*, usamos o comando

```
» figure
```

Podemos guardar a referência para esta figura se fizermos

```
» h = figure
```

Aqui, **`h`** é um valor numérico que pode ser usado posteriormente como um número de *identificação* da figura recém-criada.

Com relação ao gráfico em si, usamos o comando

```
» axes
```

para criar uma janela gráfica com um eixo desenhado; usamos o comando

```
» h = axes
```

para, similarmente, guardarmos na variável **h** a referência para o gráfico recém-criado. Podemos ainda fazer, por exemplo, diretamente

```
» h = plot(x,x.^2)
```

para guardarmos na variável **h** a referência para o gráfico de $f(x) = x^2$.

Obter a referência de um gráfico existente

Para obtermos a referência do último gráfico desenhado, podemos usar o comando

```
» fig = gcf
```

para obtermos a referência para a janela em que o último gráfico foi desenhado, ou ainda o comando

```
» h = gca
```

para obtermos a referência para o gráfico em si.

Obter/Atribuir propriedades dos gráficos

Para obtermos uma lista de *todas* as propriedades de um gráfico no MATLAB dado pela referência **h**, usamos o comando

```
» get(h)
```

Exemplo:

```
» t = -1:0.1:1
» plot(t,sin(t))
» get(gca)
```

Exemplo:

```
» x = -2*pi:0.1:2*pi
» [X,Y] = meshgrid(x)
» surf(X,Y,sin(X)+cos(Y))
» get(gca)
» get(gcf)
```

Podemos ainda obter o valor de uma propriedade específica do gráfico atual, passando este nome como argumento para a função **get**, escrevendo

```
» get(h,'Propriedade')
```

Exemplo:

```
» t = -1:0.1:1
» plot(t,sin(t))
» get(h,'type')
» get(h,'linestyle')
```

Para mais informações, procure as páginas *Figure Properties* e *Axes Properties* no *Help* do MATLAB.

2.2.7 Gráficos com funções anônimas

Suponha que queremos passar uma função como argumento para outra função: por exemplo, gostaríamos de usar um comando do tipo

```
» plot(x,f(x))
```

para a função $f(x) = x^2 + 1$.

Para isso, podemos definir uma *função anônima* usando a seguinte sintaxe:

```
» f = @(x) x.^2+1
```

e, em seguida

```
» x = -1:0.1:1
» plot(x,f(x))
```

Se quisermos criar uma função anônima com mais de uma variável, usamos

```
» f = @(x,y,z,t) x+y+z+t
```

Exemplo:

```
» x = -3:0.1:3;
» [x,y] = meshgrid(x);
» f = @(x,y) x.^2+3*y-x.*y.^2
» surf(x,y,f(x,y));
```

CAPÍTULO 3

Manipulação de arquivos e tratamento de dados

Há varias maneiras de se salvar e ler arquivos no MATLAB. O mais simples é usarmos arquivos com a extensão **.txt**, apesar de o MATLAB aceitar diversos tipos de arquivo.

3.1 Leitura ou Importação de dados

Para importarmos dados, o método mais fácil é utilizar a interface gráfica do MATLAB, selecionando

File → Import Data

Em seguida, a interface gráfica do MATLAB oferece várias opções, como selecionar células de arquivos de planilha eletrônica e transformá-las em variáveis diretamente, filtrar dados por tipo (texto ou numéricos, por exemplo), salvar os dados em matrizes ou células, e muitas outras opções.

(Para verificar os tipos de arquivo suportados e as funções disponíveis, consulte o Help.)

3.1.1 Leitura de arquivos de dados numéricos usando **load**

Para lermos um arquivo que contenha apenas dados numéricos, por exemplo chamado **dados.txt**, usamos o comando

```
» load dados.txt;
```

Em seguida, na variável **dados** estarão contidos os valores obtidos do arquivo **dados.txt**.

Se quisermos também podemos usar a sintaxe

```
» A = load('dados.txt')
```

Esse comando somente pode ser utilizado em arquivos com dados homogêneos e formatados.

3.1.2 Leitura de arquivos de dados formatados

Se tivermos um arquivo em que os dados estão

Para importarmos dados de maneira automática, podemos usar o comando

```
» importdata(arquivo, separador, ncabecalho)
```

para obtermos uma *struct* com o resultado da leitura do arquivo. Aqui, *separador* é um caractere que denota o separador de dados usado no arquivo (por exemplo, um espaço (' ') ou um ponto-e-vírgula (';')), e *ncabecalho* é um inteiro que contém o número de linhas de cabeçalho presentes no arquivo (por exemplo, nomes de variáveis e rótulos de colunas).

Se **importdata** reconhecer a extensão do arquivo, a função chama o comando apropriado do MATLAB para ler esse tipo de arquivo (por exemplo, **load** no caso de dados numéricos ou **xlsread** no caso de dados de planilhas eletrônicas). Caso contrário, **importdata** interpreta o arquivo como um arquivo de texto puro com delimitadores (formatado).

