

Guia MatLab

para alunos de Engenharia Elétrica,
Computação e Automação

por Jan Krueger Siqueira

Editora Wally Salami

Este guia foi desenvolvido para aplicações em MatLab 7.0, embora vários dos recursos também estejam disponíveis em versões anteriores.

Por outro lado, pode haver funções que não funcionem na versão 7.0, pois isso dependerá talvez de quais bibliotecas foram selecionadas durante a instalação.

Índice

Capítulo 1 – Prólogo	
1.1 – Uso do MatLab	05
1.2 – MatLab x Maple	05
Capítulo 2 – Comandos Básicos	
2.1 – Declarações e Operações Básicas	08
2.2 – Comandos de Ajuda	11
2.3 – Comandos para a Tela de Comandos	12
Capítulo 3 – Vetores e Matrizes	
3.1 – Notação e Declaração de Vetores no MatLab	14
3.2 – Operações e Funções com Vetores	16
3.3 – Matrizes: o Foco do MatLab	18
3.4 – Matrizes Multidimensionais	21
Capítulo 4 – Funções Matemáticas	
4.1 – Aritmética e Álgebra Básicas	22
4.2 – Trigonométricas	22
4.3 – Equações e Polinômios	23
4.4 – Álgebra Linear	23
4.5 – Cálculo Diferencial e Integral	24
4.6 – Sinais e Sistemas	28
4.7 – Outras Funções	31
Capítulo 5 – Plotando Gráficos	
5.1 – A Família de Funções para Plotar	33
5.2 – Alguns Detalhes: Título, Legendas, Etc	37
5.3 – Plotando em 3D	38
5.4 – Manipulando a Janela do Gráfico	39
5.5 – Acessando Dados do Gráfico	40
Capítulo 6 – O Editor de Funções e Rotinas	
6.1 – Vantagens do Editor	41
6.2 – Criando Funções e Rotinas	41
6.3 – Condicionais e Loops: “if”, “switch”, “while” e “for”	43
6.4 – Recursão e Variáveis Globais	45
6.5 – Debugando	47
Capítulo 7 – Funções de Entrada e Saída (I/O)	
7.1 – Strings	48
7.2 – Capturando e Imprimindo na Tela de Comandos	48
7.3 – Arquivos de Texto, Som e Imagem	49
7.4 – Execução do MatLab e de Outros Aplicativos	50
Capítulo 8 – Programando com Qualidade	
8.1 – Por que Isso Vale a Pena?	52
8.2 – Como NÃO Fazer	52
8.3 – Como Fazer	53
Capítulo 9 – Exemplos Práticos	

9.1 – Problema de Probabilidade	58
9.2 – Criando um Polinômio	59
9.3 – “Animação Gráfica”	60
9.4 – Biot-Savart Numérico	62
9.5 – Documentador de “*.m”s	63

Capítulo 1 – Prólogo

1.1 – Uso do MatLab

Diversos professores falam: “MatLab é uma ferramenta muito boa”, “Façam isso no MatLab” ou “Fiz uma coisa no MatLab para mostrar para vocês”. Mas o que faz desse programa uma ferramenta interessante?

Basicamente, trata-se de um software excelente para cálculos numéricos em massa, otimizado para uso de vetores e matrizes (daí o nome “MatLab”). Além disso, ele conta com ótimos recursos para plotagem, para simulação de sistemas diferenciais (Simulink) e outros muitos que jamais iremos conhecer. Se instalarmos todas as ferramentas disponíveis da versão 7.0, o espaço ocupado no HD pode ser bem maior do que o esperado (e do que o necessário).

Além do mais, seu editor de funções e rotinas oferece um espaço bem amigável aos programadores, contando inclusive com cores de identificação de código e com um debugger.

1.2 – MatLab x Maple

Talvez muitos alunos prefiram usar o programa Maple para efetuar seus cálculos de engenharia, por terem tido com ele um contato mais prematuro. Há quem faça até discussões calorosas sobre qual dos dois é o melhor software.

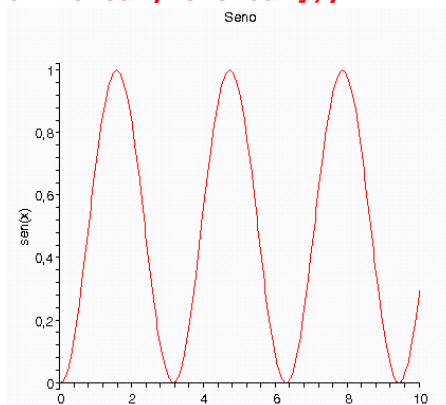
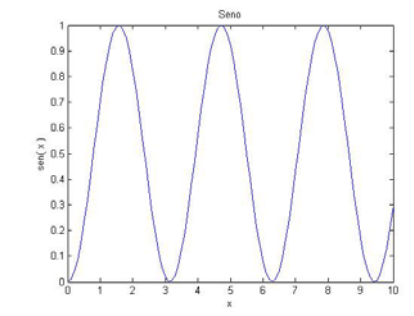
Pessoalmente, creio que cada um tem sua utilidade. Vejamos as vantagens do Maple:

Aspectos	Maple	MatLab
Expressões Literais	<pre>> sin(a)^2+cos(a)^2 ; sin(a)² + cos(a)² > simplify(%); 1 > int(sec(x),x); ln(sec(x) + tan(x)) > int(1/sqrt(2*Pi)*exp(- 0.5*x^2), x=-infinity .. infinity); 1.</pre>	<pre>>> sin(a)^2+cos(a)^2 ??? Undefined function or variable 'a'. >> syms x ; >> a = sin(x)^2 + cos(x)^2 a = sin(x)^2+cos(x)^2 >> simplify(a) ans = 1 >> int(1/sqrt(2*pi)*exp(-0.5*x^2), -inf, inf) ans =</pre>

		7186705221432913/18014398509481984 *2^(1/2)*pi^(1/2)
Exibição dos Dados	<pre>> (a^2 + 5*log(x/y)) / (sqrt(11) + Pi) ;</pre> $\frac{a^2 + 5 \ln\left(\frac{x}{y}\right)}{\sqrt{11} + \pi}$	<pre>>> syms a x y >> (a^2 + 5*log(x/y)) / (sqrt(11) + pi) ans = 562949953421312/3635653209253651*a^ 2+2814749767106560/3635653209253651 *log(x/y)</pre>

Pois é, o Maple trabalha bem com expressões literais e numa exibição melhor. Agora vejamos as vantagens do MatLab:

Aspectos	Maple	MatLab
Uso de vetores e matrizes	<pre>> A:=matrix(2,2,[3,3,3,3]);</pre> $A := \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}$ <pre>> B:=matrix(2,2,[2,2,2,2]);</pre> $B := \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$ <pre>> evalm(A.B);</pre> $\begin{bmatrix} 12 & 12 \\ 12 & 12 \end{bmatrix}$	<pre>>> A = [[3 3] ; [3 3]] B = [[2 2] ; [2 2]] A = 3 3 3 3 B = 2 2 2 2 >> A*B ans = 12 12 12 12 >> A.*B ans = 6 6 6 6</pre>
Cálculo em Massa	<pre>> A:=[0,Pi/2,Pi,3*Pi/2,2*Pi];</pre> $A := \left[0, \frac{1}{2}\pi, \pi, \frac{3}{2}\pi, 2\pi \right]$ <pre>> sin(A);</pre> <p>Error, invalid input: sin expects its 1st argument, x, to be of type algebraic, but received [0, 1/2*Pi, Pi, 3/2*Pi, 2*Pi]</p>	<pre>>> x = 0 : pi/2 : 2*pi x = 0 1.5708 3.1416 4.7124 6.2832 >> sin(x) ans = 0 1.0000 0.0000 -1.0000 -0.0000</pre>

	<pre>> for i from 0 to 4 do v[i] := sin(Pi*i/2); end do; v₀ := 0 v₁ := 1 v₂ := 0 v₃ := -1 v₄ := 0</pre>	
Funções	<pre>> funcao := x-> if x>=0 then sin(x) else 1/x end if; funcao := x → if 0 ≤ x then sin(x) else $\frac{1}{x}$ end if; > funcao(-2); -$\frac{1}{2}$ > funcao(2*Pi); Error, (in funcao) cannot determine if this expression is true or false: -2*Pi <= 0 > funcao(evalf(2*Pi,1000)); -1.795864770 10⁻¹⁰</pre>	<pre>function retorno = funcao (x) if x >=0 retorno = sin(x); else retorno = 1 / x ; end >> funcao(-2) ans = -0.5000 >> funcao(2 * pi) ans = -2.4493e-016</pre>
Gráficos	<pre>> plot((sin(x))^2, x=0..10, title=Seno,labels=["x", "sen(x)],labeldirections=[h orizontal,vertical]);</pre> 	<pre>>> plot(0 : 0.1 : 10 , sin(0 : 0.1 : 10) .^ 2) >> title('Seno') >> xlabel('x') >> ylabel('sen(x)')</pre> 

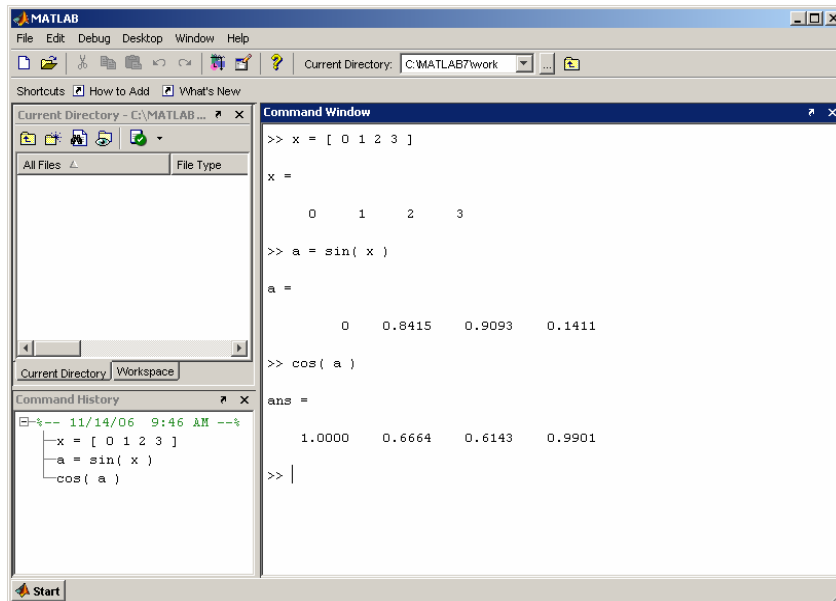
De fato, o MatLab trabalha muito bem com matrizes e cálculos em massa, além de possuir um aspecto de programação melhor. A manipulação dos gráficos pode ser mais interessante também, dependendo da aplicação.

Moral da história: se quisermos uma análise mais matemática, mais literal, usamos o Maple; se quisermos trabalhar com muito dados numéricos e gráficos, e programar com vários funções/módulos, usamos o MatLab.

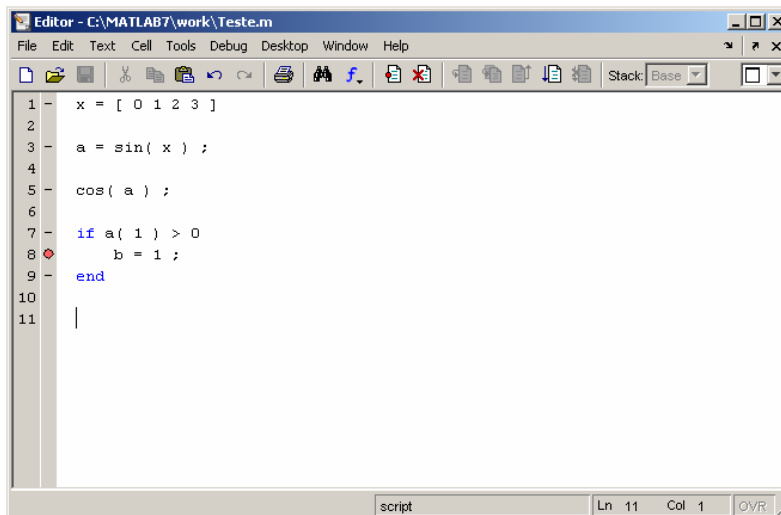
Capítulo 2 – Comandos Básicos

2.1 – Declarações e Operações Básicas

Há dois modos de se trabalhar no MatLab: usando a tela de comandos e usando o editor de texto. Para início de conversa, usaremos a tela de comandos – mas tudo o que é feito aqui pode ser usado no editor.



→ tela de comandos



→ editor de texto

OBS: no lado direito da tela de comandos, existe um quadro onde se podem visualizar os arquivos do diretório atual (Current Directory) e as variáveis declaradas até o momento (Workspace). Logo abaixo, temos o histórico dos comandos digitados.

Começamos do princípio então. Digitamos o comando sempre em frente do “>>”, e a resposta aparece embaixo, como num prompt.

Para **declarar variáveis**, fazemos o mais óbvio dos comandos.


```

>> x = 2

x =

                2

>> x = x + 2.5

x =

                4.5

>> x = -5.6 + 3*i

x =

    -5.6000 + 3.0000i

>> x = 'string'

x =

string

```

Repare que não se trata de uma linguagem tipada (ou seja, uma variável pode ser ora um número inteiro, ora uma string, etc, sem precisarmos declarar seu tipo).

Repare também que o uso do **ponto e vírgula** não é obrigatório. Quando posto, ele apenas omite a resposta na tela de comandos (mas armazena o valor). E digitando apenas o nome da variável, seu valor aparece (caso já tenha sido declarado).

```

>> a
??? Undefined function or variable 'a'.
>> a = 3 ;
>> a

a =

                3

```

A **notação científica** pode ser usada também. Para os que não sabem, $1.43e-3$ (escreva tudo junto, sempre) significa “1.43 vezes 10 elevado a -3” = 0.00143.

```

>> a = 12.67e3

a =

                12670

```

Algumas contas agora. As quatro operações básicas – **adição**, **subtração**, **multiplicação** e **divisão** – possuem os símbolos universais de todas (ou boa parte) das linguagens → +, -, * e /, respectivamente. Lembre-se que a prioridade vai para a multiplicação e divisão, de forma que usamos parêntesis nas expressões mais complexas.

Ah, sim, você pode executar mais de um comando de uma só vez. Basta escrever um deles e apertar **Shift + Enter**, e então ocorre a quebra de linha.

```

>> a = 2.6

a =

    2.6000

>> b = 1.6

b =

    1.6000

>> a + b

ans =

    4.2000

>> ( a + b ) * ( a - b ) - a / b
    a * b

ans =

    2.5750

ans =

    4.1600

```

A **potenciação** é usada com o operador “^”. Já vale a pena citar aqui que o número imaginário $\sqrt{-1}$ é representado por “i” ou por “j”, a menos que haja variáveis com esses nomes.

```

>> c = a ^ 2 + b ^ 2

c =

    9.3200

>> i ^ 2

ans =

    -1

>> j ^ 3

ans =

    0 - 1.0000i

>> i = 2

i =

    2

>> i ^ 2

```

```
ans =  
4
```

As outras operações são obtidas através de chamadas de função, como na maioria das linguagens. Basicamente, seu formato básico é *nome_da_função(argumento_1, argumento_2, ...)*. É o caso da raiz quadrada: `sqrt(numero)`.

```
>> sqrt( 4.04 )  
ans =  
2.0100  
  
>> sqrt( -4.04 )  
ans =  
0 + 2.0100i
```

As demais operações matemáticas serão apresentadas no Capítulo 4.

Algumas outras abstrações matemáticas e computacionais, como o **infinito** e o “Not a Number” (**NaN**), também estão presentes na linguagem.

```
>> 1 / 0  
Warning: Divide by zero.  
ans =  
Inf  
  
>> 0 / 0  
Warning: Divide by zero.  
ans =  
NaN
```

2.2 – Comandos de Ajuda

Antes de continuarmos com a apresentação de funcionalidades, é interessante já mostrar o caminho para aqueles que gostam de aprender sozinhos, por exploração. A maioria das versões do MatLab não possui um guia de ajuda em janela decente. No entanto, através da própria tela de comandos, podemos encontrar o que queremos com certa facilidade.

