



SUMÁRIO

1 - Apresentação do software	2
2 - Conhecendo o ambiente do aplicativo.....	2
3 - Funções básicas	3
4 - Operações com vetores	5
5 - Operações com matrizes.....	8
6 - Números complexos	11
7 - Noções gerais de polinômios e sistemas de equações lineares	12
7.1 – Polinômios.....	12
7.1.1 - Definindo-se um polinômio no MATLAB	12
7.1.2 - Cálculo das raízes de um polinômio	13
7.1.3 - Calculando os coeficientes de um polinômio partindo de suas raízes.....	13
7.1.4 - Visualizando as raízes de um polinômio.	13
7.1.5 - Operações com polinômios	15
7.2 - Sistemas de equações lineares.....	16
7.3 - Encontrando a melhor curva partindo de dados experimentais.....	17
8 – Análise Numérica	18
8.1 – Integração	18
8.2 – Diferenciação	19
8.3 – Equações Diferenciais	19
9 - Gráficos em 2D.....	20
9.1 - Funções gráficas elementares.....	20
9.2 - Plotando um vetor	21
9.3 - Plotando um número complexo	21
9.4 - Plotando uma matriz.....	21
9.5 - Plotando um vetor em função de outro	22
9.6 - Plotagens combinadas	22
9.6.1 - Usando a mesma linha de comando.....	22
9.6.2 - Adicionando linhas a um gráfico já existente.....	23
9.7 - Ajustando os limites de escala	23
9.7.1 - Corrigindo distorções	23
9.8 - Escondendo eixos e escalas	23
9.9 - Apagando figuras da janela gráfica	23
9.10 - Trabalhando com múltiplas janelas gráficas.....	24
9.11 - Construindo subgráficos	24
9.12 - Ampliando a visualização gráfica - ZOOM	24
9.13 - Plotando funções com o MATLAB	25
9.14 – Outras formas de gráficos.....	25
9.14.1 - Gráficos de barras.....	25
9.14.2 - Escala logarítmica	26
9.14.3 - Gráficos de variáveis discretas	26
9.14.4 - Gráficos em escada	27
9.14.5 - A Função compass.....	27
10 - Programando com o MATLAB	28
10.1 - Operações relacionais e lógicas	28
10.2 - Rotinas especiais de programação.....	30
10.3 - Estruturas de controle repetitivas (loops).....	30
10.4 - Estrutura de controle condicional.....	33
10.5 - Construindo funções com o MATLAB	36
10.6 - Debugando com o MATLAB.....	37
11 - Exercícios.....	39
12 - Referências bibliográficas	43
13 – Anexo: Listagem de Comandos.....	44



1 - Apresentação do software

O software MATLAB (**MATLAB** é marca registrada de **The MathWorks, Inc.**) foi originalmente desenvolvido para ser um "Laboratório para estudo de Matrizes", porém atualmente suas capacidades ultrapassam em muitas vezes as possibilidades de sua versão original. Desta forma, o MATLAB é hoje um dos principais sistemas interativos e uma das mais importantes linguagens de programação para computação técnica e científica em geral.

Sua primeira versão foi escrita na Universidade do Novo México e na Universidade de Stanford, no final da década de 70, e destinava-se a cursos de teoria matricial, álgebra linear e análise numérica. Hoje, o MATLAB integra análise numérica, cálculo com matrizes, processamento de sinais, imagens, sons, e construção de gráficos em ambiente fácil de usar onde problemas e soluções são expressos como eles são escritos matematicamente, ao contrário dos métodos de programação tradicional.

Os usos típicos do MATLAB incluem:

- ✓ Cálculos matemáticos;
- ✓ Desenvolvimento de Algoritmos;
- ✓ Modelagem, simulação e confecção de protótipos;
- ✓ Análise, exploração e visualização de dados;
- ✓ Gráficos científicos e de engenharia;
- ✓ Desenvolvimento de aplicações, incluindo a elaboração de interfaces gráficas com o usuário.

É também um sistema interativo cujo elemento básico de informação é uma matriz que não requer dimensionamento. Esse sistema permite a resolução de muitos problemas numéricos em apenas uma fração do tempo que se gastaria para escrever um programa semelhante em outra linguagem, como o Pascal e o Fortran.

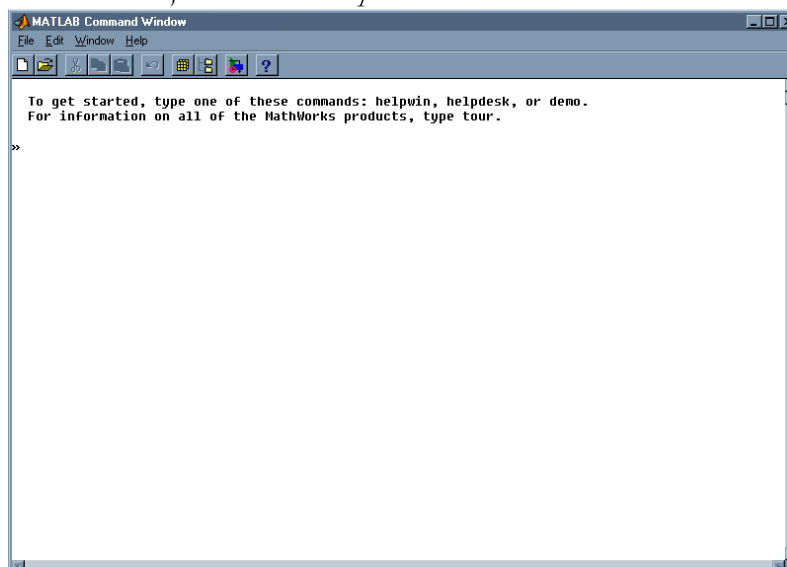
2 - Conhecendo o ambiente do aplicativo

A área de trabalho do MATLAB assemelha-se bastante ao ambiente *Windows* padrão, com a diferença que o programa oferece uma linha de *prompt* (como no antigo MS-DOS), onde o usuário pode digitar os comandos.

Os menus do MATLAB são os seguintes:

- (a) Menu FILE: permite que o usuário abra, feche e salve arquivos ou a área de trabalho. Permite ainda, que se alterem as configurações do software, que se envie arquivos para impressão, que se abra o editor de janelas e se encerre o programa;
- (b) Menu EDIT: possui as opções padrão do *Windows*, tais como undo, copy, paste. Além disso, permite que se limpe a área de trabalho;
- (c) Menu WINDOW: menu que permite a alternância entre as várias janelas abertas durante a execução do programa;
- (d) Menu HELP: contém os tópicos de ajuda e informações gerais sobre o software.

Na barra de menu, dois botões se destacam. O primeiro é o botão *Workspace Browser*, que abre uma janela contendo a relação de todas as variáveis, seus nomes, suas classes e seus tamanhos. O segundo botão é o *Path Browser*, que permite que um novo diretório seja adicionado ao *path* do MATLAB.





A área de trabalho (*prompt*), serve para entrada de comandos e parâmetros. Não há a necessidade de declararmos qualquer variável no MATLAB. A variável (ou expressão) passa a existir a partir do momento em que é definida pelo usuário, em tempo real de utilização. Um exemplo de definição de uma variável numérica segue abaixo:

```
» x = 3.1416927
```

Uma variável também pode guardar um *string*, como mostrado abaixo:

```
» y = 'teste'
```

Observe que a diferenciação entre um *string* e uma função é feita apenas pelos apóstrofes.

Ao necessitarmos de ajuda sobre qualquer função do MATLAB, basta que digitemos `help 'nome do comando ou função'`. Este procedimento mostra na área de trabalho uma breve explicação sobre o comando em questão. Por exemplo, experimente digitar `help help`.

As mensagens de erro, quando acontecem, tentam explicar os motivos pelo qual o erro ocorreu. Por exemplo:

```
» hep
??? Undefined function or variable 'hep'.
```

As regras de precedência aritmética são válidas também no MATLAB, ou seja, pela ordem temos: potenciação – multiplicação e divisão – adição e subtração. Os símbolos utilizados para representação destas operações são:

^	Exponenciação
/	Divisão à direita
\	Divisão à esquerda
*	Multiplicação
+	Adição
-	Subtração
'	Transposição

Deve-se notar que existem dois símbolos para divisão: as expressões $1/4$ e $4\backslash 1$ possuem o mesmo valor numérico, isto é, 0,25. Parênteses são usados em sua forma padrão para alterar o mesmo a precedência usual dos operadores aritméticos.

NOTA: O MATLAB faz distinção entre os caracteres maiúsculos e minúsculos.

3 - Funções básicas

As tabelas abaixo apresentam algumas funções básicas do MATLAB.

Tabela 1 – Utilitários para a janela de comandos

COMANDOS	DESCRIÇÃO
<code>format</code>	Altera o formato dos dados na tela
<code>disp</code>	Mostra matriz ou texto na tela
<code>clc</code>	Apaga janela de comandos
<code>clear</code>	Apaga variáveis
<code>home</code>	Move o cursor para o topo da tela
<code>echo</code>	Ativa/desativa exibições de linha individuais durante a execução de um arquivo .m
<code>quit</code>	Termina o programa



Tabela 2 – Utilitários para tratamento de arquivos

COMANDOS	DESCRIÇÃO
cd	Muda de diretório
delete	Deleta arquivo ou objeto gráfico
dir	Mostra diretório
exist	Confere se uma variável ou função existe
load	Carrega variáveis gravadas em disco
save	Salva variáveis em disco
type	Lista o conteúdo de um arquivo ou função
what	Mostra os nomes dos arquivos .m e .mat no diretório corrente
who	Mostra as variáveis existentes na tela de comando
chdir	Muda o diretório de trabalho

Tabela 3 – Caracteres especiais usados na janela de comandos

CARACTERES	DESCRIÇÃO
=	Comando de atribuição
[]	Delimitar elementos de matrizes e vetores
()	Alternar a ordem de precedência das expressões aritméticas
.	Ponto decimal
,	Separa argumento de funções e elementos de matrizes e vetores
;	Finalizador de linha com supressão de impressão
%	Comentário
:	Geração de um vetor com intervalos definidos
!	Execução de um programa do sistema operacional

Tabela 4 – Funções matemáticas básicas

FUNÇÃO	DESCRIÇÃO
acos	Arco-coseno
asin	Arco-seno
atan	Arco-tangente
cos	Coseno
cosh	Coseno hiperbólico
exp	Exponencial com base e
fix	Arredondamento para o inteiro mais próximo até zero
log	Logaritmo natural
log10	Logaritmo decimal
rand	Gera números aleatórios com distribuição uniforme
randn	Gera números aleatórios com distribuição normal
rat	Aproximação racional
round	Arredonda o número para o inteiro mais próximo
sign	Retorna 1 se for positivo e 0 se for negativo ou zero
sin	Seno
sinh	Seno hiperbólico
sqrt	Raiz quadrada
tan	Tangente
tanh	Tangente hiperbólica

NOTA: O MATLAB trabalha com ângulos em radianos.



4 - Operações com vetores

Chamamos de vetor \mathbf{u} a um conjunto de elementos $u_1 u_2 u_3 \dots u_n$ dispostos em linha ou em coluna. Estes elementos, são os componentes do vetor \mathbf{u} .

No MATLAB, os vetores devem ser declarados entre colchetes. Veja o exemplo abaixo:

```
» [0 1 2 3]
ans =
    0    1    2    3
```

O mesmo resultado seria obtido com:

```
» [0,1,2,3]
ans =
    0    1    2    3
```

Como visto no exemplo acima, quando não declaramos a variável que irá conter os componentes do vetor, o vetor é devolvido como uma variável chamada *ans* (procedimento *default* do MATLAB).

As operações aritméticas básicas são realizadas elemento a elemento do vetor. Por exemplo:

```
» u = [1 2 3];
» z = 3+u
z =
    4    5    6
```

Se quisermos realizar qualquer operação com apenas um elemento do vetor, basta que especifiquemos qual dos elementos deverá ser ‘tratado’. Por exemplo:

```
» z(3) = -1
z =
    4    5   -1
```

Outro aspecto importante no MATLAB é a flexibilidade de podermos realizar uma atribuição de um novo valor para um elemento (ou variável) utilizando a própria variável nesta operação. Um exemplo simples seria:

```
» z(3) = z(3) + 1
z =
    4    5    0
```

Se você quiser saber o número de elementos de um vetor, utilize a função *length*.

```
» length(z)
ans =
    3
```

A transposta do vetor é calculada utilizando-se o símbolo ‘ $'$ ’. Para obtermos um vetor coluna a partir de z , basta digitarmos o seguinte:

```
» w = z'
w =
    4
    5
    0
```



Um detalhe importante: o símbolo `*` executa o produto escalar entre dois vetores, desde que um deles seja um vetor linha e o outro um vetor coluna. Vejamos a seguir:

```
» esc = u * z'  
esc =  
14
```

Entretanto, algumas operações especiais precisam ser definidas quando queremos realizar operações entre dois vetores. Tais operações são definidas pelos seguintes símbolos:

OPERADORES	OPERAÇÕES
<code>.*</code>	Multiplicação
<code>./</code>	Divisão à direita
<code>.\</code>	Divisão à esquerda
<code>.^</code>	Exponenciação

Um exemplo de multiplicação seria:

```
» mult = u .* z  
mult =  
4 10 0
```

Um exemplo de divisão:

```
» div = u ./ z  
Warning: Divide by zero.  
div =  
0.2500 0.4000 Inf
```

Neste exemplo, ocorreu um fato interessante. O MATLAB trabalha com divisões por zero, retornando, nestes casos, com o símbolo “Inf” para infinito. O problema ocorrido nesta divisão poderia ser facilmente evitado se utilizássemos a divisão à esquerda, como segue:

```
» div = u .\ z  
div =  
4.0000 2.5000 0
```

A potenciação de um vetor por um escalar pode ser feita como segue:

```
» div = div.^ 3  
div =  
64.0000 15.6250 0
```

O MATLAB permite a geração de vetores usando intervalos. Imaginemos que vamos precisar trabalhar com um vetor igualmente espaçado de $\pi/4$ em $\pi/4$ até o valor π . Para gerar este vetor, utilizaríamos o seguinte comando:

```
» x = 0 : pi/4 : pi  
x =  
0 0.7854 1.5708 2.3562 3.1416
```

O vetor poderia ser criado com incrementos negativos:

```
» y = pi : -pi/4 : 0  
y =  
3.1416 2.3562 1.5708 0.7854 0
```

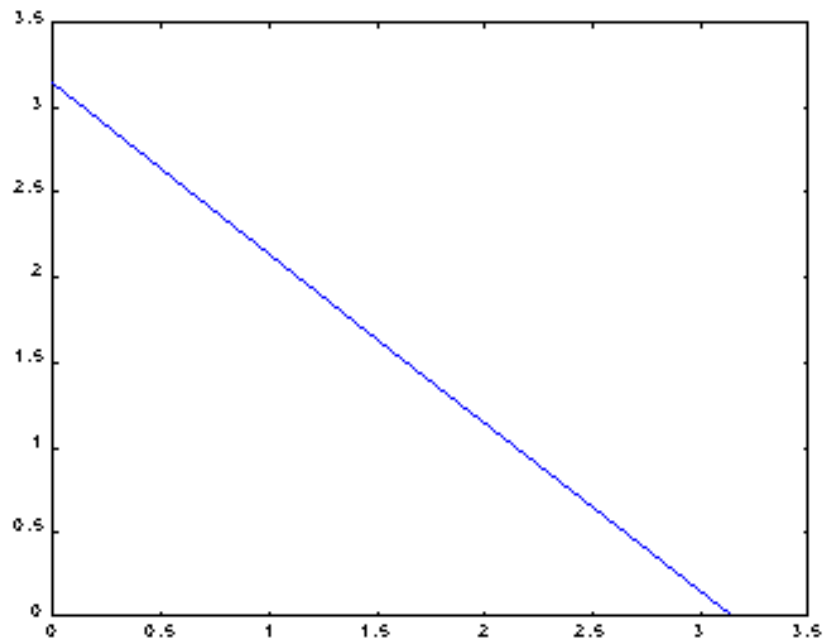
Aproveitando estes dois vetores, x e y , vamos realizar a gora uma plotagem simples. O comando de plotagem no MATLAB é o comando *plot*. Como sabemos, para a construção de um gráfico necessitamos de dois parâmetros que



irão compor os valores dos eixos coordenados horizontal e vertical. No MATLAB, o primeiro parâmetro da função *plot* é considerado como contendo os valores do eixo horizontal e o segundo como contendo os valores do eixo vertical. Assim, o comando para fazer a plotagem de y em função de x seria:

```
» plot (x,y)
```

O resultado obtido é uma nova janela, contendo o gráfico do vetor y em função do vetor x, que como esperado, é uma reta.



