# Introdução ao MATLAB

Pedro Cortez Lopes
Rafael Lopez Rangel
Luiz Fernando Martha


**CIV2801 – Fundamentos da Computação Gráfica Aplicada**

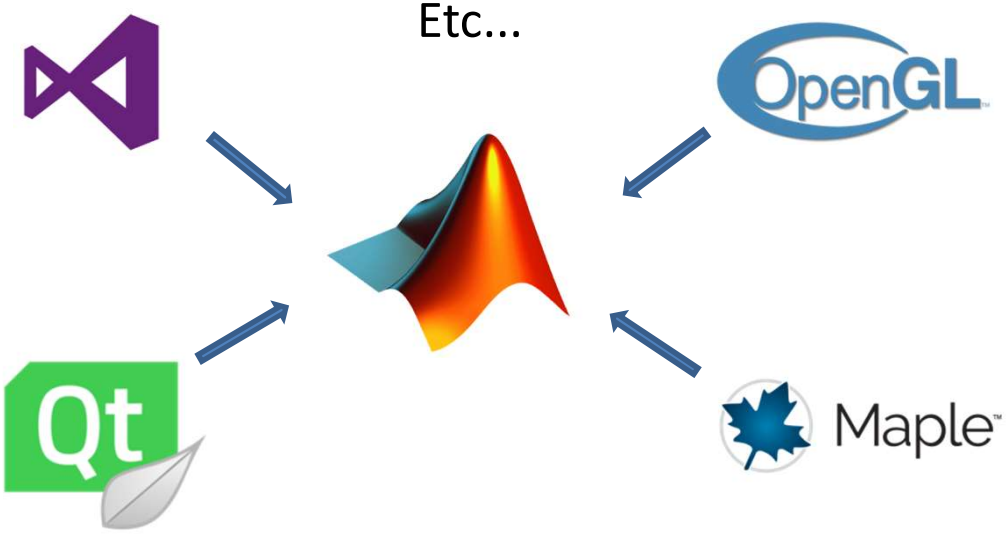
2024.2

# Introdução ao MATLAB

- **Vantagens no Uso do MATLAB para Aplicações em Engenharia**

Álgebra Computacional
+
Programação
+
Desenvolvimento de Interface Gráfica
+
Sistema Gráfico

Etc...



# Introdução ao MATLAB

- **Ambiente de Álgebra Computacional**

**Variáveis Matriciais**
Todas as variáveis são interpretadas como matrizes, inclusive escalares, permitindo que operações matriciais possam ser realizadas facilmente com comandos simples.

Ex.: Resolução de sistemas lineares [A]{X} = {B} → X = B \ A;

**Álgebra Simbólica**
Pacote que permite a resolução e manipulação de expressões e equações matemáticas com variáveis simbólicas.

# Introdução ao MATLAB

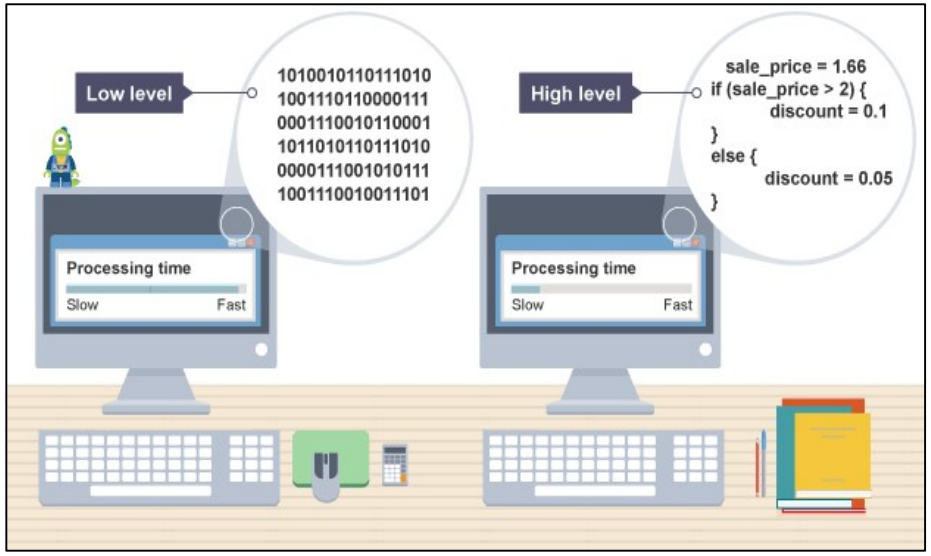
- **Ambiente de Programação**

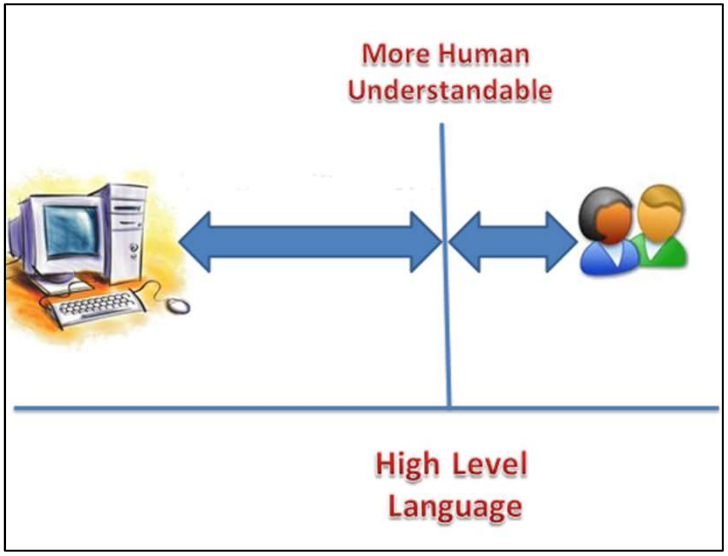
**Linguagem de programação de alto nível**

Nível de abstração elevado, longe do código de máquina e próximo à linguagem humana.

**Linguagem interpretada**

Diferente de uma linguagem compilada, em que um compilador traduz o código fonte para linguagem de máquina, uma linguagem interpretada é executada comando a comando por um interpretador



# Introdução ao MATLAB

**Linguagem multi-paradigma**

Um paradigma de programação é forma que se apresenta a estrutura e execução do código. Em MATLAB pode-se escrever o código de um programa utilizando os seguintes paradigmas de programação:

- **Programação Estruturada**: Baseia-se em um código que se reduz a estruturas sequenciais, iterativas e de decisão.

- **Programação Orientada a Objetos**: Baseia-se na composição e interação de unidades de software chamadas objetos.

- **Programação Orientada a Eventos**: Em contraste à programação orientada a fluxos, a execução do código é guiada por indicações externas chamadas eventos.

Outras vantagens que facilitam a programação em MATLAB incluem a não necessidade de declaração dos tipos das variáveis e a manutenção automática da memória (garbage collection).

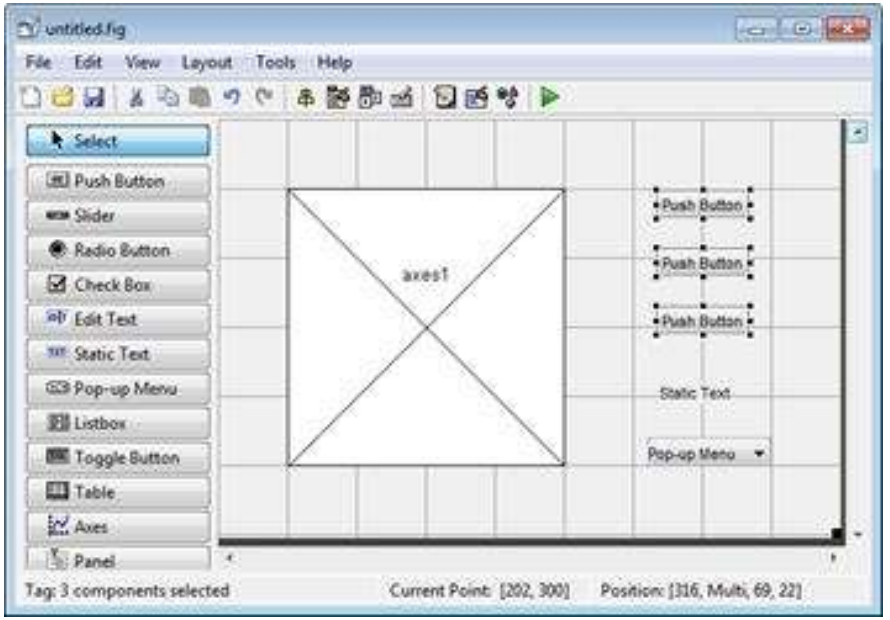
# Introdução ao MATLAB

- **Desenvolvimento de Interface Gráfica**

O MATLAB disponibiliza um ambiente interativo de desenvolvimento de interfaces gráficas.