Qual seria o comando para ajuda então? Elementar, meu caro Watson: digitamos **help nome_da_função** para sabermos mais sobre essa função! Não é genial?

```
>> help sqrt  
  
SQRT Square root.  
SQRT(X) is the square root of the elements of X. Complex  
results are produced if X is not positive.  
  
See also SQRTM.
```

Ok, mas e se eu não souber o nome da função de que preciso? Não se aflija, pois para isso existe o comando **lookfor palavra**. Com a lista de funções encontradas com *palavra* em sua descrição, usa-se em seguida o comando **help** para investigar melhor seu funcionamento. E para pesquisar uma expressão com mais de uma palavra, coloque aspas simples nela (**lookfor 'palavra1 palavra2 ...'**)

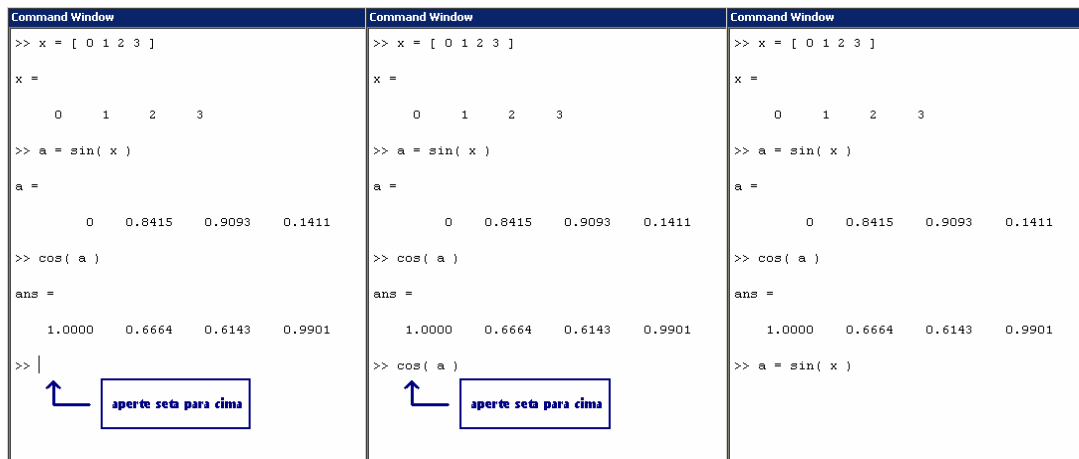
```
>> lookfor fourier
FFT Discrete Fourier transform.
FFT2 Two-dimensional discrete Fourier Transform.
FFTN N-dimensional discrete Fourier Transform.
IFFT Inverse discrete Fourier transform.
IFFT2 Two-dimensional inverse discrete Fourier transform.
IFFTN N-dimensional inverse discrete Fourier transform.
XFOURIER Graphics demo of Fourier series expansion.
DFTMTX Discrete Fourier transform matrix.
INSTDFFT Inverse non-standard 1-D fast Fourier transform.
NSTDFFT Non-standard 1-D fast Fourier transform.

>> lookfor 'square root'
REALSQRT Real square root.
SQRT Square root.
SQRTM Matrix square root.
```

Não é a oitava maravilha do mundo, mas já quebra um galho.


2.3 – Comandos para a Tela de Comandos

Diferente do que ocorre no Maple, não é possível modificar ou reexecutar um comando da linha de onde ele foi criado. Porém é possível reobtê-lo na linha corrente com as **setas cima e baixo do teclado**, tal qual no prompt do DOS, por exemplo. Isso poupa o inútil trabalho de redigitação ou de Ctrl+C e Ctrl+V.



Veja que, colocando a(s) inicial(is), pode-se retomar comandos de modo mais rápido.

Command Window	Command Window
<pre>>> x = [0 1 2 3] x = 0 1 2 3 >> a = sin(x) a = 0 0.8415 0.9093 0.1411 >> cos(a) ans = 1.0000 0.6664 0.6143 0.9901 >> a </pre>	<pre>>> x = [0 1 2 3] x = 0 1 2 3 >> a = sin(x) a = 0 0.8415 0.9093 0.1411 >> cos(a) ans = 1.0000 0.6664 0.6143 0.9901 >> a = sin(x)</pre>



E, depois de digitar tantas declarações e operações, a tela já deve estar uma zona, e a memória cheia de variáveis inúteis. Isso sem falar que poderíamos estar plotando vários gráficos também. Para limpar tudo isso sem sair do programa, usamos os comandos:

Operação	Função
Limpar Comandos da Tela	<code>clc</code>
Limpar Variáveis da Memória (ou apenas uma em específico)	<code>clear</code> <code>clear <i>variavel</i></code>
Fechar Janela de Gráfico (ou fechar todas as janelas)	<code>close</code> <code>close all</code>

Capítulo 3 – Vetores e Matrizes

Vetores e matrizes são o grande diferencial do MatLab. Este capítulo é muito importante se quisermos rapidez e eficiência nos cálculos.

3.1 – Notação e Declaração de Vetores no MatLab

Como se sabe, um vetor nada mais é do que uma fila de números com algum significado (ou não). A vantagem que ele traz aqui no MatLab é poder operar com muitos valores de uma só vez.

Primeiramente, alguns pontos importantes: declara-se o vetor com os **elementos entre colchetes (com ou sem vírgulas)**, o **primeiro index do vetor é 1** e os elementos são acessados com o **index entre parêntesis**.

```
>> vet = [ 0 , 1 , 2 , 3 ]

vet =

     0     1     2     3

>> vet = [ 0 1 2 3 4 ]

vet =

     0     1     2     3     4

>> vet( 0 )
??? Subscript indices must either be real positive integers or
logicals.

>> vet( 1 )

ans =

     0

>> vet( 4 )

ans =

     3
```

Mas e se quisermos um vetor de 0 a 9? Não precisa digitar tudo, basta usar a notação compacta *val_inicial : val_final*. E para ir de 0 a 9 de dois em dois? Usamos a notação *val_inicial : val_passo : val_final*

```
>> vet = 0 : 9

vet =

     0     1     2     3     4     5     6     7     8     9

>> vet = 0 : 2 : 9
```

```
vet =  
      0      2      4      6      8
```

E daí as brincadeiras começam. Por exemplo, pegando apenas um certo **trecho do vetor**.

```
>> vet = 0 : 9  
vet =  
      0      1      2      3      4      5      6      7      8      9  
  
>> vet( [1 3 7] )  
ans =  
      0      2      6  
  
>> vet( 2 : 4 )  
ans =  
      1      2      3  
  
>> vet( 1 : 3 : 9 )  
ans =  
      0      3      6
```

Concatenando (unindo) vetores.

```
>> vet1 = [ 1 2 3 4 5 6 ] ;  
    vet2 = [ 7 8 9 ] ;  
>> [ vet1 vet2 ]  
  
ans =  
      1      2      3      4      5      6      7      8      9  
  
>> [ vet1( 1 : 3 ) vet1( 6 ) vet2( 1 ) ]  
  
ans =  
      1      2      3      6      7
```

Podem-se **alterar elementos** de um vetor também, ou **eliminá-los**.

```
>> vet1 = [ 1 2 3 4 5 6 ] ;  
>> vet1( 1 ) = 0  
  
vet1 =  
      0      2      3      4      5      6  
  
>> vet1( 1 ) = [ ]
```

```

vet1 =
      2      3      4      5      6
>> vet1( 2 : 4 ) = [ ]
vet1 =
      2      6

```

Não é necessário decorar todas essas manhas agora. Consulte o guia conforme necessário, e aprenda conforme o uso.

3.2 – Operações e Funções com Vetores

Um ponto fundamental é fazer aritméticas básicas com vetores. Vale lembrar que, para isso, os vetores precisam ter a mesma dimensão. **Adição, subtração e multiplicação por um escalar** são bem simples.

```

>> vetA = [ 1 2 3 ] ;
      vetB = [ 1 1 1 ] ;
>> vetA + vetB

ans =
      2      3      4

>> 2 * vetA

ans =
      2      4      6

>> vetA - 2 * vetB

ans =
     -1      0      1

```

Multiplicação, divisão e potenciação já são diferentes. Tais operações são definidas também num contexto de matrizes, o que faz necessário criar uma diferenciação entre operação de vetores / matrizes e operação de elementos de vetores / matrizes.

Portanto, para **multiplicar, dividir ou elevar os elementos dum vetor**, colocamos um **ponto na frente do operador**.

```

>> vetA = [ 2 3 ] ;
      vetB = [ 4 5 ] ;
>> vetA * vetB
??? Error using ==> *
Inner matrix dimensions must agree.

>> vetA .* vetB

ans =
      8      15

```



```

>> vetA ./ vetB

ans =

    0.5000    0.6000

>> vetA .^ vetB

ans =

    16    243

>> vetB ^ 2
??? Error using ==> ^
Matrix must be square.

>> vetB .^ 2

ans =

    16    25

```

A seguir, uma tabela de funções úteis para vetores.

Operação	Função
Módulo (Norma) do Vetor	modulo = norm(vet)
Produto Escalar de Dois Vetores	prodEscalar = dot(vet1 , vet2)
Produto Vetorial de Dois Vetores (R^3)	vet_result = cross(vet1 , vet2)
Total de Elementos (Comprimento)	totalElementos = length(vet)
Soma dos Elementos	soma = sum(vet)
Produto dos Elementos	produto = prod(vet)
Média dos Elementos	media = mean(vet)
Maior dos Elementos (ou o maior elemento e seu índice)	elem = max(vet) [elem , indice] = max(vet)
Menor dos Elementos (ou o menor elemento e seu índice)	elem = min(vet) [elem , indice] = min(vet)
Encontrar Índices dos Elementos que Satisfaçam Certa Condição	vet_indices = find(condição) <exemplo: vet_indices = find(vet > 0) retorna índices dos elementos positivos >

Existem muitas mais. Use o comando “lookfor” para buscar uma outra.

Por fim, é importante citar que **muitas funções aceitam vetores e matrizes como argumento**, aplicando suas operações em todos os elementos. Isso é feito de maneira otimizada, para agilizar cálculos em massa. É o grande diferencial do MatLab.

```

>> vet = [ 0 3 6 9 ]

vet =

    0    3    6    9

>> sqrt( vet )

ans =

```

```

0    1.7321    2.4495    3.0000
>> sin( vet )
ans =
0    0.1411   -0.2794    0.4121

```

3.3 – Matrizes: o Foco do MatLab

A **declaração** e a notação de matriz são similares às de vetor. Cada linha é composta por elementos entre colchetes (opcionais, porém recomendados para facilitar visualização), e todas elas são postas, por sua vez, num outro par de colchetes, com **ponto-e-vírgulas** separando-as.

```

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

```

Para acessar um elemento, usamos a notação **matriz(linha , coluna)**

```

>> matriz = [ [1 2 3] ; [4 5 6] ; [7 8 9] ]
matriz =
    1    2    3
    4    5    6
    7    8    9
>> matriz( 1 , 2 )
ans =
    2
>> matriz( 3 , 1 )
ans =
    7

```

Para acessar uma linha / coluna inteira, usamos a notação **matriz(linha , :)** / **matriz(: , coluna)**.

```

>> matriz( 2 , : )
ans =

```

```

      4      5      6
>> matriz( : , 3 )

ans =

      3
      6
      9

```

Para calcularmos a **matriz transposta**, usamos o operador aspas simples (').

```

>> matriz'

ans =

      1      4      7
      2      5      8
      3      6      9

>> matriz_linha = [ 1 2 3 4 ]

matriz_linha =

      1      2      3      4

>> matriz_coluna = matriz_linha'

matriz_coluna =

      1
      2
      3
      4

```

Pode-se **modificar elementos** da matriz, ou então linhas e colunas inteiras:

```

>> matriz = [ [1 2 3] ; [4 5 6] ; [7 8 9] ] ;
>> matriz( 1 , 2 ) = 0

matriz =

      1      0      3
      4      5      6
      7      8      9

>> matriz( 1 , : ) = 10

matriz =

     10     10     10
      4      5      6
      7      8      9

>> matriz( [ 1 1 ] , [ 1 3 ] ) = -2

matriz =

     -2     10     -2

```

```

4      5      6
7      8      9

```

```
>> matriz( [ 1 3 ] , : ) = 5.5
```

```
matriz =
```

```

5.5000    5.5000    5.5000
4.0000    5.0000    6.0000
5.5000    5.5000    5.5000

```

As operações básicas funcionam como foi dito na seção de vetores: **adição e subtração** com “+” e “-”, sempre; **multiplicação e divisão entre elementos** com o ponto na frente (“.*” e “./”); **multiplicação de matrizes** com “*”; **divisão de matrizes** (uma matriz vezes a inversa da outra) com “/”.

```

>> matriz_A = [ [ 1 1 ] ; [ 1 2 ] ] ;
matriz_B = [ [ 0 1 ] ; [ 1 1 ] ] ;
matriz_C = [ [ 1 2 3 ] ; [ 4 5 6 ] ] ;
matriz_D = [ [ 1 1 1 ] ; [ 4 4 4 ] ] ;
>> matriz_A + matriz_B
matriz_C + matriz_D

```

```
ans =
```

```

1      2
2      3

```

```
ans =
```

```

2      3      4
8      9     10

```

```

>> matriz_A + matriz_C
??? Error using ==> +
Matrix dimensions must agree.

```

```
>> matriz_A * matriz_B
```

```
ans =
```

```

1      2
2      3

```

```

>> matriz_C * matriz_D
??? Error using ==> *
Inner matrix dimensions must agree.

```

```

>> matriz_A .* matriz_B
matriz_C ./ matriz_D

```

```
ans =
```

```

0      1
1      2

```

```
ans =
```

```

1.0000    2.0000    3.0000
1.0000    1.2500    1.5000

>> matriz_A / matriz_B

ans =

    0     1
    1     1

```

As funções aritméticas e algébricas (raiz, seno, logaritmo, ...) com matrizes atuam em cada um dos seus elementos. Já certas funções apresentadas na seção de vetores (soma, produto, média, ...) atuam geralmente nas colunas e retornam um vetor de respostas. Consulte a ajuda das funções para descobrir essas diferenças.

A seguir, uma tabela de funções úteis para matrizes.

Operação	Função
Criar Matriz N x N Identidade	<code>matriz = eye(N)</code>
Criar Matriz M x N de Zeros	<code>matriz = zeros(M , N)</code>
Criar Matriz M x N de 1s	<code>matriz = ones(M , N)</code>
Redimensionar Matriz para M x N	<code>matriz2 = reshape(matriz1 , M , N)</code>
Rodar Matriz em k*90°	<code>matriz2 = rot90(matriz1 , k)</code>
Espelhar Matriz da Esquerda para Direita	<code>matriz2 = fliplr(matriz1)</code>
Espelhar Matriz de Cima para Baixo	<code>matriz2 = flipud(matriz1)</code>

Veja que “zeros” e “ones” podem ser usadas para vetores também; basta passar o parâmetro $M = 1$. A função “fliplr” espelha o vetor, e a “flipud” não o modifica em nada.

Confira outras funções para matrizes em 4.4 – Álgebra Linear.

3.4 – Matrizes Multidimensionais

Em algumas aplicações, é interessante trabalhar com matrizes de dimensão 3 ou superior. Para tal, fazemos uma declaração em etapas. Veja o exemplo da declaração de uma matriz tridimensional 2 x 2 x 2:

```

>> matriz3D = [ 1 2 ; 3 4 ]

matriz3D =

    1     2
    3     4

>> matriz3D( : , : , 2 ) = [ 5 6 ; 7 8 ]

matriz3D( : , : , 1 ) =

    1     2
    3     4

matriz3D( : , : , 2 ) =

    5     6
    7     8

```

Capítulo 4 – Funções Matemáticas

4.1 – Aritmética e Álgebra Básicas

Não preciso explicar muito aqui. Olhe a tabela e use as funções:

Operação	Função
Potenciação	x^n
Radiciação (Raiz Quadrada)	\sqrt{x}
Exponenciação (potência do número neperiano)	$\exp(x)$
Logaritmo Neperiano (ln)	$\log(x)$
Logaritmo Decimal	$\log_{10}(x)$
Logaritmo na Base 2	$\log_2(x)$
Módulo (para números complexos também)	$\text{abs}(x)$
Fatorial (n!)	$\text{factorial}(n)$
Combinação de N, K a K	$\text{nchoosek}(N, K)$
Módulo do Resto da Divisão de A por B	$\text{mod}(A, B)$
Arredondamento(para Cima, para Baixo ou para o Mais Próximo)	$\text{ceil}(x)$ $\text{floor}(x)$ $\text{round}(x)$

Não encontrou exatamente a função que queria? Algumas manhas então:

- para calcular a raiz “n” de “x”, faça “ $x^{(1/n)}$ ”
- para calcular o logaritmo de “x” na base “n”, faça “ $\log(x) / \log(n)$ ”

4.2 – Trigonométricas

As funções a seguir trabalham com **ângulos em RADIANOS**.

(em radianos)	Direta	Inversa (Arco-)
Seno	$\sin(x)$	$\text{asin}(x)$
Co-Seno	$\cos(x)$	$\text{acos}(x)$
Tangente	$\tan(x)$	$\text{atan}(x)$

As funções a seguir trabalham com **ângulos em GRAUS (degrees)**.