Algumas funções do MATLAB são bastante úteis no tratamento de vetores. Exemplos delas são as funções *max*, *min* e *mean*. A função *max* devolve o maior dos componentes do vetor. A função *min*, por sua vez, devolve o menor dos componentes do vetor.

```
» max(z)
ans =
    5
```

```
» min(z)
ans =
    0
```

A função *mean* devolve o valor médio dos componentes do vetor.

```
» mean(z)
ans =
    3
```

Dois vetores distintos podem ser concatenados (encadeados) para formar um novo vetor de maneira bastante simples. Vejamos abaixo:

```
» conc = [u z]
conc =
    1    2    3    4    5    0
```

Para concatenarmos dois vetores coluna devemos utilizar o ';' para separar os dois vetores que serão concatenados. Todas as operações vistas para vetores linha também são válidas para vetores coluna. Maiores detalhes serão vistos nos próximos itens.

5 - Operações com matrizes

Uma matriz é um arranjo de elementos na forma de uma tabela retangular de elementos, sendo que a forma geral é um arranjo de **m** linhas e **n** colunas. No MATLAB, uma matriz pode ser definida de maneira semelhante aos vetores, diferenciando-se apenas no fato da necessidade da digitação de um 'enter' ou um ';' para a separação das diferentes colunas. Vejamos alguns exemplos:

```
» a = [1 2 3; 4 5 6; 7 8 9]
a =
     1     2     3
     4     5     6
     7     8     9
```

Ou:

```
» b = [9 8 7
        6 5 4
        3 2 1]
b =
     9     8     7
     6     5     4
     3     2     1
```

Da mesma forma que fizemos com vetores, as matrizes também podem ter seus elementos escritos como expressões. Além disso, as matrizes podem ter seus elementos identificados (e operados) individualmente. Por exemplo:

```
» b(2,1)
ans =
     6
```

Este comando devolveu em *ans* o valor do elemento da segunda linha e primeira coluna da matriz **b**. O tamanho de uma matriz qualquer pode ser obtido através da função *size*.

```
» size(a)
ans =
     3     3
```

A matriz **a** possui 3 linhas e 3 colunas. A concatenação de matrizes é bastante semelhante à concatenação dos vetores. Vejamos dois exemplos:

```
» c = [a b]
c =
     1     2     3     9     8     7
     4     5     6     6     5     4
     7     8     9     3     2     1

» d = [a; b]
d =
     1     2     3
     4     5     6
     7     8     9
     6     5     4
     3     2     1
```

Para extrairmos uma submatriz de uma matriz qualquer procedemos da seguinte forma:

```
» e = d(1:2, 1:2)
e =
     1     2
     4     5
```

Este comando retirou a primeira e segunda linhas e a primeira e segunda colunas da matriz **d** e armazenou na matriz **e**. Se quisermos que todas as linhas da matriz antiga compusessem a nova matriz bastaria colocarmos ':' no primeiro parâmetro da matriz **d**. Suponhamos agora que precisemos inserir uma linha na matriz **e**. O comando é o seguinte:

```
» e = [e; 1 0]
```




```
e =  
    1    2  
    4    5  
    1    0
```

Nas operações de adição e subtração de matrizes, a exemplo do que aconteceu quando trabalhamos com vetores, os elementos de uma matriz são somados ou subtraídos com o seu correspondente na outra matriz. Esta é a razão pela qual as duas matrizes envolvidas devem ter o mesmo número de linhas e colunas.

```
» soma = a + b  
soma =  
    10    10    10  
    10    10    10  
    10    10    10
```

```
» sub = a - b  
sub =  
    -8    -6    -4  
    -2     0     2  
     4     6     8
```

A multiplicação e divisão de uma matriz por escalares é efetuada elemento a elemento da matriz. Por exemplo:

```
» mult = 3 * a  
mult =  
     3     6     9  
    12    15    18  
    21    24    27
```

```
» div = a / 3  
div =  
    0.3333    0.6667    1.0000  
    1.3333    1.6667    2.0000  
    2.3333    2.6667    3.0000
```

A exponenciação individual dos elementos de uma matriz podem ser feitos pelo comando ' \wedge '.

```
» a.^2  
ans =  
     1     4     9  
    16    25    36  
    49    64    81
```

A multiplicação de matrizes é uma ferramenta bastante útil no MATLAB. Vale a pena ressaltar que na multiplicação de uma matriz **a** qualquer por uma matriz **b** qualquer, só pode acontecer se o número de colunas da matriz **a** for igual ao número de linhas da matriz **b** ou vice-versa. Vejamos um exemplo: Queremos multiplicar a matriz **c** pela matriz **d**. Primeiramente vamos verificar se o número de linhas de uma é igual ao número de colunas da outra.

```
» size (c)  
ans =  
     3     6
```

```
» size (d)  
ans =  
     6     3
```

Percebe-se que a multiplicação destas duas matrizes é possível. Então:

```
» c * d  
ans =  
    180    162    144  
    162    162    162  
    144    162    180
```

Perceba que o resultado teria sido diferente se tivéssemos feito **d** vezes **c**:

```
» d * c  
ans =  
     30     36     42     30     24     18  
     66     81     96     84     69     54  
    102    126    150    138    114     90
```



```
90 114 138 150 126 102
54 69 84 96 81 66
18 24 30 42 36 30
```

Este tipo de preocupação não é necessária quando estamos trabalhando com matrizes quadradas de mesmo tamanho.

A fim de mostrar a divisão de duas matrizes, vamos definir duas novas matrizes:

```
» x = [15 10 8; 7 1 0; 2 5 1]
x =
    15    10     8
     7     1     0
     2     5     1

» y = [3 -1 2; -5 1 1; 0 3 4]
y =
     3    -1     2
    -5     1     1
     0     3     4
```

Agora, vamos dividir a matriz **x** pela matriz **y**:

```
» x/y
ans =
   -2.0213   -4.2128    4.0638
   -0.5745   -1.7447    0.7234
   -1.8511   -1.5106    1.5532
```

Note que a divisão de **y** por **x** poderia ser efetuada utilizando-se a divisão à direita, como segue:

```
» x\y
ans =
   -0.7033    0.0239         0
   -0.0766    0.8325    1.0000
    1.7895   -1.2105   -1.0000
```

Houve um motivo simples para termos definido novas matrizes a fim de demonstrarmos a divisão. As matrizes **a** e **b** eram singulares, ou seja, possuíam determinante nulo. Ao dividirmos duas matrizes singulares, o resultado é uma matriz com componentes infinitos, conforme nos mostra o MATLAB:

```
» a/b
Warning: Matrix is singular to working precision.
ans =
   Inf   Inf   Inf
   Inf   Inf   Inf
   Inf   Inf   Inf
```

A potenciação de matrizes equivale a sucessivas multiplicações dela por ela mesma. Por exemplo, façamos **a** ao cubo:

```
» a ^ 3
ans =
    468    576    684
   1062   1305   1548
   1656   2034   2412
```

A transposta de uma matriz (troca das colunas pelas linhas) é obtida da mesma maneira que a transposta de um vetor. O operador é o apóstrofo ':

```
» a'
ans =
     1     4     7
     2     5     8
     3     6     9
```



Na próxima tabela são mostrados as mais importantes funções para tratar matrizes no MATLAB:

FUNÇÕES	DESCRIÇÃO
det	Determinante de uma matriz
eye	Gera uma matriz identidade
inv	Calcula a inversa da matriz
ones	Gera uma matriz unitária
rand	Gera uma matriz randômica
tril	Transforma/gera uma matriz triangular inferior
triu	Transforma/gera uma matriz triangular superior
zeros	Gera uma matriz de zeros (nula)

6 - Números complexos

Veremos agora como trabalhar com números complexos no MATLAB. Aparte imaginária é simbolizada pelas letras **i** ou **j** indistintamente:

```
» i^2
ans =
-1.0000 + 0.0000i

» j^2
ans =
-1.0000 + 0.0000i
```

Note que mesmo que utilizemos a letra **j**, na resposta oferecida pelo MATLAB sempre aparece a letra **i**.

Utilizando o sistema de coordenadas cartesianas, um número complexo pode ser definido no MATLAB da seguinte maneira:

```
» z1 = 3 + 4i
z1 =
3.0000 + 4.0000i
```

Ao trabalharmos com números complexos um ponto deve ser levado em consideração: não devemos utilizar as letras **i** e **j** para definir variáveis ou constantes, pois isto feito, elas não mais poderão ser usadas para definir complexos.

O conjugado de um número complexo (o próprio número com o sinal da parte imaginária trocado) pode ser obtido utilizando-se a função *conj*:

```
» conj(z1)
ans =
3.0000 - 4.0000i
```

As operações com números complexos utilizam os operadores usuais, como podemos ver nos exemplos a seguir:

```
» z2 = 4 + 3i
z2 =
4.0000 + 3.0000i

» z1 * z2
ans =
0 + 25.0000i

» z1 + z2
ans =
7.0000 + 7.0000i

» z1 / z2
ans =
0.9600 + 0.2800i

» z1 - z2
ans =
-1.0000 + 1.0000i
```



A potenciação, como visto com as matrizes, corresponde à multiplicação sucessiva do número por ele mesmo. Vejamos um exemplo:

```
» z1 * z1
ans =
-7.0000 +24.0000i

» z1 ^ 2
ans =
-7.0000 +24.0000i
```

Para plotarmos um número complexo, basta usarmos a função *plot* com o seguinte argumento:

```
» plot(z1)
```

O resultado será um gráfico que mostra um ponto (exatamente o ponto correspondente ao número complexo) no plano complexo.

Vejamos agora algumas funções úteis quando trabalhamos com números complexos:

FUNÇÃO	DESCRIÇÃO
real	Retorna a parte real do número complexo
imag	Retorna a parte imaginária do número complexo
abs	Retorna o módulo do vetor complexo (representação trigonométrica)
angle	Retorna o ângulo do vetor complexo (representação trigonométrica)

Os números complexos também podem formar vetores ou matrizes, como os números reais. Vejamos dois exemplos simples:

```
» vetor = [z1 z2]
vetor =
3.0000 + 4.0000i 4.0000 + 3.0000i

» matriz = [z1 z2; z2 z1]
matriz =
3.0000 + 4.0000i 4.0000 + 3.0000i
4.0000 + 3.0000i 3.0000 + 4.0000i
```

Para o cálculo da transposta existem dois operadores (') e (.'). O primeiro deles calcula a transposta do conjugado e o outro a transposta normal. Vejamos estes exemplos com a matriz definida anteriormente:

```
» matriz'
ans =
3.0000 - 4.0000i 4.0000 - 3.0000i
4.0000 - 3.0000i 3.0000 - 4.0000i

» matriz.'
ans =
3.0000 + 4.0000i 4.0000 + 3.0000i
4.0000 + 3.0000i 3.0000 + 4.0000i
```

7 - Noções gerais de polinômios e sistemas de equações lineares

7.1 – Polinômios

Polinômios são funções do tipo $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0 x^0$ onde **n** e **a** são reais e **x** pode ser real ou complexo.

7.1.1 - Definindo-se um polinômio no MATLAB

Define-se um polinômio no MATLAB colocando-se os seus coeficientes de forma ordenada em um vetor linha.

Seja o seguinte polinômio:

$$x^3 - 3x^2 + 4x - 4$$

No MATLAB ele será definido da seguinte forma:

```
» poli = [1 -3 4 -4];
```



Note que o coeficiente -4 , apesar de aparentemente não estar acompanhado da variável x , aparece entre os coeficientes. Isso ocorre porque na verdade ele está sim acompanhado de x , porém como temos x^0 que é igual a 1, torna-se por norma não representá-lo.

7.1.2 - Cálculo das raízes de um polinômio

As raízes de um polinômio $P(x)$ são os valores das variáveis para os quais a igualdade $P(x)=0$ é satisfeita. Vamos definir um outro polinômio no MATLAB;

```
» coef = [1 2 1];
```

O polinômio assim definido no MATLAB corresponde a: x^2+2x+1 .

A fim de calcularmos as raízes deste polinômio, o igualaríamos a zero e teríamos:

$$x^2+2x+1=0$$

No MATLAB, para calcular as raízes deste polinômio, devemos encontrar o valor de x para que a igualdade seja satisfeita. Para tanto, usamos o comando *roots* da seguinte forma:

```
» r = roots (coef)
r =
    -1
    -1
```

O polinômio em questão possui duas raízes iguais. Note que -1 é o valor que atribuído a x torna verdadeira a igualdade.

Suponha agora que você tenha que encontrar as raízes da equação:

$$x^3-15x=4$$

O procedimento no MATLAB é o seguinte:

```
» coef = [ 1 0 -15 -4];
» r = roots(coef)
r =
    4.0000
   -3.7321
   -0.2679
```

Nas linhas acima, note que antes de escrevermos os coeficientes do polinômio, temos que ajustar a equação igualando-se a zero. Assim:

$$x^3+15x-4=0$$

Note também que como não temos o termo x^2 colocamos zero no lugar de seu coeficiente.

7.1.3 - Calculando os coeficientes de um polinômio partindo de suas raízes

Podemos, com as raízes do polinômio **coef** utilizado anteriormente, obter os coeficientes deste mesmo polinômio utilizando a função **poly**:

```
» p = poly(r)
p =
    1.0000    0.0000   -15.0000   -4.0000
```

7.1.4 - Visualizando as raízes de um polinômio.

Seja a equação:



$$y=x^3-3x^2-6x+8$$

Deseja-se calcular os valores de x para os quais y seja igual a zero, ou seja, deseja-se calcular as raízes do polinômio e, além disso, deseja-se visualizar essas raízes graficamente.

O procedimento de cálculo das raízes do polinômio é o mesmo descrito anteriormente:

```
» coef = [ 1 -3 -6 8];  
» roots(coef)  
ans =  
    4.0000  
   -2.0000  
    1.0000
```

Uma vez definida as raízes, temos que definir o domínio de x que contém estas raízes:

```
» x=-3: .1 :5;
```

Observe que x é um vetor contendo vários números; dentre eles as raízes do polinômio.