Por exemplo, se tivermos um arquivo do tipo

teste.txt

```
melissa joao maria
12;16;18
```

O resultado ao executarmos o comando **importdata** é:

```
      » dados = importdata('teste.txt', ';', 1)
dados =
               data:  [12 16 18]
           textdata:  'melissa joao maria'
         rowheaders:  'melissa joao maria'
```

3.1.3 Leitura de dados usando processamento de texto

Se tivermos dados que não estão bem organizados, ou que estão organizados de uma maneira diferente do que os outros comandos permitem ler, podemos processar o texto manualmente usando alguns comandos específicos. Para isso, precisamos seguir uma sequência de trabalho:

1. Abrir o arquivo para leitura;
2. Ler os dados do arquivo;
3. Fechar o arquivo.

Para abrir um arquivo chamado **nome.txt**, usamos o comando

```
» arquivo = fopen('nome.txt')
```

Para fecharmos este arquivo, usamos o comando

```
» fclose(arquivo)
```

Note que aqui *arquivo* é uma variável que contém um *apontador* para o arquivo que queríamos abrir; uma vez aberto, não precisamos mais nos referir ao nome do arquivo, mas sim ao apontador que associamos a ele.

3.1.4 Processamento de dados com **textscan** ou **fscanf**

Para lermos dados de um arquivo em uma célula, usamos

```
» C = textscan(arquivo,formato)
```

Para lermos dados de um arquivo em uma matriz, usamos

```
» A = fscanf(arquivo,formato)
```

Para lermos dados de um arquivo, precisamos indicar que tipo de informação estamos procurando. Isto é feito através dos *formatos* abaixo:

- Números inteiros: **%d** ou **%u**
- Números reais: **%f** (notação decimal) ou **%e** (notação científica)
- Texto com espaços: **%c**
- Texto sem espaços: **%s**
- Nova linha: **\n** (sinaliza o fim de uma linha de dados)

Exemplo:

1. Crie um arquivo chamado **info.txt** no mesmo diretório em que está salvando seus programas, com um número inteiro dentro.
2. No console, faça

```
» arquivo = fopen('info.txt')
» a = fscanf(arquivo,'%d')
» fclose(arquivo)
```

Verifique que a variável **a** vale o mesmo que seu inteiro no arquivo.

Se quisermos ler uma lista de números inteiros, por exemplo, devemos informar o *padrão* dos dados.

Exemplo: se no arquivo temos

arquivo.txt

```
1 2 3 4 5
```

precisamos usar o comando

```
v = fscanf(arquivo,'%d')
```

Assim, **v** será um vetor *coluna*.

Se quisermos especificar o tamanho da saída dos dados, devemos acrescentar um argumento à função **fscanf**. Por exemplo, se quisermos ler uma matriz 3×3 de um arquivo, fazemos:

```
» arquivo = fopen('matriz.txt');
» A = fscanf(arquivo,'%f',[3 3]);
» fclose(arquivo);
» A = A';
```

A verdadeira matriz eh a transposta da matriz que foi lida!

Se não sabemos o tamanho da matriz que está no arquivo, não podemos informar seu formato. Mas podemos contar quantos elementos foram lidos do arquivo:

```
» [A,contador] = fscanf(arquivo,'%d')
```

Suponha que temos no nosso arquivo também o nome do campo de dados:

temperaturas.txt

```
Hora   1,  Temperatura  20.6
Hora   3,  Temperatura  21.2
Hora   5,  Temperatura  23.1
Hora   6,  Temperatura  24.5
Hora   8,  Temperatura  25.0
Hora   9,  Temperatura  25.2
Hora  10,  Temperatura  25.8
```

Para ler apenas os números desta tabela, usamos o comando

```
» A = fscanf(arquivo,'Hora %d, Temperatura %f\n',[2 7])
```

Observação Para não ler o texto, e ler apenas as idades, podemos *pular* o campo de texto com o comando

```
» idades = fscanf(arquivo,'%*s %d\n')
```

Para ler arquivos de planilha gerados pelo Microsoft Excel, usamos

```
» [dados,texto,resto] = xlsread(arquivo)
```

Para ler os dados de uma planilha específica do arquivo, usamos

```
» [dados,texto,resto] = xlsread(arquivo,planilha)
```

Em sistemas com o Microsoft Excel instalado, pode-se usar

```
» [dados,texto,resto] = xlsread(arquivo,-1)
```

para abrir uma janela do Excel e selecionar os dados a serem importados interativamente.

3.2 Escrita

3.2.1 Escrita em arquivos usando **save**

Para salvarmos alguma variável em um arquivo, podemos usar o comando

```
» save('arquivo.txt','variavel')
```

Porém, este comando salva o arquivo no formato MAT, que é um formato próprio do MATLAB, ilegível para humanos. Assim, para salvarmos em um arquivo texto simples, acrescentamos a opção **'-ascii'**.

