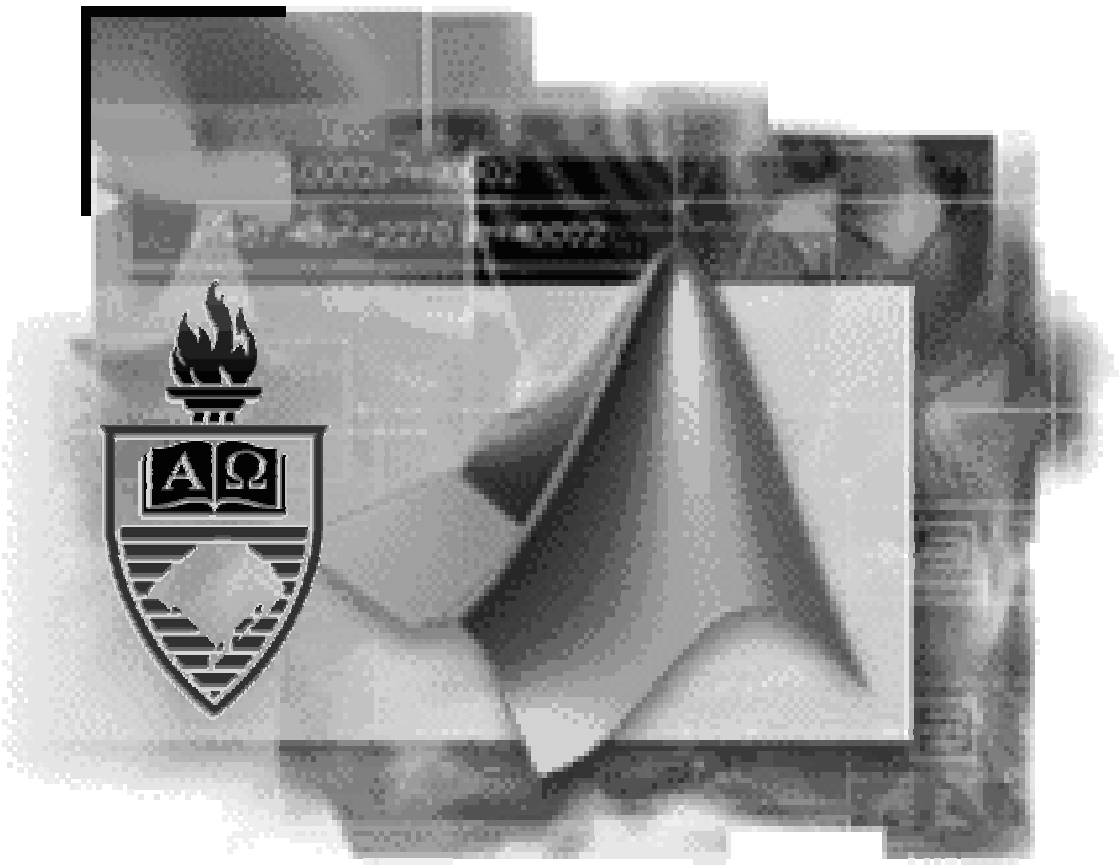
MATLAB
Ferramenta matemática para Engenharia



Luciano André Farina
Maurício Simões Posser

Laboratório de Controle e Integração de Processos

Universidade Federal do Rio Grande do Sul
Escola de Engenharia
Departamento de Engenharia Química
Semana Acadêmica de Engenharia Química

AULA 1 - CONCEITOS BÁSICOS 3

1. O QUE É O MATLAB - ESTRUTURA DOS DIRETÓRIOS 3
2. COMANDOS DE LINHAS 3
3. COMANDOS BÁSICOS 3
4. SÍMBOLOS E CONSTANTES 3
5. TRABALHANDO COM MATRIZES 3
6. GRÁFICOS 4

AULA 2 – PROGRAMAÇÃO 7

1. ALGORITMOS – NOÇÕES BÁSICAS DE PROGRAMAÇÃO 7
2. M-FILES 7
3. COMANDOS DE FLUXO E OPERADORES LÓGICOS 8
4. VETORIZAÇÃO 9
5. UTILIZAÇÃO DO DEBUGGER 9
6. EXCELLINK 10

AULA 3 – OUTRAS FORMAS DE MANIPULAR INFORMAÇÕES 11

1. OUTRAS FORMAS DE ARMAZENAR DADOS: 11

AULA 4 – SIMULINK 13

1. IDÉIA BÁSICA DO FUNCIONAMENTO DO SIMULINK 13
2. MÁSCARAS 15
3. S-FUNCTIONS 16
4. TABELAS DE AUXÍLIO 18

AULA 5 – TOOLBOXES & GUIDE 21

1. TOOLBOXES 21
2. GUIDE 25

LISTA DE EXERCÍCIOS 28

BIBLIOGRAFIA RECOMENDADA 31

Aula 1 - Conceitos Básicos

1. *O que é o MATLAB - Estrutura dos diretórios*

MATLAB é uma abreviação para MATrix LABoratory, é um ambiente de programação de alto nível, possuindo características de aplicativo (facilidade para o usuário) e de linguagem de programação (flexibilidade).

Ao contrário de linguagens clássicas como C e Fortran, no ambiente MATLAB o usuário não se preocupa com itens como declaração de variáveis, alocação de memória, utilização de ponteiros e outras tarefas de rotina.

O MATLAB apresenta uma série de funções matemáticas já implementadas que podem ser utilizadas em uma rotina construída pelo usuário. Estas funções são agrupadas de acordo com a área de interesse em *toolboxes* e armazenadas em diretórios específicos. Qualquer função a ser utilizada deve estar no diretório de trabalho ou caminho (*path*) do MATLAB.

2. *Comandos de linhas*

Todas as funções do MATLAB são executadas através de comandos digitados diretamente na área de trabalho (*workspace*) após o *prompt*.

```
» 5 + 7        » 5 * 2        » 2 / 3
» 2 \ 3        » 2 ^ 3
```

A resposta destas operações será armazenada na variável *ans*, cada operação atualizará o seu valor. Para inibir a exibição da resposta na tela, deve-se terminar a expressão com ponto e vírgula (;). Para armezenar valores em uma variável específica, deve-se atribuir o nome da variável:

```
» a = 4 + 2 ; ↵    » b = 7 * 3 ; ↵    » c = a + b ↵
```

O MATLAB pode armazenar variáveis, diferindo entre maiúsculas e minúsculas, ou seja, é *case sensitive*, deste modo as variáveis x e X são diferentes. As funções matemáticas mais utilizadas são listadas a seguir:

- funções trigonométricas: *sin; cos; tan; asin*; *acos; atan*
- logaritmo decimal: *log10*
- logaritmo natural: *log*
- exponencial: *exp*
- valor absoluto: *abs*

Listas contendo funções elementares e especiais podem ser visualizadas executando o comando ***help*** seguido das palavras *elfun* e *specfunc*.

3. *Comandos Básicos*

Alguns comandos básicos que ajudam o usuário na utilização do software são descritos a seguir:

- *help*: ajuda *on line* sobre funções do MATLAB
  » help sin » help ode23
- *who*: informa as variáveis presentes no *workspace*. O comando ***whos*** retorna as variáveis e algumas informações sobre as mesmas
- *clc*: limpa a tela
- *which*: informa o diretório de execução de uma determinada função
  » which sin » which ode23
- *diary*: salva o texto de uma seção do MATLAB no diretório corrente
- *laod/save*: carrega / salva em disco as variáveis do *workspace* como arquivo de dados (extensão *.mat*)
- *clear*: limpa variáveis do *workspace*
  » clear a; » clear all;
- *what*: retorna os arquivos relacionados ao MATLAB presentes no diretório de trabalho
- *dir* / *ls*: retorna todos os arquivos do diretório de trabalho
- *cd*: muda (ou apenas informa) o diretório corrente
  » cd .. » cd c:\temp » cd
- *why*: grande mistério da humanidade

4. *Símbolos e constantes*

Diversos símbolos representam constantes ou limites da máquina. Os principais são:

- *eps*: precisão relativa da máquina
- *realmin/realmax*: menor/maior número representável na máquina
- *inf*: representação do infinito positivo (-INF representa o infinito negativo)
- *nan*: representação de uma indeterminação matemática (p.ex.: 0/0; Inf/Inf)
- *pi*: 3.1415926535897...

5. *Trabalhando com matrizes*

O MATLAB foi desenvolvido especialmente para trabalhar com representações matriciais de elementos, tendo maior eficiência quando o usuário utiliza esta representação.

Entrada de matrizes

Matrizes podem ser geradas de diferentes modos:

- Entrando os elementos explicitamente

  ```
  » a = [1 , 2 ; 3 , 4]      » a = [1  2] ↵
                                   3  4]
  ```

  Para separar elementos de colunas diferentes pode-se usar a vírgula ou o ESPAÇO. Para separar linhas usa-se o ponto e vírgula ou o ENTER(↵).

Cada elemento da matriz é indexado, sendo possível extrair-se um elemento indicando sua posição da forma linha-coluna ou da forma seqüencial.

» a (2) = 3 » a (2,1) = 3

- Usando funções de construção

Algumas funções do MATLAB auxiliam a montagem de matrizes com características específicas. Alguns exemplos são a matriz identidade e matrizes com uns ou zeros .

» eye(3) » ones(3) » zeros(3)

- Utilizando a forma **:** de geração de seqüências.

» a = [1:3 ; 4:6]

Operações e Manipulações de matrizes

Uma forma de manipular elementos de uma matriz é utilizando a forma de indexação juntamente com comandos de laço, ou de forma mais eficiente, utilizando a forma de seqüências geradas pelo operador (**:**).

» a (: , 1) todas as linhas da coluna 1
» a (1 , :) todas as colunas da linha 1
» a ([1 3], :) todas as colunas das linhas 1 e 3

As operações de adição, subtração, multiplicação e divisão (+ - * /) seguem as regras de operações matriciais, enquanto que o operador *ponto* antecedendo as operações de multiplicação e divisão, faz com que a mesma seja realizada elemento a elemento.

```
» [2  3; 4 5] .* [1  2; 3 4]
  ans =
    [2  6
     12 20]
```

Além destas operações básicas, ainda pode-se aplicar funções trigonométricas, exponenciais e logarítmicas a matrizes, as quais retornarão uma matriz com os elementos da matriz original avaliados pela função aplicada ponto a ponto. No caso da operação potência (^), a solução segue regra da multiplicação matricial, enquanto que a operação *ponto* antecedendo (**.**^), elevará elemento a elemento na potência dada.

```
» [2  3; 4  5] .^2
  ans =
    [4  9
    16  25]
```

Outras operações e funções geralmente utilizadas são:

• : transposta conjugada
eig(A) : auto valores/vetores da matriz A
norm(A): norma da matriz A
det(A) : determinante da matriz A
inv(A) : inversa de A

Outras funções podem ser encontradas no *help matfun.*

Para manipular matrizes ainda pode-se utilizar funções específicas, como por exemplo:

diag(A)	extrai a diagonal da matriz A
diag([1 2])	gera uma matriz diagonal com os elementos da diagonal iguais a 1 e 2.
size(A)	retorna um vetor com as dimensões da matriz A
reshape(A,m,n)	redimensiona a matriz A segundo o as novas dimensões m_xn

Outras funções podem ser encontradas no *help elmat.*

STRINGS

Entende-se por *strings* o conjunto de caracteres colocados entre aspas simples.
» s = '*Este texto é uma string*'

Uma *string* é entendida pelo MATLAB como um vetor de caracteres, portanto, no exemplo acima, para pegarmos apenas a palavra *Este*, basta fazer

```
» s(1:4)
  ans =
    Este
```

Como é encarada como um vetor, também existem formas de manipular-mos *strings* através de indexação de seus elementos ou então utilizando funções especiais, como por exemplo:

eval(S)	avalia a *string* S como uma expressão do MATLAB
str2num('7')	converte a *string* 7 no número 7
num2str(7)	converte o número 7 em *string*
strcat(S,T)	concatena as *strings* S e T
strcmp(S,T)	compara as *strings* S e T

Outras funções podem ser encontradas no *help strfun.*

6. Gráficos

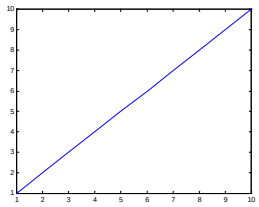
Os gráficos são considerados pelo MATLAB como figuras, nas quais pode-se visualizar de várias formas diferentes um conjunto de dados. Como são figuras, para manipulá-los, pode-se utilizar comandos de linha (diretamente através da barra de ferramentas da janela criada para o gráfico) ou então utilizando o *GUIDE*, como será visto mais adiante.