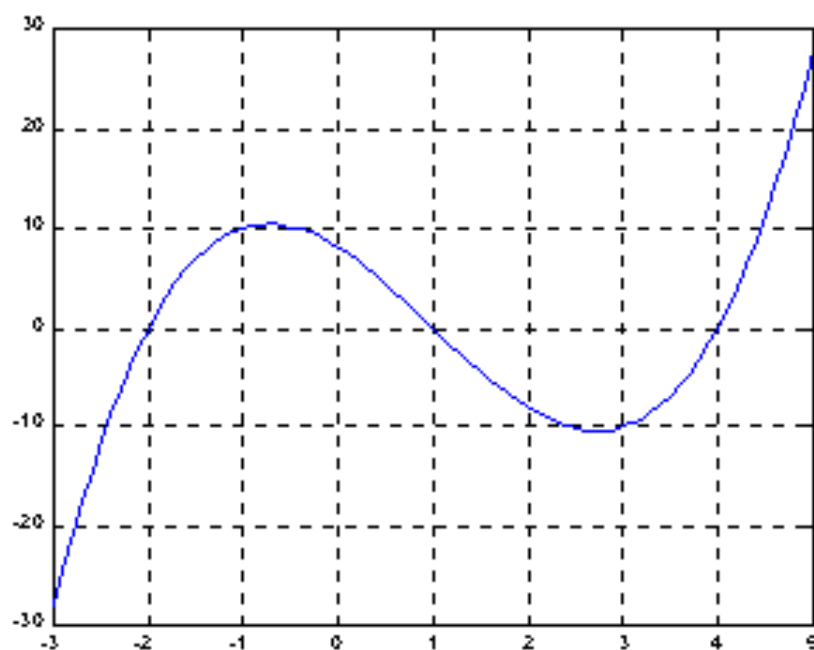
Agora vamos utilizar uma nova função que vai calcular para cada valor de x um valor correspondente para y . Essa é a função **polyval**. Ela é utilizada com dois argumentos, sendo primeiro o vetor de coeficientes e o outro é o vetor que contém as raízes que queremos *plotar*:

```
» y = polyval(coef,x);
```

Agora só nos resta fazer a *plotagem* na janela gráfica.

```
» plot(x,y)  
» grid  
» title('Gráfico de  $y=x^3-3x^2-6x+8$ ')  
» xlabel('eixo x')  
» ylabel('eixo y')  
» zoom
```

O resultado será:





Você pode observar no gráfico que, como era de se esperar, onde temos $x=4$, -2 e 1 temos $y=0$. O último comando realizado, *zoom*, permite efetuarmos um *zoom* em gráficos 2D. Utilize este recurso e realize uma aproximação na área onde a curva corta o eixo x e verifique as raízes do polinômio.

7.1.5 - Operações com polinômios

Com a mesma facilidade que operamos com números reais, vetores, matrizes e números complexos, podemos também operar com polinômios. Vejamos algumas operações possíveis:

- Multiplicação e divisão de polinômios:

Deixaremos por conta do leitor a revisão da teoria de divisão de polinômios, caso queira acompanhar os resultados obtidos no MATLAB.

Sejam dois polinômios

$$\begin{aligned}p1 &= 2s^2 + 3s + 1 \\ p2 &= 5s - 2\end{aligned}$$

Queremos $p1 \cdot p2$. Sabemos que o resultado esperado é:

$$p1 \cdot p2 = 10s^3 + 11s^2 - s - 2$$

No MATLAB teremos:

```
» p1=[2 3 1];  
» p2=[5 -2];
```

A função *conv* é utilizada para a multiplicação entre dois polinômios:

$$\begin{aligned}p3 &= \text{conv}(p1, p2) \\ p3 &= \\ 10 \quad 11 \quad -1 \quad -2\end{aligned}$$

Temos então resultado esperado.

Enquanto a função *conv* é utilizada para realizar multiplicações, o contrário, ou seja a divisão, é efetuada com a função *deconv*.

Observe como é feito:

$$\begin{aligned}\text{» } [Q \ R] &= \text{deconv}(p3, p1) \\ Q &= \\ 5 \quad -2\end{aligned}$$

$$\begin{aligned}R &= \\ 0 \quad 0 \quad 0 \quad 0\end{aligned}$$

O vetor **Q** contém os coeficientes do quociente polinomial, que no nosso caso, são os coeficientes do próprio polinômio $p2$.

O vetor **R** contém os coeficientes do polinômio que corresponde ao resto da divisão polinomial. Como a divisão de $p3$ por $p1$ é exata, o resto é zero.

Vamos agora, substituir o polinômio $p3$ pelo polinômio $p4$ assim definido:

$$p4 = 10s^3 + 11s^2 + 2s$$

No MATLAB escreve-se:

```
» p4=[10 11 2 0];
```

Realizando a divisão:



```
» [Q R]=deconv(p4,p1)
Q =
    5   -2
```

```
R =
    0    0    3    2
```

Agora já temos um resto na divisão, ou seja, o polinômio $3s+2$

7.2 - Sistemas de equações lineares

Para resolução dos sistemas de equações lineares, é fundamental o cálculo do determinante da matriz. O determinante de uma matriz é um número específico associado a toda e qualquer matriz quadrada.

No MATLAB, como visto anteriormente, o comando utilizado para o cálculo de determinantes é o comando *det*. Vale lembrar que o cálculo de determinantes só é possível para matrizes quadradas.

Vejam um exemplo:

```
» w=[2 1 3;-5 6 -1; 4 0 -2]
w =
     2     1     3
    -5     6    -1
     4     0    -2
» det(w)
ans =
   -110
```

Podemos agora propor um sistema de equações lineares para ser resolvido:

$$\begin{aligned}x_1 + 2x_2 + 3x_3 &= 4 \\ 2x_1 + 3x_2 + 4x_3 &= 5 \\ 4x_1 + 2x_2 + 5x_3 &= 1\end{aligned}$$

No MATLAB, podemos representar este sistema de equações por uma matriz 3×3 e um vetor:

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

Calcularemos agora o determinante de A:

```
» det(A)
ans =
    -5
```

Aqui vale a pena lembrarmos um aspecto importante na resolução de sistemas de equações lineares. Um sistema de equações lineares somente possui solução determinada de a matriz que o representa possuir determinante não nulo, ou seja, a matriz formada com os coeficientes das equações não pode ser uma matriz singular. No exemplo anterior, percebe-se que o sistema proposto tem solução.

A solução mais prática utilizando-se o MATLAB é multiplicar diretamente o vetor **B** pela inversa da matriz **A**. Esta operação corresponde a resolução de uma operação com as matrizes que representam o sistema de equações lineares, como visto abaixo:

$$A_{n \times n} * X_{n \times 1} = B_{n \times 1}$$

A matriz **A** contém os coeficientes do sistema, o vetor **X** representa as incógnitas e o vetor **B** contém os termos independentes do sistema. No MATLAB, temos:

```
» X = inv(A)*B
X =
   -1.4000
```



```
1.8000  
0.6000
```

De outra forma, poderíamos realizar uma divisão à esquerda entre **A** e **B**. Veja como é:

```
» X=A\B  
X =  
-1.4000  
1.8000  
0.6000
```

7.3 - Encontrando a melhor curva partindo de dados experimentais

O MATLAB nos permite fazer a interpolação polinomial partindo-se dados experimentais. Para isto utilizamos o comando **polyfit**. Os exemplos que serão vistos a seguir ilustram a utilização deste comando.

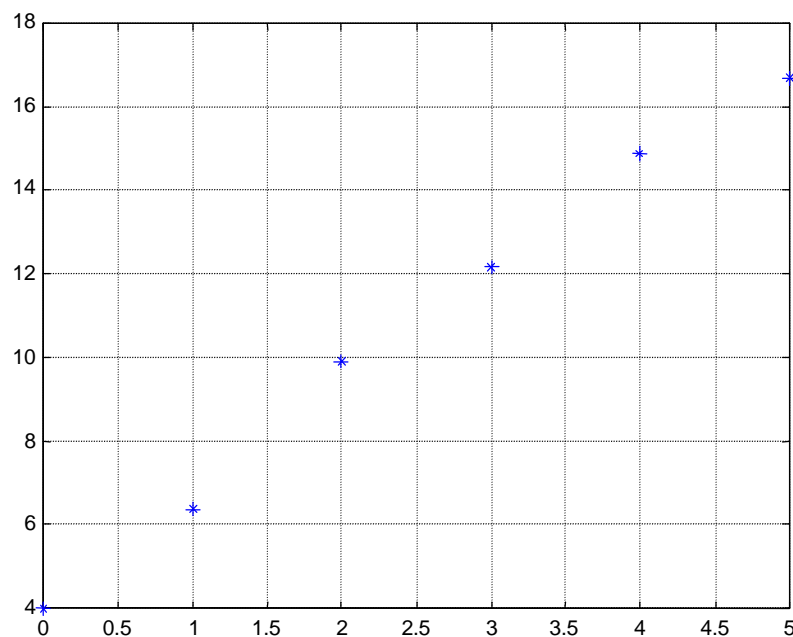
Os pontos experimentais a seguir representam a posição de um móvel em MRU ao longo do tempo.

S(cm)	4,00	6,71	9,43	12,18	14,87	17,7
t(s)	0,00	1,00	2,00	3,00	4,00	5,00

Vamos definir dois vetores com os dados experimentais:

```
» S=[4 6.4 9.93 12.18 14.87 16.7];  
» t=[0 1 2 3 4 5];
```

Agora iremos plotar os pontos para termos uma noção da evolução dos mesmos



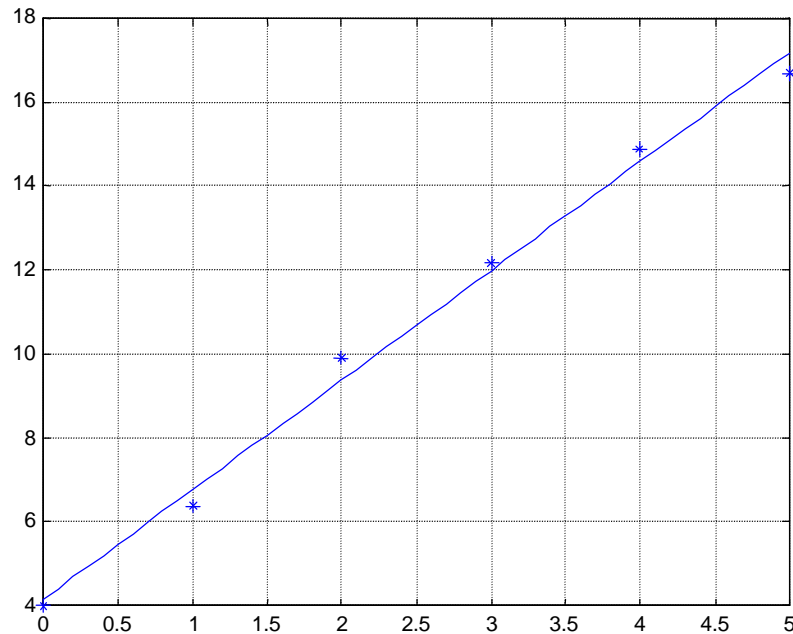
Iremos então realizar a interpolação dos dados através de um polinômio de primeira ordem, o mesmo procedimento pode ser realizado para qualquer polinômio de grau **n**.

```
» poly = polyfit(t,S,1)  
poly =  
2.6046 4.1686
```

Agora iremos traçar a reta obtida sobre os dados experimentais:

```
» hold on;  
» t=0:1:5;  
» S = polyval(poly,t);  
» plot(t,S)
```

E o resultado final será:



8 – Análise Numérica

8.1 – Integração

O MATLAB possui três funções para determinar numericamente a integral de uma função em um domínio finito: *trapz*, *quad* e *quad8*. A função *trapz* aproxima a integral de uma função somando a área de trapézios formados a partir de pontos da curva. Na verdade, este método subestima o valor da área sob o gráfico em alguns pontos e superestima em outros. Desta forma, quanto maior o número de trapézios que utilizamos, melhor será a aproximação. A função *trapz* é chamada como a função *plot*. Experimentemos com os vetores **t** e **S** utilizados no item anterior:

```
» area = trapz(t,S)  
area =  
53.7300
```

As funções *quad* e *quad8*, utilizam o conceito matemático de quadratura e técnicas de variação da largura dos trapézios conforme as características da função, ou seja, elas diminuem as larguras dos trapézios nas regiões do gráfico em que ocorrem as variações mais rápidas da função. Entretanto, seu modo de chamada é um pouco diferente da função *trapz*. Para demonstrá-lo, utilizaremos a função *humps*, *default* do MATLAB:

» area = quad('humps', -1 , 2)	» area = quad8('humps', -1, 2)
area =	area =
26.3450	26.3450

A única diferença entre *quad* e *quad8*, é que a segunda é um pouco mais rigorosa nos cálculos que a primeira.

8.2 – Diferenciação

Comparada à integração, a diferenciação numérica é muito mais difícil. A derivação deve ser, devido as suas dificuldades inerentes, evitada ao máximo, principalmente se os dados a serem derivados foram obtidos de forma experimental.

Neste caso, é melhor efetuarmos um ajuste de curva pelos mínimos quadrados e então derivar o polinômio resultante. Como exemplo, vamos considerar os dados abaixo:

```
» a = [0:1:1];
» b = [-.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11.2];
» n = 2; % Ordem do ajuste
» p = polyfit(x,y,n)
p =
   -0.0004   -0.3007    0.9489
» d = polyder(p)
d =
   -0.0007   -0.3007
```

O vetor **d** contém os coeficientes do polinômio que é resultado da derivação de **p**.

O MATLAB possui ainda a função *diff*, que dá como resultado uma derivação grosseira através do método de diferenças finitas. Por ser uma aproximação muito ruim, não iremos abordar este comando neste curso.

8.3 – Equações Diferenciais

As equações diferenciais ordinárias (EDO) descrevem como a taxa de variação das variáveis de um sistema é influenciada por variáveis do sistema e por estímulos externos, ou seja, por entradas. Quando as EDO's podem ser resolvidas analiticamente, alguns recursos do Toolbox de Matemática Simbólica podem ser usados para encontrar soluções exatas.

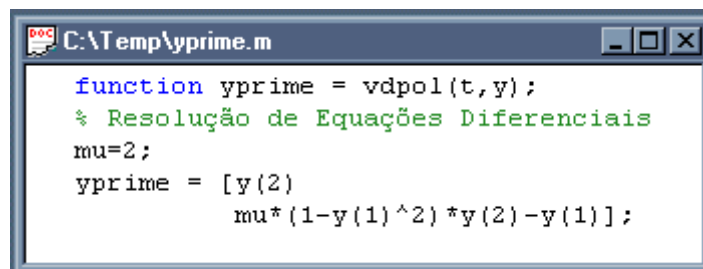
Naqueles casos em que as equações não podem ser resolvidas prontamente de forma analítica, é conveniente resolvê-las numericamente. Como exemplo, resolveremos a seguinte equação diferencial:

$$\frac{d^2x}{dt^2} - \mu(1 - x^2) \frac{dx}{dt} + x = 0$$

Para ser resolvida, a equação deve ser reescrita da seguinte forma:

$$\text{Sejam: } y_1 = x \text{ e } y_2 = \frac{dx}{dt}; \quad \text{Então: } \frac{dy_1}{dt} = y_2$$
$$\text{Assim: } \frac{dy_2}{dt} = \mu(1 - y_1^2) - y_1$$

A função a ser usada será a *ode45*, que necessita como parâmetros de entrada uma M-file de função contendo a equação na forma reescrita, o tempo atual e os valores atuais de y_1 e y_2 . Esta função deve ser editada no editor do MATLAB em o seguinte formato:



```
function yprime = vdpol(t,y);
% Resolução de Equações Diferenciais
mu=2;
yprime = [y(2)
          mu*(1-y(1)^2)*y(2)-y(1)];
```

Este arquivo deve ser salvo com um dado nome (sugerimos *yprime*), no seu diretório de trabalho. Escolhemos, em seguida, um intervalo de tempo de 0 a 30 segundos e as condições iniciais: unitária para a derivada primeira e nula para a derivada segunda. Abaixo ‘setamos’ estes parâmetros:

```
» intervalo = [0 30];  
» cond_inic = [1; 0];  
» % Usando a função ode45  
» ode45('yprime', intervalo, cond_inic);
```

A saída será um gráfico do comportamento temporal do sistema. Para termos acesso aos dados, basta colocarmos argumentos de saída, como mostrado abaixo:

```
» [t, y] = ode45('yprime', intervalo, cond_inic);
```

O vetor **t** conterá os instantes de tempo em que a solução foi calculada e **y** é uma matriz de duas colunas e $length(t)$ linhas (para este caso). Na primeira coluna de **y** estão armazenados os valores correspondentes a $y(1)$ e na segunda coluna os valores de $y(2)$.