Exemplo:

```
» dados = rand(3,4);
» save('dadosout.txt','dados','-ascii')
```

3.2.2 Escrita em arquivos usando **fprintf**

Para escrever dados em um arquivo, a sintaxe é similar ao que tínhamos para a leitura, mas devemos avisar ao MATLAB que vamos escrever neste arquivo, acrescentando a opção **'w'** ao final do comando **fopen**:

```
» arquivo = fopen('info.txt','w')
» fprintf(arquivo,'%d',1)
» fclose(arquivo)
```

Aqui, a opção **'w'** cria um arquivo novo, vazio, em que serão escritos os dados; se o arquivo já existir, os dados anteriores são apagados e os novos dados são escritos dentro do arquivo.

As outras opções de acesso a arquivos com o comando **fopen** são **'r'** (Somente leitura) ou **'a'** (acrescentar texto ao final do arquivo existente).

CAPÍTULO 4

Métodos para Análise Estatística

Agora, vamos considerar o caso em que temos um conjunto de dados e que desejamos aplicar algum método de análise estatística a estes dados. Vamos analisar alguns métodos disponíveis no MATLAB que *não* necessitam da *Statistics Toolbox*. Para os métodos que se encontram nesta toolbox, recomendamos consultar a documentação específica.

4.1 Funções básicas

Podemos calcular o valor da média aritmética simples para um conjunto de números armazenados em um vetor **x** usando o comando

```
» mean(x)
```

Para calcularmos a média aritmética simples *de cada coluna* de uma matriz e armazenarmos essas médias em um vetor linha, podemos usar o comando

```
» mean(matriz)
```

Para calcularmos a média aritmética simples *de cada linha* de uma matriz e armazenarmos o resultado em uma matriz coluna, usamos o comando

```
» mean(matriz,2)
```

Observação Se você tiver a *Statistics Toolbox* à mão, pode usar o comando **`trimmmean`** para calcular a média excluindo os k maiores e menores valores de um vetor **`X`**.

Para calcularmos a mediana de um conjunto de dados armazenados em um vetor, usamos o comando

```
» median(x)
```

Para calcularmos a mediana das colunas de uma matriz, e retornar as medianas em um vetor linha, usamos o comando

```
» median(matriz)
```

Existem duas definições para o desvio padrão:

$$s = \left(\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2 \right)^{\frac{1}{2}} \tag{4.1}$$

ou

$$s = \left(\frac{1}{n} \sum_{i=1}^{n} (x_i - \overline{x})^2 \right)^{\frac{1}{2}} \tag{4.2}$$

onde

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

Para calcularmos o desvio padrão usando a fórmula (4.1), usamos o comando

```
» std(x)
```

O resultado é a raiz quadrada da variância.

Se quisermos calcular um vetor linha contendo o desvio padrão calculado para cada coluna de uma matriz, usamos

```
» std(matriz)
```

Se quisermos calcular o desvio padrão dos elementos de um vetor usando a fórmula (4.2), usamos

```
» std(x,1)
```

Para calcularmos a variância dos elementos de um vetor, usamos o comando

```
» var(x)
```

Para calcularmos um vetor linha com as variâncias de cada coluna da matriz, usamos o comando

```
» var(matriz)
```

O comando **var** normaliza os dados por $n - 1$, se temos $n > 1$ dados. Se desejamos normalizar por n, usamos o comando

```
» var(x,1)
```

Para calcularmos a matriz de covariância entre 2 variáveis de dados, usamos o comando

```
» cov(X)
```

Podemos ainda obter outras informações desta matriz:

```
» diag(cov(X))
```

é o vetor de variâncias para cada coluna de dados (idem a **var**)

```
» sqrt(diag(cov(X)))
```

é desvio padrão (idem a **std**).

Aqui, **X** pode ser um vetor ou uma matriz. Para uma matriz $m \times n$, a matriz de covariância é $n \times n$.

Se tivermos uma matriz em que cada coluna contém observações de uma variável, podemos calcular os coeficientes de correlação entre as variáveis desta matriz usando o comando

```
» R = corrcoef(X)
```

Os coeficientes vão de -1 (correlação negativa) até 1 (correlação positiva). Valores próximos de 0 indicam que não há correlação linear entre as variáveis.