O MATLAB apresenta uma série de funções prontas para montar gráficos 2D, 3D, de barras, de dispersão, e outros. Para ver detalhadamente estas funções, basta dar um *help graph2d*, *graph3d* ou *specgraph*, enquanto que os comandos de linha para manipulá-los podem ser encontrados no *help graphics.*

2D PLOTS

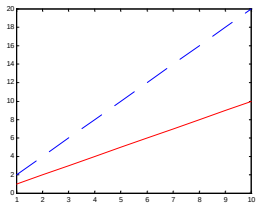
Existem várias formas de fazer gráficos 2D no MATLAB, a mais usual é utilizando o comando *plot*. Para tanto, deve-se especificar os vetores das ordenadas e das abcissas.

```
» x = [1:10] ; y = x;
» plot(x,y);
```



Este comando faz o gráfico de uma reta $y = x$. No caso da entrada ser apenas um vetor coluna, o comando *plot* o considera como ordenada e gera a abcissa automaticamente. As cores da linha, bem como seu estilo, podem ser acrescentados como argumentos desta função da seguinte forma:

```
» plot(x, x, 'r', x, 2 * x, '--b');
```



Este comando gera um gráfico com duas retas, $y=x$ e $y= 2x$, a primeira em linha contínua vermelha e a segunda descontínua azul.

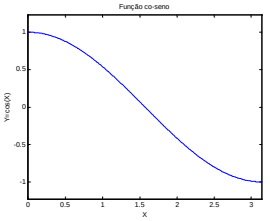
As opções que podem ser utilizadas no comando *plot* são as seguintes:

cor		marcador		tipo de linha	
y	amarelo	.	ponto	-	sólido
m	magenta	o	círculo	:	ponto
c	ciano	x	x-marca	-.	traço-ponto
r	vermelho	+	cruz	--	tracejada
g	verde	*	estrela		
b	azul	s	quadrado		
w	branco	d	diamante		
k	preto	v	triângulo		

Para mais opções basta ver no *help plot*.

Um outro comando que gera gráficos de funções bidimensionais é o *ezplot*, ele tem como argumento uma função na forma de *string* e um intervalo de variação. Neste caso não é preciso definir os vetores das ordenadas e abcissas.

```
» ezplot('cos(x)' , [0,pi]);
```



Para adicionar um título ao gráfico utiliza-se o comando *title*, enquanto que para nomear os eixos, utiliza-se o comando *xlabel* / *ylabel*. Para adicionar mais de um gráfico na mesma figura, utiliza-se o comando *hold*. Este comando funciona como uma chave liga-desliga, adicionando outras funções no mesmo gráfico enquanto não for executado o *hold off*.

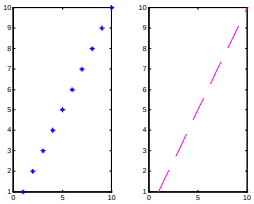
```
» title('Função co-seno');
» xlabel('X'); ylabel('Y=cos(X)');
» hold on; ezplot('sin(x)' , [0,pi]); hold off;
```

Outros comandos muito utilizados são:

- *axes*: gera eixos
- *axis*: controla a escala dos eixos
- *legend*: adiciona legenda
- *grid on/off*: adiciona/retira as linhas de grade do gráfico
- *ginput*: permite clicar no gráfico para pegar pontos
- *gtext*: adiciona textos em qualquer posição específica na figura.

Ainda é possível fazer mais de um gráfico na mesma figura com o comando *subplot*.

```
»subplot(1,2,1);plot(1:10,'*b');subplot(1,2,2);plot(1:10,'--m');
```

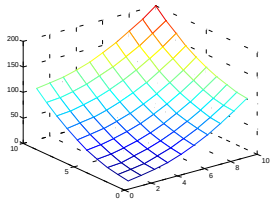


Desta forma o gráfico da reta com o marcador * será o primeiro de uma figura que permite apresentar dois gráficos, em uma linha e duas colunas: *subplot(linha,coluna,posição)*

3D PLOTS

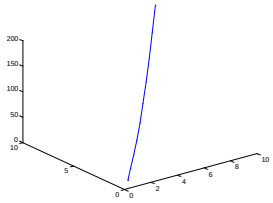
Para gerar gráficos 3D é preciso montar uma malha com os vetores das três dimensões envolvidas. O comando que gera esta malha com facilidade é o *mesgrid* que toma dois vetores como argumento para gerar uma malha que será utilizada para o cálculo da uma terceira dimensão.

```
» x = [1:10]' ; y = x;
» [X,Y] = meshgrid (x,y);
» Z = X.^2 + Y.^2;
» mesh (Z);
```



O comando *mesh* cria um gráfico 3D na forma de uma malha. Outros comandos, como o *surf* e o *surfl* criam um gráfico com os elementos da malha preenchidos, permitindo ainda a possibilidade de uma representação contínua, através do comando *shading interp*. Para representar linhas e pontos no espaço, o comando utilizado é o *plot3*, que toma três vetores como argumento.

```
» x = [1:10]' ; y = x; z = x.^2 + y.^2;
» plot3(x , y , z);
```



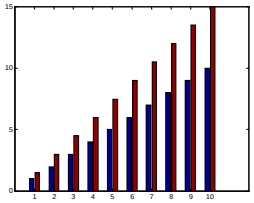
Para alterar as cores, pode-se utilizar o comando *colormap*, que toma como argumento um vetor RGB (*red green blue*), ou então vetores pré definidos como na tabela a seguir:

hot	*gray*	*bone*	*copper*	*pink*	*white*	*flag*
vga	*jet*	*prism*	*autumn*	*cool*	*hsv*	*winter*

Gráficos especiais (line, barplot...)

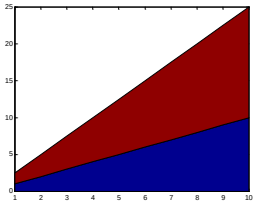
Dentre os gráficos especiais, os mais utilizados são:

```
» x=[1:10];y = [x  1.5*x];
» bar(x,y);
```



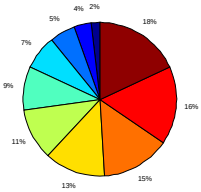
Faz um gráfico de barras da matriz *y* $m_x n$ com *m* grupos de *n* barras verticais enquanto que o vetor *x* deve ser monotonicamente crescente ou decrescente.

```
» area(x,y);
```



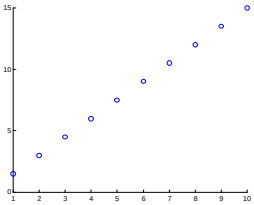
Neste caso, as colunas da matriz *y* são apresentadas como áreas preenchidas.

```
» pie(x);
```



Faz um gráfico pizza dos valores do vetor *x*, que é normalizado e apresentados como proporções.

```
» scatter(x,y(:,2));
```



Faz um gráfico de dispersão dos dados. Equivale ao *plot(x,y,'o')*.

Outros comando de manipulação utilizados em gráficos 2D e 3D estão na barra de ferramentas das próprias figuras.

abre uma figura nova	seleciona objetos	zom + / zoom -
abre uma figura	insere texto	gira o objeto
salva a figura	insere seta	
imprime a figura	insere reta	

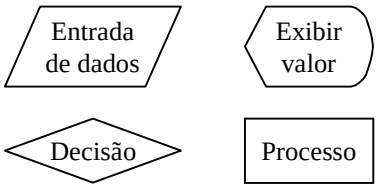
Aula 2 – Programação

1. *Algoritmos – Noções Básicas de Programação*

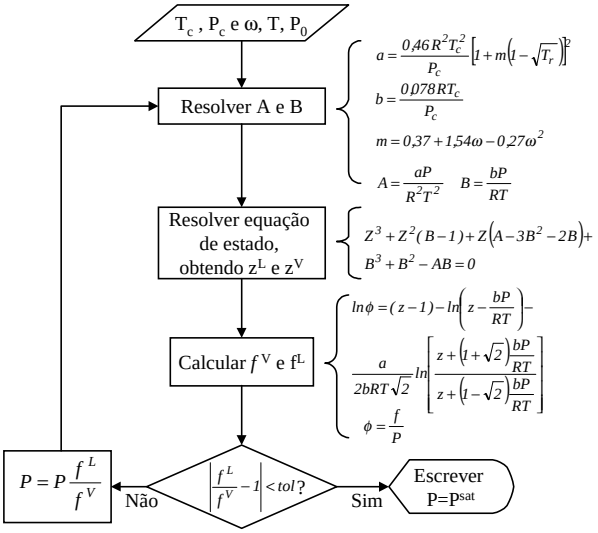
Um programa nada mais é do que uma seqüência de comandos (rotina) estabelecida de acordo com um objetivo pré-estabelecido, ou seja, é imprescindível que, antes de iniciar-se um programa, o usuário esteja completamente ciente do que ele deseja que a rotina realiza.

O primeiro passo para o desenvolvimento de uma rotina é a construção de um algoritmo para o programa, onde todos os comandos devem ser explicitados. Uma boa maneira para a construção do algoritmo é a utilização de um fluxograma. A partir do fluxograma, o único trabalho do usuário é 'traduzir' o esquema para a linguagem de programação, qualquer que seja esta.

Uma simbologia para o desenho do fluxograma é apresentada a seguir:



A seguir, utilizando esta notação, é mostrado um fluxograma com o algoritmo para cálculo da pressão de vapor de um líquido para uma dada temperatura utilizando a equação de estado de Peng Robinson.



No exemplo, as equações utilizadas são mostradas ao lado de cada bloco de processo, o que também ajuda na implementação. As entradas do processo são constantes do componente (temperatura e pressão crítica e fator acêntrico), a temperatura desejada e uma estimativa inicial para a pressão de vapor. A partir destes dados, utilizando a equação de estado de Peng-Robinson a rotina é resolvida iterativamente até que o critério de convergência do bloco de decisão interrompa o ciclo.

Para implementar esta rotina em qualquer linguagem de programação, basta atribuir os comandos devidos, de forma que o programa execute o algoritmo passo a passo.