(em graus)	Direta	Inversa (Arco-)
Seno	$\text{sind}(x)$	$\text{asind}(x)$
Co-Seno	$\text{cosd}(x)$	$\text{acosd}(x)$
Tangente	$\text{tand}(x)$	$\text{atand}(x)$

Lembrando que as funções inversas retornam valores de ângulo entre $-\pi/2$ e $\pi/2$ radianos / -90° e $+90^\circ$. Muito cuidado quando for usá-las para não errar o quadrante desejado!

Por fim, as funções hiperbólicas.

	Direta	Inversa (Arco-)
Seno Hiperbólico	$\sinh(x)$	$\operatorname{asinh}(x)$
Co-Seno Hiperbólico	$\cosh(x)$	$\operatorname{acosh}(x)$
Tangente Hiperbólica	$\tanh(x)$	$\operatorname{atanh}(x)$

4.3 – Equações e Polinômios

Embora o MatLab não trabalhe com literais, ele pode sim ser útil em equações e polinômios. Repare que não usamos expressões como " $x^2 - 3x + 4$ ", mas sim os coeficientes dessa equação: $[1 -3 4]$. Confira a tabela abaixo:

Operação	Função
Resolver Equação Polinomial (Encontrar as Raízes)	$\text{vet_raizes} = \text{roots}(\text{vet_coef})$ < coeficientes do polinômio em ordem decrescente de grau no vetor vet_coef >
Montar Polinômio a Partir das Raízes	$\text{vet_coef} = \text{poly}(\text{vet_raizes})$
Encontrar Melhor Polinômio (Mínimos Quadrados)	$\text{vet_coef} = \text{polyfit}(\text{vet_x}, \text{vet_y}, N)$ < cria polinômio de grau N que melhor se aproxima dos pontos (x,y) >
Aplicar "x" num Polinômio	$y = \text{polyval}(\text{vet_coef}, x)$

A dupla `polyfit` e `polyval` seriam bem úteis nos laboratórios de física, onde plotamos os pontos experimentais e buscamos no "olhômetro" a **melhor reta** formada por eles. Bastaria usar " $\text{vet_coef} = \text{polyfit}(\text{pontos_x}, \text{pontos_y}, 1)$ " para obter os dois coeficientes da reta. Daí usaríamos a `polyval` para obter os pontos dessa reta e plotá-los (exemplo: " $\text{plot}([-10 : 10], \text{polyval}(\text{vet_coef}, [-10 : 10]))$ ").

CUIDADO: polinômios do tipo $x^4 + 4x^2 - 5x$ devem ser escritos na forma desenvolvida (no caso, $x^4 + 0x^3 + 4x^2 - 5x + 0$) e então se passam os coeficientes (no caso, $[1 0 4 -5 0]$)! O vetor $[1 4 -5]$ representaria o polinômio $x^2 + 4x - 5$!

Confira o exemplo 2 do capítulo 9 para ver uma aplicação dessas funções.

Por fim, vale lembrar novamente que o MatLab trabalha com números complexos, de forma que toda a equação polinomial terá solução (mesmo que aproximada).

4.4 – Álgebra Linear

E, como não poderia deixar de ser, temos uma coletânea de funções de Álgebra Linear para matrizes. Confira algumas na tabela abaixo.

Operação	Função
Determinante	$\text{determinante} = \text{det}(\text{matriz})$
Inversa	$\text{matriz_inversa} = \text{inv}(\text{matriz})$
Resolver Sistema Linear ($A \cdot X = B$)	$\text{matriz_x} = \text{linsolve}(A, B)$ < A é matriz quadrada e B é matriz coluna >

	<i>< x é matriz coluna com as soluções ></i>
AutoValores / Auto Vetores (eigenvalues / eigenvectors)	matriz_autoval = eig(matriz) <i>< retorna autovalores numa matriz coluna ></i> [mat_autoval , mat_autovet] = eig(matriz) <i>< retorna autovalores numa matriz diagonal e autovetores em colunas numa outra matriz ></i>
Decomposição LU	[L , U] = lu(matriz)
Decomposição QR	[Q , R] = qr(matriz)

exemplo: resolvendo sistemas de equações

$$\begin{cases} 5a + 2b = 9 \\ 3a - b = 1 \end{cases} \Rightarrow A * X = B \Rightarrow \begin{pmatrix} 5 & 2 \\ 3 & -1 \end{pmatrix} X = \begin{pmatrix} 9 \\ 1 \end{pmatrix}$$

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

A =

```
5    2
3   -1
```

B =

```
9
1
```

```
>> matriz_x = linsolve( A , B )
```

matriz_x =

```
1
2
```

Temos também um pouco de Cálculo Funcional (operações sobre matrizes $N \times N$, usadas geralmente para resolver sistemas diferenciais).

Operação	Função
Raiz Quadrada de Matriz	matriz2 = sqrtm(matriz1)
Exponencial de Matriz	matriz2 = expm(matriz1)
Logaritmo de Matriz	matriz2 = logm(matriz1)
Trigonômicas de Matriz	matriz2 = funm(matriz1 , @SIN) matriz2 = funm(matriz1 , @COS) matriz2 = funm(matriz1 , @SINH) matriz2 = funm(matriz1 , @COSH)

4.5 – Cálculo Diferencial e Integral

Primeiramente convém lembrar mais uma vez que o MatLab é uma excelente ferramenta para números. Portanto começaremos com as funções que fazem o Cálculo Numérico.

Operação	Função
Derivada Numérica	<code>vet_derivada = diff(vet_funcao) / valor_passo</code>
Integral Definida Numérica (quadrature)	<code>valorIntegral = quad(funcao , tempoIni , tempoFim)</code>
Resolução Numérica de Equação ou Sistema Diferencial de 1ª. Ordem	<code>[vet_tempo vet_y] = ode45(funcao , [tempoIni tempoFim] , vet_condIni)</code> <onde $y' = \text{função}$ >

Calma que a gente explica com exemplos.

A função “diff” apenas retorna as subtrações de cada elemento do vetor com o elemento anterior. Lembrando que a definição da derivada é

$$\frac{dy}{dt}(t_0) = \lim_{\Delta t \rightarrow 0} \frac{y(t_0 + \Delta t) - y(t_0)}{\Delta t},$$

podemos calculá-la com o resultado da função “diff” dividido pelo intervalo dum infinitesimal – um valor suficientemente pequeno.

exemplo: **derivada** de $y = \cos(t) * t$, do instante 0 a 5s

```
>> dt = 0.001 ;
>> vet_t = 0 : dt : 5 ;
>> vet_y = cos( vet_t ) .* vet_t ;
>> vet_derivada = diff( vet_y ) / dt ;
```

A função “quad” usa um método de Simpson alternativo e recebe um parâmetro função que pode ser passado por string ou numa referencia a alguma função já existente em arquivo “.m”.

exemplo: calcular a **integral** da função anterior no mesmo intervalo de tempo

```
>> quad( 'cos( t ) .* t' , 0 , 5 )

ans =

-5.5110
```

Resolver **equações ou sistemas diferenciais** já é um pouco mais complicado. Começemos com um exemplo simples. Para resolver a equação

$$2y'(t) + y(t) = \cos(t), \text{ com } y(0) = 10$$

e obter os valores de y de 0 a 5s, precisamos antes de tudo isolar o termo de primeira ordem. Isto é, escrever

$$y'(t) = \frac{1}{2}(\cos(t) - y(t)).$$

Agora precisamos criar uma função auxiliar no Editor de Texto – ele será abordado com mais calma no capítulo 6. Vá em “File > New > M File” e digite o seguinte:

```
function retorno = funcao ( tempo , y )
    retorno = ( cos( tempo ) - y ) / 2 ;
```

Salve o arquivo como “funcao.m”.
Daí digitamos o seguinte comando:

```
>> tempoInicial = 0 ;
>> tempoFinal = 5 ;
>> condicaoInicial = 10 ;
>> [ vet_tempo , vet_y ] = ode45( @funcao , [ tempoInicial
tempoFinal ] , condicaoInicial ) ;
```

Para equações de ordem superior, o segredo é transformá-la num sistema de equações de 1ª ordem. Vejamos o exemplo.

$$y''(t) + 5y'(t) + 6y(t) = \cos(t), \text{ com } y(0) = 10 \text{ e } y'(0) = 0$$

Reescrevendo para a forma de um sistema (fazemos $y'(t) = x(t)$ e, portanto, $y''(t) = x'(t)$) e já isolando os termos de 1ª ordem, teríamos o seguinte.

$$\begin{aligned} y'(t) &= x(t), \text{ com } y(0) = 10 \\ x'(t) &= \cos(t) - 5x(t) - 6y(t), \text{ com } x(0) = y'(0) = 0 \end{aligned}$$

Repetimos o que fizemos no caso anterior: criamos uma função auxiliar no Editor de Texto, acrescentando alguns detalhes. Veja que o segundo parâmetro da função agora é um vetor com as variáveis x e y – estabeleceremos que o primeiro elemento será relativo a $x(t)$ e o segundo a $y(t)$.

```
function vet_retorno = funcao ( tempo , vet_var )
    vet_retorno = zeros( 2 , 1 ) ;
    vet_retorno( 1 ) = cos( tempo ) - 5 * vet_var( 1 ) - 6 *
vet_var( 2 ) ;
    vet_retorno( 2 ) = vet_var( 1 ) ;
```

Salve o arquivo como “funcao.m”.
Daí digitamos o seguinte comando:

```
>> tempoInicial = 0 ;
>> tempoFinal = 5 ;
>> vet_condicaoInicial = [ 0 10 ] ;
>> [ vet_tempo , vet_var ] = ode45( @funcao , [ tempoInicial
tempoFinal ] , vet_condicaoInicial ) ;
```

Os valores de x e y estarão respectivamente em `vet_var(: , 1)` e `vet_var(: , 2)`.

O MatLab vem com uma ferramenta bem mais interessante para a resolução de sistemas diferenciais: o Simulink. No entanto, este guia não tratará do assunto, já que ele é abordado em outras matérias de engenharia.

Além do cálculo numérico, existe uma biblioteca de funções que trabalha com símbolos e literais, o que permite obter derivada e integral indefinidas e resolver equações diferenciais. Entretanto ressalta-se novamente que, para essa finalidade, o uso do Maple é recomendado.

Operação	Função
Declarar Símbolo Literal	<code>syms nomeSimbolo</code>
Calcular Derivada de uma Expressão (ou derivada parcial) (ou derivada de ordem N)	<code>diff(expressao)</code> <code>diff(expressao , variavel)</code> <code>diff(expressao , N)</code>
Calcular Integral de uma Expressão (ou integral com respeito a uma variável) (ou integral definida de A a B)	<code>int(expressao)</code> <code>int(expressao , variavel)</code> <code>int(expressao , A , B)</code>
Resolver Equação Diferencial	<code>dsolve(equacao)</code> <code>dsolve(equacao , condicoesIniciais)</code>

exemplo 1: calcular derivada segunda e integral de $\frac{t}{1+x^2}$ com respeito a x

```
>> syms x ;
>> syms t ;
>> symb_expressao = t / ( 1 + x^2 )

symb_expressao =

t/(1+x^2)

>> diff( symb_expressao , x , 2 )

ans =

8*t/(1+x^2)^3*x^2-2*t/(1+x^2)^2

>> int( symb_expressao , x )

ans =

t*atan(x)
```

exemplo 2: resolver $y'(t) = -ay(t)$ com $y(0) = 1$

```
>> symb_y = dsolve( 'Dy = -a*y' , 'y(0) = 1' )

symb_y =

exp(-a*t)
```

4.6 – Sinais e Sistemas

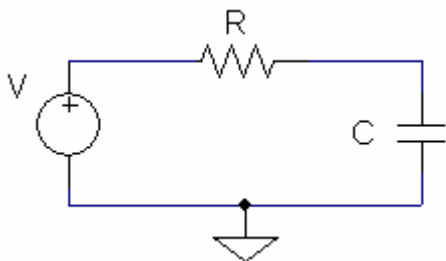
Se você pegou este guia apenas porque tem que fazer um trabalho de ELE 1030 – Sinais e Sistemas, esta é a sua chance! Só espero que outras partes desse guia tenham sido lidas também, porque senão não vai adiantar de muita coisa.

Conforme dito anteriormente, o MatLab não é muito forte com expressões literais, mas sim com as numéricas. Então todas as transformadas e funções de transferência serão aplicadas para um conjunto de pontos finitos, e não para outras funções ou sistemas.

A tabela abaixo resume as funções mais interessantes. Exemplos com elas virão na seqüência.

Operação	Função
Montar Função de Transferência	$H = \text{tf}(\text{vet_coef_num}, \text{vet_coef_denom})$ < numerador e do denominador com polinômios de s >
Resposta ao Degrau	$\text{step}(H)$ $\text{step}(H, \text{vet_tempo})$ $[\text{vet_Y} \text{ vet_tempo}] = \text{step}(H)$
Resposta ao Impulso	$\text{impulse}(H)$ $\text{impulse}(H, \text{vet_tempo})$ $[\text{vet_Y} \text{ vet_tempo}] = \text{impulse}(H)$
Resposta a Entradas Genéricas	$\text{lsim}(H, \text{vet_X}, \text{vet_tempo})$ $\text{vet_Y} = \text{lsim}(H, \text{vet_X}, \text{vet_tempo})$
Diagrama de Bode	$\text{bode}(H)$
Transformada Discreta de Fourier (Fast Fourier Transform)	$\text{vet_XF} = \text{fft}(\text{vet_X})$ < a primeira metade de vet_XF corresponde às frequências positivas >

exemplo 1: circuito RC série



$$R = 200 \, \Omega$$

$$C = 1 \, \text{mF}$$

$$H_c = \frac{V_c}{V} = \frac{1/RC}{s + 1/RC} = \frac{5}{s + 5}$$

A partir da função de transferência encontrada via análise do circuito, obteremos a resposta da tensão no capacitor para entradas impulso, degrau e exponencial decrescente na fonte.

```
>> Hc = tf( [ 5 ] , [ 1 5 ] )
```

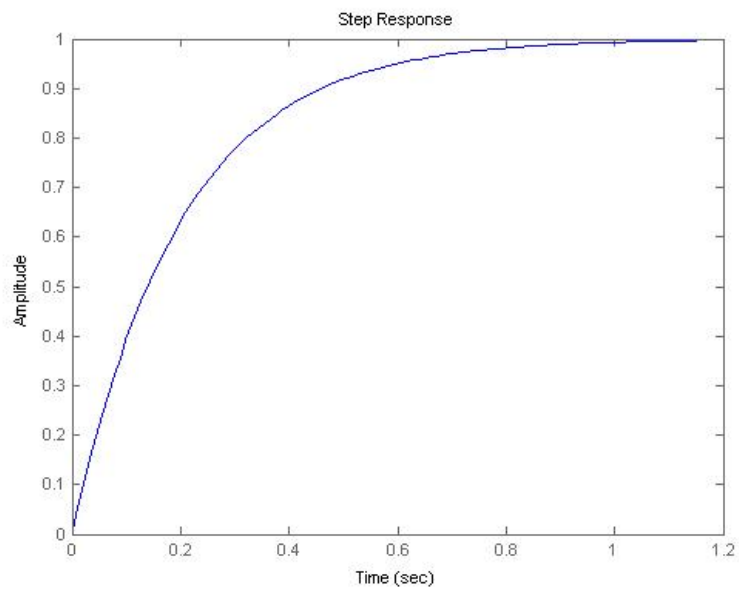
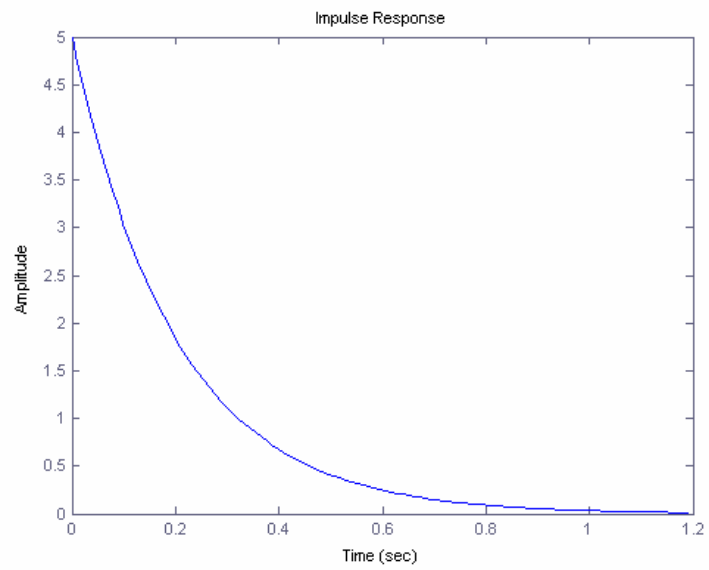
```
Transfer function:
```