9 - Gráficos em 2D

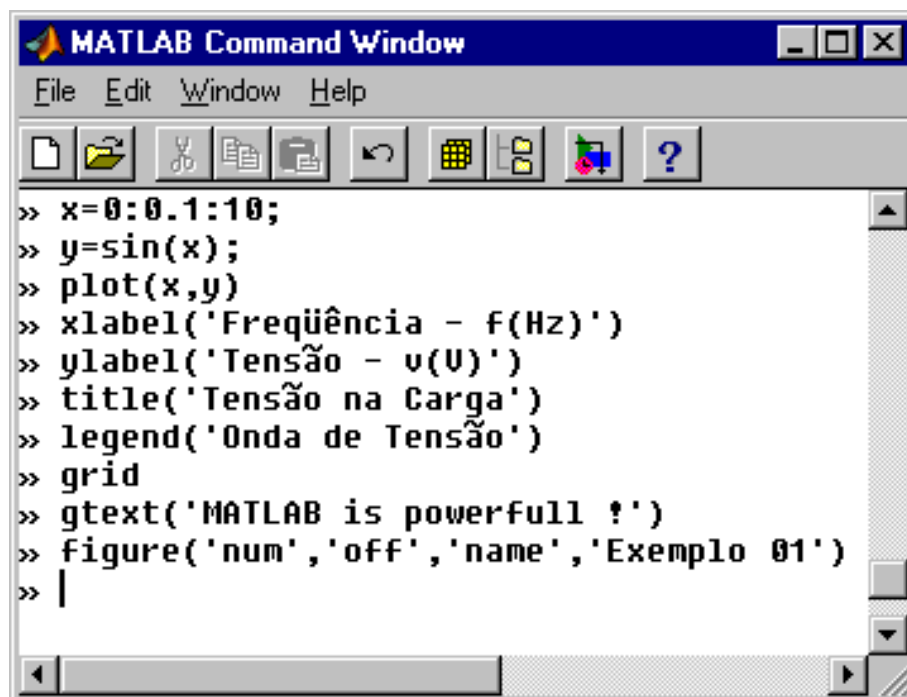
9.1 - Funções gráficas elementares

A seção gráfica do MATLAB possui variadas e sofisticadas técnicas para representar e visualizar dados. Um ponto negativo deve-se a forma com que as ferramentas são aplicadas a uma entidade gráfica, através de comandos, diferenciando-se da maioria dos *softwares* com aplicações dessa ordem, onde as alterações são efetuadas de forma mais amigável, com uso do mouse e atalhos que facilitam as ações.

Os principais comandos utilizados na edição e criação de gráficos são listados abaixo:

- `plot` - *plota* um vetor ou uma função;
- `title` - adiciona título ao gráfico ;
- `xlabel` - adiciona um rótulo ao eixo x;
- `ylabel` - adiciona um rótulo ao eixo y;
- `text` - insere um texto numa determinada posição da janela gráfica;
- `gtext` - insere um texto no gráfico usando o mouse como posicionador;
- `grid` - traça linhas de grade.

Exemplo de aplicação:



```
MATLAB Command Window  
File Edit Window Help  
» x=0:0.1:10;  
» y=sin(x);  
» plot(x,y)  
» xlabel('Frequência - f(Hz)')  
» ylabel('Tensão - v(V)')  
» title('Tensão na Carga')  
» legend('Onda de Tensão')  
» grid  
» gtext('MATLAB is powerfull !')  
» figure('num','off','name','Exemplo 01')  
» |
```

A formatação de cor e estilo da linha podem ser facilmente ajustados através da edição dos argumentos do comando `plot`.



A tabela a seguir mostra as opções.

Cor da Linha		Estilo da Linha	
Caracter	Cor	Caracter	Estilo
y	amarela	.	Ponto
m	magenta	o	Círculo
c	cyan	x	marca "x"
r	vermelha	+	Mais
g	verde	*	Asterisco
b	azul	-	Sólido
w	branca	:	Pontilhado
k	preta	--	Tracejado

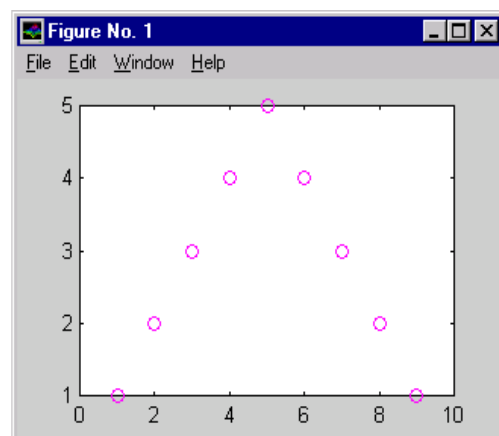
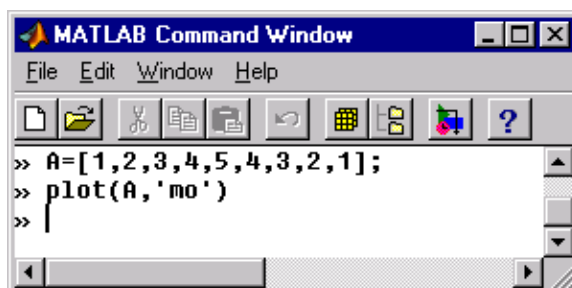
Observe o formato da linha de comando:

» plot(x,y,'ro')

Note que o argumento de cor ou formato de linha deve ser escrito entre aspas simples. Podem ser usados um argumento para cor e um para estilo de linha, ou apenas uma das opções.

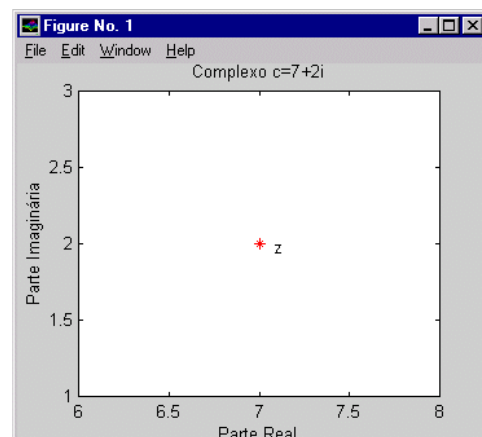
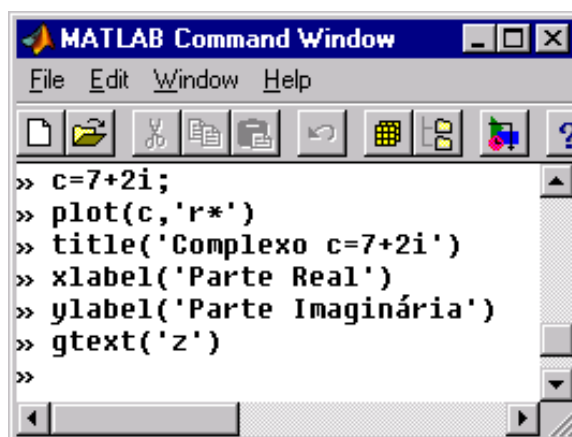
9.2 - Plotando um vetor

Considerando um vetor arbitrário x , o comando $plot(x)$ cria um gráfico em escala linear do vetor x em função de seus índices.



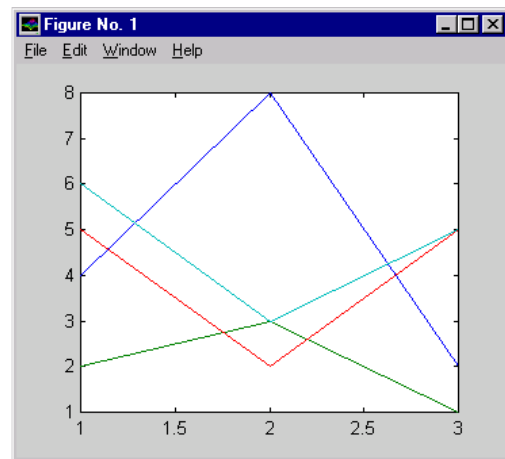
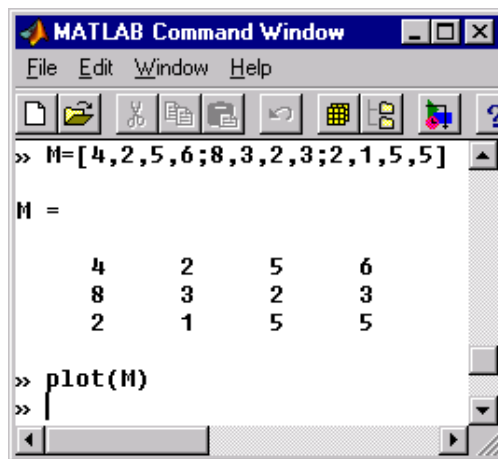
9.3 - Plotando um número complexo

Se c é um número complexo, o comando $plot(c)$ é equivalente a $plot(real(c), imag(c))$.



9.4 - Plotando uma matriz

Sendo M uma matriz, o comando $plot(M)$ irá representar uma linha gráfica para cada coluna de M .



Observe que não foi necessário adicionar nenhum argumento ao comando para criar linhas de cores diferentes.

9.5 - Plotando um vetor em função de outro

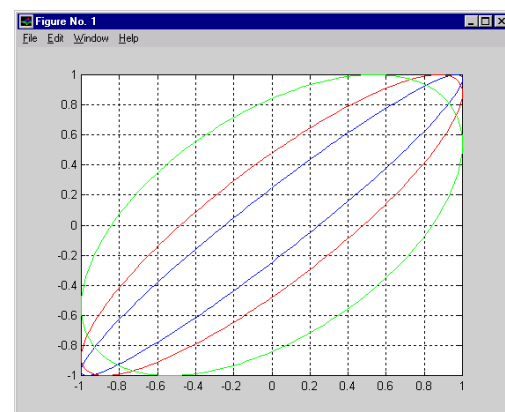
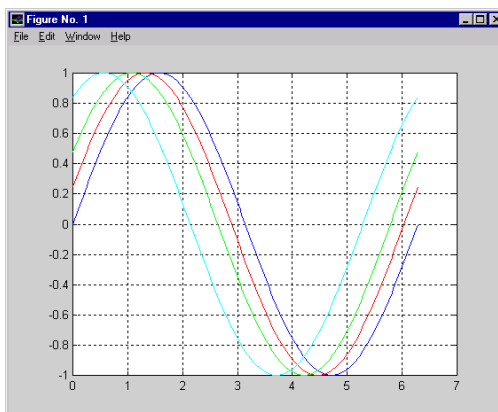
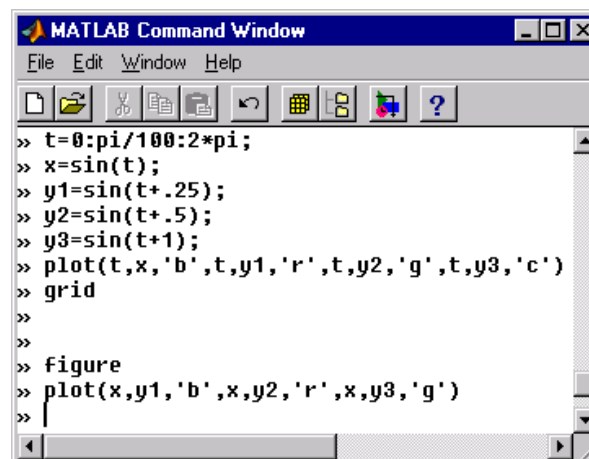
O comando `plot(x,y)` *plota* o vetor **x** em função de **y**.

Condição necessária: `length(x)==length(y)`.

9.6 - Plotagens combinadas

9.6.1 - Usando a mesma linha de comando

Podemos efetuar várias plotagens na mesma tela com apenas um comando `plot`. Veja.





9.6.2 - Adicionando linhas a um gráfico já existente

Essa característica é obtida com o comando *hold*. Com essa opção setada, o programa não apaga as linhas já existentes ao traçar outras. Essa ação pode ajustar automaticamente a escala da janela gráfica para comportar as novas curvas.

Digitando *hold off*, o MATLAB retorna ao modo default.

9.7 - Ajustando os limites de escala

O comando *axis* permite o ajuste dos limites de escalas da janela gráfica atual.

» `axis [xmin xmax ymin ymax]`

A opção *axis('auto')* restaura as configurações padrão do MATLAB.

Para verificar os limites atuais da escala de uma janela gráfica, basta digitar `v=axis`, onde *v* será um vetor com os valores das escalas.

9.7.1 - Corrigindo distorções

Variações do comando *axis('argumento')* permitem corrigir distorções indesejadas em representações gráficas. Acompanhe às linhas de comando que seguem:

» `t = 0:100;`
» `plot(sin(t),cos(t))`

O gráfico correto seria um círculo, você está visualizando um círculo? Para corrigir esse problema proceda com o comando:

» `axis('equal')`

Essa variação do comando *axis* faz com que os incrementos dos vetores em *x* e *y* sejam iguais.

Outra alternativa para corrigir o problema seria a opção

» `axis('square')`

Onde a janela gráfica tornar-se-á quadrada.

9.8 - Escondendo eixos e escalas

Em algumas aplicações apenas as curvas de dados nos interessam e podemos eliminar da janela gráfica eixos, escalas, sinais de graduação e grade com o comando:

» `axis('off')`

Para reverter:

» `axis('on')`

9.9 - Apagando figuras da janela gráfica

Suponha que após construir uma janela com curva, rótulos e escalas já configuradas você queira agora traçar outra curva na mesma janela, sem a anterior e não precisando ajustar as características da janela.

Acompanhe os comandos:

» `a = [3 5 3 7 9 3 2 1 5 2 2];`
» `plot(a)`
» `title('vetor "a"')`
» `xlabel('indices')`
» `ylabel('elementos')`
» `gtext('maior elemento')`
» `axis[0 12 0 10]`

Agora temos que plotar outro vetor, mas as características da janela ainda nos serão úteis. Vejamos então como usar o comando *cla*

```
» cla
» b = [1 2 4 7 0 4 5 3 4 5 2];
» plot(b)
```

9.10 - Trabalhando com múltiplas janelas gráficas

Para criar uma nova janela usamos o comando *figure* cada janela é identificada por um número que chamamos de *handle*.

Quando trabalhamos com mais de uma janela gráfica devemos ter o cuidado de saber qual a janela ativa no momento para evitarmos fazer alterações na janela errada.

Para verificar qual o *handle* da janela ativa no momento utilizamos a função *gcf* - *get current figure*.

```
» h = gcf
```

A variável **h** conterá o número da janela ativa.

Supondo que a janela ativa seja a 2 mas você pretende alterar a janela 5. Para tanto proceda assim:

```
» figure(5)
```

Outra forma de obter essa ação é clicar com o mouse sob a área de plotagem da figura desejada.

Para fechar determinada janela gráfica use o *close*

```
» close(3)
```

9.11 - Construindo subgráficos

Subgráficos podem ser construídos numa mesma janela com a função *subplot*.

```
» subplot(m,n,s) ou
» subplot(mns)
```

Gera uma janela com uma matriz $m \times n$ onde cada elemento da matriz é um subgráfico. O número *s* indica qual subgráfico será plotado ou ativado.