Se também quisermos saber o *p-value* de cada correlação, usamos o comando

```
» [R, P] = corrcoef(X)
```

Exemplo: Calcular a matriz de correlação e os *p-values* entre as colunas da matriz **X**:

```
» [R,P] = corrcoef(X)
```

Encontrar todos os índices da matriz de correlação para os quais o *p-value* é menor que 0.05:

```
» [i,j] = find(p<0.05)
```

Para encontrarmos a matriz de correlação entre variáveis e seus respectivos *p-values*, também podemos usar a função **corr**, com mais opções:

```
» [RHO,PVAL] = corr(X,Y,'nome',valor)
```

Exemplos:

» **[RHO,PVAL] = corr(X,Y,'type','Pearson')**

» **[RHO,PVAL] = corr(X,Y,'type','Kendall')**

» **[RHO,PVAL] = corr(X,Y,'rows','all')**

» **[RHO,PVAL] = corr(X,Y,'rows','complete')** : *pula linhas com **NaN**!*

4.2 Gráficos

Um histograma pode ser criado com o comando

```
» n = hist(Y)
```

em que o vetor **Y** é distribuido em 10 caixas igualmente espaçadas, e **n** é o número de elementos em cada caixa.

O comando

```
» n = hist(Y,nbins)
```

divide os dados em **nbins** caixas.

O comando

```
» boxplot(X)
```

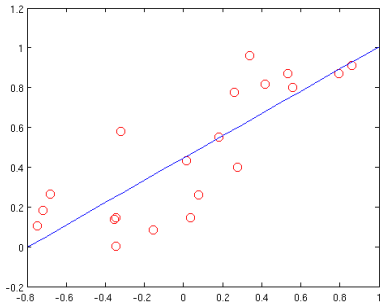
cria um gráfico de caixas dos dados em **X**. Se **X** for uma matriz, existirá uma caixa por coluna; se **X** for um vetor, existirá apenas uma caixa.

Em cada caixa:

- a marca central é a mediana;
- os limites da caixa representam o 25° e o 75° percentil;
- os marcadores externos sinalizam os pontos extremos dos dados (sem considerar *outliers*;
- os *outliers*, se existirem, serão marcados individualmente no gráfico.

4.3 Fitting

Queremos descobrir uma função (linear, polinomial ou não-linear) que aproxime um conjunto de dados:



4.3.1 Regressão

Podemos calcular automaticamente um modelo de regressão (usando quadrados mínimos) através da janela de um gráfico.

Exemplo:

```
» load census
» plot(cdate, pop, 'ro')
```

Na janela do gráfico, podemos selecionar

Tools → Basic Fitting

Podemos calcular a norma dos resíduos para um *fit* realizado através do comando

```
» sqrt(sum(resids.^2))
```

Podemos também extrapolar dados usando a interface gráfica do MATLAB, novamente em

Tools → Basic Fitting

Finalmente, podemos usar o comando

File → Generate Code

para criarmos uma função que reproduz o gráfico obtido.

Alternativamente, o comando

```
» p = polyfit(x,y,n)
```

encontra os coeficientes do polinômio $p(x)$ de grau n que aproxima os dados $y(i) = p(x(i))$, em um sentido de mínimos quadrados. O vetor **p** resultante contém os coeficientes do polinômio em ordem descendente de potências.

O comando

```
» [p,S] = polyfit(x,y,n)
```

retorna os coeficientes do polinômio em **p** e uma estrutura **S** que pode ser usada com o comando **polyval**.

A estrutura **S** contém os campos **R**, **df** e **normr**.

Se os dados **y** são aleatórios, uma estimativa da covariância de **p** é

```
» (inv(R)*inv(R)')*normr^2/df
```

O comando

```
» y = polyval(p,x)
```

retorna o valor de um polinômio de grau **n** (armazenado no vetor **p**) em **x**. O comando

```
» [y,delta] = polyval(p,x,S)
```

usa a estrutura **S** gerada pelo comando **polyfit** para gerar **delta**, que é uma estimativa do desvio padrão do erro obtido ao se tentar calcular **p(x)**.

Exemplo: Se quisermos fazer uma regressão linear em um conjunto de pontos (x, y), usamos o comando

```
» p = polyfit(x,y,1)
```

O resultado é um vetor **p** que contém os coeficientes da reta

$$y = p_1 x + p_2$$

Exemplo:

```
» x = 1:1:20;
» y = x + 10*rand(1,20);
» p = polyfit(x,y,1);
» plot(x,y,'ro')
» hold on
» t = 0:0.1:20;
» plot(t,polyval(p,t))
```

Exemplo (com resíduos):

```
»  x = 1:1:20;
»  y = x + 10*rand(1,20);
»  p = polyfit(x,y,1);
»  fitted = polyval(p,x);
»  res = y-fitted;
»  subplot(2,1,1), plot(x,y,'ro','markersize',8)
»  hold on
»  t = 0:0.1:20;
»  subplot(2,1,1), plot(t,polyval(p,t))
»  subplot(2,1,2), bar(x,res)
```

O comando

```
»  scatter(X,Y)
```

faz um gráfico dos pontos com coordenadas **X** e **Y**, usando círculos como marcadores.

Se usarmos

```
»  scatter(X,Y,S,C)
```

podemos especificar a área de cada marcador em **S**.

Outras opções:

- **`scatter(...,marcador)`** usa o marcador escolhido (p. ex. **`'+'`** ou **`'*'`**)
- **`scatter(...,'filled')`** preenche os marcadores.