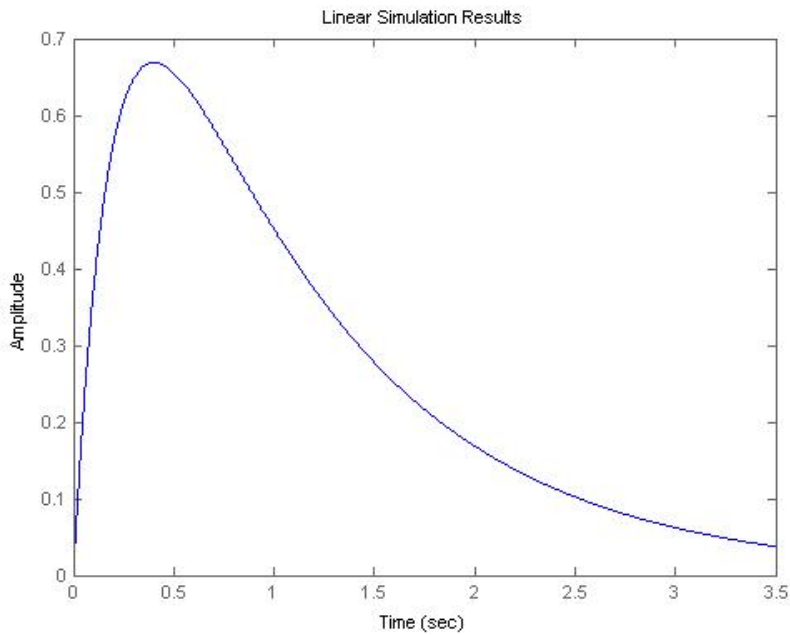
```
5
```

```
-----
```

```
s + 5
```

```
>> impulse( Hc ) ;  
>> step( Hc ) ;  
>> vet_tempo = [ 0 : 0.001 : 3.5 ] ;  
>> lsim( Hc , exp( -1 * vet_tempo ) , vet_tempo ) ;
```





Os gráficos gerados são esses acima. Confira que as curvas condizem com as expressões teóricas (fica como exercício a demonstração).

exemplo 2: análise de uma função de transferência genérica

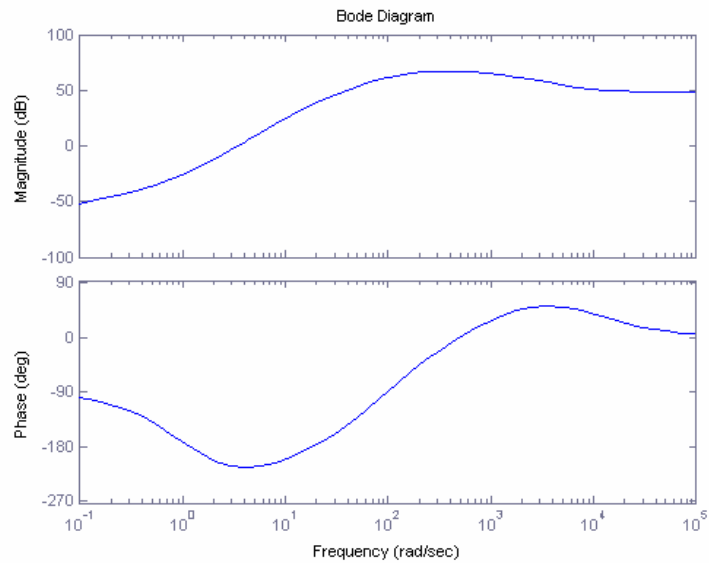
$$H(s) = \frac{250s(s+1)^2(s+10000)}{(s+10)(s+100)^2(s+1000)}$$

Para obtermos o diagrama de Bode, temos que montar a função de transferência primeiro, como no exemplo anterior. Entretanto, os polinômios do numerador e denominador não estão desenvolvidos (temos a forma fatorada, ao invés disso). A fim de evitar dois trabalhos de distributivas, basta usar a função “poly” apresentada em 4.3, que retorna os coeficientes do polinômio dadas as raízes (cuidado: as raízes duplas devem ser passadas duas vezes!).

```
>> vet_raizesNumerador = [ 0 1 1 10000 ] ;
>> vet_raizesDenominador = [ 10 100 100 1000 ] ;
>> vet_coefPolinomioNumerador = 250 * poly( vet_raizesNumerador ) ;
>> vet_coefPolinomioDenominador = poly( vet_raizesDenominador ) ;
>> H = tf( vet_coefPolinomioNumerador , vet_coefPolinomioDenominador )
```

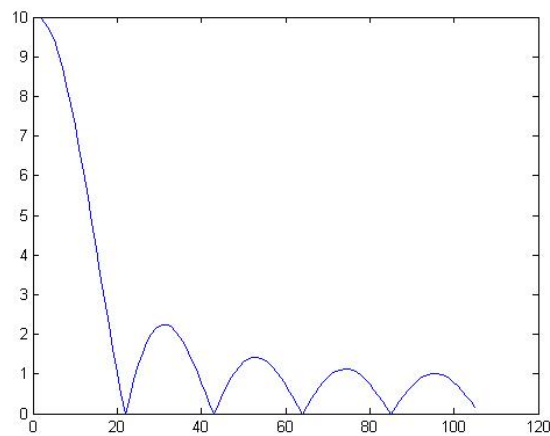
```
Transfer function:
250 s^4 - 2.501e006 s^3 + 5e006 s^2 - 2.5e006 s
-----
s^4 - 1210 s^3 + 222000 s^2 - 1.21e007 s + 1e008
```

```
>> bode( H ) ;
```



exemplo 3: módulo da transformada de fourier de $\text{rect}(t)$:

```
>> vet_y = [zeros( 1 , 100 ) ones( 1 , 10 ) zeros( 1 , 100 )] ;
>> vet_Y = fft( vet_y ) ;
>> vet_moduloY = abs( vet_Y ) ;
>> plot( vet_moduloY( 1 : 105 ) ) ;
```



Completando a análise de Sinais e Sistemas, apresentamos uma das várias opções de filtros existentes no MatLab.

Operação	Função
Construir Filtro de Ordem N e Frequência de Corte Wc (ou passa-banda entre W1 e W2)	[vet_coefNum vet_coefDenom = butter(N , Wc , 'low') [vet_coefNum vet_coefDenom = butter(N , Wc , 'high') [vet_coefNum vet_coefDenom = butter(N , [W1 W2])
Aplicar Filtro num Sinal	vet_y = filter(vet_coefNum , vet_coefDenom , vet_x)

4.7 – Outras Funções

Algumas opções para fatorações e números primos:

Operação	Função
Fatorar um Número	<code>vet_fatores = factor(numero)</code>
Mínimo Múltiplo Comum – MMC (Least Common Multiple – LCM)	<code>mmc = lcm(num1, num2)</code> <code>vet_mmc = lcm(vet1 , vet2)</code> <code>matriz_mmc = lcm(matriz1 , matriz2)</code>
Máximo Divisor Comum – MDC (Greatest Common Divider – GCD)	<code>mdc = gcd(num1, num2)</code> <code>vet_mdc = gcd(vet1 , vet2)</code> <code>matriz_mdc = gcd(matriz1 , matriz2)</code>
Checar se um número ou os elementos de um vetor ou matriz são primos (retorno: 0 = não é primo, 1 = é primo)	<code>condicao = isprime(numero)</code> <code>vet_condicoes = isprime(vet)</code> <code>mat_condicoes = isprime(matriz)</code>
Obter Números Primos Menores ou Iguais a N	<code>vet_primos = primes(N)</code>

OBS: as funções de MMC e MDC fazem a operação sobre 2 números de cada vez (no caso de vetores, sobre o 1º elemento de um com o 1º elemento do outro, e assim sucessivamente, retornando um vetor de MMC / MDC). Para fazer com 3 ou mais números juntos, chame a função mais de uma vez (exemplo: MMC entre 4, 5 e 12 → `mmc = lcm(lcm(4 , 5) , 12)`).

Agora, algumas opções para gerar números aleatórios:

Operação	Função
Gerar Matriz M x N com Elementos em Distribuição Uniforme entre 0 e 1	<code>matriz = rand(M , N)</code>
Gerar Matriz M x N com Elementos em Distribuição Normal (Gaussiana) com média 0 e $\sigma^2 = 1$	<code>matriz = randn(M , N)</code>
Gerar Números numa Distribuição Qualquer	<code>numero = random(string_Distrib , ...)</code> < consulte help random para maiores detalhes >

Embora não se possam declarar números em outras **bases numéricas**, o MatLab possui funções de conversão entre elas, através do uso de strings.

Conversão de Bases	Funções
Binário → Decimal Decimal → Binário	<code>numDecimal = bin2dec(string_numBinario)</code> <code>string_numBinario = dec2bin(numDecimal)</code>
Hexadecimal → Decimal Decimal → Hexadecimal	<code>numDecimal = hex2dec(string_numHexa)</code> <code>string_numHexa = dec2hex(numDecimal)</code>
Hexadecimal → Double Double → Hexadecimal (modo como números reais são armazenados na memória dum computador)	<code>numDouble = hex2num(string_numHexa)</code> <code>string_numHexa = num2hex(numDouble)</code> < <i>string_numHexa deve ter 16 caracteres (tendo menos, zeros são acrescentados à direita)</i> >

Capítulo 5 – Plotando Gráficos

5.1 – A Família de Funções para Plotar

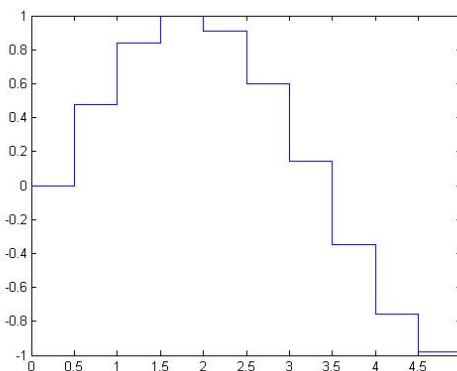
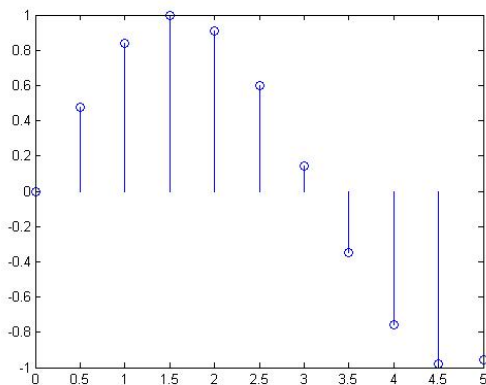
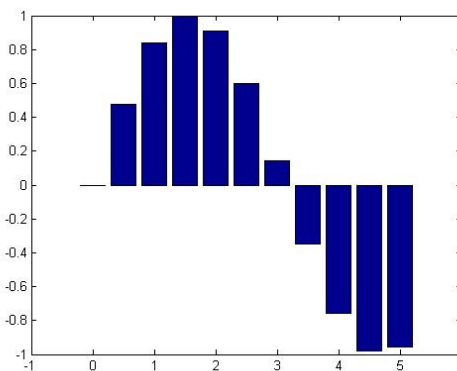
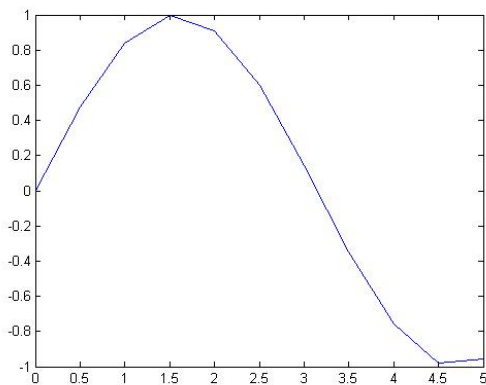
O MatLab oferece inúmeras opções para criarmos e editarmos gráficos. Veja na tabela abaixo, primeiramente, algumas funções para plotarmos.

Modo de Plotar	Função
Ligando Pontos	<code>plot(vet_x , vet_y)</code>
Barras	<code>bar(vet_x , vet_y)</code>
Valores Discretos Isolados	<code>stem(vet_x , vet_y)</code>
Degraus	<code>stairs(vet_x , vet_y)</code>
Log-Log (base 10)	<code>loglog(vet_x , vet_y)</code>
Semi-Log (base 10) (só x ou y com eixo logarítmico)	<code>semilogx(vet_x , vet_y)</code> <code>semilogy(vet_x , vet_y)</code>
Coordenadas Polares	<code>polar(vet_theta , vet_raio)</code>

Tomemos como exemplo os vetores `vet_x` e `vet_y` a seguir:

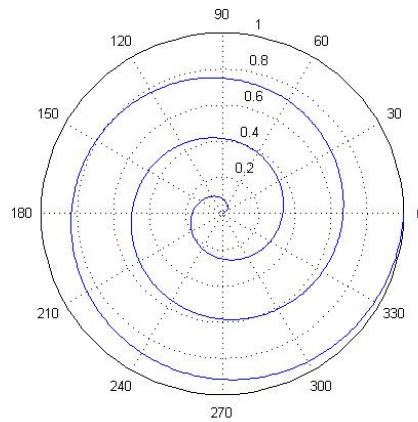
```
>> vet_x = 0 : 0.5 : 5 ;  
    vet_y = sin( vet_x ) ;
```

Se plotássemos esses pontos com as 4 primeiras funções, teríamos o seguinte:



Podemos desenhar uma espiral com as coordenadas polares

```
>> vet_raio = 0 : 0.001 : 1 ;
    vet_theta = 0 : 6*pi/1000 : 6*pi ;
>> polar( vet_theta , vet_raio ) ;
```

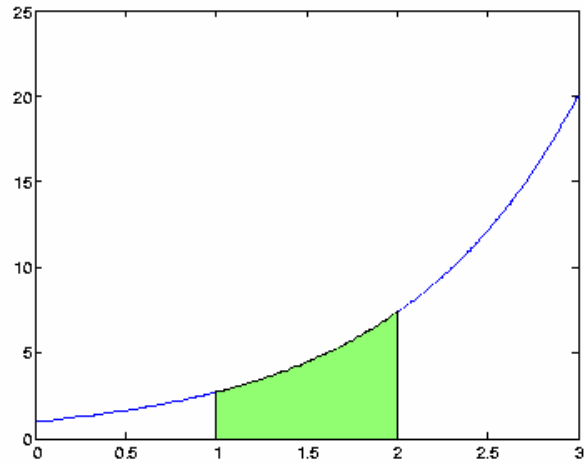


Não acabou não. Tem mais funções para plotar.