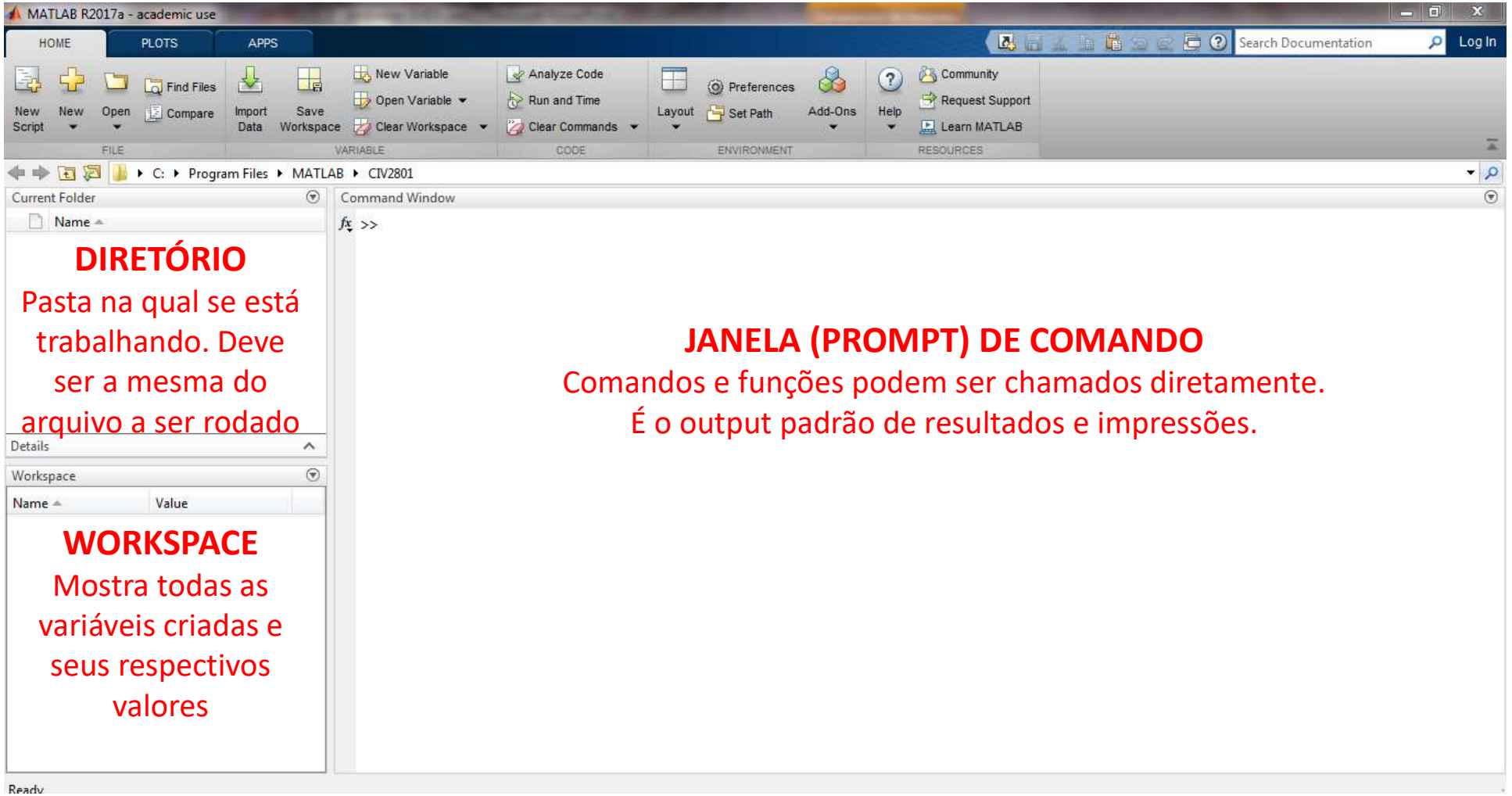
Neste ambiente, os componentes gráficos são adicionados manualmente por um sistema *drag-and-drop*, e um arquivo baseado no paradigma da Programação Orientada a Eventos é criado automaticamente com templates de funções básicas que devem ser preenchidas com o código que controlará a interface.

É possível criar arquivos executáveis a partir do código dos programas escritos usando arquivos MATLAB.

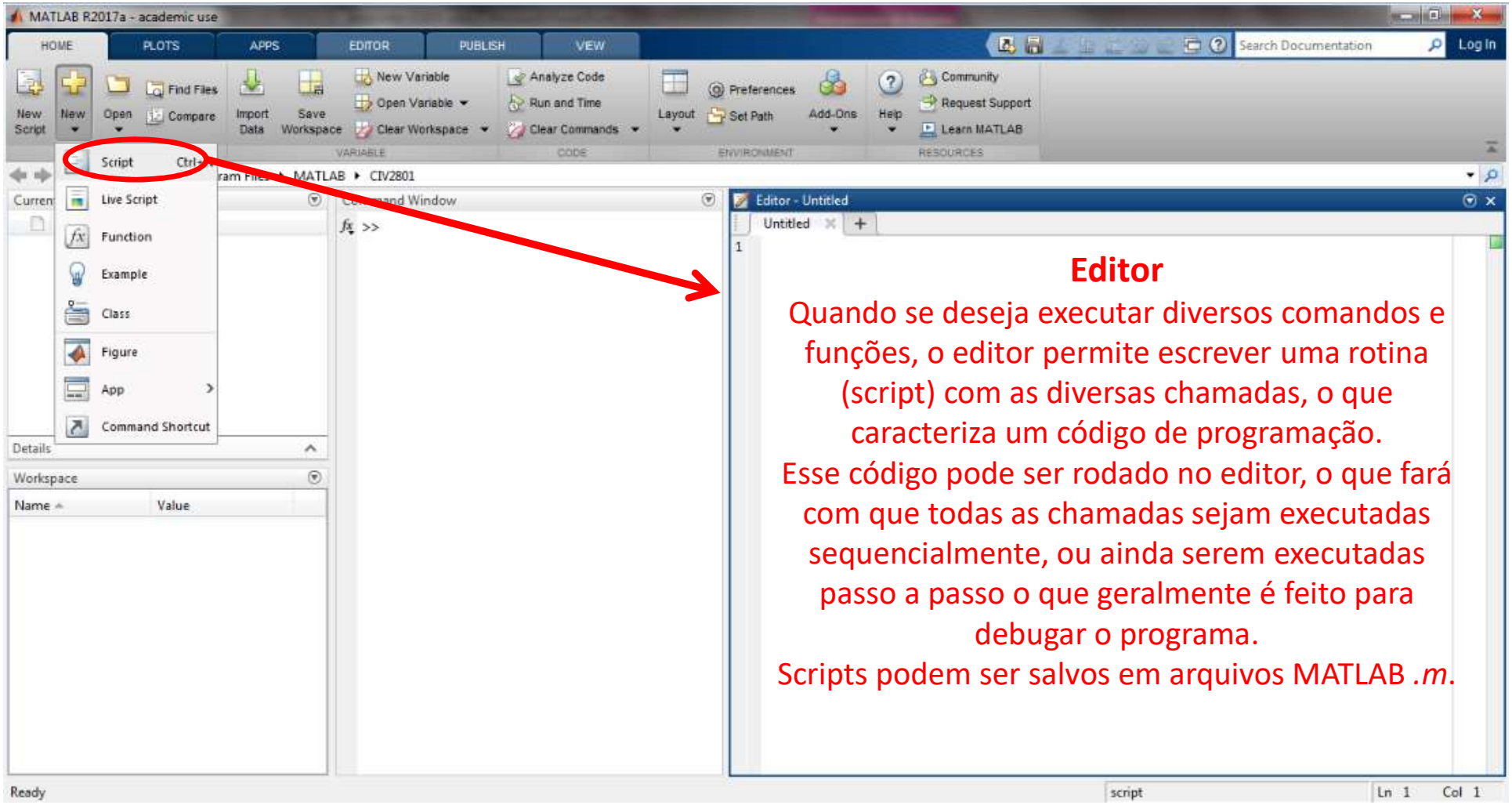


# Introdução ao MATLAB

- **Janela Principal**



# Introdução ao MATLAB




## Editor

Quando se deseja executar diversos comandos e funções, o editor permite escrever uma rotina (script) com as diversas chamadas, o que caracteriza um código de programação.