2. *M-FILES*

A linguagem de programação do MATLAB utiliza exatamente os mesmos comandos descritos na aula anterior, do mesmo modo que eles seriam utilizados se digitados no *workspace*, porém os comandos ficam agrupados e a execução de todos eles, em seqüência, pode ser realizada digitando-se apenas o nome atribuído à seqüência, que pode ser salva em um arquivo de texto com a extensão .m. Os arquivos contendo rotinas de execução para o MATLAB são denominados *m-files*.

Para inserir comentários no texto, usa-se o símbolo %. O uso de comentários permite um melhor entendimento da rotina por qualquer usuário.

A ajuda (explicação sobre o funcionamento da rotina) de cada *m-file* também é definida utilizando linhas de comentário. As primeiras linhas do programa, se marcadas com o marcador % são exibidas na tela quando o usuário digira *help nomearq*, onde 'nomearq' é o nome do arquivo de texto.

Há duas opções de utilização dos *m-files* pelo MATLAB: usando os chamados *script-files* ou usando as *functions*. Ambas consistem da utilização das funções básicas do aplicativo na seqüência desejada, sendo que o que as diferencia são o local onde as variáveis são armazenadas durante os cálculos e a forma de entrada e saída de dados.

Script-files

A utilização de um *script-file* funciona exatamente da mesma forma que a execução direta de comandos no *prompt* do MATLAB. Quando um *script-file* é dodado, cada comando presente no arquivo de texto é executado.

Os *script-files* podem operar com variáveis presentes no *workspace* ou com dados criados durante a execução. Uma vez que este tipo de *m-file* não possui argumentos de entrada e saída, as variáveis criadas são armazenadas no *workspace*do MATLAB.

Os *script-files* podem, inclusive, retornar elementos gráficos, usando comandos como *plot*.

A seguir é proposto um exemplo de *script-file* para determinar a fórmula molecular de um composto orgânico a partir da sua fórmula centesimal e do peso molecular do composto. O *m-file* pode ser escrito em qualquer editor de texto, sendo armazenado com a extensão .m. Para ser executado, deve-se digitar no *prompt* do MATLAB o nome do arquivo (sem a extensão), sendo que o arquivo deve estar situado no diretório de trabalho ou então estar no caminho (*path*) do MATLAB.

```
% Programa:
% Fórmula centesimal --> Fórmula molecular

% Inicialização
clc, clear all , clear global
% Pesos moleculares dos elementos
pm=[ 12.01 1.01 16 14.01 ];
% Entrada de dados
comp=input('Entre com o vetor de composições
percentuais (C,H,O,N) ');
pmc=input('Entre com o peso molecular do composto ');
% Execução
prorp=pmc*comp/100;
formula=round(prorp./pm)
```

Alguns comandos utilizados no *script-file* são comentados a seguir:

- Inicialização: os utilizados limpam a tela e as variáveis (locais e globais)
- Entrada de dados: o comando *input* solicita que o usuário entre dados, que são armazenados em uma variável determinada.

Functions

As rotinas escritas como *functions* são também executadas a partir do *prompt* do MATLAB, e então criam um *workspace* (a partir de agora denominado *f-workspace*) próprio, onde apenas algumas (ou mesmo nenhuma) variáveis do *workspace* do MATLAB entram, onde todas as variáveis geradas são armazenadas e de onde apenas as variáveis desejadas retornam. A forma de execução das *functions* segue a seguinte sintaxe:

» ['saídas'] = funcao('entradas')
Deste modo, apenas as variáveis descritas no grupo 'entradas' entram no *f-workspace* e apenas as descritas em 'saídas' retornam ao *workspace* do MATLAB. funcao é o nome do *m-file* que contém a rotina que determina 'saídas' a partir de 'entradas'. Tanto um grupo quanto o outro podem ser qualquer tipo de variável do MATLAB (escalares, vetores, matrizes, estruturas, *strings*, *cell arrays*, ...).

A utilização de *functions* é bastante útil por permitir a execução da mesma rotina para vários conjuntos de entrada sem a necessidade de alterar o arquivo de texto e por não poluir o *workspace* do MATLAB com as variáveis utilizadas em cálculos intermediários da rotina.

A seguiu um exemplo de *function*, utilizando o mesmo algoritmo do exemplo anterior:

```
function [formula] = analise_f(comp,pmc)
% [formula] = analise_f(comp,pmc)
% Fórmula centesimal --> Fórmula molecular
% Esta function determina a formula centesimal
% de um composto orgânico a partir do vetor de
% composições percentuais comp=[C,H,O,N] e do
% peso molecular do composto (pmc)

% Pesos moleculares dos elementos
pm=[ 12.01 1.01 16 14.01 ];
% Execução
prorp=pmc*comp/100;
formula=round(prorp./pm);
```

Variáveis globais versus variáveis locais

A definição de variáveis locais vem exatamente da possibilidade de utilização de variáveis que estão presentes apenas no *workspace* de trabalho. São variáveis locais todas aquelas que podem ser utilizadas quando se trabalha em um *workspace*. Por exemplo, na execução de uma *function* podem ser utilizadas as variáveis utilizadas como argumentos de entrada e as variáveis criadas pela rotina, então estas são as variáveis locais do *f-workspace* em questão.

Existem variáveis que podem ser declaradas como sendo globais e, deste modo, podem ser utilizadas como variáveis locais em diferentes *workspaces*. Para declarar uma variável como global deve-se utilizar o comando

» global variavel
Para utilizar esta variável em um *f-workspace* o mesmo comando deve ser dado na *function* que o utiliza.

3. Comandos de Fluxo e Operadores Lógicos

Determinadas rotinas utilizam uma seqüência de comandos repetidas vezes para a resolução do algoritmo, como é o caso do primeiro exemplo mostrado nesta aula, o cálculo da pressão de vapor de um líquido utilizando uma equação de estado. Neste exemplo, de acordo com a resposta do bloco de decisão, a rotina deve ser encerrada ou os cálculos devem ser reiniciados, com um valor atualizado de um dos parâmetros (no caso a própria pressão de vapor). Para realizar este *loop* se utilizam os chamados comandos de fluxo, que realizam esta função. A decisão de qual caminho seguir é executada pelos denominados operadores lógicos, que indicam à rotina qual seqüências de comandos deve ser seguida de acordo com a avaliação de algum parâmetro.

O comando de fluxo mais geralmente utilizado nas rotinas do MATLAB é o laço *for* e os operadores lógicos mais comuns são o *if* e o *switch*. A seguir apresentamos uma breve descrição destes e do comando *while*, que opera ao mesmo tempo como comando de fluxo e operador lógico.

Laço FOR

O laço *for* executa uma seqüência de operações por um determinado número de vezes.

O comando *for* define o início do laço, determinando o intervalo em que uma variável vai ser avaliada. Esta variável pode ser usada em uma seqüência de operações, definidas pelas linhas de comando do *m-file* que ficam entre os comandos inicial e final do laço (*end*).

Para quebrar o laço em determinado ponto pode-se utilizar o comando *break* (normalmente utilizado após um operador lógico).

A seguir um exemplo que demostra a utilização do comando *for*:

```
» for i=1:4;
    A(i,1)=i;
    A(i,2)=i^2;
  end
```

```
» A
A =
    1    1
    2    4
    3    9
    4   16
```

Operador lógico IF

O operador *if* funciona como um condicional, avaliando uma operação relacional* e determinando uma seqüência de comandos a ser seguida, definida a partir do ponto onde a expressão avaliada pela operação relacional se torna verdadeira.

Várias expressões podem ser avaliadas para a definição da seqüência de comandos a ser seguida, para isso se utilizam os operadores *elseif* ou *else*. O comando *elseif* determina uma nova condição a ser avaliada, enquanto o comando *else* determina a seqüência de operações caso nenhuma das condições anteriores tenha sido satisfeita.

Para indicar o término de uma operação lógica *if* deve-se inserir o comando *end* à rotina.

A seguir um exemplo ilustrativo da utilização dos comandos *if, elseif, else*.

```
» for i=1:3;
    for j=1:3;
      if i == j
        B(i,j)=0;
      elseif i>j
        B(i,j)=-1;
      else
        B(i,j)=1;
      end
    end
  end
» B
B =
    0    1    1
   -1    0    1
   -1   -1    0
```

Operador while

O operador *while* combina as funções de operador lógico do *if* e de laço do *for*.

Em suma, o operador define um laço de modo que a seqüência de operações compreendida entre o início e o final do intervalo (determinada pelo *end*) seja realizada até que uma determinada condição seja satisfeita.

Operador lógico SWITCH

O operador *switch* trabalha de maneira muito parecida ao *if*, porém a sintaxe e o modo de implementação são diferentes. Com este comando pode-se avaliar apenas uma expressão por vez, porém ela é mais eficientemente avaliada para diferentes condições.

O comando *switch* inicia o operador, indicando a expressão a ser avaliada. Com o comando *case* determina-se qual seqüência de comandos deve ser seguida, sendo esta terminada com um novo *case*, com um *otherwise* (equivalente ao comando *else* do operador *if*) ou ainda com o terminador da operação de avaliação lógica *end*.

A seguir um exemplo, utilizando o resultado do anterior.

```
»  for i=1:3;
     for j=1:3;
       switch B(i,j)
       case 0
         C{i,j}='zero';
       case {1,-1}
         C(i,j)={'nonzero'};
       end
     end
   end

» C
C =
    'zero'       'nonzero'    'nonzero'
    'nonzero'    'zero'       'nonzero'
    'nonzero'    'nonzero'    'zero'
```

4. Vetorização

O MATLAB é um *software* desenvolvido para trabalhar com matrizes. Quando se está construindo uma rotina de cálculo e se deseja utilizar uma ferramenta de *loop*, onde o objetivo é calcular uma expressão diversas vezes, é muito mais eficiente trabalhar utilizando o conceito de operações matriciais ou vetoriais.

Para demonstrar o que queremos dizer, vamos apresentar uma rotina simples, implementada de dois modos, onde se evidencia a diferença de eficiência entre uma implementação utilizando *loops* sua versão vetorial.

	tic
	x = 0;
	for k = 1:1001
	  y(k) = log10(x);
	  x = x + .01;
	end
	toc
		tic
		x = 0:.01:10;
		y = log10(x);
		toc

Tempo computacional	0.1900	0.0600

Utilizando a forma vetorizada, para este exemplo, tem-se uma redução de 70 % no tempo computacional.

5. Utilização do DEBUGGER

O MATLAB possui um editor de textos que está configurado para trabalhar com sua linguagem de programação, o DEBUGGER.

Para rodar o editor, digite *edit* na linha de comando, um documento em branco será aberto. Case se deseje rodar o editor e abrir um documento de texto, basta digitar *edit* seguido do nome do arquivo.

As principais vantagens da utilização deste editor são a visualização gráfica dos textos digitados e a possibilidade de se trabalhar simultaneamente com o MATLAB e o editor.

* As possíveis operações relacionais são mostradas digitando-se *help relop* na linha de comando do MATLAB

Os recursos de visualização identificam cada tipo de função do MATLAB com uma cor diferente, além de possuir recursos (fracos) de auto-tabulação. O estilo dos textos gerados é o mesmo mostrado nos exemplos deste capítulo.