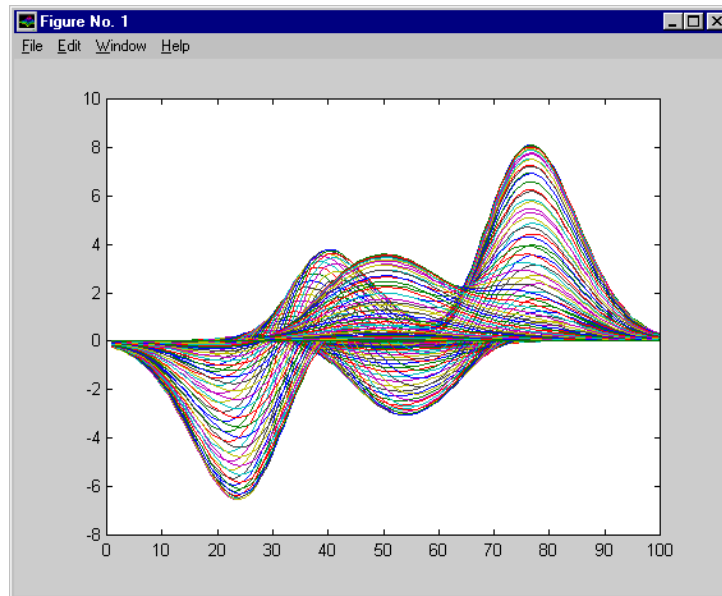
```
» t=0:0.1:30;
» subplot(3,2,1)
» plot(sin(t))
» subplot(3,2,2)
» plot(cos(t),'r')
» subplot(3,2,3)
» plot(tan(t),'g')
» subplot(3,2,4)
» plot(sinc(t),'m')
» subplot(3,2,5)
» plot(sin(t+pi/2))
» subplot(3,2,6)
» plot(cos(t+pi))
```

9.12 - Ampliando a visualização gráfica - ZOOM

Vamos explicar as funções do *zoom* com um exemplo.

Fazendo uso da função *peaks* criamos um matriz 100 x 100, a seguir vamos plotá-la.

```
» ma = peaks(100); Atenção não esqueça do ponto-e-vírgula!
» plot(ma)
```

Responda agora com base no gráfico: qual o maior valor que a função alcança no eixo y com uma precisão de três casas decimais?

Observando a figura, notamos que o valor fica próximo a 8, ocorrendo entre 70 e 80 no eixo x.

Para um resultado mais apurado damos um "zoom" no local. Poderíamos usar o comando axis

» axis [70 80 7 9]

Mas essa opção não se faz muito útil, pois para cada aproximação deveríamos redigitar os parâmetros para o comando.

Tente agora com:

» zoom

Clique na região a ser ampliada, formando um retângulo com o botão esquerdo do mouse pressionado.

Algumas variações do comando zoom podem ser vistas com o *help*.

» help zoom

9.13 - Plotando funções com o MATLAB

Para plotar a função $\exp(x)$ no intervalo $[0 \ 10]$, poderíamos criar um vetor x e um vetor y em seguida plotar um em função do outro. Porém para funções matemáticas já definidas no programa, ou novas funções definidas pelo usuário temos o comando *fplot*.

» fplot('exp', [0 10])

» grid

Note, entre apóstrofes '*'*', temos a função. Entre colchetes [*]*, o intervalo de análise.

9.14 – Outras formas de gráficos

9.14.1 - Gráficos de barras

Para visualizar dados em forma de barras, para aplicações onde tal característica seja útil, utilizamos o comando *bar*.

Sua aplicação é fácil, veja:

» b=[1 2 3 4 5 4 3 2 1];

» bar(b)

Importante: as configurações para cor e estilo de linha vistas para o comando *plot* são válidas aqui também.

9.14.2 - Escala logarítmica

Em muitas aplicações a utilização de escalas lineares em um ou ambos os eixos de um gráfico limitam a forma com que as informações são apresentadas. Para otimizar a representação utilizamos escalas logarítmicas.

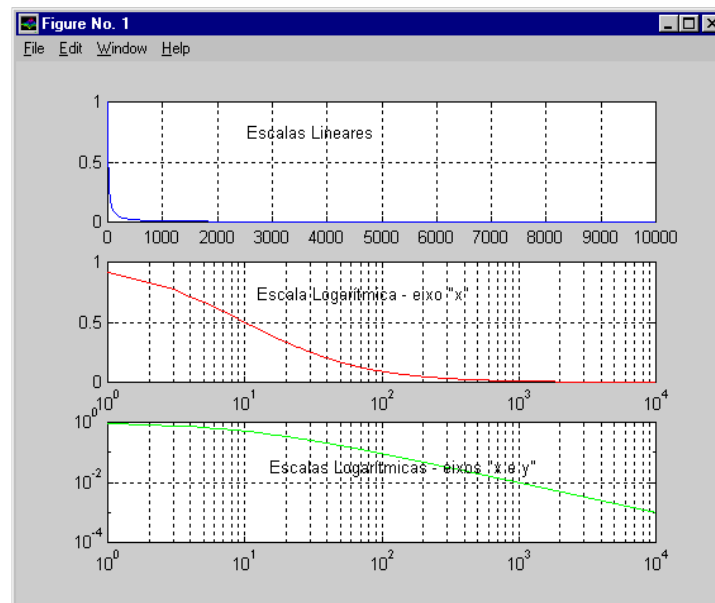
Vamos direto a uma aplicação:

```
» t=0:1e6;  
» y=1./(1+t./10);
```

Plotar numa mesma janela em três gráficos separados a função

- (i) em escala linear (ambos os eixos) → `plot(x,y)`
- (ii) escala logarítmica no eixo x → `logx(x,y)`
- (iii) escala logarítmica nos dois eixos → `loglog(x,y)`

O resultado deve ser esse:



9.14.3 - Gráficos de variáveis discretas

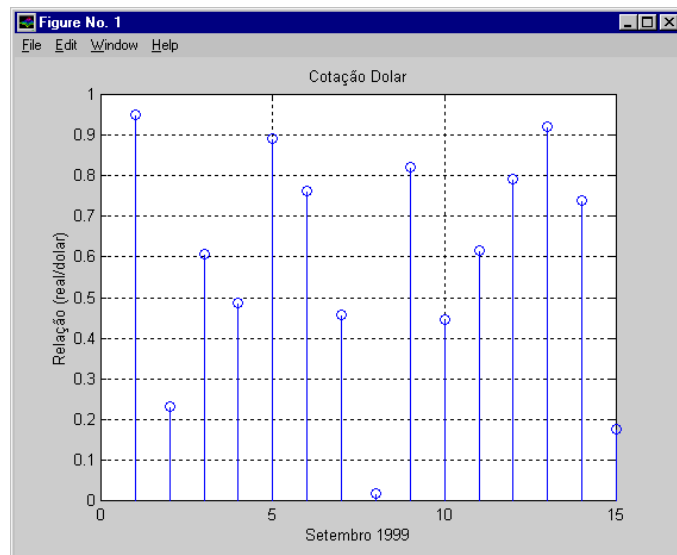
Uma forma de representar fenômenos ou quantidades que não apresentam-se numa forma contínua é plotando as "amostras" discretas.

Como exemplo de variável discreta poderíamos citar a cotação de uma moeda, em sua variação diária.

A função do MATLAB que permite tal representação é o `stem`.

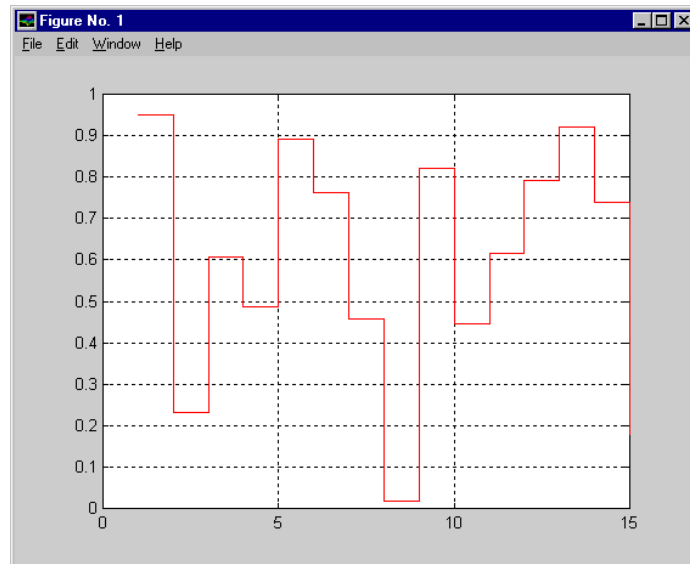
```
» dias=1:15;  
» cota = rand(15);  
» stem(dias,cota)
```

Para maior clareza, utilize os comando estudados para nomear os eixos e título do gráfico.



9.14.4 - Gráficos em escada

Uma variação que une características dos comandos já vistos `bar` e `stem`, é o comando `stairs`. Execute o comando `stairs` para os dados de cotação da moeda, visto acima.

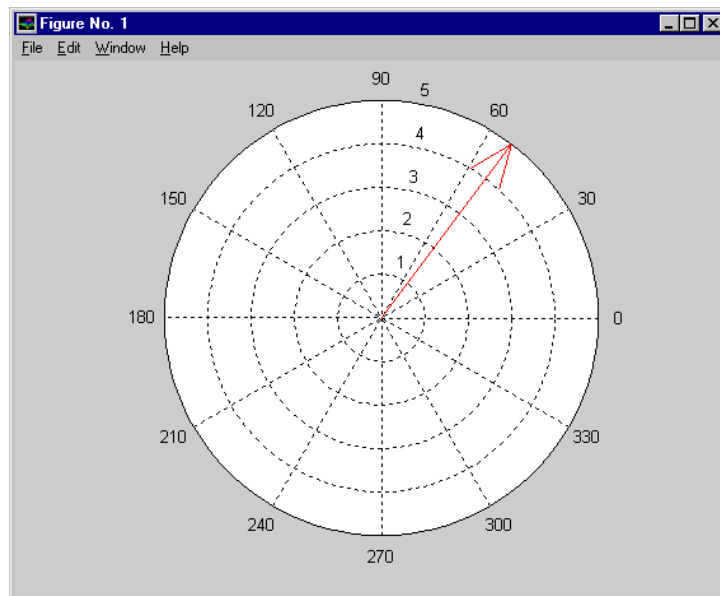


9.14.5 - A Função `compass`

Para representar números complexos com amplitude e ângulo.

```
» a=3+j*4  
» compass(a)
```

Note que a janela apresenta o módulo e a fase do número "c".



10 - Programando com o MATLAB

Hoje, o conhecimento de linguagens de programação diferencia profissionais aos olhos do mercado. A programação esta cada vez mais presente dentro da engenharia (simulações, cálculos complexos e rotineiros, análise e projeto de sistemas, etc) e saber programar se tornou uma necessidade. Nesta parte do curso, estaremos tratando da programação com o MATLAB utilizando-se de suas diversas funções e ferramentas que facilitam e simplificam muito a estrutura dos programas.

Tópicos a serem abordados:

- Operações Relacionais e Lógicas;
- Rotinas de Programação;
- Estruturas de Controle Repetitivas;
- Estruturas de Controle Condicionais;
- Construindo Novas Funções;
- Debugando com o MATLAB.

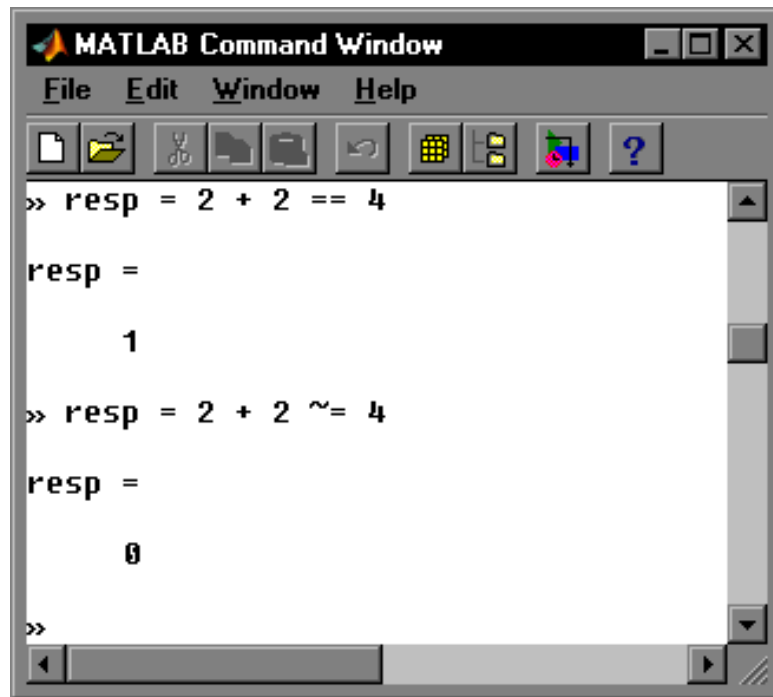
Para o estudo seguinte, parte-se do princípio que todos já tenham noções básicas de rotinas e estruturas de programação.

10.1 - Operações relacionais e lógicas

Os operadores relacionais são aqueles que respondem a uma determinada operação com verdadeiro ou falso, que são simbolizados respectivamente por 1 e 0. A tabela a seguir relaciona operador com a descrição do operador.

OPERADOR	DESCRIÇÃO
<	Menor que
<=	Menor ou igual
>	Maior que
>=	Maior ou igual
==	Igual
~=	Diferente

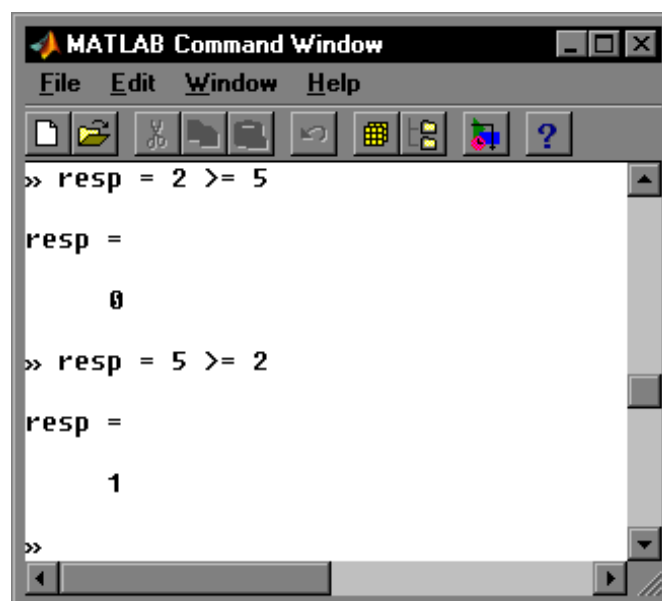
Para os que ainda não tiveram contato com este raciocínio, os exemplos a seguir demonstram algumas operações.



```
MATLAB Command Window
File Edit Window Help
» resp = 2 + 2 == 4
resp =
    1
» resp = 2 + 2 ~= 4
resp =
    0
»
```

No *prompt*, quando perguntamos se $2 + 2$ é igual a 4, o MATLAB nos dá como resposta a afirmativa (1). Já depois quando perguntamos se $2 + 2$ é diferente de 4, o MATLAB nos responde que não (0).

Veja alguns outros exemplos:



```
MATLAB Command Window
File Edit Window Help
» resp = 2 >= 5
resp =
    0
» resp = 5 >= 2
resp =
    1
»
```

O MATLAB responde aos operadores lógicos da mesma forma que responde aos operadores relacionais acima descritos. Para verdadeiro tem-se **1** e falso **0**. No MATLAB temos os seguintes operadores lógicos descritos na tabela:

OPERADOR	DESCRIÇÃO
&	AND
	OR
~	NOT

O operador AND ‘&’ responde sim (1) apenas quando as duas ou mais expressões envolvidas na questão forem verdadeiras, caso contrário a resposta é não (0). Veja o exemplo no prompt abaixo onde no primeiro caso temos a primeira expressão verdadeira e a segunda expressão dois falsa, que torna a resposta falsa (0). ($1 \& 0 = 0$).



```
MATLAB Command Window
File Edit Window Help
[Icons]
>> resp = 3 > 2 & 3 > 5
resp =
    0
>> resp = 3 > 2 | 3 > 5
resp =
    1
>> resp = ~ resp
resp =
    0
>> |
```

Já o operador OR ‘ | ’, mostrado no segundo caso do prompt, responde sim (1) se pelo menos uma das expressões for verdade.