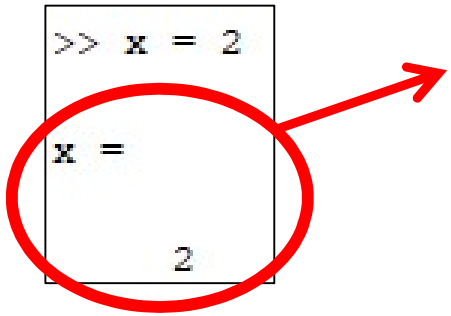
Esse código pode ser rodado no editor, o que fará com que todas as chamadas sejam executadas sequencialmente, ou ainda serem executadas passo a passo o que geralmente é feito para debugar o programa.

Scripts podem ser salvos em arquivos MATLAB *.m*.

# Introdução ao MATLAB

- **Definição de variáveis**

A definição de variáveis no MATLAB se dá pela simples atribuição de valor, conforme exposto abaixo.




Toda vez que um novo valor for atribuído a uma variável, uma resposta como esta será impressa na janela de comando. Para evitar que isso aconteça, basta encerrar a linha de comando com ponto-e-vírgula.

A alocação de memória é dinâmica, ou seja, a variável é declarada e salva no *workspace* no momento em que algum valor é atribuído à mesma.

# Introdução ao MATLAB

- **Definição de vetores e matrizes**

Assim como variávies escalares (que são tratadas como matrizes 1x1), a definição de vetores e matrizes se dá pela simples atribuição de valores.

```
>> b = [1, 2, 3]

b =

     1     2     3
```

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

A =

     1     2     3
     4     5     6
```

Quando usado entre chaves, ponto-e-vírgula indica o fim de uma linha em uma matriz.

A atribuição de valores também pode ser feita termo a termo, como exposto abaixo.

```
>> c(1,1) = 2;
>> c(1,2) = 3;
>> c(2,1) = 7;
>> c

c =

     2     3
     7     0
```

Note que, no MATLAB, a indexação começa em 1, contrastando com a convenção usual adotada por outras linguagens de programação, que iniciam a indexação em 0.

Termos não informados da matriz são preenchidos com 0.

# Introdução ao MATLAB

- **Definição de vetores**

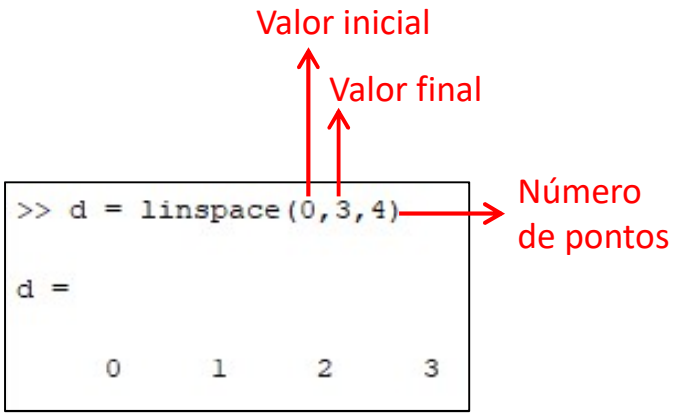
Vetores podem ser definidos pelo uso de dois pontos, gerando valores em intervalos regulares entre dois números, como exposto abaixo.

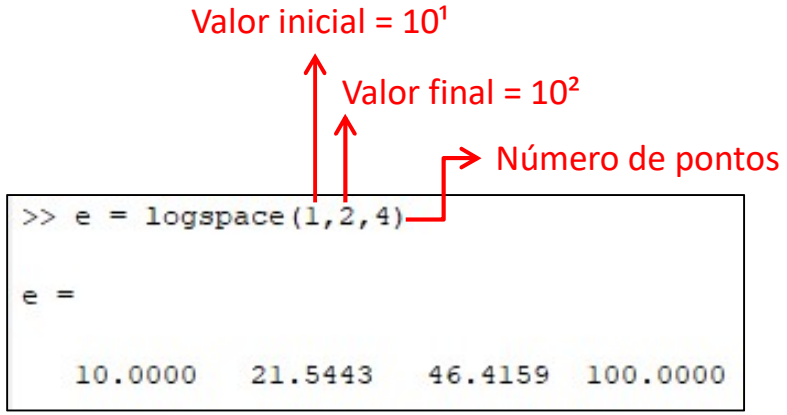
```
>> a = 1:4

a =

     1     2     3     4
```

```
>> b = 2:0.5:4

b =

    2.0000    2.5000    3.0000    3.5000    4.0000
```

```
>> c = 1:-1:-2

c =

     1     0    -1    -2
```

Seguindo uma lógica similar, é possível definir vetores com as funções *linspace* (define um vetor linearmente espaçado entre dois valores) e *logspace* (define um vetor logaritmicamente espaçado entre dois valores) .



# Introdução ao MATLAB

- **Definição de matrizes**

É possível definir matrizes automaticamente por meio de funções. Algumas das mais usuais são as seguintes.

**zeros/ones**

As funções *zeros* e *ones* definem matrizes totalmente compostas por 0 e 1, respectivamente. Se apenas um parâmetro de entrada *dim* for fornecido, será montada uma matriz (*dim* **X** *dim)*, caso *n* parâmetros sejam fornecidos, será montada uma matriz de ordem *n* com dimensões ($dim_1$ **X** $dim_2$ **X** ... **X** $dim_{n-1}$ **X** $dim_n$).

```
>> A = zeros(3)

A =

     0     0     0
     0     0     0
     0     0     0
```

```
>> B = ones(1,3)

B =

     1     1     1
```

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

C =

     0     0
     0     0
     0     0
```

```
>> D = ones(3,3,2)

D(:,:,1) =

     1     1     1
     1     1     1
     1     1     1


D(:,:,2) =

     1     1     1
     1     1     1
     1     1     1
```

# Introdução ao MATLAB

- **Definição de matrizes**

**eye**

A função *eye* retorna uma matriz identidade com dimensões de acordo com os parâmetros fornecidos.

```
>> A = eye(3)

A =

     1     0     0
     0     1     0
     0     0     1
```

```
>> B = eye(3,2)

B =

     1     0
     0     1
     0     0
```

**rand**

A função *rand* retorna uma matriz de valores aleatórios contidos no intervalo (0,1) com dimensões de acordo com os parâmetros fornecidos.

```
>> C = rand(2)

C =

    0.8147    0.1270
    0.9058    0.9134
```

```
>> D = rand(2,4)

D =

    0.6324    0.2785    0.9575    0.1576
    0.0975    0.5469    0.9649    0.9706
```

# Introdução ao MATLAB

- **Definição de matrizes**

**diag**

A função *diag* retorna uma matriz diagonal baseada em um vetor fornecido como parâmetro de entrada.

```
>> A = diag([1, 2, 3])

A =

     1     0     0
     0     2     0
     0     0     3
```

Caso o dado de entrada seja uma matriz, *diag* retorna sua diagonal principal.

**sparse**

Caso o parâmetro de entrada seja uma matriz, a função *sparse* retorna uma matriz esparsa, isto é, guarda apenas os valores diferentes de 0 e suas posições. As operações matriciais continuam válidas para matrizes esparsas.

A função *sparse* também pode ser usada para gerar uma matriz esparsa a partir de vetores que indiquem tripletes, com as posições e valores dos termos a serem guardados.

```
>> B = sparse(A)

B =

   (1,1)        1
   (2,2)        2
   (3,3)        3
```

```
>> C = sparse([1,2,3],[3,1,2],[4,7,6])
                    i     j     valor
C =

   (2,1)        7
   (3,2)        6
   (1,3)        4
```

```
>> D = full(C)

D =

     0     0     4
     7     0     0
     0     6     0
```

A função *full* retorna uma matriz cheia, a partir de uma matriz esparsa.

# Introdução ao MATLAB

- **Indexação de vetores e matrizes**

No MATLAB, é adotada a convenção de que a indexação inicia de 1, ou seja, um vetor {v} de *n* termos, por exemplo, vai do termo *v(1)* ao termo *v(n)*.

```
>> v = 2:2:10

v =

     2     4     6     8    10
```

```
>> [ v(1), v(end)]

ans =

     2    10
```

A expressão *end* retorna o último termo da linha/coluna em questão.

É possível ainda trabalhar com a indexação vetorial, ou seja, utilizar vetores como índices para referir-se a múltiplos termos de um vetor ou matriz.

```
>> v(2:4)

ans =

     4     6     8
```

```
A =

     8     9     2
     9     6     5
     1     0     9
```

```
>> b = diag(A([1,2,2],[3,1,2]))

b =

     2
     9
     6
```

# Introdução ao MATLAB

- **Operações matriciais**

Por trabalhar com variáveis matriciais, o MATLAB confere certa simplicidade a essas operações, que, em outras linguagens computacionais, deveriam ser implementadas pelo usuário, ou importadas de bibliotecas feitas por terceiros.