Modo de Plotar	Função
Ligando Pontos e Preenchendo Área Abaixo do Gráfico	area(vet_x , vet_y)
Segurar / Soltar Gráfico para Plotar Mais de uma Curva Juntas	hold hold on hold off
Criar Nova Janela de Gráfico	figure
Dividir Janela de Plotagem em M x N Partes e Selecionar Parte P	subplot(M , N , P)
Gráfico Estatístico de Fatias de Pizza (Pie Chart)	pie(vet_valores) pie(vet_valores , vet_fatiasSobressaidas)
Gráfico de Cores (para Matrizes)	imagesc(matriz)
Legenda para o Gráfico de Cores	colorbar

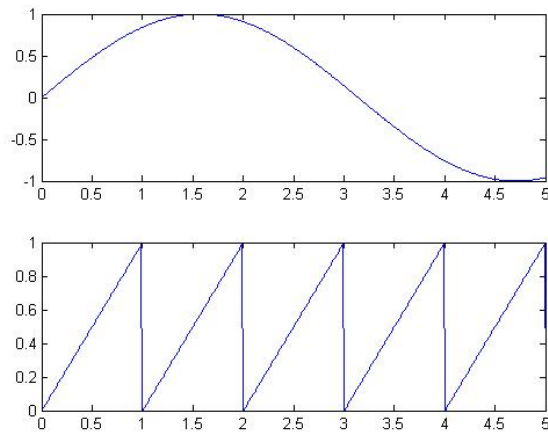
Digamos que se quer **selecionar a região** de 1 a 2 **abaixo do gráfico** $\exp(x)$, para dar um exemplo didático sobre integral. Poderíamos fazer do seguinte modo.

```
>> vet_x1 = 0 : 0.01 : 3 ;
    vet_x2 = 1 : 0.01 : 2 ;
    vet_y1 = exp( vet_x1 ) ;
    vet_y2 = exp( vet_x2 ) ;
>> plot( vet_x1 , vet_y1 ) ;
>> hold ;
>> area( vet_x2 , vet_y2 ) ;
```



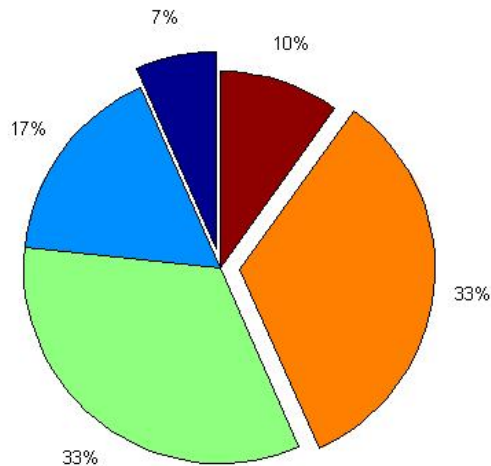
Um exemplo para a função **subplot**, que pode usada em conjunto com qualquer uma das funções de plotagem:

```
>> vet_x = 0 : 0.01 : 5 ;
    vet_y1 = sin( vet_x ) ;
    vet_y2 = mod( vet_x , 1 ) ;
>> subplot( 2 , 1 , 1 ) ;
>> plot( vet_x , vet_y1 ) ;
>> subplot( 2 , 1 , 2 ) ;
>> plot( vet_x , vet_y2 ) ;
```



Um exemplo de um **diagrama pizza** seria o abaixo. Ele calcula os percentuais com base na soma total (no caso, $2 + 5 + 10 + 10 + 3 = 30$) e imprime os valores ao lado das fatias. O segundo argumento passado é opcional, e serve apenas para destacar certas fatias.

```
>> vet_valores = [ 2 5 10 10 3 ] ;
>> vet_fatiasSobressaidas = [ 1 0 0 1 0 ] ;
>> pie( vet_valores , vet_fatiasSobressaidas ) ;
```

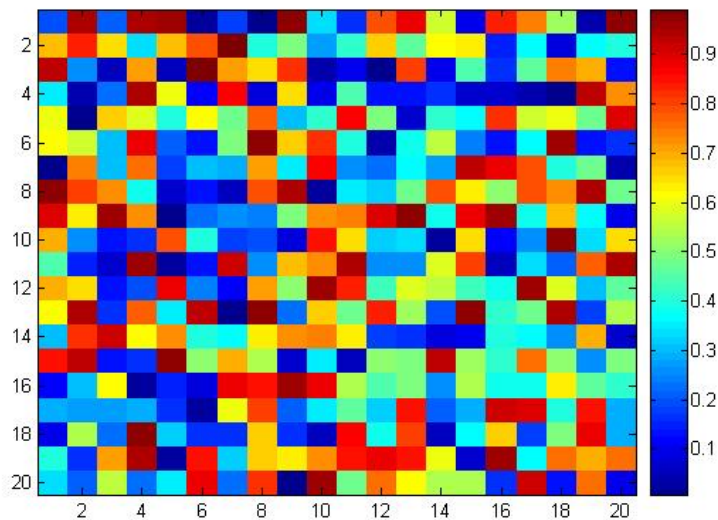


Para acrescentar legenda referente às fatias, veja o comando “legend” em 5.2.

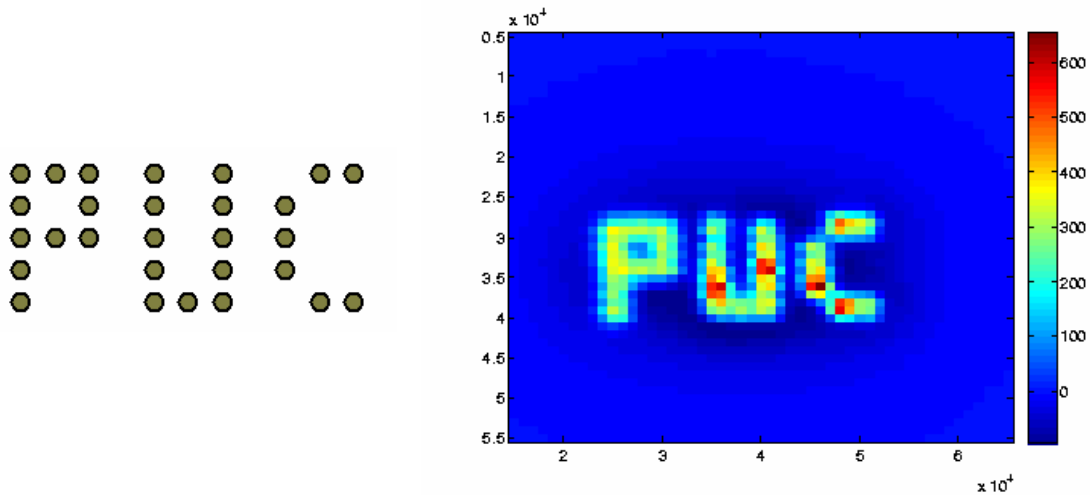
A função “imagesc” é bem útil quando se deseja **mapear alguma superfície**. Por exemplo, tendo os valores de temperatura de uma certa área contidos numa matriz, podemos plotar sua “imagem térmica” – valores mais altos apresentarão cores mais avermelhadas; os mais baixos, cores azuladas.

O exemplo a seguir tenta obter a “imagem” de uma matriz 20 x 20 com valores aleatórios entre 0 e 1.

```
>> matriz_aleatoria = rand( 20 , 20 ) ;
>> imagesc( matriz_aleatoria ) ;
>> colorbar ;
```



Um exemplo mais prático da função “imagesc” foi feito com um medidor de campo magnético em superfícies. O ensaio teste, com um arranjo de ímãs circulares formando a palavra “PUC”, teve o seguinte resultado:



5.2 – Alguns Detalhes: Cores, Título, Legendas, Etc

Linhas pontilhadas? Linhas mais grossas? Cor vermelha? Plotar apenas os pontos isolados, sem traço algum? Tudo isso pode ser passado como parâmetros extras para personalizar o gráfico.

exemplos:

```

plot( vet_x , vet_y , 'r' ) → linha vermelha
plot( vet_x , vet_y , '*' ) → pontos (asteriscos) sem ligação
plot( vet_x , vet_y , 'r*' ) → asteriscos vermelhos sem ligação
plot( vet_x , vet_y , 'k:' ) → linha preta pontilhada
plot( vet_x , vet_y , 'LineWidth' , 2 ) → linha de grossura 2

```

Consulte “**help plot**” para descobrir mais opções.

Logo que os pontos estiverem na tela, é importante identificar os eixos e o gráfico em si. Pode-se ajustar opções na própria janela também. Eis algumas ferramentas básicas:

Detalhe	Função
Título	title(string)
Identificando Eixos	xlabel(string) ylabel(string) zlabel(string)
Grade Pontilhada	grid
Ajuste Automática dos Eixos (fixando ou adaptando segundo as curvas)	axis manual axis tight
Legenda (útil para quando há mais de uma curva)	legend(string_curva1 , string_curva2 , ...)
Limite dos Eixos	xlim([valorInicial valorFinal]) ylim([valorInicial valorFinal]) zlim([valorInicial valorFinal])

Lembrando que as **strings** são palavras entre **aspas simples**, e não duplas como em outras linguagens.

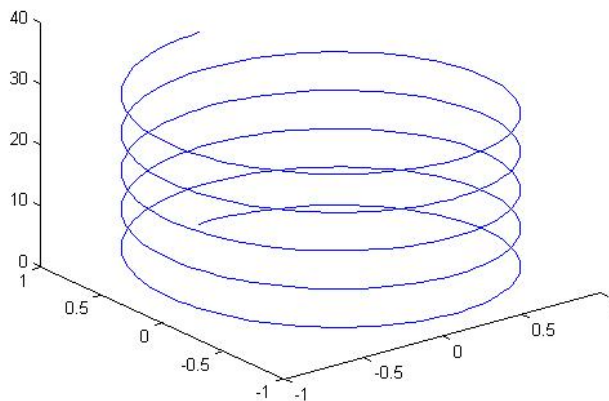
5.3 – Plotando em 3D

Há dois modos de se plotar em três dimensões: com linhas ou superfícies.

Modo de Plotar	Função
Linhas (Pontos Ligados) em 3D	<code>plot3(vet_x , vet_y , vet_z)</code>
Superfície em 3D	<code>surf(matriz_x , matriz_y , matriz_z)</code>
Combinando Vetores X e Y para Gerar Matrizes para Plotar Superfícies	<code>[matriz_x , matriz_y] = meshgrid(vet_x , vet_y)</code>

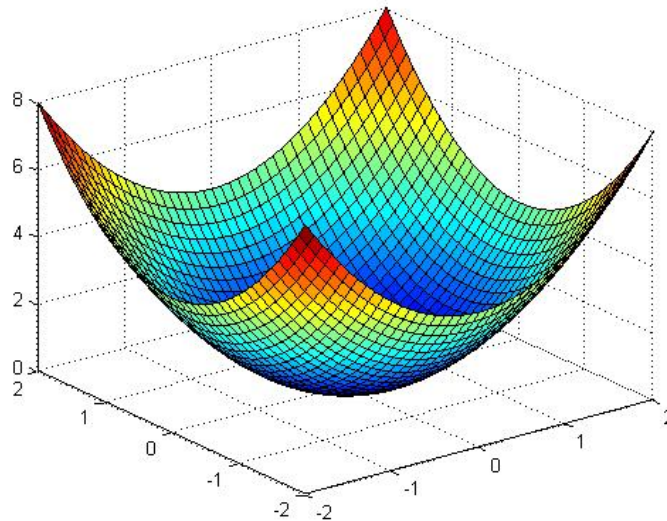
Plotando uma espiral helicoidal:

```
>> vet_t = 0 : pi/50 : 10*pi ;  
>> plot3( sin( vet_t ) , cos( vet_t ) , vet_t ) ;
```



E uma parabolóide ($z = x^2 + y^2$). A função “meshgrid” cria um mapa quadrado xy de pontos entre -2 e 2 a partir de vetores, para substituímos então na expressão de z e obtermos assim uma superfície.

```
>> vet_x = -2 : 0.1 : 2 ;  
>> vet_y = -2 : 0.1 : 2 ;  
>> [ matriz_x , matriz_y ] = meshgrid( vet_x , vet_y ) ;  
>> matriz_z = matriz_x .^2 + matriz_y .^2 ;  
>> surf( matriz_x , matriz_y , matriz_z ) ;
```

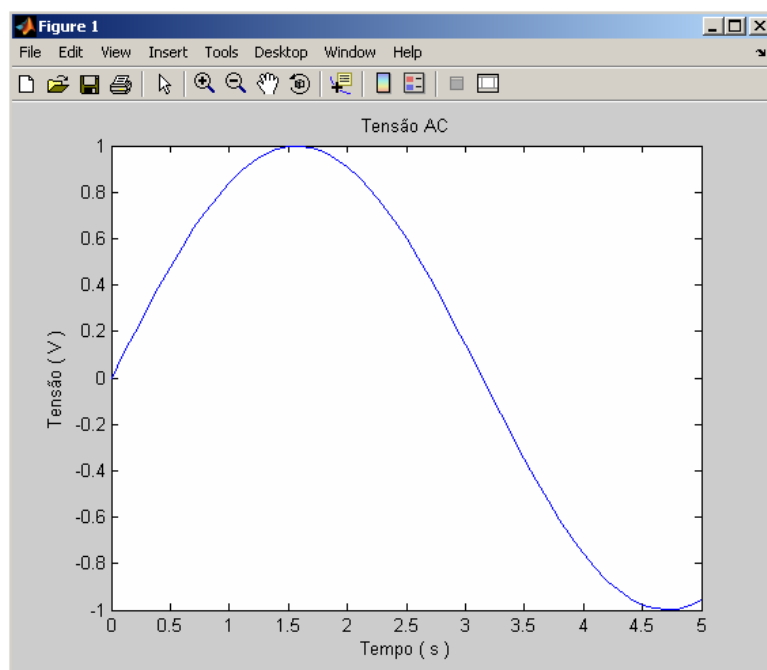


5.4 – Manipulando a Janela do Gráfico

A seguir, algumas funções que podem configurar opções da janela em que plotamos.

Operação	Comando
Ajustar Fonte dos Eixos, Título e Legenda	<code>set(gca , 'fontsize' , <i>numeroFonte</i>)</code>
Ajustar Tamanho e Posição da Janela (em pixels, com origem no canto superior esquerdo da tela)	<code>set((gcf , 'Position', <i>posicaoX</i> , <i>posicaoY</i> , <i>tamanhoX</i> , <i>tamanhoY</i>)</code>

Perceba, no entanto, que várias das opções apresentadas até agora estão disponíveis não só através da chamada de funções. Pela própria janela do gráfico, encontramos menus e ícones de configurações.



- pelo menu Insert, podemos colocar manualmente legendas, títulos, etc, além de setas e outros elementos
- ativando o ícone com o cursor do mouse e clicando duas vezes na curva, um menu de opções é aberto; pode-se configurar ali cor, espessura e tipo de linha (ou então deletar a curva pressionando “Delete”)
- os ícones de lupa permitem ajustar o zoom (você pode desenhar um retângulo com o zoom +, a fim de aumentar uma região específica); o de mãozinha ajusta a posição
- o ícone “data cursor” possibilita a seleção de um ponto para a visualização de seu valor (após clicar em um ponto, pode-se andar pelas adjacências com as setas esquerda e direita do teclado)

5.5 – Acessando Dados dos Gráficos

Se por algum motivo neste mundo você quiser capturar os pontos de uma tela para processá-los, existem pelo menos duas maneiras de fazer.

Operação	Função
Capturar Pontos com o Mouse	<code>[vet_x , vet_y] = getpts</code> <i><após o comando, clique na tela para capturar pontos e pressione ENTER quando terminar; aperte BACKSPACE para “descapturar” último ponto></i>
Capturar Todos os Pontos da Curva Já Plotada	<code>vet_x = get(get(gca , 'children') , 'XData')</code> <code>vet_y = get(get(gca , 'children') , 'YData')</code>

Se houver mais de uma curva na tela, o comando `get(get (...) ...)` retornará uma matriz coluna de “cells”. Para convertê-la a uma matriz normal, use o comando “cell2mat”.

Capítulo 6 – O Editor de Funções e Rotinas

6.1 – Vantagens do Editor

Utilizaremos o editor de texto sempre que precisarmos executar uma seqüência grande ou mais complexa de comandos. As vantagens são:

- poder salvar seu trabalho
- não precisar repetir os comandos um a um quando algo der errado no meio do processo
- usar com facilidade comandos de loop e condicionais
- criar funções que poderão ser usadas por outras funções ou rotinas
- poder usar breakpoints para DEBUGs
- aspecto visual melhor

Para **abrir o editor**, basta digitar “edit” na tela de comandos ou clicar em File>New. Os arquivos são salvos com a **extensão “.m”**.

6.2 – Criando Funções e Rotinas

Convenções básicas para o editor:

```
% comentários são iniciados por "%" e ficam verdes até o fim da linha

%{
  bloco
  de
  comentários
}%

'strings ficam entre ASPAS SIMPLES e são roxas'

'uma string não fechada por descuido fica vermelha

% para continuar o comando na linha de baixo, use "..."
funcao( parametro1 , parametro2 , ...
        parametro3 ) ;
```

Uma **rotina** é simplesmente uma seqüência de comandos. Basta digitar um em cada linha, conforme se fazia até agora – veja que declarar variáveis sem o ponto-e-vírgula fará com que seus valores apareçam na tela de comandos. Para executar tudo, clique no ícone "Run" (ou "Save and Run") ou aperte F5. É necessário salvar seu trabalho antes de rodá-lo.