Por fim o operador NOT ‘ ~ ’ inverte seu argumento. Note que resposta da última operação que é sim (1) efetuado com o OR é invertida e passa a valer (0).

10.2 - Rotinas especiais de programação

Antes de confeccionarmos nosso primeiro programa, precisamos conhecer as rotinas especiais de programação do MATLAB que são comuns à várias linguagens. Estas rotinas associadas aos inúmeros recursos oferecidos pelo MATLAB propiciam ao usuário desenvolver desde simples programas feitos em calculadoras de bolso, como HP, até os mais sofisticados programas desenvolvidos em linguagens de alto nível, como C++.

ROTINAS	DESCRIÇÃO
for end	Gera um loop enumerável
while <cond> End	Gera um loop enquanto uma condição (<cond>) for verdadeira
if <cond> elseif end	Seqüência de comandos executáveis condicionalmente
if <cond> else end	Seqüência de comandos executáveis condicionalmente
Switch case<cond> otherwise end	Seqüência de comandos executáveis condicionalmente
break	Sair fora de um loop for ou while
return	Retornar de uma função (arquivo.m)
pause	Para num ponto do programa até que se aperte qualquer tecla do teclado

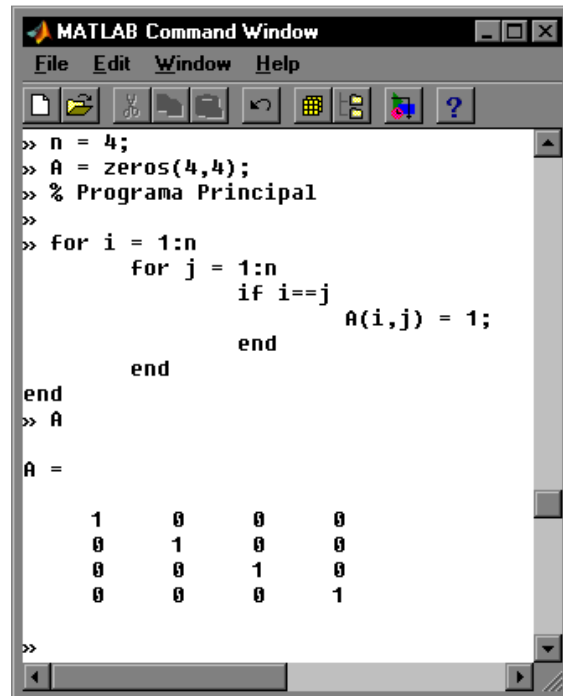
(*) Todas as funções / comandos do MATLAB devem ser escritos com letras minúsculas.

10.3 - Estruturas de controle repetitivas (loops)

➤ loop for

Partiremos direto para a utilização do **for** no exemplo a seguir. Vamos editar este pequeno programa sem utilizar ainda dos recursos de M-file que serão abordados na sequência. Fez-se isso, para mostrar que rotinas repetitivas também podem ser digitadas e executadas diretamente no *prompt* do MATLAB. Neste exemplo, construiremos uma matriz zero (nula) 4 X 4 e em seguida a transformaremos em uma matriz unidade (identidade), onde substituiremos por 1 os termos nulos da sua diagonal principal. Veja na tela abaixo a sequência de comandos:

Aqui, escolhemos **n** como um contador e geramos a matriz nula **A**.



```
MATLAB Command Window
File Edit Window Help
>> n = 4;
>> A = zeros(4,4);
>> % Programa Principal
>> for i = 1:n
    for j = 1:n
        if i==j
            A(i,j) = 1;
        end
    end
end
>> A
A =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
>>
```

Depois partimos para o programa principal. Quando digitamos a primeira linha e teclamos < ENTER >, o cursor fica esperando novas instruções até que o comando seja fechado com **end**. O sinal de *prompt* só aparece com o fechamento de todos os loops abertos.

Veja que depois que fechamos todos os comandos abertos com seus respectivos **ends**, voltamos ao sinal de *prompt* onde podemos chamar a matriz **A** e conferir o funcionamento do programa.

A rotina **for** funciona aqui da mesma forma que funciona nas linguagens mais usuais, ou seja, a variável **i** é incrementada a cada *loop* e as instruções de programa são executadas repetitivamente até que se constate que **i > n**. Observe que quando se termina uma linha de instrução de comando é colocado o ponto e vírgula ‘;’ que suprime a impressão na tela. Experimente não usar o ponto e vírgula e executar o programa.

(*) erros de digitação ou de programação ocorridos em linhas anteriores a atual dentro de estruturas de repetição não poderão ser corrigidos sem nova digitação integral do programa.

➤ loop **while**

Comandos repetitivos também podem ser organizados dentro de uma estrutura **while**. O loop **while** tem a seguinte forma:

```
while <condição>
    comandos;
end
```

A condição é testada e sendo verdadeira os comandos são executados. Depois de realizados todos os comandos, novamente testa-se a condição que, sendo verdadeira, força uma nova execução dos comandos. A rotina permanece sendo executada até que a condição seja falsa.

O exemplo a seguir introduz a criação de M-files explorando os recursos da estrutura de controle **while**.

Quando vamos escrever no MATLAB uma sequência de comandos, tanto podemos escrever estes comandos diretamente na linha de *prompt*, como fizemos até agora, como podemos escrever uma rotina de comandos em arquivos especiais que são chamados de M-files. Os M-files são arquivos executáveis no MATLAB e são chamados do *prompt* bastando-se digitar o nome do arquivo e <ENTER>. Assim, todos os comandos listados

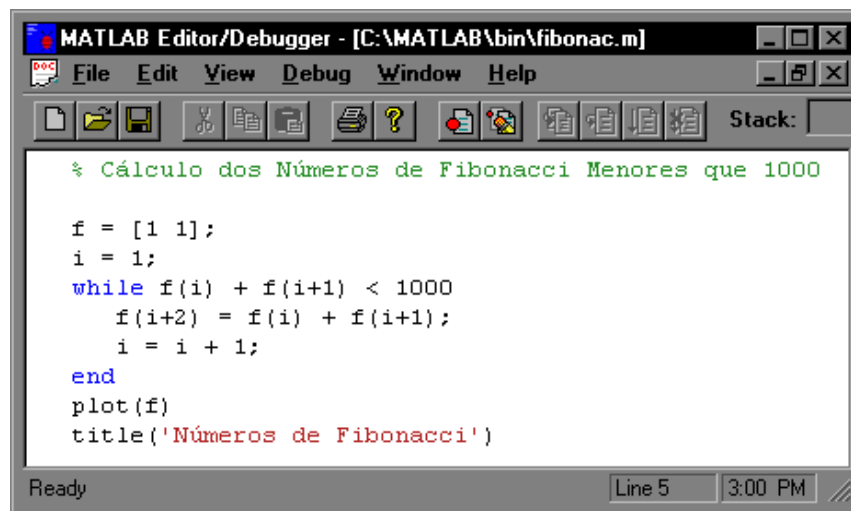


no arquivo são executados sequencialmente. Editar um arquivo M-file é muito simples. Para fazê-lo você pode utilizar o editor de textos de sua preferência.

Vamos criar um programa em um arquivo (M-file) com a função de gerar esta sequência de números. Este arquivo será chamado de `fibonac.m`.

Para criar o M-file, vá até o menu **File**, submenu **New** e escolha **M-file**. Assim, você se encontrará no editor de textos do próprio MATLAB onde as linhas de comandos do programa devem ser escritas.

Números de Fibonacci: É uma sequência de números onde os dois primeiros números são iguais a 1, o terceiro igual a soma dos dois primeiros, o quarto é a soma do segundo com o terceiro, o quinto é igual a soma do terceiro e do quarto e assim por diante.

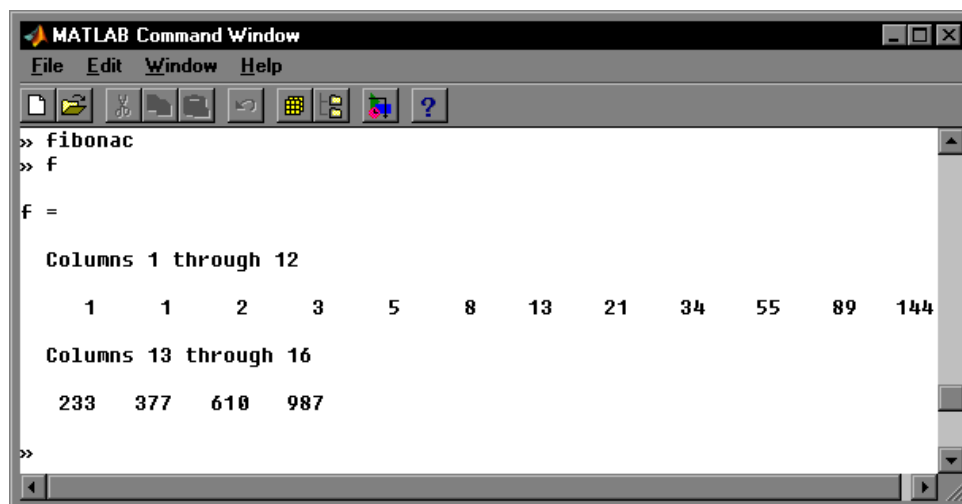


```

% Cálculo dos Números de Fibonacci Menores que 1000

f = [1 1];
i = 1;
while f(i) + f(i+1) < 1000
    f(i+2) = f(i) + f(i+1);
    i = i + 1;
end
plot(f)
title('Números de Fibonacci')
```

Para executar mais facilmente seus programas, salve-os no subdiretório **bin** do MATLAB, assim ao abrir o MATLAB seus programas já poderão ser executados diretamente chamando-se os arquivos da linha de prompt sem precisar redirecionar diretórios. Uma vez salvo os M-files, basta digitar o nome do arquivo e executar.



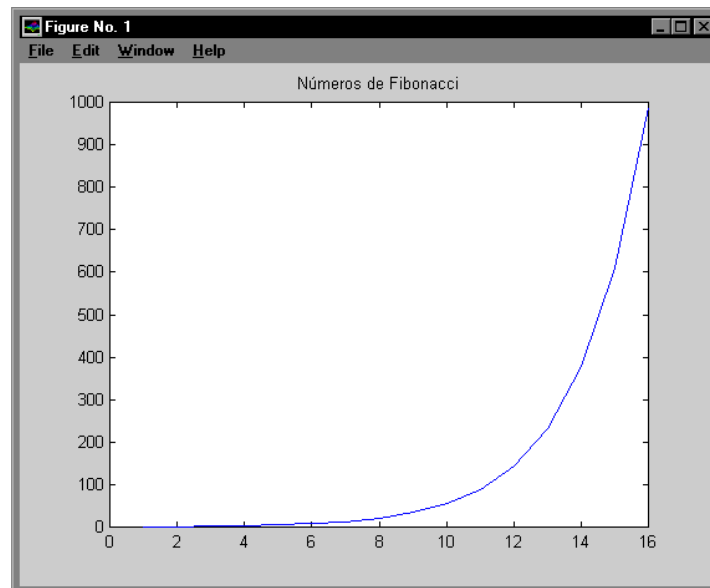
```

>> fibonac
>> f
f =

Columns 1 through 12
    1     1     2     3     5     8    13    21    34    55    89   144

Columns 13 through 16
   233   377   610   987
```

Como saída do programa temos o vetor `f` com a sequência de números de Fibonacci que são mostrados no gráfico seguinte.



10.4 - Estrutura de controle condicional

As estruturas de controle condicionais envolvem tomadas de decisão para a execução de comandos.

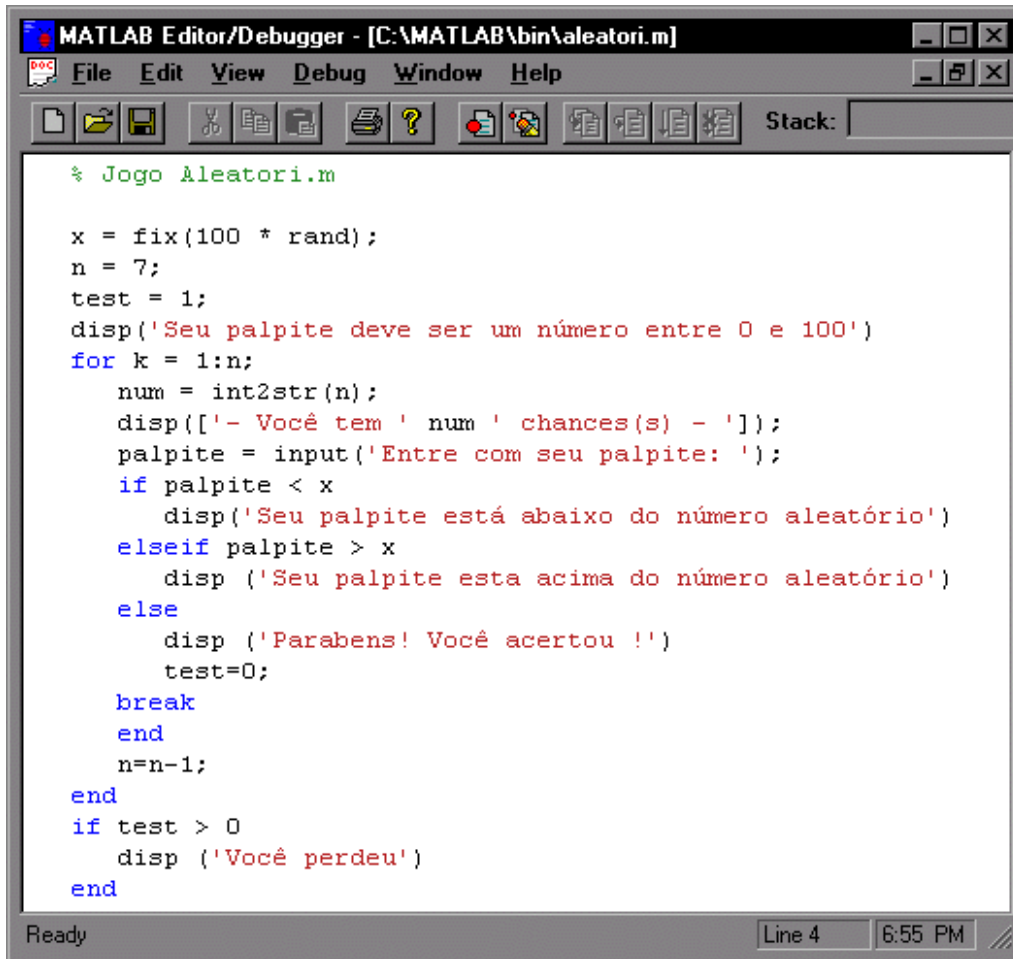
➤ **if - elseif - else - end**

```
if < condição 1 >
    comandos 1
elseif < condição 2 >
    comandos 2
else
    comandos 3
end
```

Onde os comandos 1, 2 e 3 podem ser executados de acordo com os resultados de suas respectivas condições. Se a condição 1 é verdadeira, comandos 1 são executados, se a condição 2 é verdadeira, então comandos 2 são executados e por último se nem condição 1 ou 2 é verdadeira, então comandos 3 é que são executados. O entendimento desta última sequência de condições e comandos, principalmente saber diferenciar o **else** do **elseif**, é muito importante.

(*) No exemplo da estrutura com 'if' e 'elseif' somente os comandos associados a primeira condição verdadeira encontrada são executados. As expressões condicionais seguintes não são testadas e o resto da estrutura if-elseif-else-end é ignorada. Além disso, não é necessário que o comando final 'else' esteja presente.