**Soma e subtração**

```
>> A = [1:3;
        4:6;
        7:9]

A =

     1     2     3
     4     5     6
     7     8     9
```

```
>> B = rand(3)

B =

    0.9572    0.1419    0.7922
    0.4854    0.4218    0.9595
    0.8003    0.9157    0.6557
```

```
>> C = A + B

C =

    1.9572    2.1419    3.7922
    4.4854    5.4218    6.9595
    7.8003    8.9157    9.6557
```

```
>> D = A - B

D =

    0.0428    1.8581    2.2078
    3.5146    4.5782    5.0405
    6.1997    7.0843    8.3443
```

# Introdução ao MATLAB

- **Operações matriciais**

## Multiplicação

```
>> A = [1:3;
        4:6;
        7:9]

A =

     1     2     3
     4     5     6
     7     8     9
```

```
>> b = logspace(0,2,3)

b =

     1    10   100
```

```
>> C = b * A

C =

   741   852   963
```

```
>> Vector = [ 1 ; 5; 10]

Vector =

     1
     5
    10
```

```
>> Result = A * Vector

Result =

    41
    89
   137
```

Atenção às dimensões das matrizes e vetores multiplicados!

$$A_{i,k} \times B_{k,j} = C_{i,j}$$

# Introdução ao MATLAB

- **Operações matriciais**

**Transposição**

```
A =

     2     8     9
     0     6     0
     0     3     4

>> A'

ans =

     2     0     0
     8     6     3
     9     0     4
```

```
b =

     1     2     3

>> b'

ans =

     1
     2
     3
```

# Introdução ao MATLAB

- **Operações matriciais**

**Solução de sistemas lineares**

$$\begin{cases} 2x_1 + 5x_2 + 9x_3 = 1 \\ 7x_1 + 6x_2 + 5x_3 = 2 \\ 2x_1 + 8x_2 + x_3 = 8 \end{cases} \rightarrow \begin{bmatrix} 2 & 5 & 9 \\ 7 & 6 & 5 \\ 2 & 8 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \\ 8 \end{Bmatrix}$$

```
A =

     2     5     9
     7     6     5
     2     8     1
```

```
b =

     1
     2
     8
```

```
>> x = A \ b

x =

   -0.3848
    1.1516
   -0.4431
```

O comando **** se referencia a um algoritmo que escolhe a melhor solução para o sistema em questão, a partir de propriedades de [A] (deve ser bem condicionada). A solução {x} também pode ser encontrada por *x = inv(A) * b*, onde o comando *inv(A)* retorna a matriz inversa de [A], mas esse procedimento não é computacionalmente eficiente.

# Introdução ao MATLAB

- **Operações vetoriais**

Existem funções próprias de operações vetoriais, duas das mais usuais são as seguintes.

**cross**
Retorna o produto vetorial entre dois vetores.

**norm**
Retorna a norma de um vetor.

```
>> x = [2 0 0]

x =

     2     0     0
```

```
>> y = [0 2 0]

y =

     0     2     0
```

```
>> z = cross(x,y)

z =

     0     0     4

>> norm(z)

ans =

     4
```

# Introdução ao MATLAB

- **Operações vetoriais**

É possível operar termo a termo dentro de vetores, seguindo o nome dos mesmos por um ponto na linha de comando.

```
>> x = linspace(0,4,5)

x =

     0     1     2     3     4
```

```
>> a = logspace(0,1,5)

a =

    1.0000    1.7783    3.1623    5.6234   10.0000
```

```
>> y = x.^3 + x.*a - 2

y =

   -2.0000    0.7783   12.3246   41.8702  102.0000
```

# Introdução ao MATLAB

- **Operações polinomiais**

No MATLAB, há como trabalhar com polinômios alocando seus coeficientes em vetores. Existem funções bastante práticas disponíveis, algumas delas são expostas a seguir.

**polyval**

Retorna os valores de um polinômio para determinado intervalo de valores de seu domínio, *y = p(x)*.

```
>> p = [1 4 4]; % p(x) = x² + 4x + 4
>> x = 0:5;
>> y = polyval(p,x)

y =

     4     9    16    25    36    49
```

O uso de % separa os comentários do código.

# Introdução ao MATLAB

- **Operações polinomiais**

**roots**
Retorna as raízes de dado polinômio.

**poly**
Retorna os coeficientes de um polinômio a partir de suas raízes.

**polyder**
Retorna os coeficientes da derivada de dado polinômio.

```
>> p = [1, -3, 2];  % p(x) = x² - 3x + 2
>> r = roots(p)

r =

     2
     1
```

```
>> r = [2, 2, 3];  % raízes de um polinômio
>> p = poly(r)

p =

     1    -7    16   -12

>> % p(x) = x³ - 7x² + 16x - 12
>> p_der = polyder(p)

p_der =

     3   -14    16

>> % p'(x) = 3x² - 14x + 16
```

# Introdução ao MATLAB

- **Expressões comparativas**

Existem expressões lógicas, que não atribuem valor a variáveis. São importantes para o funcionamento de estruturas condicionais. As mais usuais estão expostas abaixo.

| | | |
|---|---|---|
| **==** | → | igual a |
| **~=** | → | diferente de |
| **>** | → | maior que |
| **>=** | → | maior ou igual a |
| **<** | → | menor que |
| **<=** | → | menor ou igual a |
| **~** | → | não |
| **&&** | → | e |
| **\|\|** | → | ou |

# Introdução ao MATLAB

- **Expressões condicionais**

Para restringir o acesso a linhas de comando mediante condições específicas, podem ser utilizadas duas estruturas condicionais, *if-else* ou *switch-case*.

**if-else**

A mais comumente usada, baseia-se em informar condições para que determinadas linhas de código sejam executadas, podendo haver outras linhas que devam ser executadas caso tais condições não sejam atendidas .

```
>> x = 2;
>> if x < 1
     y = x^2 - 2;
   elseif x >= 1 && x < 2
     y = x;
   else
     y = - x^2 + 2;
   end
>> y

y =

    -2
```

# Introdução ao MATLAB

- **Expressões condicionais**

**switch-case**

Funciona como um interruptor, executa determinados blocos de código mediante condições específicas. Não trabalha com o caso onde nenhuma condição é atendida.

```
>> caminho = 2;
>> switch caminho
       case 1
            tempo = 30;
       case 2
            tempo = 45;
       case 3
            tempo = 40;
   end
>> tempo

tempo =

    45
```

# Introdução ao MATLAB

- **Expressões de loop**

Laços podem ser criados no MATLAB, de forma que uma ação seja repetida por um número controlado de vezes ou até que determinada condição seja violada, utilizando as expressões *for* ou *while*, respectivamente.

**for**

As ações dentro do laço serão repetidas *n* vezes, onde, em cada repetição, uma variável auxiliar terá um valor de dado vetor, de dimensão *n*.

```
>> x = ones(1,5);
>> for i = 0:4
       x(i+1) = x(i+1) + i;
   end
>> x

x =

     1     2     3     4     5
```

```
>> loop = [2,1,4,-1];
>> n = size(loop,2);  % computa a dimensão de 'loop'
>> x = zeros(1,n);    % pré-dimensiona 'x', aumenta eficiência computacional no laço
>> index = 1;         % inicializa contador
>> for i = loop
       x(index) = i * abs(i);
       index = index + 1;
   end
>> x

x =

     4     1    16    -1
```

A função *abs* retorna o valor absoluto de uma variável.

ATENÇÃO! Pela funcionalidade matricial do MATLAB, em muitos casos, é possível evitar o uso de laços com poucas linhas de comando, o que confere, além de simplicidade, maior eficiência ao código.

```
>> loop = [2,1,4,-1];
>> x = loop.*abs(loop)

x =

     4     1    16    -1
```

# Introdução ao MATLAB

- **Expressões de loop**

**while**

As ações dentro do laço serão repetidas até que dada expressão condicional deixe de ser verdadeira. Deve-se ter atenção para evitar a possibilidade de laços infinitos.