A possibilidade de utilização simultânea do editor com o MATLAB auxilia na procura de eventuais erros de sintaxe no programa, uma vez que permite a execução de rotinas linha a linha, colocação de pontos de quebra e execução de intervalos do programa entre outros.

São disponíveis ainda uma série de comandos de edição comuns, como 'localizar', 'voltar', 'copiar', ...

6. ExcelLink

Para facilitar o trabalho de fluxo de dados entre o MATLAB e programas do *MS-Office*, foi criado o *ExcelLink*, uma macro desenvolvida para o *MS-Excel* que permite o envio e recebimento de dados do MATLAB.

Executando o arquivo *excllink.xla*[*] o *MS-Excel* é executado, surgindo um conjunto de botões aparece na barra de ferramentas.

O MATLAB deve ser executado a partir do *Excel* para ativar o *link.*

Pode-se enviar dados para o MATLAB selecionando uma ou várias células e usando o botão p*utmatrix*, então aparecerá uma janela pedindo o nome da variável a ser armazenada com estes dados.

Para receber dados deve-se utilizar o botão *getmatrix* e informar a variável onde estão armazenados os dados que quer-se buscar.

É possível ainda utilizar funções do MATLAB no *Excel*. Para tanto, usa-se o botão *evalstring*. As variáveis de saída devem ser armazenadas no *workspace* do MATLAB, então deve-se atribuir um nome à variável para depois importá-la para o *Excel*.

[*] As macros desenvolvidas utilizam o MS Windows 95 ou MS Windows NT 4.0; MS Excel 97; e MATLAB 5.1 superior.

Aula 3 – Outras formas de manipular informações

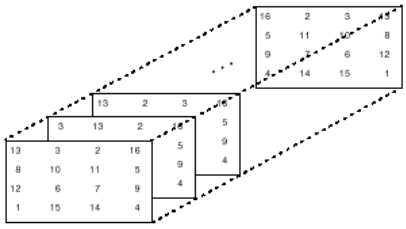
1. Outras formas de armazenar dados:

Além das formas de armazenar dados já abordadas, existem outras maneiras de guardar informações e acessá-las proporcionando ao programador uma grande versatilidade em termos de construção do rotinas.

A seguir serão apresentados três destas maneiras, *Multidimensional Arrays*, *Structures* e *Cell Arrays*, ou seja, Matrizes Multidimensionais, Estruturas e Listas. Cada uma destas formas possui uma sintaxe específica para sua construção e manipulação, permitindo assim a escolha da maneira mais adequada ao programa de guardar e manipular informações.

MULTIDIMENSIONAL ARRAYS

Matrizes multidimensionais são uma extensão do conceito de matrizes bidimensionais, que são acessadas utilizando apenas dois índices (linha, coluna); para o caso de uma matriz tridimensional, o número de índices necessários para localizar um elemento específico na matriz passa a ser igual a três. Desta forma, para uma matriz com dimensão igual a *n*, um elemento específico estará definido por *n* índices. A figura a seguir apresenta uma forma visual de interpretar matrizes multidimensionais.



Este tipo de matriz pode ser construída com os mesmos comando utilizados para matrizes bidimensionais, como *ones*, *rand*, e outros, porém agora com mais de dois argumentos que especificam as dimensões.

```
» rand(2,2,3)
ans(:,:,1) =
   0.5466   0.6946
   0.4449   0.6213
ans(:,:,2) =
   0.7948   0.5226
   0.9568   0.8801
ans(:,:,3) =
   0.1730   0.2714
   0.9797   0.2523
```

Uma matriz tridimensional, por exemplo, pode representar a temperatura em pontos de uma malha retangular em um espaço tridimensional.

A maneira pela qual as matrizes multidimensionais são manipuladas é equivalente às matrizes bidimensionais, porém agora deve-se utilizar um número de índices compatível com o número de dimensões para referenciar as posições desejadas na matriz.

CELL ARRAYS

As listas (ou disposição em células) são uma classe especial de matrizes, cujos elementos, ou células, contém matrizes do MATLAB, permitindo que matrizes de diferentes dimensões possam ser armazenadas em uma única variável. A sintaxe para criar uma lista (*Cell Array*) é a seguinte:

z = { A B C ... }

onde *A, B,C,etc.*, podem ser matrizes, outras listas, uma estrutura ou um *string*, por exemplo.

```
» a=1:3; b=[5 6;7 8]; c={a b}; d.a=a; d.b=b; z={a b c d};
» a
  a =
     1     2     3
» b
  b =
     5     6
     7     8
» c
  c =
    [1x3 double]    [2x2 double]
» d
  d =
    a: [1 2 3]
    b: [2x2 double]
» z
  z =
    [1x3 double]   [2x2 double]   {1x2 cell}   [1x1 struct]
```

Neste caso que foi apresentado, *a* e *b* são matrizes, *c* é uma lista com duas matrizes, *d* é uma estrutura com dois campos, um contendo a matriz *a* e outro a matriz *b*; enquanto que *z* também será uma lista, porém contendo duas matrizes, uma lista e uma estrutura.

Para manipulação dos elementos de uma lista também se utiliza as chaves, que serve para indicar em que célula se encontra determinado objeto.

```
» z{2}
  ans =
     5     6
     7     8
```

Desta forma, pode-se capturar a matriz *b* que está armazenada na lista *z*, mas se queremos capturar um elemento da matriz (p.ex.: o número 6), uma das possibilidades é a seguinte:

```
» z{2}(1,2)
  ans =
     6
```

Ou seja, vai especificando cada vez mais para poder capturar o elemento desejado, sempre seguindo a sintaxe referente objeto que se está trabalhando, por exemplo:

```
» z{3}{2}(1,2)
  ans =
     6
```

Este comando também retorna o número 6 da matriz *b*, porém neste caso a matriz *b* está dentro da lista *c* na

posição dois, que por sua vez ocupa a posição três da lista *z*.

Para criar uma lista de matrizes vazias pode-se utilizar o comando *cell*, que tem como argumento as dimensões desejadas para a lista.

STRUCTURES

As estruturas são uma classe de matrizes do MATLAB que permitem armazenar dados de naturezas diferentes em um mesmo lugar. As estruturas diferem das Listas por possuírem nomes que identificam seus campos. Uma forma de criar uma estrutura é a seguinte:

```
» dados.nome='fulano';
» dados.idade=25;
» dados.fones={[316 33 33] [316 44 44]};
» dados
  dados =
    nome: 'fulano'
    idade: 25
    fones: {[316 33 33]  [316 44 44]}
```

Como as estruturas são uma classe de matrizes, ainda pode-se adicionar elementos a uma mesma estrutura. Para tanto, deve-se proceder da seguinte maneira:

```
» dados(2).nome='ciclano';
» dados(2).idade=23;
» dados(2).fones={[999 88 88] [777 66 55]};
» dados
  dados =
    1x2 struct array with fields:
      nome
      idade
      fones
» dados(1)
  ans =
    nome: 'fulano'
    idade: 25
    fones: {[316 33 33]  [316 44 44]}
» dados(2)
  ans =
    nome: 'ciclano'
    idade: 23
    fones: {[999 88 88]  [777 66 55]}
```

Ou ainda,

```
» b(1,1).a=[35 36];
» b(1,2).a=[37 38];
» b(2,1).a=[39 40];
» b(2,2).a=[41 42];
» b
  b =
    2x2 struct array with fields:
      a
» b(1:2,1).a
  ans =
    35    36
  ans =
    39    40
```

Uma outra forma de criar uma estrutura é utilizando o comando *struct*, que toma como argumentos os nomes dos campos seguidos dos seus respectivos valores.

```
» e = struct('Disciplina','controle','Nota',9.5)
  e =
    Disciplina: 'controle'
      Nota: 9.5000
```

Aula 4 – Simulink

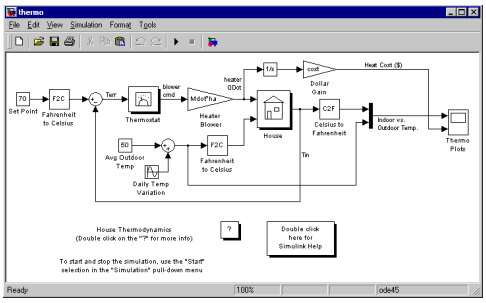
1. *Idéia Básica do funcionamento do Simulink*

SIMULINK é uma ferramenta interativa, baseada em estrutura de diagrama de blocos, voltada para a modelagem, análise e controle de sistemas. A interface gráfica do aplicativo permite que sejam criados modelos a patir de um conjunto de blocos já existente (*blocklibrary*) ou então criar blocos que executam funções editadas pelo usuário.

Para mostrar o potencial do aplicativo, vamos utilizar um exemplo.

Demonstração do potencial do aplicativo

No MATLAB, em todos os pontos, existem exemplos demonstrativos (*demos*) que mostram exemplos de utilização de cada tópico. Um destes *demos*, referente ao SIMULINK, é a simulação de um modelo termodinâmico de uma casa. O exemplo pode ser explorado digitando-se *thermo* no *prompt* do MATLAB. O diagrama de blocos do exemplo será mostrado na tela.



O exemplo modela a termodinâmica de uma casa utilizando um equacionamento simples. Para rodar a simulação, clique em ▶ na barra de ferramentas da janela do SIMULINK.

O termostato controla o funcionamento do aquecedor, que procura manter a temperatura da casa em 70 ºF (*setpoint*). A temperatura externa média é de 50 ºF, tendo variação modelada como um seno de amplitude 15 ºF e período de 24 horas.

O exemplo utiliza subsistemas para simplificar o modelo. Um subsistema é um bloco que representa um conjunto de blocos. Este exemplo contém cinco subsistemas, um representando o termostato (*Thermostat*), um representando a casa (*House*) e três convresores de temperatura, de ºF para ºC e vice-versa (*Fahrenheit to Celsius* e *Celsius to Fahrenheit*).

O subsistema *House* utiliza as temperaturas externa e interna com ação do aquecedor para atualizar a temperatura interna da casa. Um duplo clique no ícone revela a modelagem, mostrando os bolcos que compõem o subsistema.

O subsistema *Thermostat* modela o funcionamento do termostato, regulando as posições 'ligado' e 'desligado' do aquecedor de acordo com os limites de temperatura especificados no bloco *Relay*, que pode ser editado com um duplo clique no subsistema e no bloco em questão.

Os blocos de conversão de temperatura são subsistemas mascarados, que efetuam a operação matemática desejada. Para visualizar a implementação, seleciona-se o bloco e digita-se *Ctrl+u* (ver sob a máscara).

O custo de aquecimento é determinado integrando a taxa de aquecimento e multiplicando este por um fator de custo.

No bloco *Thermo Plots* são plotados o custo de aquecimento e as temperaturas interna e externa à casa.

Algumas modificações podem ser utilizadas para analisar o comportamento do sistema. Sugerimos as seguintes:

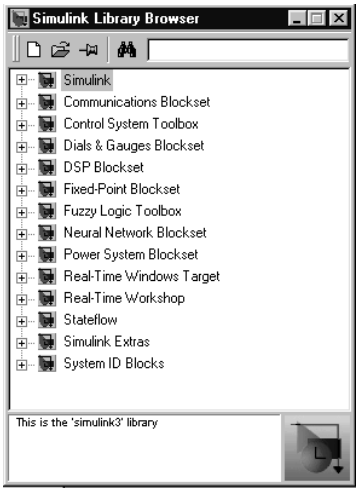
- Modificar o tempo de simulação (no menu: *Simulation* / *Parameters* / *Stop time*)
- Modificar o intervalo de visualização dos gráficos
- Mudar o *Setpoint* da temperatura da casa para, por exemplo, 80 ºF, analisando a resposta do sistema
- Alterar a temperatura externa média
- Alterar o modelo do distúrbio, modificando sua amplitude ou ainda o tipo de distúrbio

Construção de um modelo simples

Este exemplo demonstra como utilizar comandos e ações para construir modelos próprios.