Já a **função** recebe (ou não) parâmetros, e retorna (ou não) um ou mais valores. Algumas regras devem ser respeitadas:

- sua declaração deve constar no início do arquivo; os comentários que estiverem antes da declaração são exibidos na tela de comandos quando se digita “help função”
- tudo que estiver abaixo dela, até a declaração de uma possível outra função, será executado quando ela for chamada (não há chaves ou parênteses para limitar seu conteúdo)
- apenas a primeira função do arquivo pode ser chamada externamente; as funções que estiverem abaixo da primeira servirão apenas como funções auxiliares, já que só conseguirão ser acessadas pela primeira
- o MatLab associa a função ao arquivo “.m”; ou seja, deve-se obrigatoriamente salvar o arquivo com o mesmo nome da função antes de utilizá-la

A declaração de uma função que recebe dois parâmetros e retorna um valor seria como está mais abaixo. No caso, a variável “resultado” é o retorno, e sua última atribuição é a que será de fato retornada a quem chamou a função. Podem existir outros comandos após essa última atribuição.

```
function resultado = funcao ( parametro1 , parametro2 )
(...)
    resultado = ( ... )
(...)
```

Já a declaração de uma função que recebe apenas um parâmetro e não retorna nada seria assim:

```
function funcao ( parametro1 )
(...)
```

Se quiser **retornar**, digamos, **2 argumentos** e não receber parâmetro nenhum:

```
function [ resultado1 , resultado2 ] = funcao ( )
(...)
    resultado1 = ( ... )
(...)
    resultado2 = ( ... )
(...)
```

E uma nova opção: pode-se declarar o **recebimento de uma quantidade indefinida de parâmetros**. Isso é útil, por exemplo, quando sua função recebe uma função do usuário como parâmetro, e passa a ela os parâmetros definidos por ele. Veja o exemplo.

```
function funcao ( função_usuario , varargin )
(...)
```

```
função_usuario( varargin{ : } ) ;
(...)
```

O parâmetro "varargin", que contém os inúmeros argumentos da função do usuário, deve vir por último na declaração da função.

6.3 – Condicionais e Loops: “if”, “switch”, “while” e “for”

Este tópico já deve ser bem conhecido para os alunos de Engenharia que já fizeram Estrutura de Dados. Mesmo assim, vale a pena apresentar a sintaxe e explicar os conceitos.

A **estrutura “if” executa blocos de comandos** somente se a **condição for verdadeira**. Veja a tabela de possibilidades abaixo

<pre>if condicao bloco end</pre>	<pre>if condicao1 bloco1 elseif condicao2 bloco2 elseif condicao3 bloco3 (...) elseif condicaoN blocoN end</pre>	<pre>if condicao1 bloco1 elseif condicao2 bloco1 else bloco3 end</pre>
<p>Executa bloco de comandos somente se a condição for verdadeira</p>	<p>Executa bloco 1 somente se a condição 1 for verdadeira. Caso contrário, executa bloco 2 se a condição 2 for verdadeira. E assim em diante.</p>	<p>Executa bloco 1 somente se a condição 1 for verdadeira. Caso contrário, executa bloco 2 se a condição 2 for verdadeira. Caso contrário, executa bloco 3.</p>

Uma alternativa a estrutura “if” é a **estrutura “switch”**. Ela é mais interessante para executar comandos com base no valor de uma variável ou expressão.

```
switch expressao
case valor1
    bloco1
case { valor2 , valor3 }
    bloco2
otherwise
    bloco3
end
```

No caso acima, o bloco 1 é executado se o valor da expressão for “valor1”. Caso contrário, se for “valor2” OU “valor3”, o bloco 2 é executado. Caso contrário, o bloco 3 é executado. Perceba que, diferente do que ocorre na linguagem C, os blocos 2 e 3 não são executados caso o valor 1 seja verdadeiro! Não há, portanto, necessidade de breaks!

Para **executar o mesmo bloco de comandos mais de uma vez**, usamos as estruturas de loop (laço). No caso da **estrutura “while”**, o bloco é executado enquanto a condição for verdadeira.

CUIDADO: se a condição for sempre verdadeira, o bloco rodará eternamente, até que alguém aborte o programa.

```
while condicao
    bloco
end
```

Finalmente, a **estrutura “for”** é usada para executar um bloco de comandos numa quantidade definida de vezes. Através de um contador (ou “iterador” → por isso a letra “i” é usada) declarado no início como um vetor, estabelecemos quantas vezes o bloco será executado. Ao mesmo tempo, declaramos os valores do contador ao longo dos ciclos. Observe alguns casos possíveis:

<code>for i = 1 : N</code> <code> bloco</code> <code>end</code>	<code>for i = 0 : 2 : N</code> <code> bloco</code> <code>end</code>	<code>for i = [2 5 6]</code> <code> bloco</code> <code>end</code>
Executa bloco de comandos N vezes; a cada ciclo, i assume valor 1, 2, 3, ..., N.	Executa bloco de comandos N/2 vezes; a cada ciclo, i assume valor 0, 2, 4, ..., N.	Executa bloco de comandos 3 vezes; a cada ciclo, i assume valor 2, 5 e 6.

OBS: vale lembrar mais uma vez que o primeiro índice de vetores e matrizes é 1. A tabela abaixo reúne algumas das condições mais comuns em “if”s e “while”s.

Condição (Pergunta)	Notação
X é igual a Y?	<code>X == Y</code>
X é diferente de Y?	<code>X ~= Y</code>
X é maior do que Y?	<code>X > Y</code>
X é menor ou igual a Y?	<code>X <= Y</code>
X é divisível por Y? (ou seja, resto da divisão é nulo?)	<code>mod(X , Y) == 0</code>
X é inteiro?	<code>mod(X , 1) == 0</code>
X é par?	<code>mod(X , 2) == 0</code>
X é número primo?	<code>isprime(X)</code>
X é real (não-complexo)?	<code>isreal(X)</code>
X é um escalar (não-vetor)?	<code>length(X) == 1</code>

É possível juntar duas condições numa só:

Condição (Pergunta)	Notação(ões)
Condições A e B são ambas verdadeiras?	<code>A & B</code> <code>and(A , B)</code>
Condição A e/ou condição B é verdadeira?	<code>A B</code> <code>or(A , B)</code>
Condição A ou condição B é verdadeira? (Exclusive OR)	<code>xor(A , B)</code>
Condição A não é verdadeira?	<code>~A</code> <code>not(A)</code>

Combinando essas condições, é prudente usar parêntesis para estabelecer a ordem de prioridade das verificações.

exemplo: testar se X é 2 ou múltiplo de 8 e 10, escrevemos:

```
if ( X == 2 ) | ( mod( X , 8 ) == 0 & mod( X , 10 ) == 0 )
```

Veja agora um exemplo de função que usa alguns dos conceitos até aqui apresentados.

```
function retorno = primoAnterior ( numero )

if length( numero ) ~= 1 | ~isreal( numero ) | numero < 3
    % erro: foi passado um vetor, um complexo ou um numero menor
do que 3
    retorno = NaN ;

else
    % pega maior inteiro anterior ao número recebido
    numeroAtual = ceil( numero - 1 ) ;

    while ~isprime( numeroAtual )
        numeroAtual = numeroAtual - 1 ;
    end

    retorno = numeroAtual ;
end
```

Para encerrar este tópico, é muito importante frisar que o uso de “for” e “while” pode e deve ser evitado na maioria dos casos. Isso porque o MatLab possui funções otimizadas para receber vetores e matrizes, efetuando assim o cálculo em todos os elementos de uma só vez! Usar loops e chamar a função para cada elementos individualmente deixa o código mais extenso e complicado, e pode até retardar o processamento.

A tabela abaixo mostra alternativas para quem está acostumado demais com algoritmos da linguagem C:

Objetivo	Estilo C	Estilo MatLab
Calcular seno dos elementos de um vetor	<pre>for i = 1 : length(vet) vet2(i) = sin(vet(i)) ; end</pre>	<pre>vet2 = sin(vet) ;</pre>
Obter soma de todos os elementos de um vetor	<pre>soma = 0 for i = 1 : length(vet) soma = soma + vet(i) ; end</pre>	<pre>soma = sum(vet) ;</pre>
Zerar elementos de 3 em 3 de um vetor	<pre>for i = 1 : length(vet) if mod(i , 3) == 0 vet(i) = 0 ; end end</pre>	<pre>total = length(vet) vet(3 : 3 : total) = 0</pre>

6.4 – Recursão e Variáveis Globais

Há quem prefira evitar o uso de loops e programar funções com o uso de recursão. OK, sem problemas, o MatLab aceita essa opção. Vejamos como ficaria o exemplo anterior com a função chamando ela mesma.

```
function retorno = primoAnterior ( numero )

if length( numero ) ~= 1 | ~isreal( numero ) | numero < 3
```

```

    % erro: foi passado um vetor, um complexo ou um numero menor
do que 3
    retorno = NaN ;

else
    % pega maior inteiro anterior ao número recebido
    numeroAtual = ceil( numero - 1 ) ;

    if isprime( numeroAtual )
        retorno = numeroAtual ;
    else
        retorno = primoAnterior( numeroAtual ) ;
    end
end
end

```

Rotinas, conforme foi dito, são apenas seqüências de comandos que teriam o mesmo efeito se digitássemos e executássemos seu conteúdo na tela de comandos. Portanto, quando se declara uma variável numa rotina, ela permanece no workspace do MatLab até que o comando "clear" seja executado (ou até que se saia do próprio MatLab), já se tornando global e podendo ser acessada e modificada por qualquer outra rotina.

Já as variáveis declaradas numa função existem somente enquanto a função é executada. Para que ela trabalhe com variáveis globais, é necessário usar a declaração "global *variavel*", tanto para declará-las quanto para acessá-las ou modificá-las.

exemplo: supondo os seguinte arquivos

rotina1.m:

```

clear ;

rotina2 ;
funcao1 ( ) ;
global b

a
b
c

```

rotina2.m:

```

a = 1 ;

```

funcao1.m:

```

function funcao1 ( )
    global b
    b = 2 ;
    c = 3 ;

```

rodando rotina1.m, o seguinte trecho aparece na tela de comandos:

```

a =

    1

```

```
b =  
    2  
  
??? Undefined function or variable "c".  
  
Error in ==> rotinal at 9  
c
```

6.5 – Debugando

Durante a execução, quando há um erro de "gramática da linguagem" (parênteses não fechado, falta de vírgulas, etc) ou má digitação de nomes, uma mensagem de erro na tela de comandos geralmente esclarece o motivo da falha, que pode ser então imediatamente consertada – repare que sempre aparece o número da linha onde ocorreu o erro (vide exemplo do item anterior, "Error in ==> rotinal at 9").

No entanto, é muito comum na vida de um programador se deparar com um erro de causas desconhecidas no código. Uma ferramenta boa para ajudar na solução desses problemas é o debugador (debugger). Basicamente, inserem-se pontos de pausa de execução ("set/clear breakpoint") para avaliar as condições das variáveis. Executando linha a linha ("step"), o mistério é mais facilmente encontrado.



Dica: podemos descobrir o valor assumido por uma variável apontando-lhe com o mouse no Editor de Texto, durante o Modo Debug.

Capítulo 7 – Funções de Entrada e Saída

7.1 – Strings

Antes de entrarmos efetivamente no tema deste capítulo, é interessante falar algo sobre strings, que nada mais são do que uma forma de trabalharmos com textos no código. Usamos strings até agora apenas para escrever legenda e título de gráficos, ou em algum parâmetro numa função ou outra. Para esses e outros propósitos, já podemos apresentar alguns detalhes extras.

E mais uma vez: as **strings no MatLab são declaradas com aspas simples**, e não duplas como em outras linguagens.

Operação com Strings	Função
Concatenação (União)	<code>string = [string1 string2 ...]</code>
Criar Matriz Coluna de Strings (Concatenação Vertical)	<code>matString = strvcat(string1 , string2 , ...)</code> <code>matString = strvcat(matString1 , matString2 , ...)</code>
Comparação (Se São Iguais ou Não)	<code>condicao = strcmp(string1, string2)</code>
Conversão Número-String e String-Número	<code>string = num2str(numero)</code> <code>numero = str2num(string)</code>
Procurar um Trecho numa String (Índices Iniciais de Ocorrências)	<code>vet_indices = strfind(string , string_trecho)</code>
Substituir um Trecho numa String por Outro	<code>stringNova = strep(string , string_trecho , string_substituta)</code>
Executar Comando Contido numa String	<code>eval(string_comando)</code>

Para criar um **conjunto de strings**, usamos uma matriz coluna. Infelizmente, o MatLab exige que nela todas as strings tenham o mesmo tamanho. A função `strvcat`, presente na tabela acima, adiciona-lhes espaços em branco no fim até que todas tenham o mesmo comprimento. Isso se torna interessante para a função “`legend`”, por exemplo, que aceita tanto N argumentos para nomear as curvas quanto uma matriz com esses nomes todos.

Aliás, é possível incluir caracteres especiais nas strings de títulos e legendas de gráficos. Veja alguns na tabela abaixo:

Caractere Especial	Código
α	<code>\alpha</code>
β	<code>\beta</code>
θ	<code>\theta</code>
Ω	<code>\Omega</code>
ω	<code>\omega</code>
σ	<code>\sigma</code>
π	<code>\pi</code>
λ	<code>\lambda</code>
ρ	<code>\rho</code>
μ	<code>\mu</code>
γ	<code>\gamma</code>

Próximo Caractere Sobrescrito	^
Próximo Caractere Subscrito	_

Aqui temos, por fim, alguns caracteres usados na manipulação de arquivos texto.

Caractere Especial	Código
Quebra de Linha	\n
Tabulação	\t
Caractere \	\\
Caractere %	%%

7.2 – Capturando e Imprimindo na Tela de Comandos

Embora não sejam muito utilizadas, as funções equivalentes ao “printf” e “scanf” existem no MatLab. Confira a tabela abaixo:

Operação	Função
Exibir Mensagem na Tela de Comandos	disp(string_mensagem)
Exibir Mensagem de Erro na Tela de Comandos e Abortar Rotina / Função	error(string_mensagem)
Exibir Mensagem e Aguardar Usuário Entrar com Numero ou Texto na Tela de Comandos	numero = input(string_mensagem) string = input(string_mensagem , 's')

7.3 – Arquivos de Texto, Imagem e Som

Uma ótima característica do MatLab é poder juntar suas ferramentas matemáticas com arquivos de multimídia, o que possibilita inclusive a integração com outros aplicativos.

Começemos com algumas funções de manipulação de arquivos, similares às da linguagem C.

Operação	Função
Abrir “.m” de uma Função no Editor	open nomeFuncao
Carregar Números dum Arquivo ASCII	matriz = load(string_arquivoTexto)
Abrir / Criar Arquivo Texto	fid_arquivo = fopen(string_arquivoTexto , string_objetivo) <i>string_objetivo:</i> <i>‘r’ = ler o arquivo (read)</i> <i>‘w’ = criar e escrever no arquivo (write)</i> <i>‘a’ = continuar escrevendo no arquivo (append)</i>
Imprimir Texto no Arquivo	fprintf(fid_arquivo, string_texto) fprintf(fid_arquivo, string_texto, ...)
Verificar se Arquivo Chegou ao Fim (File: End of File)	condição = feof(fid_arquivo)
	fscanf(string_formatos , variavel1 , variavel2 , ...)

Capturar Números / Caracteres do Arquivo para Variáveis	<p><i>string_formatos:</i> '<i>%d</i>' = captura inteiro '<i>%f</i>' = captura numero de casas decimais (float) '<i>%c</i>' = captura caractere (char) '<i>%s</i>' = captura palavra (string)</p>
Capturar Próxima Linha (Get Line) do Arquivo	string_linha = fgetl(fid_arquivo)
Fechar Arquivo	fclose(fid_arquivo)

Confira o exemplo 5 do Capítulo 9 para ver uma aplicação dessas funções.

Uma opção interessante também é usar as ferramentas apresentadas em 4.6 – Sinais e Sistemas, tais como filtros e transformada de Fourier, com arquivos de som.

Operação	Função
Carregar Arquivo “.wav”	vet_som = wavread(string_nomeArquivo) [vet_som , taxaAmostragem , bitsAmostra] = wavread(string_nomeArquivo)
Reproduzir Arquivo “.wav”	wavplay(vet_som , taxaAmostragem)
Gravar Arquivo “.wav”	wavwrite(vet_som , taxaAmostragem , bitsAmostra , string_nomeArquivo)
Abrir Arquivo de Som numa Janela de Gráfico (com opção de reprodução)	pw(string_nomeArquivo)

As mesmas ferramentas de Sinais e Sistemas também podem ser usadas com arquivos de imagem. Elas são geralmente carregadas em matrizes $M \times N \times 3$, onde cada matriz $M \times N$ representa intensidades RGB (Red-Green-Blue – vermelho, verde e azul).

Operação	Função
Carregar Arquivo de Imagem	matriz3d_imagem = imread(string_nomeArquivo, string_extensao) <i>string_extensao:</i> ' <i>jpeg</i> ' ' <i>bmp</i> ' ' <i>gif</i> ' (...)
Imprimir Imagem na Tela de Gráfico	image(matriz3d_imagem)
Gravar Arquivo de Imagem	imwrite(matriz3d , string_nomeArquivo , string_extensao)

7.4 – Execução do MatLab e de Outros Aplicativos

A tabela abaixo reúne dois comandos interessantes para o controle da execução de funções e rotinas: a opção de abortá-la manualmente (caso esteja demorando muito para processar) ou de interrompê-la por alguns instantes.