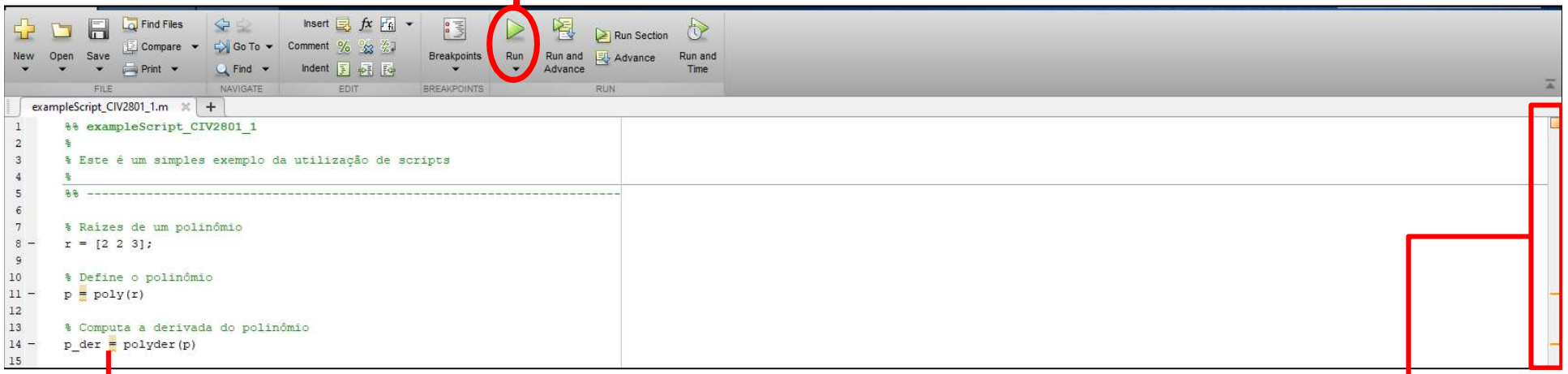
```
>> flag = 0;  % inicializa variável auxiliar
>> i = 1;  % inicializa contador
>> x = zeros(1,5);
>> while flag == 0
       x(i) = i;
       i = i + 1;
       if i > 5
         flag = 1;
       end
   end
>> x

x =

     1     2     3     4     5
```

# Introdução ao MATLAB

- **Uso de scripts**

*Scripts* podem ser utilizados para realizar uma sequência de comandos, ao invés de digitá-los e processá-los individualmente via *prompt*. São salvos em arquivos com a extensão *.m*.

Para rodar o *script*, basta clicar em *run*.



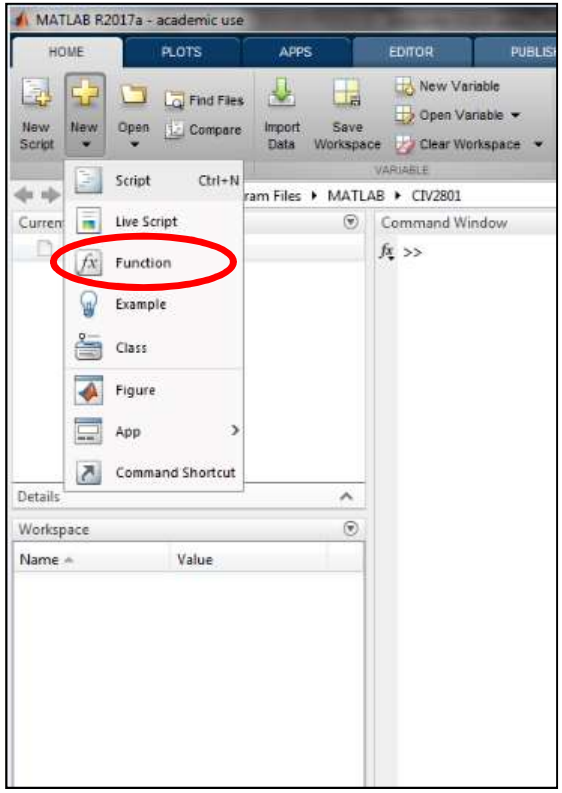

A linha está sinalizada em laranja pela falta de ponto-e-vírgula.

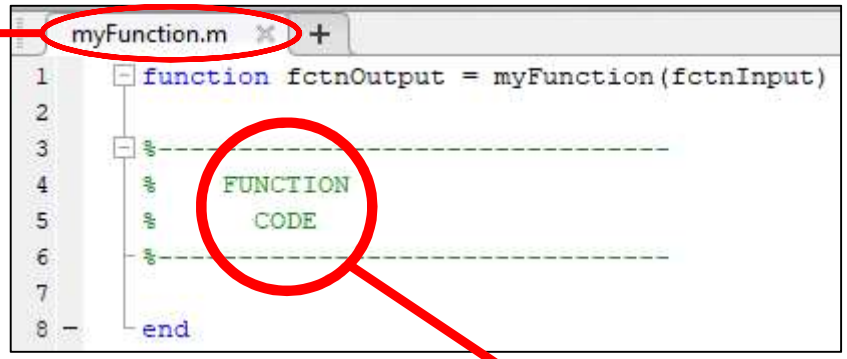
Esta barra inidica avisos e erros. O sinal verde significa que não foram identificados problemas, laranja significa que existem avisos (o código pode ter problemas para rodar), vermelho significa que existem erros (o código não irá rodar).

# Introdução ao MATLAB

- **Definição de funções**

É possível, no MATLAB, salvar blocos de código em arquivos texto com a extensão *.m*, para que sejam chamados e reutilizados posteriormente, são chamados de funções. Basta definir e salvar a função em um *script*, como demonstado abaixo.




O arquivo *.m* deve ser salvo com o mesmo nome da função.

Todas as variáveis declaradas no interior de uma função são perdidas após sua execução, a menos que sejam retornadas pela mesma.

# Introdução ao MATLAB

- **Exemplo de definição de função**

getPolyIntsect.m

```
%% getPolyIntsect
% Esta função calcula as posições das interseções entre duas curvas 2D a
% partir de polinômios que as descrevem, dados em forma vetorial.
% Ex.: p(x) = x³ - 2x² + 4x - 1 -> p = [1, -2, 4, -1]
%
% INPUT:
% * p1 -> vetor da primeira curva
% * p2 -> vetor da segunda curva
% OUTPUT:
% * intsects -> Coordenadas e valores dos polinômios nas interseções
%               [ x , p(x) ]
%
%% ------------------------------------------------------------------------
function intsects = getPolyIntsect(p1,p2)

% Obtém ordens dos polinômios
o1 = size(p1,2) - 1;
o2 = size(p2,2) - 1;

% Computa a maior ordem e identifica o polinômio de maior ordem
[order,whichPoly] = max([o1,o2]);

% Define vetor auxiliar para compatibilizar dimensões de p1 e p2
aux_p = zeros(1,order+1);

% Redefine o polinômio de menor ordem
switch whichPoly
    case 1
        aux_p(-o2+end:end) = p2;
        p2 = aux_p;
    case 2
        aux_p(-o1+end:end) = p1;
        p1 = aux_p;
end

% Descarta vetor auxiliar
clear aux_p

```

Sumário da função. Não é obrigatório para o funcionamento da mesma, ainda assim, é importante para o entendimento de usuários futuros sobre o código que será utilizado.

A função *max* retorna [valor máx., índice do valor máx.].

Continua no próximo slide.

# Introdução ao MATLAB

- **Exemplo de definição de função**

```
39      % Define o polinômio da diferença entre p1 e p2
40 -    p = p2 - p1;
41
42      % Calcula raízes de p (ou seja, as coordenadas das interseções)
43 -    intsects = roots(p);
44
45      % Checa se existem interseções
46 -    if ~isempty(intsects)
47          % Descarta interseções complexas
48 -        stop = 0; % inicia 'flag' auxiliar
49 -        n = 1; % inicia contador auxiiar
50 -        while stop == 0
51              % Checa se a coordenada da n-ésima interseção é complexa
52 -            if ~isreal(intsects(n))
53 -                intsects(n) = []; % Deleta n-ésima interseção do vetor intsects
54 -                n = n - 1; % Atualiza o contador
55 -            end
56              % Checa se o contador chegou ao tamanho do vetor intsects
57 -            if n == size(intsects,1) || isempty(intsects)
58 -                stop = 1; % Sinaliza o fim do 'while'
59 -            else
60 -                n = n + 1; % Atualiza o contador
61 -            end
62 -        end
63 -    end
64
65      % Avalia o valor dos polinômios nas posicões (x) das interseções
66 -    if ~isempty(intsects)
67 -        intsects(:,2) = polyval(p1,intsects);
68 -    end
69 -    end
```

A função *isempty* checa se a variável é vazia, retornando um valor lógico (0 ou 1).

A função *isreal* checa se a variável é real, retornando um valor lógico (0 ou 1).