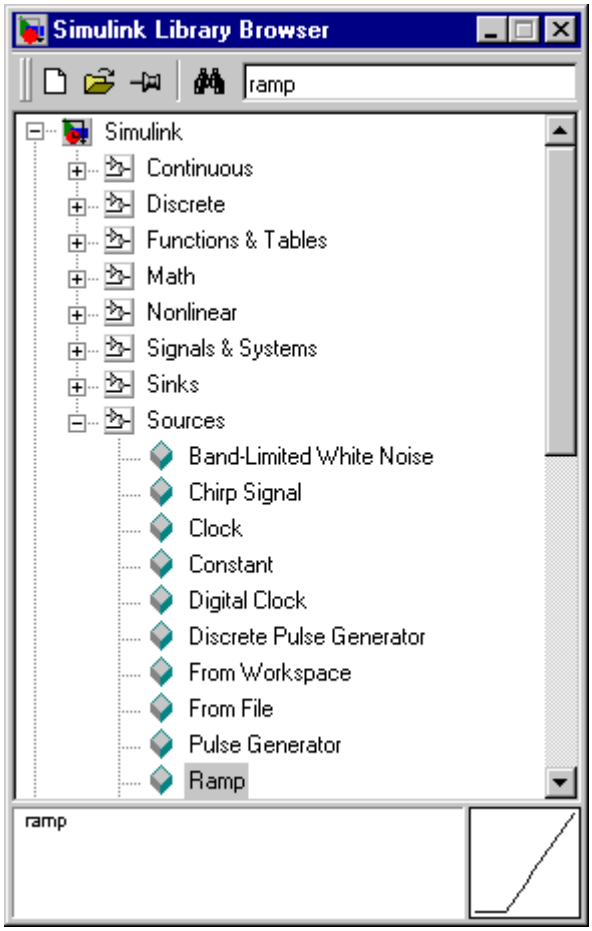
Vamos montar um modelo para simulação dinâmica da função x^2, de sua primeira derivada e de sua integral, mostrando na tela os resultados.

Antes de tudo deve-se iniciar o aplicativo, digitando *simulink* no *prompt* do MATLAB ou clicando no ícone  na barra de ferramentas. Deve surgir, então, o *library browser*, onde estão armazenados os blocos básicos do aplicativo.



Para iniciar um novo modelo, deve-se clicar no ícone  do *library browser*. Então deve surgir um documento em branco do SIMULINK, intitulado *Untitled*.

Agora o usuário deve buscar no *library browser* os blocos que deseja utilizar em seu modelo. Os blocos são agrupados de acordo com o *toolbox* ao qual estão vinculados (*Simulink*, *Comunications Blockset*, ...) e então divididos por tipos.



É possível procurar-se por um bloco informando o nome deste ao lado do ícone de procura (binóculo) e clicando nele. Outra forma é expandir os grupos com um duplo clique no item desejado (*Sources* ou *Simulink*, por exemplo) até encontrar o bloco que se quer. Na parte inferior da janela aparece o ícone do bloco marcado e breves informações sobre o mesmo (as mesmas informações são mostradas com um duplo clique no bloco quando este está na janela de trabalho).

Depois de encontrado o bloco, pode-se arrastá-lo para a janela de trabalho ou então utilizar o '*Ctrl+C* / *Ctrl+V*'.

Voltando ao nosso objetivo, a criação do modelo, devemos escolher os blocos a serem utilizados. Existem várias maneiras de implementar um mesmo modelo, de forma que o roteiro aqui sugerido não é único.

Queremos construir uma função polinomial, sua primeira derivada e sua integral, simulando-as e imprimindo seu resultado. Um modo simples para isto é utilizando o bloco *MATLAB Fcn* para a equação polinomial, os blocos *Derivative* e *Integrative* para calcular a derivada e a integral e o bloco *Slope* para plotar os gráficos. Para controlar o incremento na função pode-se usar o bloco *Ramp*. Todos os blocos podem ser encontrados com a ferramenta de procura, e então devem ser arrastados para a janela de trabalho.

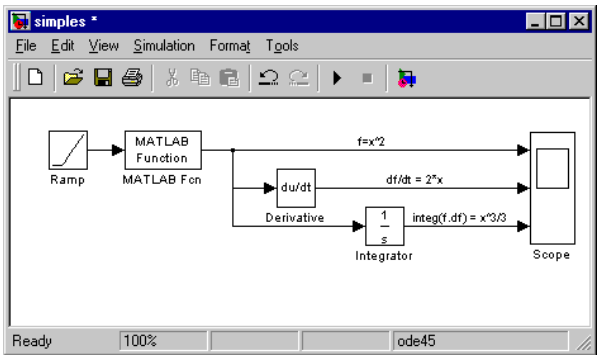
Uma vez que quer-se plotar 3 gráficos, pode-se copiar mais duas vezes o bloco *Slope* na janela de trabalho (usando '*Ctrl+C* / *Ctrl+V*') ou então alterando as propriedades do bloco. Para tanto clique duas vezes no bloco e então selecione  na barra de ferramentas. Uma janela, intitulada *'Scope' properties* aparecerá, e então o número de eixos pode ser modificado em *Number of axes*.

O bloco *MATLAB Fcn* deve ser editado colocando-se no campo *MATLAB function* a função que se deseja utilizar. No caso quer-se calcular o quadrado do valor de entrada (*u*), então pode-se colocar no campo a expressão *u^2*.

Para conectar dois blocos, deve-se arrastar a saída do primeiro até a entrada do segundo. As entradas e saídas têm a forma de setas (>). Pode-se ainda utilizar uma mesma linha (que carrega consigo um vetor ou um escalar) como entrada para outros blocos, para tanto deve-se, pressionando *Ctrl*, clicar na linha e então arrastar até o ponto desejado.

Pode-se nomear linhas ou então inserir comentários na janela de trabalho, bastando para tanto um duplo clique sobre a linha ou em qualquer espaço não ocupado.

Com isso, acreditamos que o leitor será capaz de montar o modelo desejado, como mostrado a seguir.



Pode-se agora testar o modelo rodando a simulação e alterando parâmetros como o polinômio, as funções de trabalho, o intervalo de simulação.

Para salvar o modelo, clicar em *Salvar como* e escolher o diretório. Os arquivos SIMULINK têm a extensão .mdl.

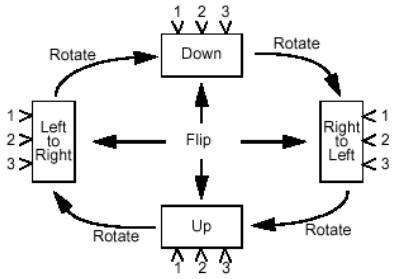
Blocos e operações interessantes

Um grande número de blocos está disponível na biblioteca do SIMULINK. Alguns destes, juntamente com algumas dicas de como melhor utilizar o aplicativo são colocadas a seguir:

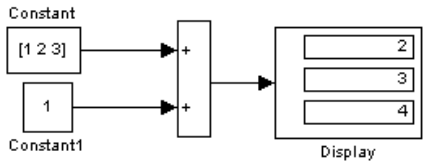
- Desenhar e/ou alterar malha gráfica auxiliar para a criação do modelo:
Para inserir uma malha auxiliar, digitar no path do MATLAB o comando
» set_param('<model name>','showgrid','on')
onde <model name> é o nome do modelo do SIMULINK.
Para alterar a malha, digitar:
»set_param('<model name>','gridspacing',<number of pixels>)

- Duplicando blocos em um modelo:
Para duplicar um bloco, selecione o bloco com o mouse e, pressionando Ctrl, leve a cópia do bloco até o local desejado.

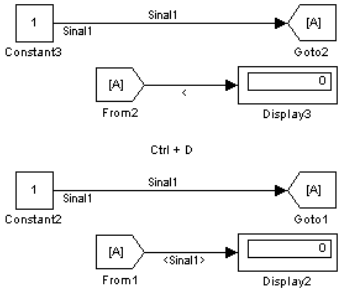
- Mudando a orientação de blocos:
Para rodar um bloco no SIMULINK pode-se usar os comandos do menu Format: Rotate block (roda 90 º) ou Flip block (roda 180 º). As teclas de atalho Ctrl+R e Ctrl+F também podem ser usadas. Para blocos com mais de uma entrada ou saída, a ordem destas segue o exemplo da figura a seguir.



- Vetores versus escalares
A maioria dos blocos do SIMULINK pode trabalhar tanto com vetores quanto com escalares. Quando, em blocos com mais de uma entrada, utilizam-se os dois tipos de expressão, o bloco faz uma combinação deles, como mostra o exemplo.



- Propagação de um sinal e nomes de correntes
Um sinal pode ser levado de um ponto a outro da janela de trabalho usando os blocos *Goto* e *From*. O nome de uma corrente também pode ser levada, evitando confusões, para tanto deve-se digitar o sinal 'menor que' no nome da corrente e então pressionar *Ctrl+D*.



- Criando subsistemas
A criação de subsistemas objetiva despoluir modelos que possuem um grande número de blocos. Em um subgrupo, como visto no exemplo da termodinâmica de uma casa, pode possuir várias entradas e saídas, de acordo com as entradas e saídas do subsistema. Pode-se adicionar um subsistema colocando-se o bloco *Sub System* na janela de trabalho ou selecionando-se um conjunto de blocos e, então, agrupando-os usando o comando *Create Subsystem* no menu *Edit* ou usando o atalho *Ctrl+G*.

2. Máscaras

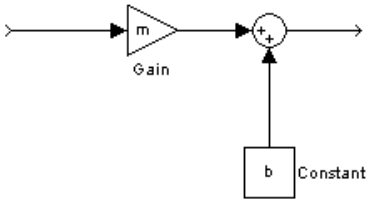
O uso de máscaras permite a criação de caixas de diálogo e ícones específicos para subsistemas. Algumas vantagens do uso de máscara são listados a seguir.

- Simplificação na utilização do modelo pela simplificação na entrada de parâmetro, uma vez que em um subsistema mascarado o usuário pode inserir todos os parâmetros de um conjunto de blocos em uma única caixa de diálogo.
- Permite a construção de uma interface própria para com o usuário, definindo-se a descrição, texto de ajuda e ícone do bloco.
- Previne a modificação acidental de subsistemas, uma vez que estes ficam protegidos atrás de uma interface.

Exemplo: criação de uma máscara para um sistema simples

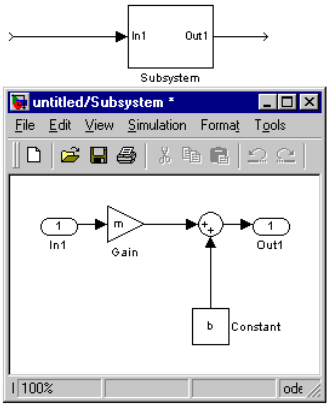
Para mostrar a utilização de máscaras em um modelo, vamos utilizar um exemplo simples: a criação de uma caixa de diálogo para um bloco que retorna a solução de uma equação de reta do tipo y = m.x + b.