Veja o seguinte exemplo onde utilizaremos o editor de textos do MATLAB para escrever o M-file que chamaremos de aleatori.m.



```
% Jogo Aleatori.m

x = fix(100 * rand);
n = 7;
test = 1;
disp('Seu palpite deve ser um número entre 0 e 100')
for k = 1:n;
    num = int2str(n);
    disp(['- Você tem ' num ' chances(s) - ']);
    palpite = input('Entre com seu palpite: ');
    if palpite < x
        disp('Seu palpite está abaixo do número aleatório')
    elseif palpite > x
        disp('Seu palpite está acima do número aleatório')
    else
        disp('Parabéns! Você acertou !')
        test=0;
        break
    end
    n=n-1;
end
if test > 0
    disp('Você perdeu')
end
```

A primeira linha de comentário dá informações sobre o programa. Depois de salvo este M-file, ao escrevermos na linha de prompt **help aleatori** o conteúdo de comentários escrito depois de (%) aparece na tela.

Depois das linhas de comentários, utilizamos a função **rand** que gera um número aleatório entre 0,0 e 1,0. Este número multiplicado por 100 depois é arredondado e sua parte decimal é truncada pelo comando **fix**. Fazendo assim, o valor que é atribuído a **x** fica entre 0 e 100.

Na linha seguinte, define-se como 7 o número de chances possíveis para que o jogador acerte o valor de **x**. A variável teste, que é necessária ao final do programa na decisão de perdedor ou ganhador, é atribuído o valor 1.

No loop **for** que só se repete ao máximo 7 vezes, número de chances, são colocados os comandos que verificam se o palpite do jogador convém com o valor de **x**.

Utilizamos a função **int2str** para transformar o inteiro **n** em uma string (cadeia de caracteres) que é guardada na variável **num**. Você já vai entender o porque desta transformação ao observar as duas formas da função **disp** na linha seguinte.

```
disp('Seu palpite deve ser um número entre 0 e 100: ');
```

Nesta forma da função **disp** o termo entre apóstrofes (‘ ’) é imprimido na tela na linha de prompt.

```
disp('Você tem direito a' num 'palpite(s)');
```

Nesta Segunda forma, concatena-se o texto 1, o conteúdo da variável e o texto 2. O que ocorre é que esta variável, para poder ser imprimida junto com os textos, deve estar sob forma de string. Daí a necessidade de se transformar **n** em uma string através da função **str2num**.

Depois, a função **input** da linha mostrada abaixo, atribui a variável **palpite**, o conteúdo digitado na linha de prompt. O texto que aparecerá na tela do MATLAB ao se executar esta função é o que fica entre os apóstrofes (‘ ’).

```
palpite = input('Entre com o palpite ');
```

Entrando na estrutura de controle condicional, temos as seguintes rotinas:

- 1) Se *palpite* é menor que **x**, a mensagem ‘Seu palpite está abaixo do número aleatório’ vai para tela.
- 2) Se *palpite* é maior que **x**, a mensagem ‘Seu palpite está acima do número aleatório’ é impressa na tela.

3) Se *palpite* é igual a *x*, a mensagem de 'Parabéns! Você acertou!' vai para a tela.

Caso o jogador não acertar o número dentro das sete chances oferecidas, o loop *for* é encerrado, *k* = 7, e a última estrutura de controle condicional constata a condição de teste maior que 0 como verdade, teste = 1, e executa o comando imprimindo na tela 'Você perdeu!'. Perceba que este programa apenas indica, mas não proíbe que o jogador entre com valores de palpite tanto abaixo de 0 como acima de 100. Pense num acréscimo a ser feito neste programa que solucione tal problema.

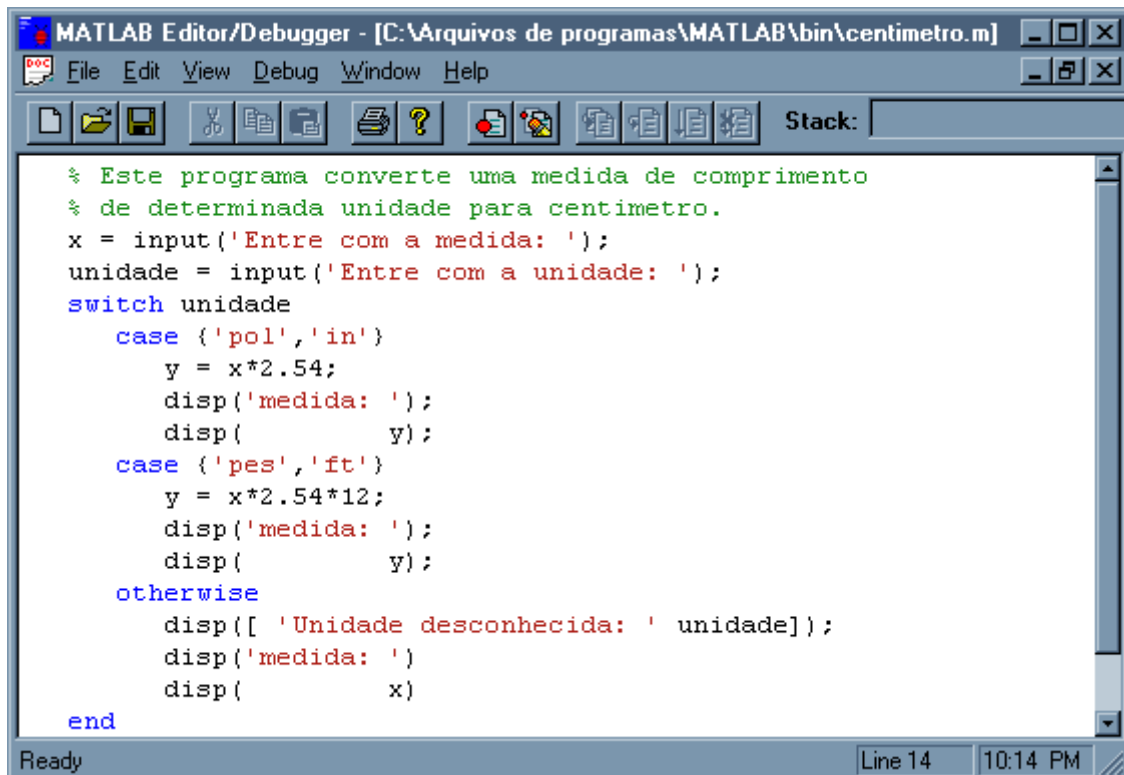
➤ switch-case-end

Quando seqüências de comandos devem ser condicionalmente executadas, com base no uso repetido de um teste de igualdade com um argumento comum, uma estrutura *switch-case-end* pode ser usada. A estrutura tem a seguinte forma:

```
switch <argumento>
    case <teste_argumento1>
        comandos1
    case {<teste_argumento2, teste_argumento3, teste_argumento4>}
        comandos2
    otherwise
        comandos 3
end
```

Aqui, o argumento é um escalar ou um string de caracteres. Se o argumento é escalar, o comando *case* executa o comando *<argumento> == <teste_argumento1>*. Se for um string de caracteres, o teste é executado com o comando *strcmp(<argumento>, <teste_argumento>)*. Nesta estrutura, no comando *case*, o argumento é comparado com os testes de argumento e, quando estes são iguais, a seqüência de comandos desta comparação é executada. Se todas as comparações *case* forem falsas, então o comando *otherwise*, que é opcional, é executado.

Um exemplo de estrutura *switch-case-end* é demonstrado abaixo, onde criou-se o M-file *centimetro.m* para converter valores de comprimento em diferentes unidades para centímetro.



```
MATLAB Editor/Debugger - [C:\Arquivos de programas\MATLAB\bin\centimetro.m]
File Edit View Debug Window Help

% Este programa converte uma medida de comprimento
% de determinada unidade para centimetro.
x = input('Entre com a medida: ');
unidade = input('Entre com a unidade: ');
switch unidade
    case {'pol','in'}
        y = x*2.54;
        disp('medida: ');
        disp(      y);
    case {'pes','ft'}
        y = x*2.54*12;
        disp('medida: ');
        disp(      y);
    otherwise
        disp([ 'Unidade desconhecida: ' unidade]);
        disp('medida: ')
        disp(      x)
end

Ready Line 14 10:14 PM
```



10.5 - Construindo funções com o MATLAB

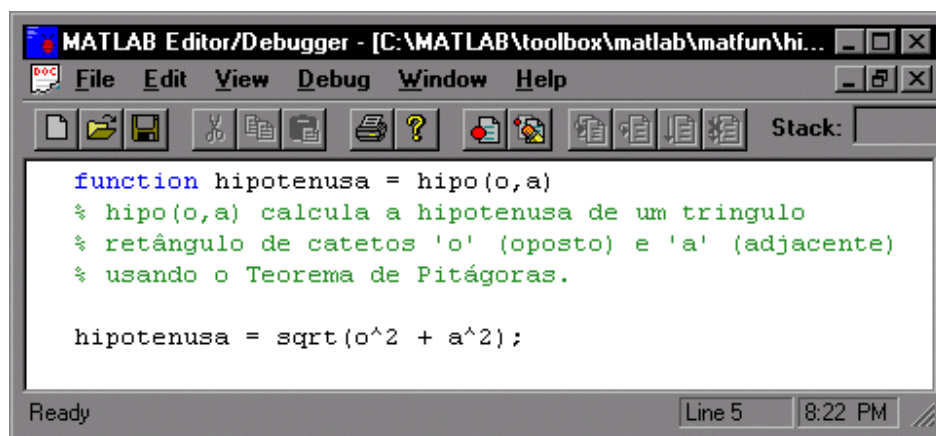
Até esta parte do curso, já citamos e utilizamos inúmeras funções do MATLAB (`fix()`, `input()`, `plot()`, etc...), agora veremos que podemos criar nossas próprias funções. Os arquivos destas funções também são chamados de M-files, porém a diferença destes para os já vistos **script files** (arquivos com extensão '.m' em sua forma simples - `aleatori.m`), é que nestes arquivos a primeira palavra que deve ser escrita após eventuais comentários é **function**. Por isto o nome **function files**.

Diferentemente dos *script files* que operam com variáveis de forma global na sessão de trabalho, os *function files* trabalham com parâmetros locais, os quais são declarados entre parênteses e separados por vírgulas logo após a palavra **function**. Além destes, qualquer variável utilizada dentro de uma *function file* é local.

Variáveis Globais : são aquelas válidas tanto dentro como fora do arquivo onde são geradas.

Variáveis Locais: são aquelas válidas somente dentro do arquivo onde são geradas.

Vejam um exemplo simples de uma função editada para se calcular o valor da hipotenusa de um triângulo retângulo a partir de seus valores de cateto oposto e adjacente. Para editar a função, utilize o editor de textos do próprio MATLAB da mesma forma como feito anteriormente para os **script files**.



```
function hipotenusa = hipo(o,a)
% hipo(o,a) calcula a hipotenusa de um triângulo
% retângulo de catetos 'o' (oposto) e 'a' (adjacente)
% usando o Teorema de Pitágoras.

hipotenusa = sqrt(o^2 + a^2);
```

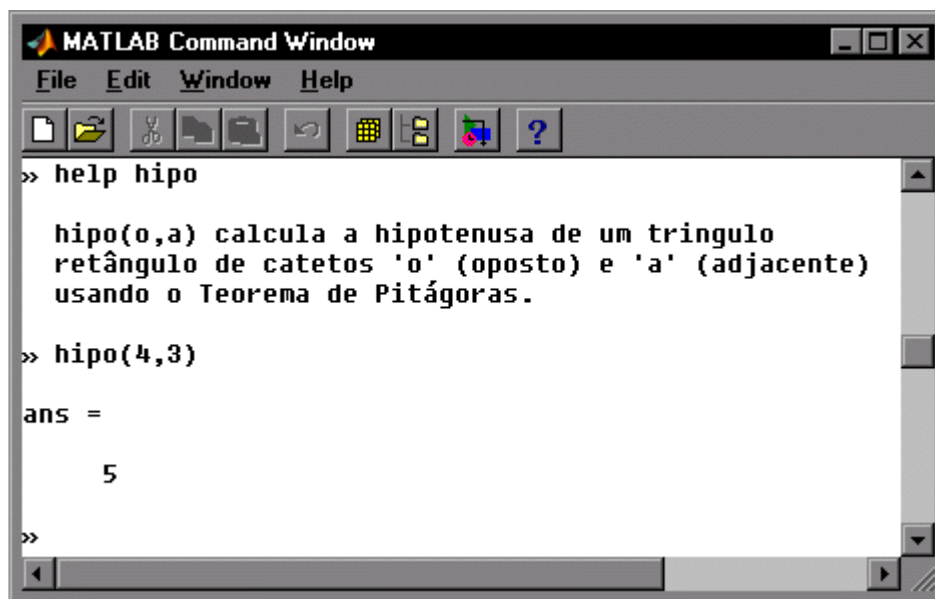
Depois de escrita a função, temos algumas opções para salva-la:

1) *Salvando no subdiretório **Bin*** - ao iniciar o MATLAB você se encontrará no caminho `c:\MATLAB\bin` e as funções salvas neste diretório são compiladas a cada execução. Isto significa que você poderá fazer alterações nas funções salvas desta maneira e senti-las na próxima execução. As funções salvas só poderão ser executadas dentro do diretório **bin**, onde foram salvas.

2) *Salvando nos subdiretórios específicos de funções do MATLAB* - os subdiretórios contidos no caminho `c:\MATLAB\toolbox\MATLAB`, como **elfun**, **elmat**, **funfun**, etc, são utilizados especificamente para guardar funções do MATLAB. Quando MATLAB é iniciado, ele varre estes subdiretórios catalogando, compilando e guardando em memória, todas as funções neles contidas. Desta maneira, o usuário pode executar a função diretamente em qualquer diretório apenas chamando-a. Ao se criar uma nova função ou modifica-la, para que o MATLAB a tenha cadastrada e atualizada, deve-se reiniciar o MATLAB.

3) *Salvando as funções em subdiretórios exclusivos* - caso você prefira criar um subdiretório exclusivo para o armazenamento de suas funções, para executá-las você precisará se encontrar no domínio deste subdiretório onde as gravou. A cada execução o MATLAB compila e executa a função da mesma forma que ocorre no subdiretório **bin**. Eventuais alterações podem ser feitas e são atualizadas na execução.

Para o exemplo criado da função que calcula a hipotenusa, estaremos a salvando-a no subdiretório de trabalho (que foi 'setado' com o auxílio da função `chdir`).



A screenshot of the MATLAB Command Window. The title bar reads 'MATLAB Command Window'. The menu bar includes 'File', 'Edit', 'Window', and 'Help'. Below the menu bar is a toolbar with icons for file operations and help. The command prompt shows the following sequence of commands and output:

```
>> help hipo  
  
hipo(o,a) calcula a hipotenusa de um triângulo  
retângulo de catetos 'o' (oposto) e 'a' (adjacente)  
usando o Teorema de Pitágoras.  
  
>> hipo(4,3)  
  
ans =  
  
5  
  
>>
```

Elabore agora uma função (*divide.m*) que dê como resposta o resto de uma divisão inteira entre dois números.

Por fim, você deve concluir que programar com MATLAB é muito simples e que, além de todas as funções já oferecidas pelo MATLAB, você poderá criar outras diversas funções de acordo com sua necessidade.

10.6 - Debugando com o MATLAB

Nesta última parte do curso, veremos alguns comandos que auxiliam o programador no trabalho de criação de novos programas. Frequentemente quando estamos programando em uma linguagem qualquer, nos deparamos com alguns erros de programação que, por vezes, são de fáceis localização e correção. Porém, às vezes o programador pode cometer um erro sutil, tornando-se difícil e demorado o trabalho de detecção e correção do mesmo. Exatamente aí é que precisamos 'debugar' o programa, ou seja, fazer um rastreamento, uma depuração, um acompanhamento da execução do programa de tal forma que se localizem os erros.