# Introdução ao MATLAB

- **Uso de funções**

Quaisquer funções definidas podem ser chamadas no *prompt*, em *scripts* ou mesmo por outras funções, pelo nome da mesma seguido de seus argumentos de *input*.

```
>> intersections = getPolyIntsect([1 0 0],[5 -6]) % x² = 5x - 6

intersections =

    3.0000    9.0000
    2.0000    4.0000
```

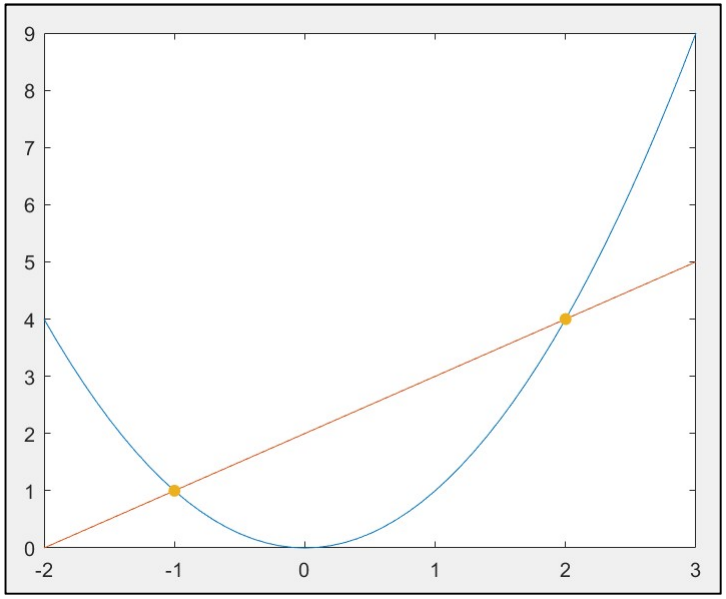

A função chamada deve estar no diretório corrente, ou um caminho até a mesma deve ser especificado por meio do comando *addpath*.

# Introdução ao MATLAB

- **Funções de plotagem**

O MATLAB oferece funcionalidades de desenho e plotagem de relativo fácil uso. Comandos como *plot* permitem a visualização de curvas e gráficos, bem como a modelagem gráfica de objetos.

```
>> pol_1 = [1, 0, 0]; % x²
>> pol_2 = [1, 2];    % x + 2
>> intersections = getPolyIntsect(pol_1,pol_2); % x² = x + 2
>> x = -2:0.1:3;
>> plot(x,polyval(pol_1,x))
>> hold on
>> plot(x,polyval(pol_2,x))
>> hold on
>> scatter(intersections(:,1),intersections(:,2),'filled')
```




A função *plot* tem como entrada vetores *x* e *y*, onde cada par de coordenadas $(x_i,y_i)$ representa um ponto a ser conectado por retas.

O comando *hold on* garante que o último *plot* seja mantido no *canvas* (eixos de plotagem).

A função *scatter* tem como entrada vetores *x* e *y*, onde cada par de coordenadas $(x_i,y_i)$ representa um ponto a ser plotado. A especificação *'filled'* faz com que esses pontos sejam círculos cheios.

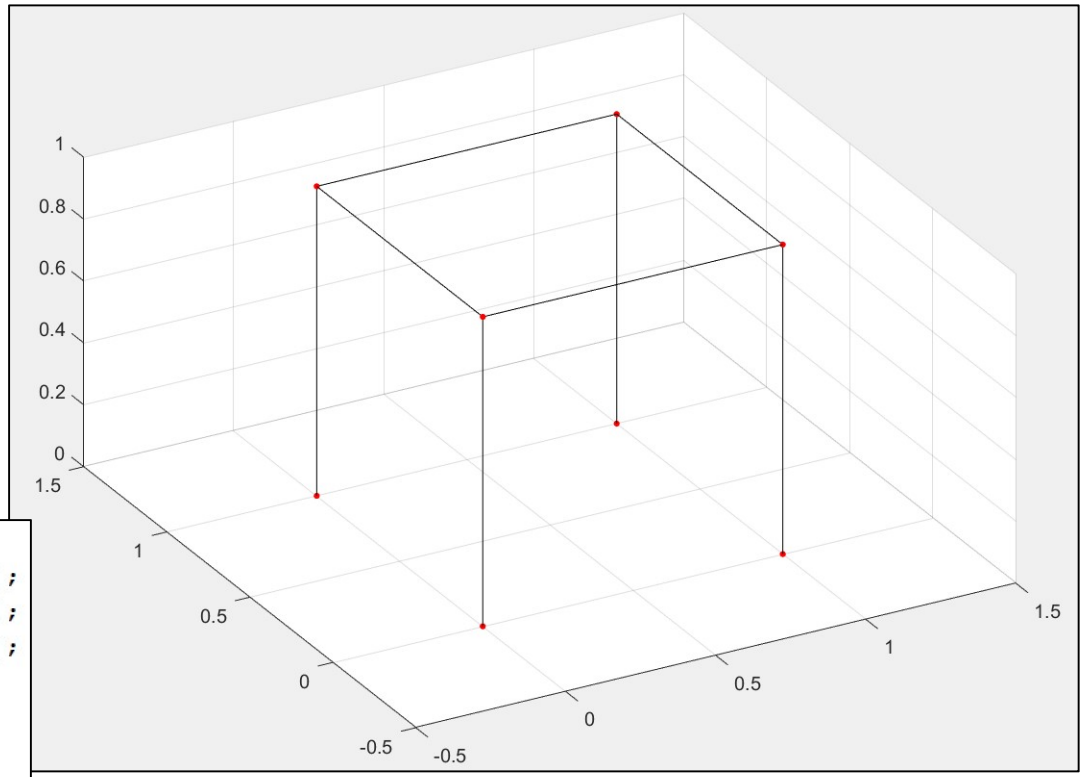
# Introdução ao MATLAB

- **Funções de plotagem**

As funções gráficas do MATLAB podem ser usadas para visualizar modelos estruturais, como, por exemplo, um pórtico tridimensional.

```
>> nodes = [0 0 0;     % [x y z]
            0 0 1;
            1 0 1;
            1 0 0;
            0 1 0;
            0 1 1;
            1 1 1;
            1 1 0];
>> elems = [1 2;       % [initial node, final node]
            2 3;
            3 4;
            5 6;
            6 7;
            7 8;
            2 6;
            3 7];
```

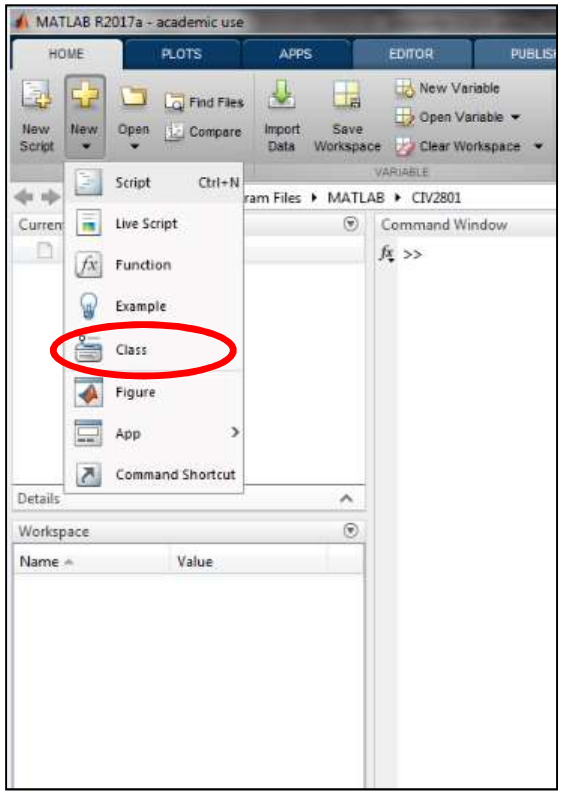
```
>> for i = 1:size(elems,1)
      x = [nodes(elems(i,1),1), nodes(elems(i,2),1)];
      y = [nodes(elems(i,1),2), nodes(elems(i,2),2)];
      z = [nodes(elems(i,1),3), nodes(elems(i,2),3)];
      plot3(x,y,z,'color',[0 0 0])
      hold on
      scatter3(x,y,z,10,[1 0 0],'filled')
      hold on
      if i == size(elems,1)
         axis equal
         axis([-0.5 1.5 -0.5 1.5 0 1])
         grid on
      end
   end
```



# Introdução ao MATLAB

- **Definição de classes**

É possível trabalhar com o paradigma da *programação orientada a objetos* no MATLAB. Classes podem ser definidas, como exposto abaixo, para detalhar propriedades e métodos de objetos.




```
myClass.m
1  classdef myClass < handle
2      properties
3          props = [];
4          %-------------------------------------
5          %            myClass
6          %           Properties
7          %-------------------------------------
8      end
9
10     methods
11         % Constructor method
12         function object = myClass(inputProperties)
13             object.props = inputProperties;
14         end
15         %-------------------------------------
16         %            myClass
17         %            Methods
18         %-------------------------------------
19         % Destructor method
20         function clean(object)
21             object.props = [];
22         end
23     end
24 end
```