Além disso, é possível invocar um programa de dentro do próprio MatLab. Exemplo: com as funções apresentadas no item anterior, pode-se escrever um arquivo texto de instruções e chamar um aplicativo que execute certas operações e retorne seus resultados num arquivo de saída, o qual será processado em seguida pelo MatLab.

Operação	Função / Comando
Abortar Execução de Rotina / Função / Comando	<i>aperte Ctrl + C na tela de comandos</i>
Pausar Execução (por tempo determinado ou aguardando usuário apertar ENTER)	pause(tempoEmSegundos) pause
Invocando Aplicativo (similar ao prompt do DOS)	<i>!caminhodoPrograma</i> <i>!comandoDOS</i>

Ou seja, para abrir um arquivo “*.txt” no Notepad, executamos um comando similar ao seguinte:

```
>> !notepad c:/arquivo.txt
```

OBS: o processamento do MatLab permanecerá em espera enquanto o programa invocado não for finalizado (ou enquanto não houver uma abortagem manual do comando).

Capítulo 8 – Programando com Qualidade

8.1 – Por que Isso Vale a Pena

Eu sei que muita gente nem vai ler este capítulo direito. Parece frescura, mas acreditem: seguir os conselhos abaixo pode ajudar bastante a vida com a programação.

Não basta saber todas os comandos e ser um gênio de algoritmos: para programar direito, precisamos de método e disciplina.

Alunos de computação fazem uma matéria chamada INF1628 – Programação em Ponto Grande, que ensina conceitos gerais para organização e eficiência em nossos códigos no computador. Embora tal curso utilize a linguagem C, muitas das idéias podem ser aproveitadas para o MatLab.

Agora que já passamos por várias seções sobre funções, vamos analisar sobre como se utiliza tudo isso no editor. Os questionamentos levantados são importantes principalmente para o caso de uma rotina grande e complexa:

- como escolher nome para variáveis e constantes?
- onde e como inserir comentários de código?
- quais as verificações a serem feitas para evitar erros na execução?
- como testar tudo?

Ao invés de simplesmente sair escrevendo no editor, pare e pense no que está enunciado nos tópicos a seguir. Lembre-se: o seu código poderá ser lido e utilizado por outras pessoas, ou por você mesmo no futuro. Quando isso acontecer, é melhor que seu arquivo “.m” esteja claro e correto.

8.2 – Como NÃO Fazer

Veja o código abaixo como exemplo de como NÃO fazer. Veja se é possível entender o que é gerado, para que ele serve.

```
function [x,y]=circulo(a,r,p)

%figure;

if a==1
    x=r*(2*rand(1,p)-1);
    y=sqrt(1-x.^2).*r*(2*rand(1,p)-1) ;
    %xlim([-1 1]);
    %ylim([-1 1]);
    %plot(x,y,'*');

elseif a==2
    r=r*rand(1,p);
    t=2*pi*rand(1,p);
    x=r.*cos(t);
    y=r.*sin(t);
    %xlim([ -1 1 ]) ;
```

```

%ylim([ -1 1 ]) ;
%plot(x,y,'*') ;
elseif a==3
    r=r*sqrt(rand(1,p));
    t=2*pi*rand(1,p);
    x=r.*cos(t);
    y=r.*sin(t);
%subplot(3,1,3);
%figure;
%xlim([-1 1]);
%ylim([-1 1]);
%hold on;
%plot(x,y,'*');
end

```

Se você não entendeu nada, não tem problema. Explicando: essa função “circulo” retorna 2 vetores (“x” e “y”) com “p” elementos. Cada par de elementos {x,y} é um ponto aleatório distribuído num círculo de raio “r” (ou seja, $x^2 + y^2 \leq r^2$), segundo uma modalidade “a”. Se a = 1, os pontos tendem a se concentrar mais nas bordas horizontais; se a = 2, os pontos estarão mais concentrados no centro; se a = 3, os pontos estarão uniformemente distribuídos.

E esse trabalho todo com uma função razoavelmente curta! Imagina se fosse uma biblioteca de funções e rotinas mais pesada!

Listemos as críticas:

- o nome da função e das variáveis informa pouco ou nada sobre seu significado
- não há nenhuma instrução de como se utiliza a função ou para que ela serve. Mesmo que se descubra, perde-se muito tempo analisando o código. E os usuários geralmente não estão interessados em saber o funcionamento a fundo.
- o parâmetro “a” poderia ser solicitado ao usuário como uma string relacionada ao modo de distribuição, ao invés de exigir valores numéricos arbitrários como 1, 2 e 3. Isso já evitaria ter que decorar o que cada número faz.
- não há verificação com respeito da validade dos parâmetros recebidos (por exemplo: se a quantidade de pontos “p” for negativa, pode gerar erros estranhos)
- não há comentários explicativos em lugar nenhum
- a tabulação dos comandos está bem confusa, e não há separação de trechos em blocos de finalidade
- poderia haver um espaçamento entre parêntesis, variáveis, números, sinais, etc...
- existe repetição de código
- algumas expressões (provavelmente de DEBUG, para testar a função) estão comentadas. Isso não é prático, pois exige que linhas sejam comentadas e descomentadas a cada teste.

8.2 – Como Fazer

Solucionemos um a um os problemas apresentados.

Primeiramente, vamos melhorar a **nomenclatura** usada. O ideal é que o papel de determinada variável ou função dentro do programa seja evidenciado logo por seu nome. Agilizamos, assim, a compreensão se a necessidade de muitos comentários explicativos.

Deste modo, a definição da função ficaria assim:

```
function [ vet_x , vet_y ] = GerarDistribuicaoCirculo
    ( modoDistribuicao , raio , qtdPontos )
```

Eu sei, os nomes ficaram grandes. Eu sei, demora para digitar nomes tão compridos. No entanto, tal declaração da função já esclarece bastante sobre sua utilidade.

Agora aceite esta regra: ANTES de começar o código, deve-se saber para que exatamente ele vai servir, o que ele vai produzir. E a melhor maneira de especificar tudo isso é fazendo o **cabeçalho** – um trecho só de comentários contendo informações básicas iniciais. De quebra, já preparamos a explicação para os usuários sobre do que se trata o que virá adiante.

Informações importantes que o cabeçalho deve conter:

- o nome da rotina/função
- o autor e a data de quando foi desenvolvida
- uma descrição breve do que é gerado no processo
- a descrição dos parâmetros a serem recebidos (no caso de funções)
- bibliotecas exigidas para a execução.

Portanto, o arquivo “.m” da função “circulo” deveria começar assim:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FUNÇÃO GERARDISTRIBUICAOCIRCULO %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Autor: Karl Marx , 1900
%
% Descrição da Função:
% São retornados pontos aleatórios (x,y) contidos num círculo, cujos
% parâmetros são passados pelo usuário. As diversas Funções de
% Densidade de Probabilidade (fdp) estão descritas em "Parâmetros"
%
% Chamada da Função:
% [ vet_x , vet_y ] = GerarDistribuicaoCirculo( modoDistribuicao ,
%                                             raio , qtdPontos )
%
% Parâmetros:
% modoDistribuicao:
%   'concentradoBordasHorizontais' --> p(x,y) = 1 / (4*raio*
%                                             sqrt(raio^2-x^2))
%   'concentradoCentro'           --> p(x,y) = 1 / (2*pi*raio*
%                                             sqrt(x^2+y^2))
%   'uniforme'                    --> p(x,y) = 1 / (pi*raio^2)
% raio                            --> raio do círculo
% qtdPontos                       --> quantidade de pontos gerados
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A questão da melhora no parâmetro “a” (atual “modoDistribuição”) já está resolvida acima (a função solicita strings agora). Porém, apesar das especificações, é prudente verificar se o usuário não chamou a função com **argumentos inválidos**. Então, logo após o cabeçalho e antes de começarmos os cálculos, escrevemos:

```
if margin ~= 3
    error( ' *** ERRO *** Quantidade de argumentos passados inválida!' );
end

if length( raio ) ~= 1 | ~isreal( raio ) | raio < 0
    error( ' *** ERRO *** Parametro "raio" inválido!' );
end
```

```

if length( qtdPontos ) ~= 1 | ~isreal( qtdPontos ) | ...
    qtdPontos < 0 | mod( qtdPontos , 1 ) ~= 0
    error( '*** ERRO *** Parametro "qtdPontos" inválido!' ) ;
end

```

Como algumas funções têm quantidade de argumentos variável, o MatLab permite que as chamemos passando menos parâmetros do que os constados na declaração. Eis por que se verificou se foram recebidos exatamente três (através da variável “nargin”, já existente no MatLab para tal propósito).

As variáveis “raio” e “qtdPontos” não podem ser complexas, vetores ou negativas, e esta última deve ser inteira (não existem quantidade de pontos fracionária).

A checagem de “modoDistribuição” pode ser feita, por uma questão de praticidade, após os if/else:

```

if strcmp( modoDistribuicao , 'concentradoBordasHorizontais' )
    (...)
elseif strcmp( modoDistribuicao , 'concentradoCentro' )
    (...)
elseif strcmp( modoDistribuicao , 'uniforme' )
    (...)
else
    error( '*** ERRO *** Parametro "modoDistribuicao" inválido!' ) ;
end

```

Ainda assim, as mensagens de erro poderiam ser mais explícitas quanto à causa da falha (exemplo: se recebido um raio complexo, dir-se-ia explicitamente que um valor real é obrigatório).

O resto do código pode ser melhorado esteticamente até ficar assim.

```

if strcmp( modoDistribuicao , 'concentradoBordasHorizontais' )
    vet_x = raio * ( 2 * rand( 1 , qtdPontos ) - 1 ) ; % valores entre [-raio
raio]
    vet_y = sqrt( 1 - vet_x .^ 2 ) .* ( 2 * rand( 1 , qtdPontos ) - 1 ) ;
elseif strcmp( modoDistribuicao , 'concentradoCentro' )
    vet_r = raio * rand( 1 , qtdPontos ) ;
    vet_theta = 2 * pi * rand( 1 , qtdPontos ) ;

    vet_x = vet_r .* cos( vet_theta ) ;
    vet_y = vet_r .* sin( vet_theta ) ;
elseif strcmp( modoDistribuicao , 'uniforme' )
    vet_r = raio * sqrt( rand( 1 , qtdPontos ) ) ;
    vet_theta = 2 * pi * rand( 1 , qtdPontos ) ;

    vet_x = vet_r .* cos( vet_theta ) ;
    vet_y = vet_r .* sin( vet_theta ) ;
else
    error( '*** ERRO *** Parametro "modoDistribuicao" inválido!' ) ;
end

```

É interessante colocar um **prefixo nas variáveis** quando elas são vetores, matrizes, etc., porque assim se toma mais cuidado com as operações possíveis de serem feitas, além de tornar o código mais claro.

A parte de código **DEBUG** comentada poderia ganhar uma parte especial. O trecho nada mais é do que uma forma de verificar, dentro da própria função, se os dados gerados estão corretos (através de, no caso, uma plotagem).

```
% constantes
    DEBUG = 0 ; % modo DEBUG desativado

(...)

if DEBUG
    figure ;
    plot( vet_x , vet_y , '*' ) ;
    xlim( [ -raio raio ] ) ;
    ylim( [ -raio raio ] ) ;
    titulo = [ 'Modo ' modoDistribuicao ] ;
    title( titulo ) ;
end
```

Desta maneira, o código completo ficaria assim:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FUNÇÃO GERARDISTRIBUICAOCIRCULO %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Autor: Karl Marx , 1900
%
% Descrição da Função:
% São retornados pontos aleatórios (x,y) contidos num círculo, cujos
% parâmetros são passados pelo usuário. As diversas Funções de
% Densidade de Probabilidade (fdp) estão descritas em "Parâmetros"
%
% Chamada da Função:
% [ vet_x , vet_y ] = GerarDistribuicaoCirculo( modoDistribuicao ,
%                                             raio , qtdPontos )
%
% Parâmetros:
% modoDistribuicao:
%   'concentradoBordasHorizontais' -->  $p(x,y) = 1 / (4*raio*sqrt(raio^2-x^2))$ 
%   'concentradoCentro'           -->  $p(x,y) = 1 / (2*pi*raio*sqrt(x^2+y^2))$ 
%   'uniforme'                    -->  $p(x,y) = 1 / (pi*raio^2)$ 
% raio                             --> raio do círculo
% qtdPontos                         --> quantidade de pontos gerados
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% constantes
    DEBUG = 0 ; % modo DEBUG desativado

% verificando parametros recebidos
    if nargin ~= 3
        error( ' *** ERRO *** Quantidade de argumentos passados inválida!' ) ;
    end

    if length( raio ) ~= 1 | ~isreal( raio ) | raio < 0
        error( ' *** ERRO *** Parametro "raio" inválido!' ) ;
    end

    if length( qtdPontos ) ~= 1 | ~isreal( qtdPontos ) | ...
        qtdPontos < 0 | mod( qtdPontos , 1 ) ~= 0
        error( ' *** ERRO *** Parametro "qtdPontos" inválido!' ) ;
    end

if strcmp( modoDistribuicao , 'concentradoBordasHorizontais' )
    vet_x = raio * ( 2 * rand( 1 , qtdPontos ) - 1 ) ; % valores entre [-raio
raio]
    vet_y = sqrt( 1 - vet_x.^ 2 ) .* ( 2 * rand( 1 , qtdPontos ) - 1 ) ;
```



```

elseif strcmp( modoDistribuicao , 'concentradoCentro' )
    vet_r = raio * rand( 1 , qtdPontos ) ;
    vet_theta = 2 * pi * rand( 1 , qtdPontos ) ;

    vet_x = vet_r .* cos( vet_theta ) ;
    vet_y = vet_r .* sin( vet_theta ) ;

elseif strcmp( modoDistribuicao , 'uniforme' )
    vet_r = raio * sqrt( rand( 1 , qtdPontos ) ) ;
    vet_theta = 2 * pi * rand( 1 , qtdPontos ) ;

    vet_x = vet_r .* cos( vet_theta ) ;
    vet_y = vet_r .* sin( vet_theta ) ;

else
    error( '*** ERRO *** Parametro "modoDistribuicao" inválido!' ) ;

end

if DEBUG
    figure ;
    plot( vet_x , vet_y , '*' ) ;
    xlim( [ -raio raio ] ) ;
    ylim( [ -raio raio ] ) ;
    titulo = [ 'Modo ' modoDistribuicao ] ;
    title( titulo ) ;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FIM DA FUNÇÃO %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

Capítulo 9 – Exemplos Práticos

9.1 – Problema de Probabilidade

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ROTINA Regua %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Autor: Nobuhiro Watsuki , 1994
%
% Descrição da Rotina:
% Um samurai retalhador da Era Meiji, insatisfeito com os rumos da
% revolução, decide descontar sua insatisfação numa inocente régua
% ocidental de tamanho unitário. Primeiro ele a parte num ponto
% aleatório X e, não satisfeito, parte o pedaço à esquerda num outro
% ponto aleatório Y.
% Através de valores gerados aleatoriamente, faremos diversos testes
% para gerar os gráficos de Distribuição de Probabilidade e de
% Densidade de Probabilidade para Y.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% aumentar numero de casas exibidas
format long ;

% constantes (altere para aumentar/diminuir precisão dos testes)
QTD_TESTES = 20000 ;
QTD_PONTOS = 2000 ;

% testar casos para vários Y entre 0 e 1
% pontos onde a régua é quebrada pela 1a. vez ( 0 < x < 1 )
vet_x = rand( 1 , QTD_TESTES ) ;

% pontos onde a régua é quebrada pela 2a. vez ( 0 < y < x )
vet_y = vet_x .* rand( 1 , QTD_TESTES ) ;

% contando casos para montar Fy( Y ) = P( y <= Y )
vet_Y = 0 : ( 1 / QTD_PONTOS ) : 1 ;

for i = 1 : QTD_PONTOS + 1
    vet_casosFavoraveis( i ) = length( find( vet_y <= vet_Y( i ) ) ) ;
end

vet_Fy_Y = vet_casosFavoraveis / QTD_TESTES ;

% montando Função de Densidade de Probabilidade derivando numericamente Fy
vet_py_Y = diff ( vet_Fy_Y ) / ( 1 / QTD_PONTOS ) ;

% OBS: função diff gera vetor com 1 valor a menos
vet_py_Y = [ vet_py_Y 0 ] ; % acrescentando 1 elemento ao vetor