Primeiramente é necessário montar o modelo, como já descrito. O modelo deve ser semelhante ao esquema a seguir:



Posteriormente deve-se criar um subsistema para este modelo, selecionando todos os blocos e clicando em *Create Subsystem* no menu *Edit* (ou utilizando o atalho *Ctrl+G*). O modelo será substituído por um único bloco. A

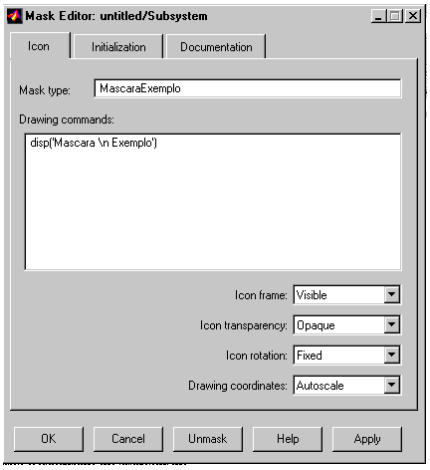
figura abaixo mostra o bloco gerado e o modelo, que pode ser visualizado com um duplo clique no primeiro.



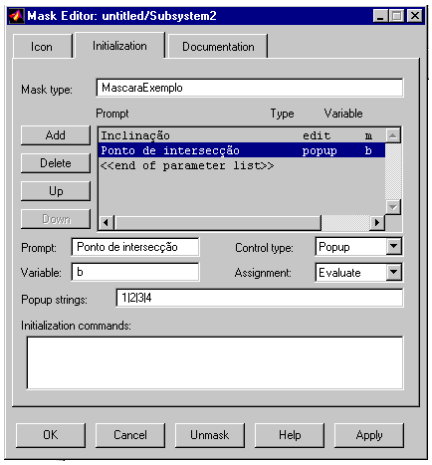
Os blocos *In1* e *Out1* representam a entrada e a saída do modelo, que coincidem com a entrada e a saída do bloco *Subsystem*.

Agora é possível mascarar o subsistema, para isso clique em *Mask Subsystem* no menu *Edit* (Ctrl+M). Abrirá então a janela de edição de máscaras, que permite a criação do ícone e da caixa de diálogo para o subsistema, podendo ser aberta usando novamente o atalho *Ctrl+M* com o sistema selecionado.

A primeira pasta do editor contém as opções de edição do ícone. No campo *Mask type* pode-se entrar uma expressão referente ao subsistema, como o nome. No campo *Drawing commands* define-se a aparência do bloco. Pode-se utilizar comandos gráficos (*line*, *plot*, ...) ou então texto (*disp*, *text*). Ainda se encontra nesta pasta opções de aparência do bloco, que podem ser configuradas de acordo com a aplicação. A seguir é mostrada uma sugestão de configuração para o subsistema estudado.

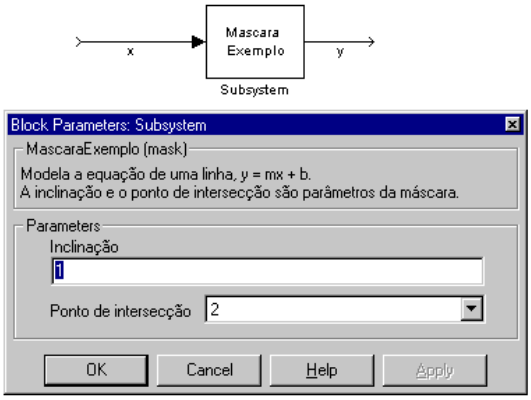


A segunda pasta contém os parâmetros de inicialização da máscara, onde a caixa de diálogo é definida. Para inserir um campo da caixa de diálogo, clicar em *Add*, então um novo campo será adicionado à lista. Este campo deve ser definido, escolhendo-se o tipo (editável, caixa de checagem ou lista de opções) em *Control type*, o nome e o símbolo da variável associada ao campo (*Prompt* e *Variable*) e a ação a ser tomada para o valor do campo (*Evaluate* calcula a variável e *Literal* a mantém como *string*). Para o exemplo y = m.x + b uma sugestão de janela, demonstrando o uso do menu *popup* é mostrado a seguir.



A última pasta contém informações sobre o documento. No campo *Block description* deve ser inserido um texto comentário que explica ao usuário o que o bloco faz e no campo *Block help* pode ser inserido um texto de ajuda, com informações mais detalhadas, uma vez que estas só serão exibidas a pedido do usuário.

A figura abaixo mostra o resultado obtido seguindo as instruções acima:

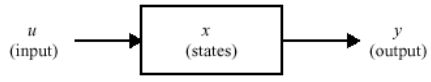


3. *S-functions*

As chamadas *S-functions* (*System Functions*) são descrições de sistemas dinâmicos, escritas em uma linguagem de programação própria (ou em C e compiladas como *MEX-files*), que interage com os *solvers* do SIMULINK. Seu uso mais comum está vinculado ao seguintes pontos:

- Criação de novos blocos para o SIMULINK
- Incorporação de códigos em C a uma simulação
- Descrição de um sistema como uma seqüência de equações matemáticas ao invés de utilizar a álgebra de blocos
- Utilização de animações gráficas (ver *penddemo*).

Todos blocos do SIMULINK trabalham com entradas (u), saídas (y) e elementos internos, ou estados (x). O esquema a seguir demonstra cada elemento do bloco:



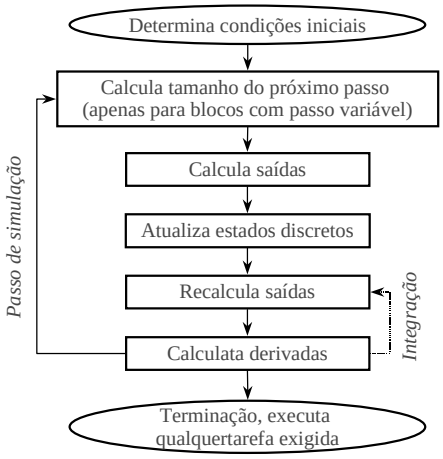
A relação matemática entre os elementos pode ser sintetizada por

$$y = f(t, x, u)$$
$$\partial x / \partial t = \dot{x} = g(t, x, u)$$

ou seja, os estados nada mais são do que as variáveis do modelo que são derivadas no tempo.

Nos arquivos de *S-functions* os vetores de estado são divididos em duas partes, uma primeira que contém apenas estados contínuos e uma segunda que contém estados discretos.

O SIMULINK faz várias chamadas de cada bloco do modelo durante fases específicas de simulação, executando tarefas como cálculo das saídas, atualização de estados, ou determinação de derivadas. Chamadas adicionais são feitas no princípio e término de uma simulação para executar a inicialização e a terminação. A figura a seguir ilustra como o SIMULINK executa uma simulação. Primeiro, o é realizada a inicialização de cada bloco, inclusive das *S-functions*. Então o SIMULINK entra no *loop* de simulação, onde cada passagem pelo *loop* é chamado um passo de simulação. Durante cada passo da simulação, o SIMULINK executa seu bloco *S-function*. Isto continua até o simulação esteja completa.



O SIMULINK informa às *S-functions* o estágio atual da simulação, ou seja, qual parte da rotina deve ser executada, através das *flags*. Deste modo cada parte da rotina deve ser iniciada com um controlador de fluxo (*if*, *case*) A tabela a seguir mostra cada estágio da simulação, a respectiva rotina na *S-function* e o valor da *flag*.

Estágio da simulação	Rotina da S-Function	Flag
Inicialização	mdlInitializeSizes	flag = 0
Cálculo do próximo passo (opcional)	mdlGetTimeOfNext VarHit	flag = 4
Cálculo das saídas	mdlOutputs	flag = 3
Atualização dos estados discretos	mdlUpdate	flag = 2
Cálculo das derivadas	mdlDerivatives	flag = 1
Término da simulação	mdlTerminate	flag = 9

Deste modo, quando a *flag* vale zero vai ser realizada a inicialização das variáveis. Neste ponto, a *S-function* deve retornar o tamanho dos vetores de entrada, saída e estados. Para isso deve conter a expressão

sys = ['nº de estados contínuos, nº de estados discretos', 'nº de saídas', 'nº de entradas', '*flag* para utilização da entrada para determinação do próximo passo', 'número de tempos de amostragem']

Normalmente, para sistemas contínuos, se utilizam apenas as posições 1, 3 e 4 da matriz. As demais posições são substituídas por zero.

Quando *flag* vale um devem ser informadas as equações diferenciais do modelo e quando *flag* vale 4 devem ser informadas as equações de saída da *S-function.*

Para melhor compreender o funcionamento das *S-functions* vamos utilizar dois exemplos simples.

Criação de uma S-function simples para integração do sinal de entrada

Vamos criar uma *S-function* que realiza a integração do sinal de entrada. Nesta simples implementação utilizamos o conceito de *flags* e a seqüência do fluxograma.

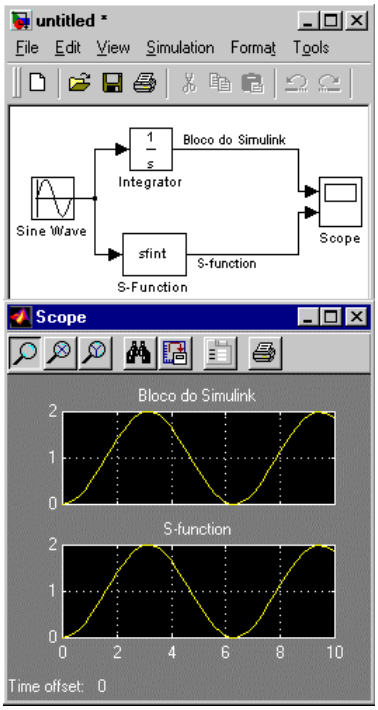
O modelo a ser construído, então, é

$$\dot{x} = \partial x / \partial t = u$$
$$y = x = \int \partial x / \partial t \, dt = \int u \, dt$$

Para escrever a *S-function*, então, devemos abrir o *Debugger* e digitar a rotina:

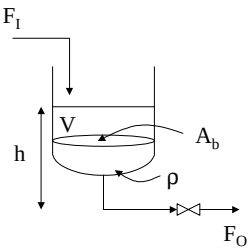
```
function [sys, x0] = sfint(t,x,u,flag)


if flag  == 0,
  x0 = 0;                  % Condições iniciais
  sys = [1 0 1 1 0 0]';    % Definição do
                           % tamanho dos vetores
elseif abs(flag) == 1,
  sys = u;                 % dx/dt = u
elseif flag == 3,
  sys = x;                 % y = x
else
  sys = [];                % Não utilizar
                           % demais flags
end
```

Para testar a rotina implementada, montar o seguinte modelo no SIMULINK. No campo *S-function name*, do bloco *S-function*, entre o nome do arquivo .m (*sfint*). O resultado obtido é também mostrado abaixo:



Criação de uma S-function simples para a modelagem não linear de um tanque

Deseja-se fazer a modelagem não linear de um tanque onde a entrada e saída de líquido são controladas pela ação da gravidade. Deseja-se analisar a variação do nível e da vazão de saída do tanque (resposta do sistema) frente a variações na alimentação (perturbação no sistema).