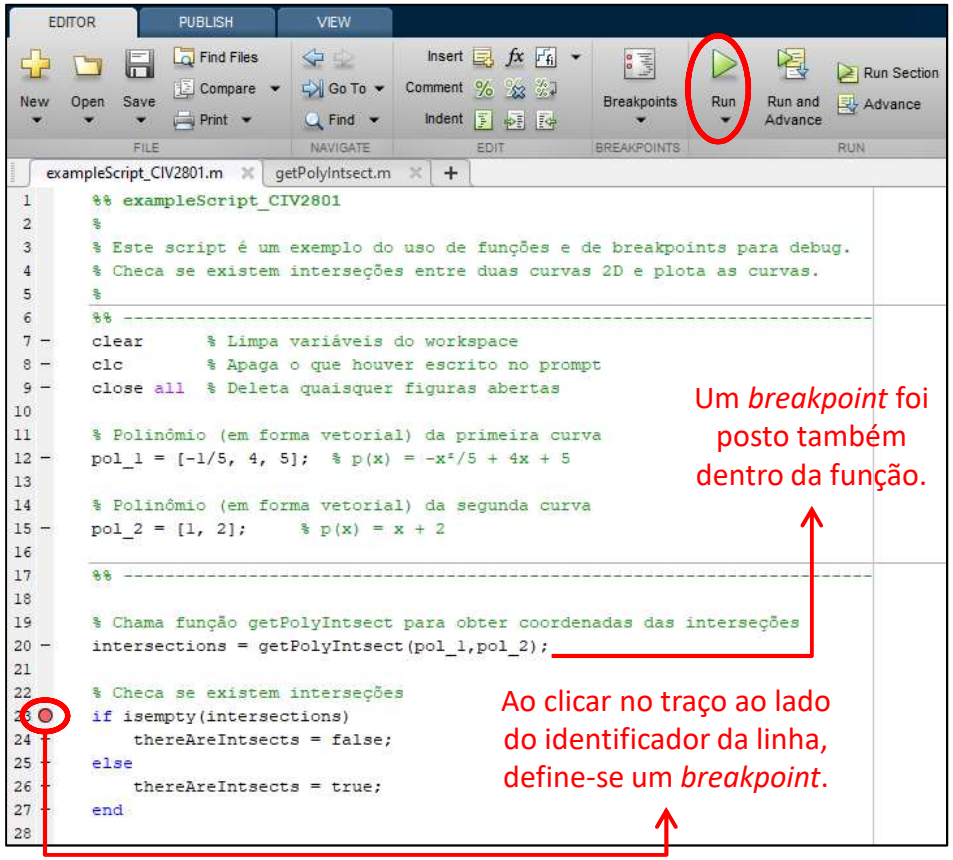
O arquivo *.m* deve ser salvo com o mesmo nome da classe.

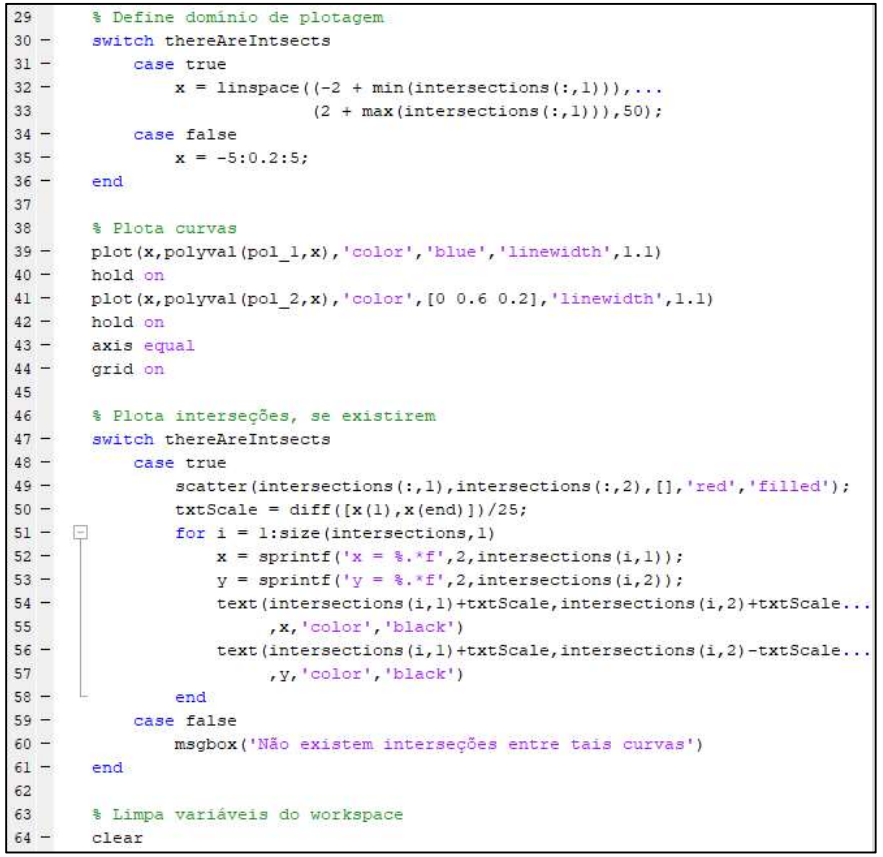
Estrutura do MATLAB onde as classes são definidas.

# Introdução ao MATLAB

- **Debug**

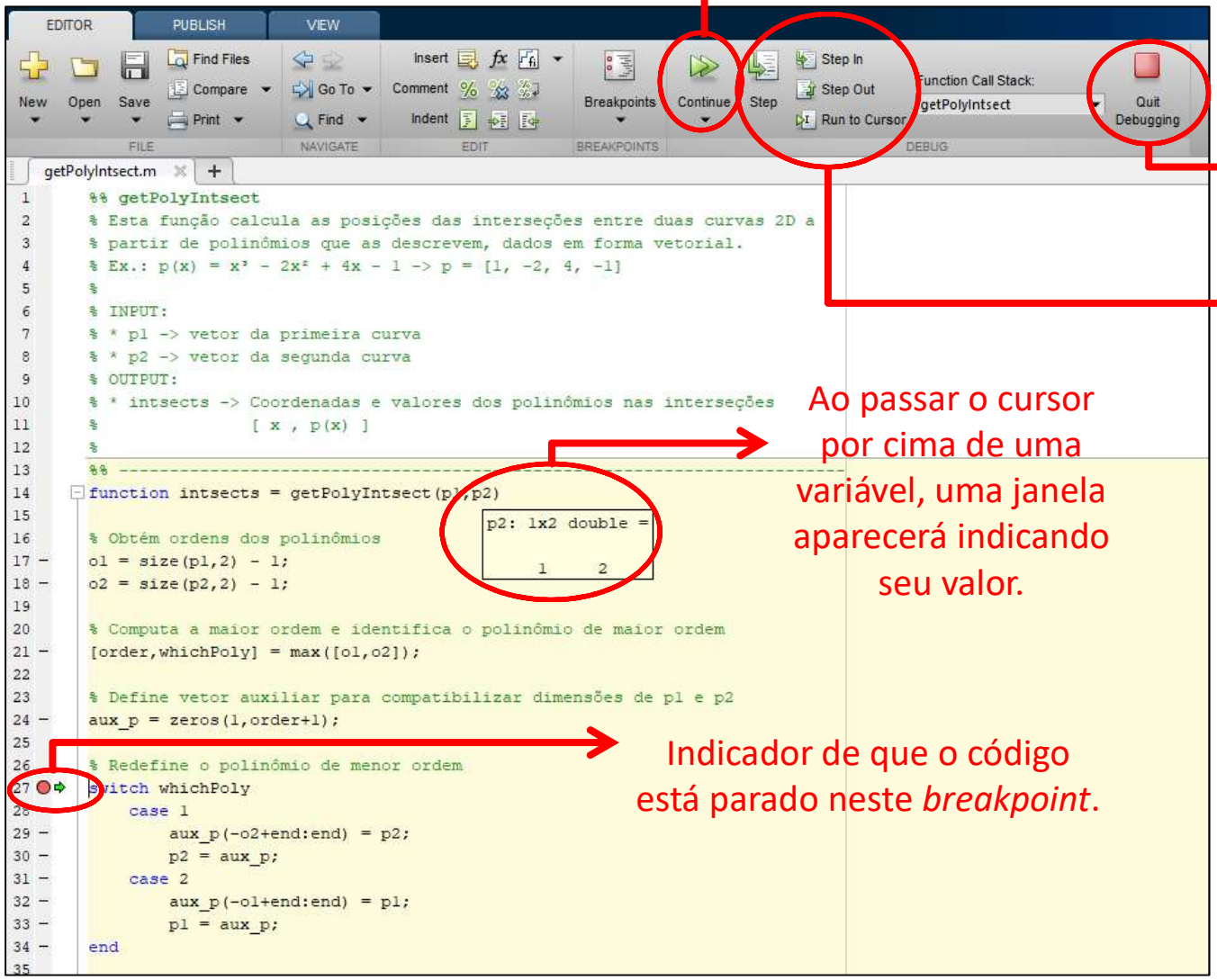
Uma das grandes vantagens providas pelo MATLAB é a sua ferramenta de *debug*. A mesma consiste em um conjunto de recursos que permitem analisar blocos de códigos pausadamente, por meio de *breakpoints*, com o intuito de identificar eventuais erros no código. Tal funcionalidade pode ser utilizada em quaisquer códigos que descrevam ações sequenciais, como *scripts*, *funções* e *métodos* de *classes*, seu uso pode ser visto no *script* a seguir.