% plotando Fy_Y
figure ;

vet_pontosY = [ 0 : 1 / QTD_PONTOS : 1 ] ;
vet_pontos2Y = [ 0 : 10 / QTD_PONTOS : 1 ] ;

subplot( 2 , 1 , 1 ) ;
plot( vet_pontosY , vet_Fy_Y ) ;
xlabel( 'y' , 'fontsize' , 13 ) ;
ylabel( 'F_y( Y )' , 'fontsize' , 13 ) ;
title( 'Problema da Régua: caso da segunda partição à esquerda' , ...
'fontsize' , 13 ) ;

% plotar curva teórica
```

```

        hold ;
        plot( vet_pontos2Y , - vet_pontos2Y .* ...
              ( log( vet_pontos2Y ) - 1 ) , '*r' ) ;
        legend( 'Simulada' , 'Teorica' ) ;
        hold ;

% plotando py_Y
subplot( 2 , 1 , 2 ) ;
plot( vet_pontosY , vet_py_Y ) ;
xlabel( 'y' , 'fontsize' , 13 ) ;
ylabel( 'p_y( Y )' , 'fontsize' , 13 ) ;

% plotar curva teórica
hold ;
plot( vet_pontos2Y , - log( vet_pontos2Y ) , '*r' ) ;
legend( 'Simulada' , 'Teorica' ) ;
hold ;

% deleta variáveis usadas acima
clear ;

```

%% FIM DA ROTINA %%%

9.2 – Criando um Polinômio Graficamente

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ROTINA DesenharPolinomio %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Autor: Pablo Picasso, 1931
%
% Descrição da Rotina:
%   Demonstração da captura de pontos e do uso de polinômios. O usuário
%   selecionará N > 1 pontos da tela e será desenhado um polinômio que
%   os contenha. Sua equação aparecerá na legenda.
%   Essa rotina pode ser modificada para, por exemplo, traçar a melhor
%   reta com base nesses pontos.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% tela em branco para a captura de pontos
handle_grafico = figure ;

xlim( [ -10 10 ] ) ;
ylim( [ -10 10 ] ) ;

set( gca , 'fontsize' , 13 ) ;
grid ;
title( 'Selecione 2 ou mais pontos na tela e pressione ENTER' ) ;

% captura dos pontos
[ vet_XSelecionados , vet_YSelecionados ] = getpts ;

if length( vet_XSelecionados ) <= 1
    close( handle_grafico ) ;
    error( '*** ERRO *** SELECIONE 2 OU MAIS PONTOS!' ) ;
end

% gerando o polinômio que contém pontos selecionados
grauPolinomio = length( vet_XSelecionados ) - 1 ;
vet_coeficientes = polyfit( vet_XSelecionados , vet_YSelecionados , ...
                           grauPolinomio ) ;

% gerando pontos do polinômio a partir dos coeficientes encontrados
vet_XPolinomio = [ -10 : 0.05 : 10 ] ;
vet_YPolinomio = polyval( vet_coeficientes , vet_XPolinomio ) ;

% string contendo o polinômio
string_polinomio = '' ;

```

```

for i = 1 : grauPolinomio

    if vet_coeficientes( i + 1 ) > 0
        string_sinal = ' + ' ;
    elseif vet_coeficientes( i + 1 ) < 0
        string_sinal = ' - ' ;
    end

    if i < grauPolinomio
        string_expoente = [ '^' num2str( grauPolinomio + 1 - i ) ] ;
    else
        string_expoente = ' ' ;
    end

    if vet_coeficientes( i ) ~= 0
        string_polinomio = [ string_polinomio ...
            num2str( vet_coeficientes( i ) ) ...
            'x' string_expoente string_sinal ] ;
    end

end

% final da string: termo independente não tem 'x^0 + '
if vet_coeficientes( grauPolinomio + 1 ) ~= 0
    string_polinomio = [ string_polinomio ...
        num2str( vet_coeficientes( grauPolinomio + 1 ) ) ] ;
end

% plotando nova figura com pontos selecionados e o polinômio gerado
plot( vet_XSelecionados , vet_YSelecionados , 'ro' , ...
    'LineWidth' , 1.5 ) ;
hold ;
plot( vet_XPolinomio , vet_YPolinomio , 'b' , 'LineWidth' , 2 ) ;

xlabel( 'x' ) ;
ylabel( 'y' ) ;
title( 'Polinômio gerado com a captura dos pontos' ) ;
legend( 'Pontos Selecionados' , string_polinomio ) ;

hold ;

% limpa variáveis da memória
clear ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FIM DA ROTINA %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

9.3 – “Animação Gráfica”

Lembrando que as equações do círculo e da elipse são, respectivamente:

$$(x - x_0)^2 + (y - y_0)^2 \leq r^2 \qquad \frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1$$

ou

$$\begin{matrix} x = r \cos(\theta) + x_0 \\ y = r \sin(\theta) + y_0 \end{matrix}, \text{ com } \begin{matrix} r \in [0 \text{ raio}] \\ \theta \in [0 \text{ } 2\pi] \end{matrix} \qquad y = \pm b \sqrt{1 - \frac{x^2}{a^2}} + y_0$$

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ROTINA OlhosDeRessaca %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Autor: Machado de Assis , 1900
%

```

```

% Descrição da Rotina:
% Demonstração literária da geração de números aleatórios e da plotagem
% no MatLab.
% Geram-se, para a esquerda e para a direita, um círculo e uma elipse
% incompleta nas extremidades horizontais, com pontos aleatórios.
% Enquanto a elipse tem seus pontos uniformemente distribuídos, o
% círculo os tem mais densamente no centro.
% Se você cursa ou já cursou "Probabilidade e Estatística" ou "Modelos
% Probabilísticos para Engenharia Elétrica", tente demonstrar o
% resultado através da análise teórica das equações utilizadas.
% A plotagem é feita aos poucos. Pegue uma pipoca e assista à animação.
%
% Citação:
% Retórica dos namorados, dá-me uma comparação exata e poética para o que
% foram aqueles olhos de Capitu. Não me acode imagem capaz de dizer,
% sem quebra da dignidade do estilo, o que foram e me fizeram. Olhos
% de ressaca? Vá, de ressaca. É o que me dá idéia daquela feição nova.
% Traziam não sei que fluido misterioso e enérgico, uma força que
% arrastava pra dentro, como a vaga que se retira da praia, nos dias
% de ressaca. Para não ser arrastado, agarrei-me às outras partes
% vizinhas, às orelhas, aos braços, aos cabelos espalhados pelos
% ombros, mas tão depressa buscava as pupilas, a onda que saía delas
% vinha crescendo, cava e escura, ameaçando envolver-me, puxar-me e
% tragar-me.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% constantes
    QTD_PONTOS = 400 ;
    TEMPO_ENTRE_PONTOS = 0.001 ;
    X0_OLHO = 2 ;
    X0_SOMBR = 0.5 ;

% configurando olho esquerdo
% íris (círculo de raio unitário deslocado para esquerda)
    vet_r = rand( 1 , QTD_PONTOS ) ; % gera de 0 a 1
    vet_theta = 2 * pi * rand( 1 , QTD_PONTOS ) ; % gera de 0 a 2*Pi

    vet_irisEsqX = vet_r .* cos( vet_theta ) - X0_OLHO ;
    vet_irisEsqY = vet_r .* sin( vet_theta ) ;

% pálpebras (elipse incompleta deslocada para direita)
    vet_maisOuMenos = 2 * ( rand( 1 , QTD_PONTOS ) > 0.5 ) - 1 ;

    vet_palpebraEsqX = 1.6 * ( 2 * rand( 1 , QTD_PONTOS ) - 1 ) ;
    vet_palpebraEsqY = vet_maisOuMenos .* ( 1.6 * sqrt( 1 - ...
        vet_palpebraEsqX .^ 2 / ( 2 ^ 2 ) ) ) ;

    vet_palpebraEsqX = vet_palpebraEsqX - X0_OLHO ;

% sobrancelha
    vet_sobrancelhaEsqX = ( -3.5 ) * rand( 1 , QTD_PONTOS ) - X0_SOMBR ;
    vet_sobrancelhaEsqY = 1.6 * sqrt( 1 - vet_sobrancelhaEsqX .^ 2 ...
        / ( 6 ^ 2 ) ) + 2 ;

% configurando olho direito
% íris (círculo de raio unitário deslocado para direita)
    vet_r = rand( 1 , QTD_PONTOS ) ; % gera de 0 a 1
    vet_theta = 2 * pi * rand( 1 , QTD_PONTOS ) ; % gera de 0 a 2*Pi

    vet_irisDirX = vet_r .* cos( vet_theta ) + X0_OLHO ;
    vet_irisDirY = vet_r .* sin( vet_theta ) ;

% pálpebras (elipse incompleta deslocada para direita)
    vet_maisOuMenos = 2 * ( rand( 1 , QTD_PONTOS ) > 0.5 ) - 1 ;

    vet_palpebraDirX = 1.6 * ( 2 * rand( 1 , QTD_PONTOS ) - 1 ) ;
    vet_palpebraDirY = vet_maisOuMenos .* ( 1.6 * sqrt( 1 - ...
        vet_palpebraDirX .^ 2 / ( 2 ^ 2 ) ) ) ;

```

```

vet_palpebraDirX = vet_palpebraDirX + X0_OLHO ;

% sobrançelha
vet_sombrancelhaDirX = 3.5 * rand( 1 , QTD_PONTOS ) + X0_SOMBR ;
vet_sombrancelhaDirY = 1.6 * sqrt( 1 - vet_sombrancelhaDirX .^ 2 ...
    / ( 6 ^ 2 ) ) + 2 ;

% plotando ponto a ponto com pausas para visualização
figure ;
hold ;

xlim( [ -( X0_OLHO + 2 ) ( X0_OLHO + 2 ) ] ) ;
ylim( [ -3 5 ] ) ;

title( 'Olhos de Ressaca' , 'FontSize' , 13 ) ;
xlabel( '(por favor, aguarde o término da animação)' ) ;

for i = 1 : QTD_PONTOS
    plot( vet_irisEsqX( i ) , vet_irisEsqY( i ) , '.' ) ;
    plot( vet_irisDirX( i ) , vet_irisDirY( i ) , '.' ) ;

    if rem( i , 2 ) == 0
        plot( vet_palpebraEsqX( i ) , vet_palpebraEsqY( i ) , '.' ) ;
        plot( vet_sombrancelhaEsqX( i ) , vet_sombrancelhaEsqY( i ) , ...
            '.' ) ;

        plot( vet_palpebraDirX( i ) , vet_palpebraDirY( i ) , '.' ) ;
        plot( vet_sombrancelhaDirX( i ) , vet_sombrancelhaDirY( i ) , ...
            '.' ) ;
    end

    pause( TEMPO_ENTRE_PONTOS ) ;
end

xlabel( 'Animação concluída!' ) ;

% limpando variáveis utilizadas
clear ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FIM DA ROTINA %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

9.4 – Biot-Savart Numérico

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FUNÇÃO CalcularCampoDipolo %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Autor: Biot-Savart , século XIX
%
% Descrição da Função:
%   Calcula o campo magnético H dum dipólo, num ponto qualquer dado em
%   coordenadas cilíndricas.
%
% Chamada da Função:
%   Hz = CalcularCampoDipolo( i , a , rho , z ) ;
%
% Parâmetros Recebidos:
%   i   - corrente que circula no dipolo
%         (sentido anti-horário)                -> em Ampères
%   a   - raio da espira (dipolo)                -> em metros
%   rho - coordenada RHO do ponto (distância ao eixo Z) -> em metros
%   z   - coordenada Z do ponto                  -> em metros
%
% Valor retornado:
%   Hz  - valor do componente Z do campo        -> em Ampère * metro
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

function Hz = CalcularCampoDipolo ( i , a , rho , z )

    if nargin ~= 4
        error( '*** ERRO *** Devem ser passados exatos 4 argumentos!' ) ;
    end

    if ( length( i ) ~= length( a ) | length( i ) ~= length( rho ) | ...
        length( i ) ~= length( z ) ...
        ) & ...
        ( ~( length( i ) == 1 & length( a ) == 1 & length( rho ) == 1 ) & ...
          ~( length( i ) == 1 & length( z ) == 1 & length( rho ) == 1 ) & ...
          ~( length( a ) == 1 & length( z ) == 1 & length( rho ) == 1 ) & ...
          ~( length( i ) == 1 & length( a ) == 1 & length( z ) == 1 ) ...
        )
        error( [ '*** ERRO *** Os 4 parâmetros devem ter a mesma ' ...
                 'dimensão, ou apenas 1 deve ser não-escalar!' ] ) ;
    end

    if ~isreal( i )
        error( '*** ERRO *** Parametro 1 ("i") inválido!' ) ;
    end

    if ~isreal( a ) | a < 0
        error( '*** ERRO *** Parametro 2 ("a") inválido!' ) ;
    end

    if ~isreal( rho ) | rho < 0
        error( '*** ERRO *** Parametro 3 ("rho") inválido!' ) ;
    end

    if ~isreal( z )
        error( '*** ERRO *** Parametro 4 ("z") inválido!' ) ;
    end

    k = 4 * a .* rho ./ ( ( a + rho ) .^ 2 + z .^ 2 ) ;
    [ K , E ] = ellipke( k ) ;

    termo1 = i ./ ( 2 * pi * sqrt( ( a + rho ) .^ 2 + z .^ 2 ) ) ;
    termo2 = K + ( a .^ 2 - rho .^ 2 - z .^ 2 ) .* E ./ ...
              ( ( a - rho ) .^ 2 + z .^ 2 ) ;

    Hz = termo1 .* termo2 ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FIM DA FUNÇÃO %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

9.5 – Documentador de “*.m”s

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ROTINA Documentador %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Autor: PPG-Man, o dono do "calabouço", 1970
%
% Descrição da Rotina:
% Demonstração da manipulação de arquivos textos, com a utilidade de
% englobar num ".doc" os cabeçalhos de todos os arquivos ".m" de um
% diretório.
%
% Citação:
% Nunca se tem tempo para fazer direito, mas sempre se tem tempo para
% fazer de novo.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% constantes
NOME_ARQUIVO = 'documentacao.txt' ;
DIRETORIO = './' ; % diretório atual

```

```

% buscando arquivos ".m" no diretório
vet_struct_arquivosM = dir( [ DIRETORIO '*.m' ] ) ;

if isempty( vet_struct_arquivosM )
    error( '*** ERRO *** Nao ha arquivo M no diretorio!' ) ;
end

% criando arquivo de documentação
fid_arquivoDocumentacao = fopen( [ DIRETORIO NOME_ARQUIVO ] , 'w' ) ;

fprintf( fid_arquivoDocumentacao , 'Documentos do diretorio ' ) ;
fprintf( fid_arquivoDocumentacao , DIRETORIO ) ;
fprintf( fid_arquivoDocumentacao , ':\n\n' ) ;

% lendo cada arquivo ".m"
for i = 1 : length( vet_struct_arquivosM )
    string_nomeArquivo = [ DIRETORIO vet_struct_arquivosM( i ).name ] ;
    fid_arquivoM = fopen( string_nomeArquivo , 'r' ) ;

    % OBS: todo o cabeçalho do arquivo pode ser obtido mais facilmente
    % com "string_cabecalho = help( string_arquivoM )", mas o caminho
    % mais complicado foi feito por razões didáticas

    if fid_arquivoM == -1
        error( '*** ERRO *** Arquivo ".m" não pôde ser aberto!' ) ;
    end

    % pulando possíveis linhas sem comentários
    string_linhaArquivoM = '' ;
    while ~feof( fid_arquivoM ) & ...
        ( isempty( string_linhaArquivoM ) | ...
          string_linhaArquivoM( 1 ) ~= '%' )

        string_linhaArquivoM = fgetl( fid_arquivoM ) ;
    end

    % copiando linhas de comentários iniciais
    while ~feof( fid_arquivoM ) & ...
        ~isempty( string_linhaArquivoM ) & ...
        string_linhaArquivoM( 1 ) == '%'

        string_linhaArquivoM = strrep( string_linhaArquivoM , ...
            '%' , '%%' ) ;
        string_linhaArquivoM = strrep( string_linhaArquivoM , ...
            '\' , '\\\' ) ;

        fprintf( fid_arquivoDocumentacao , string_linhaArquivoM ) ;
        fprintf( fid_arquivoDocumentacao , '\n' ) ;
        string_linhaArquivoM = fgetl( fid_arquivoM ) ;
    end

    fprintf( fid_arquivoDocumentacao , '\n\n' ) ;

    fclose( fid_arquivoM ) ;
end

% fechando (salvando) arquivo da documentação
fclose( fid_arquivoDocumentacao ) ;

% chamando NotePad para abrir documentação
string_comando = [ '!notepad ' DIRETORIO NOME_ARQUIVO ]
eval( string_comando ) ;

% limpando variaveis usadas
clear ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FIM DA ROTINA %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```