O comando

```
»  scatterhist(x,y)
```

cria um *scatter plot* dos dados nos vetores **x** e **y** e também um histograma em cada eixo do gráfico.

Exemplo:

```
»  x = randn(1000,1);
»  y = exp(.5*randn(1000,1));
»  scatterhist(x,y)
```

O comando

```
»  lsline
```

acrescenta uma reta calculada através de regressão linear (mínimos quadrados) *para cada plot/scatter plot na figura atual*.

Atenção: dados conectados com alguns tipos de reta (**`'-'`**, **`'-'`** ou **`'.-'`**) são *ignorados* por **`lsline`**.

Exemplo:

```
» x = 1:10;
» y1 = x + randn(1,10);
» scatter(x,y1,25,'b','*')
» hold on
» y2 = 2*x + randn(1,10);
» plot(x,y2,'mo')
» y3 = 3*x + randn(1,10);
» plot(x,y3,'rx:')
» y4 = 4*x + randn(1,10);
» plot(x,y4,'g+-')
» lsline
```

Se o vetor **p** contém os coeficientes de um polinômio em ordem descendente de potências, o comando

```
» refcurve(p)
```

adiciona uma curva de referência polinomial com coeficientes **p** ao gráfico atual.

Se **p** é um vetor com $n+1$ elementos, a curva é dada por

$$y = p(1)x^n + p(2)x^{n-1} + \ldots + p(n)x + p(n+1)$$

O comando

```
» gline
```

permite ao usuário adicionar manualmente um segmento de reta à última figura desenhada.

A reta pode ser editada manualmente na ferramenta de edição de gráficos do MATLAB.

4.3.2 Comandos em Toolboxes

Se sua instalação do MATLAB possuir a Statistics Toolbox, o comando

```
» polytool(x,y)
```

ajusta uma reta aos vetores **x** e **y** e mostra um gráfico interativo do resultado.

```
» polytool(x,y,n)
```

ajusta um polinômio de grau **n** aos dados.

Se sua instalação possuir a Curve Fitting Toolbox, você pode fazer o ajuste de curvas de maneira interativa, usando o comando

```
» cftool
```

CAPÍTULO 5

Resolução de equações lineares e não-lineares com MATLAB

Neste capítulo, vamos ver um apanhado dos comandos mais utilizados para a resolução de sistemas lineares, resolução de equações não-lineares e de sistemas não-lineares de equações.

5.1 Comandos básicos de álgebra linear

Primeiramente, vamos ver alguns comandos básicos para matrizes:

- Para calcularmos o determinante de uma matriz quadrada **A**, usamos o comando

```
» det(A)
```

- Para calcularmos os autovalores e autovetores de uma matriz quadrada **(**A), usamos o comando **eig(A)**.
- Se digitarmos simplesmente

```
» eig(A)
```

  o resultado é um vetor contendo os autovalores de *A*. Se usarmos a forma

```
» [V,D] = eig(A)
```

  o resultado é uma matriz **V** cujas colunas são autovetores de *A*, e uma matriz diagonal **D** cujos elementos não nulos são os autovalores de *A*.

- Para calcularmos a inversa de uma matriz *quadrada e inversível* **A**, usamos o comando

```
» inv(A)
```

5.2 Resolução de Sistemas Lineares no MATLAB

Aqui, vamos supor que queremos resolver um sistema linear, ou seja, um problema do tipo

> Encontrar $x \in \mathbb{R}^n$ tal que
>
> $$Ax = b$$
>
> com $A \in \mathbb{R}^{m \times n}$ e $b \in \mathbb{R}^m$.

Existem algumas maneiras de resolver este tipo de problema no MATLAB.

5.2.1 Usando a inversa: **inv**

Primeiramente, se $m = n$ (ou seja, a matriz do problema é quadrada) e A for inversível, podemos simplesmente invertê-la para encontrar

$$x = A^{-1}b.$$

No MATLAB, teremos então

```
»  x = inv(A)*b
```

No entando, sabemos que esse método é o menos eficiente de todos, tanto em desempenho quanto em termos de instabilidade numérica. Assim, vamos procurar outros métodos para resolvermos esse problema.

5.2.2 O operador \

Se, não for desejável encontrar a inversa da matriz A, ou se ela não for quadrada, e não quisermos nos preocupar com o método utilizado na resolução do sistema (em outras palavras, estamos interessados em uma solução, mesmo que aproximada, e não no método utilizado para a resolução), podemos usar o operador \ no MATLAB. Para nosso sistema $Ax = b$, teríamos então

```
»  x = A\b
```

Este operador atua da seguinte maneira:

- Se A for quadrada e inversível, então x é a solução do sistema encontrada por **`x = inv(A)*b`**.
- Se A for quadrada, mas não for inversível, uma mensagem de erro é mostrada. O MATLAB tentará resolver o sistema, mas o resultado obtido nem sempre é confiável. Você pode verificar isso avaliando o resíduo, que pode ser bastante grande em casos de mal condicionamento extremo.
- Se A for $m \times n$ com $m \neq n$, então x é a solução no sentido de mínimos quadrados do sistema, ou seja, x minimiza a norma do resíduo $r = \|Ax - b\|$.