Uma maneira simples de se detectar erros é analisar o valor das variáveis usadas e verificar se estas correspondem ao esperado. Trabalhando com **script files**, variáveis globais, isso pode ser feito tranquilamente. Porém quando trabalhamos com **function files**, variáveis locais, isso não é possível, uma vez que as variáveis usadas dentro destas não são acessíveis após execução.

Veremos três comandos que são válidos tanto para **scripts** como para **function files**.

Echo - No menu **options** podemos escolher **turn echo on** e isto faz com que as linhas de comandos sejam mostradas na tela à medida que o programa é executado. Esta opção é válida somente para **script files**. Para desativar este comando selecione **turn echo off**.

Para se fazer o controle deste comando via teclado durante a execução de um **script file**, digita-se na linha de prompt **echo on** para ativar e **echo off** para desativar.

Para estender este comando a um **function file** e se permitir que os comandos dentro desta apareçam na tela, digitamos na linha de *prompt* os comandos **echo file on**, onde em *file* substitui-se o nome da função.

Para que todas as **function files** sejam ativadas, temos o comando **echo on all**.

Assim, muitos os erros podem ser mais facilmente detectados, pois ao ocorrerem no programa, na tela estará sendo mostrado em que linha de comando do programa o problema acontece.

Keyboard - Também de muita utilidade quando estamos debugando o programa, o **keyboard** deve ser incluído numa linha do programa em análise fazendo com que a sequência de execução seja momentaneamente suspensa, deixando o teclado do computador disponível para que se verifique ou até se altere valores de variáveis.

Este comando vale igualmente tanto para **script files** como para **function files** e na sequência, para se retornar à execução do programa, digita-se **return** seguido do <enter>.

Perceba que a grande utilidade deste recurso consiste no fato de que o programador pode interferir durante a execução do programa alterando e conferindo variáveis. Com a prática você vai notar a importância deste comando para a solução de problemas em programas.



Pause – Este comando faz com que a execução de um programa seja interrompida até que o programador aperte qualquer tecla. Este artifício não permite que o programador atue alterando variáveis durante execução, mas é um bom recurso quando se deseja analisar, por exemplo, uma sequência de ‘plots’ que constituem a evolução de um sistema. Para utilizar este comando, deve-se incluir **pause** numa linha do programa em questão. Escrevendo-se **pause(n)**, interrompe-se a execução por **n** segundos.



11 - Exercícios

Exercício 01:

Você decidiu comprar um carro novo por R\$ 18.500,00. O vendedor está lhe oferecendo duas opções de financiamento: (1) uma taxa mensal de juros de 0,24%, paga durante quatro anos, ou (2) uma taxa mensal de juros de 0,74%, também paga durante quatro anos, mas com um desconto de fábrica de R\$ 1.500,00. Qual das duas opções é o melhor negócio?

Dados: valor mensal da prestação, $P \rightarrow P = A \left[\frac{R(1+R)^M}{(1+R)^M - 1} \right]$, onde A é o valor do empréstimo, R é

a taxa de juros mensal e M é o número de meses.

Solução:

```
C:\Temp\Exercicio1.m

format bank;
A = 18500;
M = 12*4;
Desconto = 1500;

% Primeira opção

R = 0.24/100;
P1 = A*(R*(1+R)^M/((1+R)^M-1))
Total_1= P1*M

% Segunda opção

R = 0.74/100;
P2 = (A-Desconto)*(R*(1+R)^M/((1+R)^M-1))
Total_2 = P2*M

Diferenca = Total_2 - Total_1
Dif_percentual = Diferenca/Total_1*100
```

Exercício 02:

O polônio, um elemento radioativo, tem uma meia-vida de 140 dias (perde metade da sua massa em 140 dias). Começando com 10 gramas de polônio hoje, quanto vai restar ao fim de cada uma das próximas dez semanas?

Dado: quantidade restante = quantidade inicial * $(0,5)^{\text{tempo}/\text{meia vida}}$

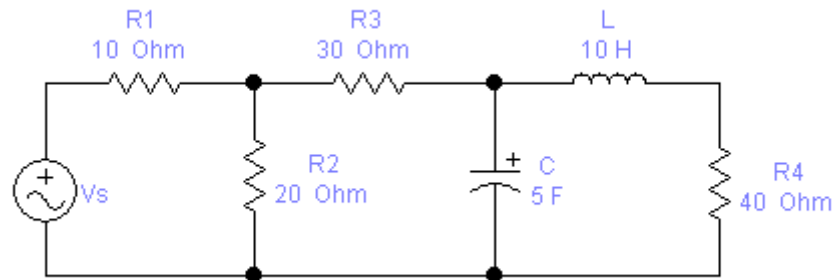
Solução:

```
C:\Temp\Exercicio2.m

format bank;
quant_inicial = 10;
meia_vida = 140;
tempo = 7:7:10*7
quant_rest = quant_inicial*.5.^(tempo/meia_vida)
```

Exercício 03:

Resolver o circuito mostrado na figura abaixo e apresentar as formas de onda para v_s , v_{R2} , v_C , v_{R4} .



Onde $v_s = 5 \cos(\omega t) [V]$

$f = 60 \text{ Hz}$

$\omega = 377 \text{ rad/s}$

Solução:

i. Escrevendo as três equações de malha para o circuito, temos:

$$V_s = (R_1 + R_2)I_1 - R_2I_2 \quad (1)$$

$$-R_2I_1 + (R_2 + R_3 - \frac{j}{\omega C})I_2 + \frac{j}{\omega C}I_3 = 0 \quad (2)$$

$$-R_2I_1 + (R_2 + R_3 - \frac{j}{\omega C})I_2 + \frac{j}{\omega C}I_3 = 0 \quad (3)$$

ii. Montando as matrizes de impedância e o vetor independente (fonte):

$$\text{imp} = \begin{bmatrix} R_1 + R_2 & -R_2 & 0 \\ -R_2 & R_2 + R_3 - \frac{j}{\omega C} & \frac{j}{\omega C} \\ 0 & \frac{j}{\omega C} & R_4 + j\omega L - \frac{j}{\omega C} \end{bmatrix} = \begin{bmatrix} 30 & -20 & 0 \\ -20 & 50 - j530 \cdot 10^{-6} & j530 \cdot 10^{-6} \\ 0 & 0 + j530 \cdot 10^{-6} & 40 + j3769 \end{bmatrix}$$

$$\text{source} = \begin{bmatrix} V_s \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}$$

iii. Resolvendo o sistema de equações lineares acima obtemos um "vetor de correntes"

$$V = RI$$

$$\text{Source} = \text{Imp} * \text{Curent}$$

$$\text{Curent} = \text{Source} * [\text{Imp}]^{-1}$$

Entre com os dados no MATLAB, corrigindo eventuais erros nas linhas abaixo:

```
>> imp = [30, -20, 0; -20, 50-530E-6j, 530E-6j; 0, 530E-6j, 40-3769j]
>> source = [5, 0, 0]
```

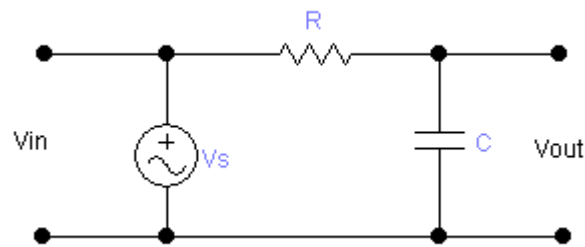
Relembrando alguns comandos já vistos, execute a inversão da matriz imp

```
>>
```

Com a matriz inversa, calculamos o "vetor de correntes".

Exercício 4:

A proposta do exercício é levantar o comportamento do circuito acima em CA.

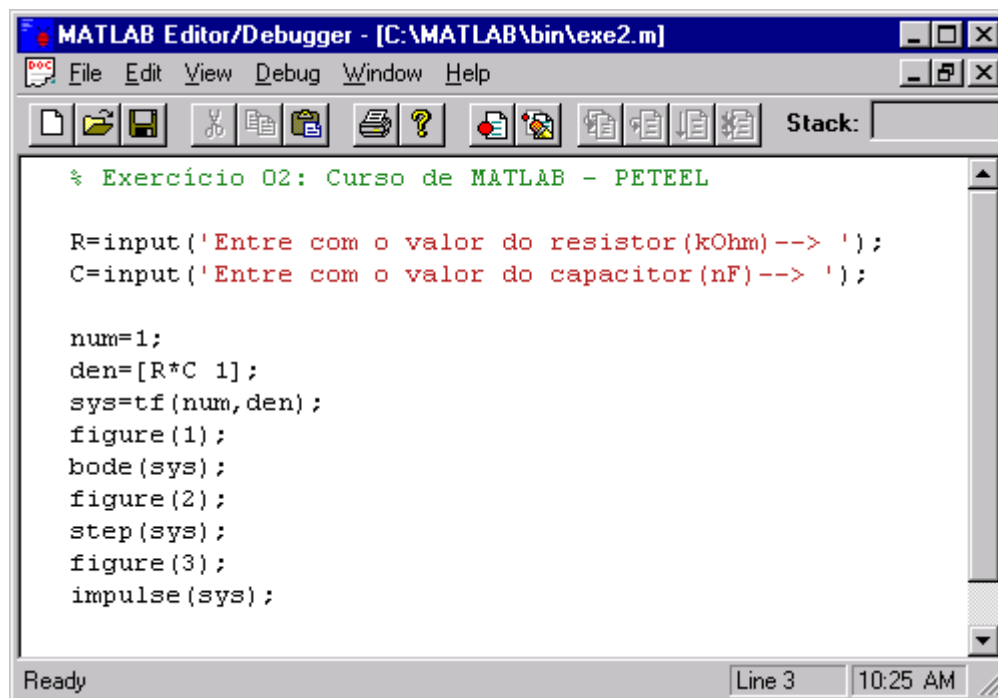


Sendo $R=1\text{K}\Omega$, $C=1\mu\text{F}$.

A análise do circuito fornece a seguinte função de transferência:

$$\frac{v_o}{v_i} = \frac{1}{1 + j\omega RC}$$

Vamos proceder com a solução do exercício criando um m-file:



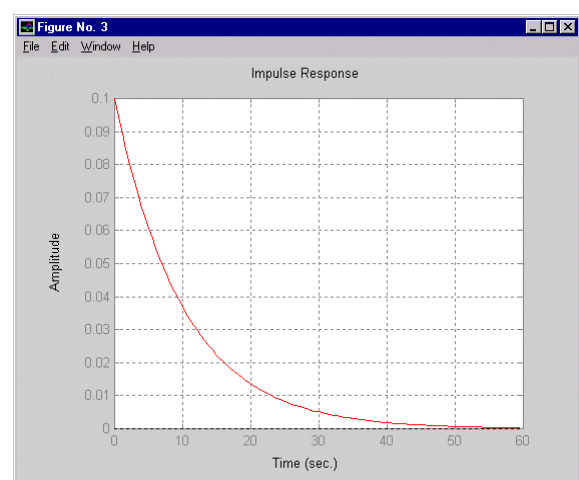
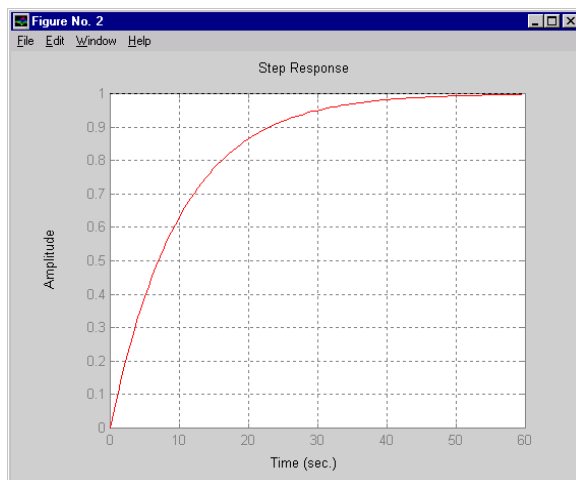
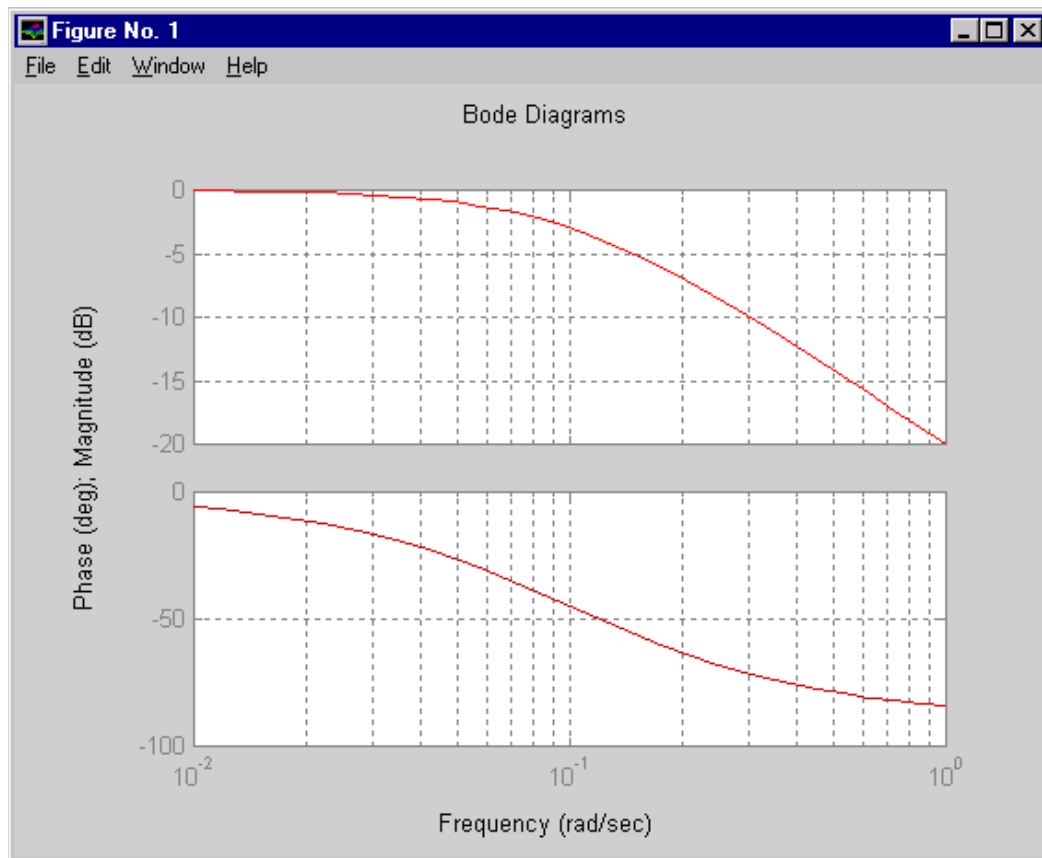
```
% Exercício 02: Curso de MATLAB - PETEEL

R=input('Entre com o valor do resistor(kOhm)--> ');
C=input('Entre com o valor do capacitor(nF)--> ');

num=1;
den=[R*C 1];
sys=tf(num,den);
figure(1);
bode(sys);
figure(2);
step(sys);
figure(3);
impz(sys);
```

The image shows a screenshot of the MATLAB Editor/Debugger window. The title bar reads "MATLAB Editor/Debugger - [C:\MATLAB\bin\exe2.m]". The menu bar includes File, Edit, View, Debug, Window, and Help. Below the menu bar is a toolbar with various icons. The main text area contains the MATLAB code for Exercise 02. The status bar at the bottom shows "Ready", "Line 3", and "10:25 AM".

Execute o programa e confira os resultados.





12 - Referências bibliográficas

- 1 – HANSELMAN, Duane & LITTLEFIELD, Duane. “MATLAB 5 – Guia do Usuário”
Editora Makron Books – São Paulo – 1ª Edição – 1999.
- 2 – GEROMEL, J. C. “PC-MATLAB”
The MathWorks, Inc. 1986.
- 3 – Fontes na Internet.



13 – Anexo: Listagem de Comandos