Considerando que o líquido possui massa específica constante, que o tanque é isotérmico e de mistura perfeita, o balanço mássico do tanque pode ser descrito por

$$F_O - F_I = \rho \frac{dV}{dt}$$

Modelando a hidráulica do sistema por $F_O = K\sqrt{h}$ e sabendo que $V = A_b h$, e equação diferencial do sistema pode ser reescrita como

$$\frac{dh}{dt} = \frac{F_I - K\sqrt{h}}{\rho A_b} \qquad h(t, F_I)$$

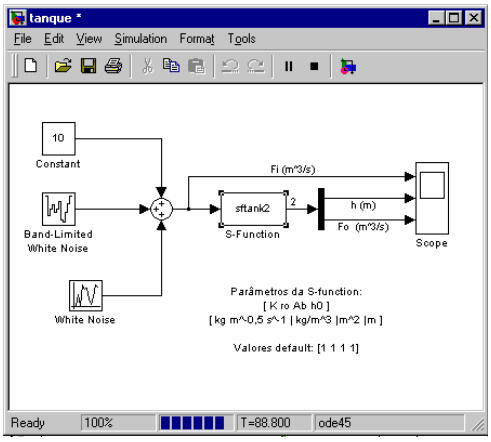
$$h(t_0) = h_0$$

Agora podemos implementar este modelo como *S-function*. Um modo de implementação é apresentado a seguir:

```
function [sys, x0] = sftank(t,x,u,flag,par)

if ~exist('par')
   par = [1 1 .2 100]; % Valores default
end

K = par(1);  %kg m^-0,5 s^-1
ro = par(2); %kg/m^3
A = par(3);  %m^2
h0 = par(4); %m

switch flag
case  0
   x0 = h0;              % Condições iniciais
   sys = [1 0 2 1 0 0]'; % Tamanho dos vetores
case 1
   sys = (u-K*sqrt(x))/(ro*A); % EDO do sist.
case 3
   sys(1) = x;         % h (nível do tanque)
   sys(2) = K*sqrt(x); % Fo (vazão de saída)
case {1, 2, 4, 9}
   sys = [];           % Não utilizar flags
end
```

A implementação sugerida utiliza os parâmetros (constantes) informados no próprio bloco da *S-function* ou então os valores padrão. Diferença entre este exemplo e o anterior é a utilização do controlador de fluxo *switch/case* para a atualização das *flags*.

Para testar a *S-function* criada, sugerimos a construção de um modelo do SIMULINK como o mostrado na figura a seguir, que possibilita a verificação do comportamento dinâmico do sistema frente a perturbações na alimentação.



Com o modelo construído pode-se simular o comportamento de um ou mais tanques com diferentes dimensões, condições iniciais e disposição (em série ou em paralelo).

Pode-se ainda criar uma máscara para o bloco, construindo um ícone próprio e uma caixa de diálogo para a entrada dos parâmetros.

4. Tabelas de auxílio

Nesta seção são mostrados os nomes e funções de alguns blocos do SIMULINK.

Table 1: Sources Library Blocks

Band Limited W hite Noise	Introduce white noise into a continuous system.
Chirp Signal	Generate a sine wave with increasing frequency.
Clock	Display and provide the simulation time.
Constant	Generate a constant value.

Digital Clock	Generate simulation time at the specified sampling interval.
Digital Pulse Generator	Generate pulses at regular intervals.
From File	Read data from a file.
FromWorkspace	Read datafrom amatrixdefined in the workspace.
Pulse Generator	Generate pulses at regular intervals.
Ramp	Generate a constantly increasing or decreasing signal.
Random Number	Generate normally distributed random numbers.
Repeating Sequence	Generate a repeatable arbitrary signal.
Signal Generator	Generate various waveforms.
Sine Wave	Generate a sine wave.
Step	Generate a step function.
Uniform Random Number	Generate uniformly distributed random numbers.

Table 2: Sinks Library Blocks

Display	Show the value of the input.
Scope	Display signals generated during a simulation.
Stop Simulation	Stop the simulation when the input is nonzero.
To File	Write data to a file.
To Workspace	Write data to a matrix in the workspace.
XY Graph	Display an XY plot of signals using a MATLAB figure window.

Table 3: Discrete Library Blocks

Discrete	Filter Implement IIR and FIR filters.
Discrete State-Space	Implement a discrete state-space system.
Discrete-Time Integrator	Perform discrete-time integration of asignal.
Discrete Transfer Fcn	Implement a discrete transfer function.
Discrete Zero-Pole	Implement a discrete transfer functionspecified in terms of poles and zeros.
First-Order Hold	Implement a first-order sample-and-hold.
Unit Delay	Delay a signal one sample period.
Zero-Order Hold	Implement zero-order hold of one sampleperiod.

Table 4: Continuous Library Blocks

Derivative	Output the time derivative of the input.
Integrator	Integrate a signal.
Memory	Output the block input from the previous time step.
State-Space	Implement a linear state-space system.
Transfer Fcn	Implement a linear transfer function.
Transport Delay	Delay the input by a given amount of time.
Variable Transport Delay	Delay the input by a variable amount of time.
Zero-Pole	Implement a transfer function specified in terms of poles and zeros.

Table 5: Math Library Blocks

Abs	Output the absolute value of the input.
Algebraic Constraint	Constrain the input signal to zero.
Combinatorial Logic	Implement a truth table.
Complex to Magnitude-Angle	Output the phase and magnitude of a complex input signal.
Complex to Real-Imag	Output the real and imaginary parts of a complex input signal.
Derivative	Output the time derivative of the input.
Dot Product	Generate the dot product.
Gain	Multiply block input.
Logical Operator	Perform the specified logical operation on the input.
Magnitude-Angle to Complex	Output a complex signal from magnitude and phase inputs.
Math Function	Perform a mathematical function.
Matrix Gain	Multiply the input by a matrix.
MinMax	Output the minimum or maximum input value.
Product	Generate the product or quotient of block inputs.
Real-Imag to Complex	Output a complex signal from real and imaginary inputs.
Relational Operator	Perform the specified relational operation on the input.
Rounding Function	Perform a rounding function.
Sign	Indicate the sign of the input.
Slider Gain	Vary a scalar gain using a slider.
Sum	Generate the sum of inputs.
Trigonometric Function	Perform a trigonometric function.

Table 6: Functions & Tables Library Blocks

Fcn	Apply a specified expression to the input.
Look-Up Table	Perform piecewise linear mapping of the input.
Look-Up Table (2-D)	Perform piecewise linear mapping of two inputs.
MATLAB Fcn	Apply a MATLAB function or expression to the input.
S-Function	Access an S-function.

Table 7: Nonlinear Library Blocks

Backlash	Model the behavior of a system with play.
Coulomb & Viscous Friction	Model discontinuity at zero, with linear gain elsewhere.
Dead Zone	Provide a region of zero output.
Manual Switch	Switch between two inputs.
Multiport Switch	Choose between block inputs.
Quantizer	Discretize input at a specified interval.
Rate Limiter	Limit the rate of change of a signal.
Relay	Switch output between two constants.
Saturation	Limit the range of a signal.
Switch	Switch between two inputs.

Table 8: Signals & Systems Library Blocks

Bus Selector	Output selected input signals.
Configurable	Represent any block selected from a

Subsystem	specified library.
Data Store Memory	Define a shared data store.
Data Store Read	Read data from a shared data store.
Data Store Write	Write data to ashareddatastore.
Data Type Conversion	Convert a signal to another data type.
Demux	Separate a vector signal into output signals.
Enable	Add an enabling port to a subsystem.
From	Accept input from a Goto block.
Goto	Pass block input to From blocks.
Goto Tag Visibility	Define the scope of a Goto block tag.
Ground	Ground an unconnected input port.
Hit Crossing	Detect crossing point.
IC	Set the initial value of a signal.
Inport	Create an input port for a subsystem or an external input.
Merge	Combine several input lines into a scalar line.
Model Info	Display revision control information in a model.
Mux	Combine several input lines into a vector line.
Outport	Create an output port for a subsystem or na external output.
Probe	Output an input signal's width, sample time, and/or signal type.
Subsystem	Represent a system within another system.
Terminator	Terminate an unconnected output port.
Trigger	Add a trigger port to a subsystem.
Width	Output the width of the input vector.

Aula 5 – Toolboxes & GUIDE

1. *Toolboxes*

A idéia desta seção é apresentar sucintamente algumas das funções consideradas como as mais utilizadas pelo estudante de graduação e pós-graduação no decorrer do curso e também as potencialidades desta grande ferramenta: o MATLAB. Portanto não é o objetivo desta seção entrar em maiores detalhes, os quais podem ser obtidos em manuais gigantescos que existem para cada um dos *toolboxes* que serão apresentados ou então uma ajuda razoável que pode ser obtida no *help on line*.

Symbolic Math

Este *toolbox* permite trabalhar com funções na forma simbólica, retornando uma solução analítica para os problemas. De um modo geral, existem duas formas de se trabalhar com expressões simbólicas; a primeira é utilizando *strings* enquanto que a outra é utilizando objetos simbólicos. Os objetos simbólicos são uma classe definida pelo MATLAB que permite sua manipulação através de funções específicas do *symbolic toolbox*.

A seguir são apresentadas algumas das funções mais utilizadas neste *toolbox*. Para mais detalhes sobre estas funções ou para procurar outras funções existentes, pode-se utilizar o *help on line* do MATLAB procurando por *symbolic*.

diff

Esta função calcula a derivada simbólica de uma função que está na forma de *string*.

```
» diff('sin(a)','a')
  ans =
    cos(a)
```

No caso do argumento desta função ser um vetor, ela retornará a diferença entre os elementos i_{+1} e i. E no caso de matrizes, pode-se definir a ordem (o) da diferença em se ela será aplicada sobre as linhas (1) ou colunas(2). A sintaxe neste caso é a seguinte:

```
» diff (X, o, 1 ou 2);
```

int

Calcula a integral de uma função. Esta função pode fazer diferentes tipos de cálculos de acordo com os argumentos de entrada:

int(S) calcula a integral definida de S em relação à variável definida pela função *findsym*. S é uma *string* ou uma variável *sym*.
int (S,v) calcula a integral indefinida de S e, relação a *string* v.
int (S,a,b) calcula a integral definida de S em relação a sua variável simbólica para o intervalo [a b].
int (S,v,a,b) calcula a integral definida de S em relação à v para o intervalo [a b].

limit

Calcula o limite de uma função. A sintaxe é a seguinte:

limit (F, x, a, 'right' ou 'left');
Esta expressão calcula o limite da função F (*string*) para x→ a pela direita ou pela esquerda.