5.2.3 Decomposição LU: **lu**

Sabe-se da álgebra linear que se for possível encontrar a decomposição LU de uma matriz A, é possível resolver o sistema $Ax = b$ através de dois sistemas triangulares, um superior e um inferior, da seguinte forma:

$$Ax = b \Leftrightarrow (LU)x = b \Leftrightarrow \begin{cases} Ly &= b \\ Ux &= y \end{cases}$$

Para encontrarmos a decomposição LU de uma matriz A no MATLAB, usamos o comando

```
» [L,U] = lu(A)
```

Podemos em seguida resolver o sistema $Ax = b$ fazendo

```
» y = L\b
» x = U\y
```

5.2.4 **linsolve**

Se não quisermos nos preocupar com o algoritmo usado na resolução do sistema, temos uma outra opção no comando **linsolve**.

```
» x = linsolve(A,b)
```

resolve o sistema linear $Ax = b$ usando a fatoração LU com pivotamento parcial de A, caso a matriz seja quadrada, e a fatoração QR de A com pivotamento nas colunas, caso contrário.

5.2.5 Métodos Iterativos para Sistemas Lineares

O MATLAB também conta com alguns algoritmos para resolver sistemas lineares de forma iterativa, isto é, a partir de um ponto inicial x_0, obtém-se uma aproximação para a solução exata do sistema através da realização de algum procedimento várias vezes, até que alguma condição seja satisfeita.

Se a matriz A for grande e esparsa, métodos diretos (usando fatoração) são, em geral, pouco eficientes. Nestes casos, podemos usar um dos seguintes métodos, disponíveis no MATLAB:

- **pcg** Gradiente conjugado precondicionado
- **bicg** Gradiente bi-conjugado (gradiente conjugado para matrizes não necessariamente simétricas)
- **gmres** Generalized minimum residual method (com restarts)
- **lsqr** LSQR (quadrados mínimos)

Para mais informações sobre estes métodos, pode-se consultar no *help* do MATLAB a página intitulada *Systems of Linear Equations*, no item *Iterative Methods for Solving Systems of Linear Equations*.

5.3 Resolução de equações não-lineares

5.3.1 Equação não linear a uma variável: **fzero**

Aqui, o problema que nos interessa é encontrar uma raiz da equação

$$f(x) = 0$$

onde f é uma função de uma variável real. Para encontrarmos a solução deste problema para uma função f qualquer, a partir de um ponto $x0$, podemos usar o comando **fzero(função,x0)**. No entanto, um cuidado especial com a função a ser minimizada é necessário.

No MATLAB, para passarmos uma função como argumento de outra função (no caso, queremos passar $f(x)$ como argumento da função **fzero**) precisamos obter a *referência* da função (assim como tínhamos obtido a referência a um objeto gráfico anteriormente).

Primeiramente, a função para a qual gostaríamos de encontrar uma raiz deve estar em um arquivo próprio, definida como uma função do MATLAB, no formato

```
[y]  = minhafuncao(x)
     comandos
```

Em seguida, chamamos a função **fzero** no console para minimizar a função **minhafuncao** a partir do ponto **x0**, escrevendo

```
»  fzero(@minhafuncao,x0)
```

Exemplo: Para encontrar uma das raizes de $f(x) = x^2 - 4$, previamente definida em um arquivo de nome **quadratica.m**, a partir do ponto $x = 6$, usamos o comando

```
» fzero(@quadratica,6)
```

O algoritmo da função **fzero** usa uma combinação dos métodos da bissecção, secante e interpolação quadrática inversa.

5.3.2 Raizes de um polinômio: **roots**

Para encontrar as raizes de um polinômio de grau n da forma

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

primeiramente representamos este polinômio como um vetor *linha* **p** no MATLAB, cujas componentes são os coeficientes dos termos em ordem descendente de grau, ou seja,

```
»  p = [a_n  a_{n-1}  ···  a_2  a_1  a_0]
```

Em seguida, usamos o comando

```
»  r = roots(p)
```

resultando em um vetor coluna **r** com as raizes deste polinômio.

Exemplo: $p(x) = t^3 + 2t^2 - 5t - 6$ (Figura 5.1).

```
»  p = [1 2 -5 -6]
»  roots(p)
```

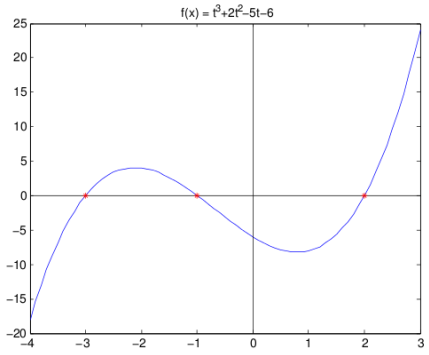


Figura 5.1: $p(x) = t^3 + 2t^2 - 5t - 6$ e suas raizes.