# Introdução ao MATLAB

- **Debug**




Continua até o próximo ponto, ou até o fim do código, se não houver mais *breakpoints*.

Interrompe o código no ponto em que parou.

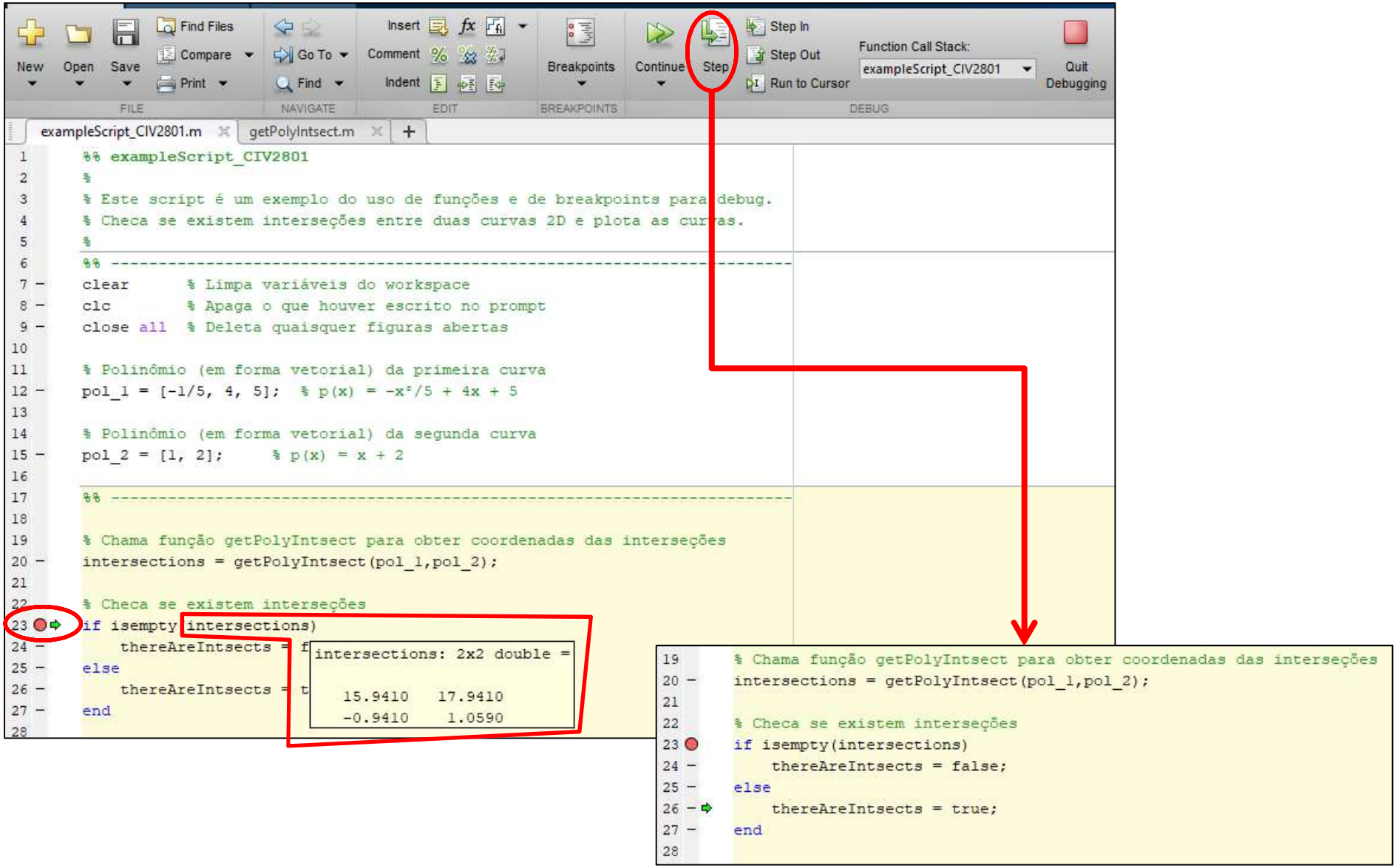
Continua na próxima linha, dentro ou fora de funções, ou segue até a posição do cursor.

Ao passar o cursor por cima de uma variável, uma janela aparecerá indicando seu valor.

Indicador de que o código está parado neste *breakpoint*.

# Introdução ao MATLAB

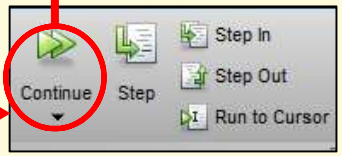
- **Debug**

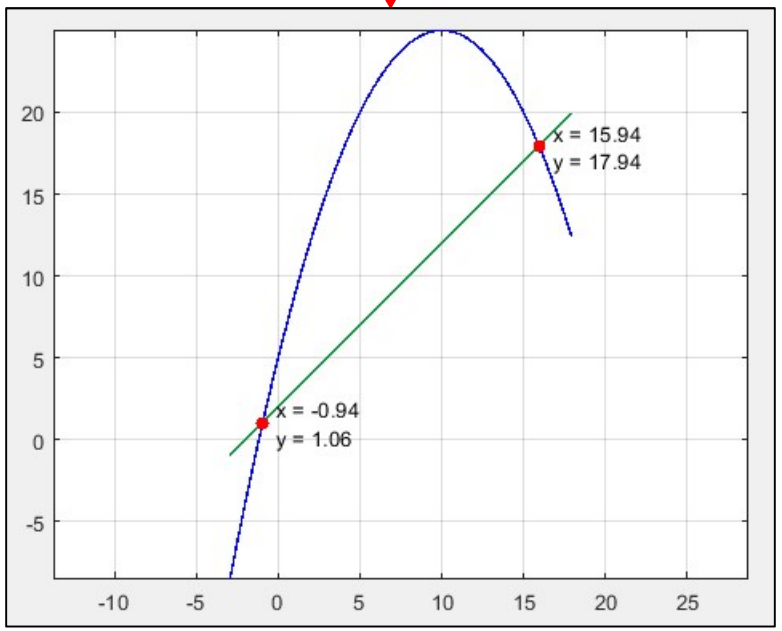


# Introdução ao MATLAB

- **Debug**

```
22      % Checa se existem interseções
23      if isempty(intersections)
24          thereAreIntsects = false;
25      else
26          thereAreIntsects = true;
27      end
28
29      % Define domínio de plotagem
30      switch thereAreIntsects
31          case true
32              x = linspace((-2 + min(intersections(:,1))),...
33                           (2 + max(intersections(:,1))),50);
34          case false
35              x = -5:0.2:5;
36      end
37
38      % Plota curvas
39      plot(x,polyval(pol_1,x),'color','blue','linewidth',1.1)
40      hold on
41      plot(x,polyval(pol_2,x),'color',[0 0.6 0.2],'linewidth',1.1)
42      hold on
43      axis equal
44      grid on
45
46      % Plota interseções, se existirem
47      switch thereAreIntsects
48          case true
49              scatter(intersections(:,1),intersections(:,2),[],'red','filled');
50              txtScale = diff([x(1),x(end)])/25;
51              for i = 1:size(intersections,1)
52                  x = sprintf('x = %.*f',2,intersections(i,1));
53                  y = sprintf('y = %.*f',2,intersections(i,2));
54                  text(intersections(i,1)+txtScale,intersections(i,2)+txtScale...
55                       ,x,'color','black')
56                  text(intersections(i,1)+txtScale,intersections(i,2)-txtScale...
57                       ,y,'color','black')
58              end
59          case false
60              msgbox('Não existem interseções entre tais curvas')
61      end
62
63      % Limpa variáveis do workspace
64      clear
```



# Introdução ao MATLAB

- **Links Úteis**

Lista de funções pré-definidas:
https://www.mathworks.com/help/matlab/functionlist.html

Tutorial MATLAB:
https://www.tutorialspoint.com/matlab/index.htm