5.3.3 Sistema de equações não lineares: **fsolve**

Para encontrarmos a solução de um sistema de equações não lineares da forma

$$F(x) = 0$$

onde $F : \mathbb{R}^n \to \mathbb{R}^m$, usamos a função **fsolve**. Neste caso, similarmente à função **fzero**, devemos escrever a função $F(x)$ em um arquivo **.m** separadamente, e usar a sintaxe

```
» fsolve(@minhafuncao,x0)
```

Exemplo: Resolver o sistema de equações

$$\begin{cases} y_1 &= 3x_1^2 + 4x_2^2 - 16 \\ y_2 &= 2x_1^2 - 3x_2^2 - 5 \end{cases}$$

```
» fsolve(@funcs,[1;1])
```

Observação Este comando faz parte da *Optimization Toolbox*, que pode não estar disponível na sua instalação do MATLAB.

5.4 Otimização: Minimização de funções

Agora, queremos resolver o problema

$$\text{minimizar } f(x).$$

5.4.1 Minimização de uma função de uma variável: **fminbnd**

Para encontrarmos o mínimo de uma função de uma variável dentro de um intervalo $[a, b]$, usamos o comando

```
» x = fminbnd(@funcao,a,b)
```

Se quisermos também saber o valor da função no ponto de mínimo, usamos a sintaxe

```
» [x,fval] = fminbnd(@funcao,a,b)
```

5.4.2 Minimização de uma função de várias variáveis: **fminsearch**

Para encontrarmos o mínimo de uma função real de várias variáveis, a partir de um ponto inicial **x0**, usamos o comando

```
»  x = fminsearch(@funcao,x0)
```

Se quisermos também saber o valor da função no ponto de mínimo, usamos a sintaxe

```
»  [x,fval] = fminsearch(@funcao,x0)
```

CAPÍTULO 6

Outros comandos úteis

6.1 Interpolação polinomial: **interpn**

Para interpolarmos um conjunto de pontos em 1D, 2D, 3D ou em *n* dimensões, usamos os comandos

- **interp1**
- **interp2**
- **interp3**
- **interpn**

respectivamente.

6.1.1 Interpolação 1D: **interp1**

O comando

```
»  yi = interp1(x,Y,xi,method)
```

interpola os dados (**x**,**Y**) nos novos pontos **xi**, usando o método **method**, que pode ser:

- **'nearest'** Vizinho mais próximo
- **'linear'** Interpolação linear (default)
- **'spline'** Splines cúbicos
- **'cubic'** Interpolação por polinômios de Hermite

Exemplo:

```
»  x = 0:10;
»  y = sin(x);
»  xi = 0:.25:10;
»  yi = interp1(x,y,xi);
»  plot(x,y,'o',xi,yi);
»  hold on;
»  zi = interp1(x,y,xi,'nearest');
»  plot(xi,zi,':k');
»  wi = interp1(x,y,xi,'spline');
»  plot(xi,wi,'m');
»  ui = interp1(x,y,xi,'cubic');
»  plot(xi,ui,'-g')
```

6.1.2 Interpolação 2D: **interp2**

O comando

```
»  ZI = interp2(X,Y,Z,XI,YI,method)
```

interpola os dados (**X**,**Y**,**Z**) nos novos pontos (**XI**,**YI**) usando o método **method**, que pode ser

- **'nearest'** Vizinho mais próximo
- **'linear'** Interpolação linear (default)
- **'spline'** Splines cúbicos
- **'cubic'** Interpolação cúbica, se os dados forem uniformemente espaçados; senão, é o mesmo que **spline**.

6.2 Aproximação polinomial: **polyfit**

O comando

```
»  p = polyfit(x,y,n)
```

encontra os coeficientes do polinômio $p(x)$ de grau n que aproxima os dados $y(i) = p(x(i))$, em um sentido de mínimos quadrados. O vetor **p** resultante contém os coeficientes do polinômio em ordem descendente de potências.

Exemplo: Tentar interpolar com um polinômio os mesmos pontos do exercício anterior, ou seja,

```
»  x = 0:10;
»  y = sin(x);
```

6.3 Integração Numérica

6.3.1 Integração numérica geral: **integral**

Para calcularmos uma aproximação numérica de $\int_a^b f(x)dx$, usamos o comando

```
»  q = integral(fun,a,b)
```

em que **fun** é uma referência a uma função.

Exemplo: Calcular a integral imprópria $f(x) = e^{-x^2}(\ln(x))^2$ entre 0 e ∞.

```
»  fun = @(x) exp(-x.^2).*log(x).^2;
»  q = integral(fun,0,Inf)
```

6.3.2 Integração numérica finita: **quad**

Para calcularmos uma aproximação numérica de $\int_a^b f(x)dx$ com a e b finitos pela quadratura de Simpson (adaptativa), usamos o comando

```
»  q = quad(fun,a,b)
```

em que **fun** é uma referência a uma função.

6.3.3 Integração numérica discreta: **trapz**

Se tudo o que conhecemos sobre a função é seus valores em um conjunto de pontos, podemos aproximar o valor da sua integral $\int_a^b f(x)dx$ usando o comando **trapz**:

```
»  x = 0:pi/100:pi;
»  y = sin(x);
»  z = trapz(x,y)
```

6.4 Diferenciação Numérica: **gradient**

O *gradiente* de uma função $f : \mathbb{R}^n \to \mathbb{R}$ é dado por

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial F}{\partial x_n}(x) \right).$$

Para calcularmos o gradiente de uma função de uma variável, procedemos da seguinte maneira.

```
»  x = a:h:b;
»  f = funcao(x);
»  g = gradient(f,h)
```

O comando **gradient** calcula numericamente a derivada de f em função da variável x nos pontos escolhidos.

Para calcularmos o gradiente de uma função de duas variáveis, o procedimento é equivalente. A diferença é que agora precisamos gerar uma malha de pontos usando o comando **meshgrid**.

```
»  x = a:hx:b;
»  y = c:hy:d;
»  [x,y] = meshgrid(x,y);
»  f = funcao(x,y)
»  [gx,gy] = gradient(f,hx,hy)
```

6.5 Resolução de Equações Diferenciais Ordinárias

6.5.1 Problemas de Valor Inicial

Uma equação diferencial ordinária (EDO) é uma equação que envolve uma ou mais derivadas de uma variável dependente y com respeito a uma única variável independente t ($y = y(t)$). Com frequência, $y(t)$ é um vetor com componentes $y = (y_1(t), y_2(t), \ldots, y_n(t))$.

O MATLAB resolve equações diferenciais ordinárias de primeira ordem dos seguintes tipos:

- EDOs explícitas, do tipo $y' = f(t, y)$
- EDOs linearmente implícitas, do tipo $M(t, y)y' = f(t, y)$, em que $M(t, y)$ é uma matriz
- EDOs implícitas, do tipo $f(t, y, y') = 0$

Para resolvermos equações diferenciais de ordem superior, precisamos escrevê-las como um sistema de equações de primeira ordem (como fazemos no curso de cálculo).

Geralmente, temos uma família de soluções $y(t)$ que satisfaz a EDO. Para obtermos uma solução única, exigimos que a solução satisfaça alguma condição inicial específica, de forma que $y(t_0) = y_0$ em algum valor inicial t_0.

$$y' = f(t, y)$$
$$y(t_0) = y_0$$

No MATLAB, temos solvers para três tipos de problemas de valor inicial. Em todos os casos, a sintaxe para resolver uma equação diferencial é

```
»  [t,y] = solver(odefun,tspan,y0,options)
```

onde **`solver`** é substituido por uma das opções que veremos em seguida. Os argumentos de entrada são sempre os seguintes:

- **`odefun`**: O *handle* para uma função que avalia o sistema de EDOs em um ponto. Esta função deve estar na forma **`dydt = odefun(t,y)`**, onde **`t`** é um escalar e **`dydt`** e **`y`** são vetores coluna.
- **`tspan`**: vetor especificando o intervalo de integração. O solver impõe a condição inicial em **`tspan(1)`**, e integra de **`tspan(1)`** até **`tspan(end)`**.
- **`y0`**: vetor das condições iniciais para o problema.
- **`options`**: Struct de parâmetros opcionais que modificam as propriedades padrão de integração.

Os argumentos de saída são

- **`t`**: vetor coluna das variáveis independentes (pontos no intervalo desejado)
- **`y`**: vetor ou matriz contendo, em cada linha, a solução calculada no ponto contido na linha correspondente de **`t`**.

6.5.2 Solvers

Os métodos disponíveis estão divididos de acordo com o tipo de problema que resolvem:

- Problemas Não-Stiff:
  - **`ode45`** (Runge-Kutta, passo simples)
  - **`ode23`** (Runge-Kutta, passo simples)
  - **`ode113`** (Adams-Bashforth-Moulton, passo múltiplo)
- Problemas Stiff:
  - **`ode15s`** (numerical differentiation formulas (NDFs), passo múltiplo)
  - **`ode23s`** (Rosenbrock, passo único)
  - **`ode23t`** (Trapezoide)
  - **`ode23tb`** (Runge-Kutta)

Para equações implícitas da forma

$$f(t, y, y') = 0,$$

pode-se usar o solver **`ode15i`** (consultar a documentação para mais detalhes).