Antes de executar esta função, deve-se transformar as variáveis envolvidas no cálculo em objetos simbólicos através da função *syms*.

```
» syms x
» limit((x-2)/(x^2-4),x,2)
   ans =
     1/4
```

solve

Resolve simbolicamente equações algébricas. As equações e as variáveis desconhecidas deverão ser expressões simbólicas ou *strings*.

```
» solve('p * sin(x) = r','x')
  ans =
    asin(r/p)


» [x,y] = solve('x^2 + x * y + y = 3','x^2 - 4*x + 3 = 0')
  x =
    [ 1]
    [ 3]
  y =
    [   1]
    [ -3/2]
```

dsolve

Calcula a solução simbólica de equações diferenciais ordinárias. As equações devem estar na forma de *strings* e a variável a ser diferenciada em relação a variável independente é antecedida pela letra *D*. No caso da ordem ser maior que um, utiliza-se a notação: *D2, D3,* etc. Por definição, a variável independente é *t*, mas pode ser alterada na chamada da função colocando a nova variável na última posição da chamada da função. As condições iniciais são especificados por equações do tipo: *'y(a)=b'* ou *'Dy(a)=b'* , onde *y* é uma das variáveis dependentes enquanto *a* e *b* são constantes. Se o número de condições iniciais for menor que o de variáveis dependentes, serão colocadas constantes arbitrárias (C1, C2,...) na solução final.

```
» dsolve('Dx = - a * x')
  ans =
    C1*exp(- a * t)

»  x = dsolve('Dx = - a * x', 'x(0) = 1','s')
  x =
    exp(- a * s)


» y = dsolve('(Dy)^2 + y^2 = 1','y(0) = 0')
  y =
    [  sin(t)]
[ - sin(t)]
```

```
»   S = dsolve('Df = f + g', 'Dg = -f + g', 'f(0) = 1','g(0) = 2')
  S =
    f: [1x1 sym]
    g: [1x1 sym]
» [S. f S. g]
  ans =
    [  exp(t)*(cos(t)+2*sin(t)), - exp(t)*(sin(t)-2*cos(t))]
```

laplace

Calcula a transformada de Laplace de uma função em relação a variável *t* (padrão). Se a função for dada em termos de *s*, então *laplace* retorna a sua inversa em *t*.

As variáveis utilizadas nesta função devem ser objetos simbólicos, portanto sempre devem ser declaras antes através do comando *syms*.

```
» syms a s t w x
» laplace(t^5)
  ans =
    120/s^6
» laplace(exp(a * s))
  ans =
    1/(t - a)
» laplace(sin(w * x),t)
  ans =
    w/(t^2+w^2)
» laplace(diff(sym('F(t)')))
  ans =
    s * laplace(F(t), t, s)-F(0)
```

Optimization Toolbox

Neste *toolbox* encontram-se funções para cálculos de otimização, ou seja, problemas minimização com ou sem restrições, minimização de funções multi-objetivos, método dos mínimos quadrados linear e não linear, cálculo de zeros, etc. Todos estes algoritmos possuem opções específicas inerentes aos métodos utilizados, a tolerância permitida, dentre outras. Para definir quais serão estas opções, o MATLAB utiliza uma estrutura que guarda todas as informações necessárias que serão utilizadas por uma função específica. Esta estrutura pode ser definida utilizando o comando *optimset*, que é apresentado em seguida.

Maiores detalhes sobre algumas das funções apresentadas ou para procurar outras funções, pode-se utilizar o *help on line* do MATLAB procurando por *optim*.

optimset

Cria uma estrutura com as opções que serão utilizadas pelas funções do *Optimization Toolbox*. A sua sintaxe é a seguinte:

options = optimset*(‘par1’,valor1,’par2’,valor2,...)*

Outra forma de definir os parâmetros é utilizar os parâmetros padrões utilizados pela função que se deseja utilizar. Para isso, deve-se entrar uma *string* com o nome da função como argumento para *optimset*.

```
» optimset ( ‘ fzero ‘ );
```

Para fazer uma cópia das opções antigas e alterando apenas algumas, deve-se proceder da seguinte maneira:

```
» new_opts = optimset(old_optos, ‘par1’,valor1);
```

A lista de todas as opções disponíveis pode ser encontrada no *help on line* procurando por *optimset*.

fzero

Este comando calcula os zeros de uma função escalar. Tem como argumentos básicos a função que se deseja determinar os zeros e um vetor com os intervalos da região onde será feita a busca ($sign(f(x_1)) \neq sign(f(x_2))$). Geralmente a função é um *M-file* criado especificamente para este problema, que deve pegar valores reais como argumento e retornar valores reais. Outra opção é utilizando o comando *inline*, que cria uma função objeto para ser utilizada pelo comando *fzero*.

```
» f=inline('x^2-6*x+5');
» fzero(f,[0 2])
  Zero found in the interval: [0, 2].
    ans =
      1
» fzero(f,7)
  Zero found in the interval: [4.76, 8.5839].
    ans =
      5
```

fsolve

Resolve equações não lineares do tipo: *F(X) = 0*, utilizando o método dos mínimos quadrados, onde *F* e *X* podem ser vetores ou matrizes. A função *F* deve ser implementada na forma de um *M-file*. A sintaxe deste comando é a seguinte:

X = fsolve(‘função.m’, X0, opções, P1, P2, ...);

onde *X0* é a matriz com os valores para inicializar o problema, *opções* é o conjunto de opções definidas com o *optimset* e *P1, P2,...* são os parâmetros que são passados para a função.

fminbnd

Resolve o problema de minimização de uma função escalar não linear com restrição. A sua sintaxe geral é a seguinte::

X = fminbnd(FUN, x_1, x_2, opções, P1, P2, ..)

onde *X* é o valor do mínimo local pertencente ao intervalo (x_1 , x_2) da função *FUN* que em geral é um *M-file*. A outra forma de resolver é utilizando o comando *inline*.

```
» f =inline('(x-2)^3+5*(x-2)^2+13');
» x = fminbnd(f,0,6)
  Optimization terminated successfully:
  the current x satisfies the termination criteria using
  OPTIONS.TolX of 1.000000e-004
  x =
    2.0000
```

lsqnonlin

Resolve problemas não lineares de mínimos quadrados com a seguinte forma:

min{sum[FUN(X).^2]}

onde *X* e os valores retornados pela função *FUN* podem ser vetores ou matrizes. A sintaxe geral é a seguinte:

X = lsqnonlin(FUN, X0,LB,UB, opções, P1, P2, ..)

onde *X0* é a matriz de inicialização, enquanto que *LB e UB* são os limites inferiores e superiores, respectivamente, das variáveis desejadas. A função *FUN* que em geral é um *M-file*, deve retornar um vetor com as funções objetivos. A outra forma de resolver é utilizando o comando ***inline.***

```
» x = lsqnonlin(inline('sin(x . * x)'),0.3,-0.5, 0.5)
  Optimization terminated successfully:
  Relative function value changing by less than
  OPTIONS.TolFun
    x =
      0.0139
```

Control System Toolbox

Este *toolbox* apresenta uma série de funções destinadas a facilitar quem trabalha na área de controle de processos. Os comandos que serão apresentados fazem parte do conhecimento da teoria de controle básica, que "deve estar no sangue" de quem um dia pretende passar pela disciplina de Controle de Processos ministrada pelo Prof. Jorge O. Trierweiler (UFRGS).

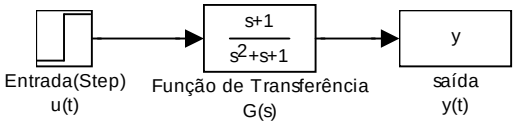
Antes de iniciar a apresentação dos comandos, convém dar uma breve introdução do que vem a ser uma função de transferência e alguns outros conceitos.

Uma função de transferência serve basicamente para avaliar uma entrada e dar o valor da saída correspondente aquela entrada. A sua origem vem da modelagem de um processo qualquer (linear ou não), o qual se deseja controlar. Como o a modelagem geralmente consiste de um conjunto de equações diferenciais lineares ordinárias em relação ao tempo, pode-se aplicar a transformada de Laplace, a qual reduz o problema de se resolver um sistema de equações diferencias a um problema algébrico relativamente simples. Na resolução deste problema é que as variáveis de interesse (entrada/saída) são relacionadas por uma função, a qual recebe o nome de função de transferência. Convém salientar que no caso do modelo ser não linear, pode-se, utilizando a expansão em série de Taylor, linearizar este modelo um ponto qualquer, geralmente o ponto de interesse é o correspondente ao estado estacionário do sistema, onde a planta é operada.

Uma função de transferência se resume basicamente a uma função racional de dois polinômios, aos quais é usual a designação *num*(numerador, cujas raízes eqüivalem aos zeros do sistema) e *den* (denominador, cujas raízes eqüivalem aos pólos do sistema, e que por sua vez estão relacionados com a dinâmica do processo). Portanto, uma função de transferência tem a seguinte aparência:

$$\frac{K(\tau_o S+1)}{(\tau_1\tau_2 S^2+1)+(\tau_1+\tau_2)S+1}$$

Os polinômios são colocados desta forma pois pode-se identificar mais facilmente as constantes de tempo τ bem como as raízes do denominador e do numerador (pólos e zeros, respectivamente). Em malha aberta, ou seja, sem estar controlando o sistema, a representação em diagrama de blocos é a seguinte:



A seguir serão apresentadas algumas funções do *control toolbox* que permitem uma análise rápida e clara das características principais de um sistema qualquer. A seqüência na qual elas são apresentadas, segue aproximadamente a ordem de uma análise real. Para maiores detalhes sobre algumas das funções apresentadas ou para procurar outras funções, pode-se utilizar o *help on line* do MATLAB procurando por *control*.

tf

Gera um objeto do modelo *LTI* que possui como uma das suas informações a função de transferência, tendo como argumentos os coeficientes do numerador e o denominador. Um objeto *LTI* é uma das formas de representação para modelos lineares invariantes no tempo, que pode ser manipulado pelas diversas ferramentas disponíveis no *control toolbox*. A sintaxe geral é a seguinte:

G = tf(NUM,DEN);

No caso de haver uma entrada e uma única saída (SISO), *NUM* e *DEN* são vetores com os coeficientes do polinômio do numerador e do denominador, respectivamente. Para o caso de mais de uma entrada e mais de uma saída (MIMO), *NUM* e *DEN* devem ser colocados em uma lista (*cell array*) em posições específicas determinadas pelas entradas e saídas que a função relaciona.

```
» num=[1 1];den=[1 1 1];
» G = tf(num, den)
  Transfer function:
       s + 1
    ---------------
    s^2 + s + 1
```

step

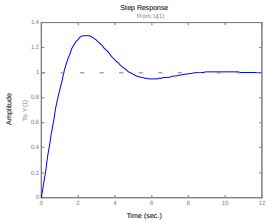
Esta função aplica um sinal degrau na entrada do sistema, que é representado por uma função de transferência, por exemplo, e então avalia a saída apresentando o resultado graficamente.

Para sistemas MIMO, esta função mantém aplica a função degrau em uma das entradas mantendo constante as demais, avaliando seu efeito nas saídas. Existem inúmeras opções que esta função permite utilizar, portanto é conveniente dar uma olhada no *help on line* do MATLAB, procurando por *step*.

Se fizermos:

```
» step(G);
ou
» step(num,den);
```

O resultado será o seguinte:



zpkdata

Este comando pega de um objeto *LTI* e retorna os valores dos zeros, pólos e ganho do sistema, que são informações úteis na análise de estabilidade do modelo e posterior construção do controlador.

```
» [z,p,k] = zpkdata(G)
  z =
    [-1]
  p =
    [2x1 double]
  k =
    1
»p{1}
  ans =
   -0.5000 + 0.8660i
   -0.5000 - 0.8660i
```

pole

Retorna os valores dos pólos do sistema. Toma como argumento um objeto *LTI*. Este comando eqüivale a calcular os autovalores da matriz A na representação no espaço de estado utilizando o comando *eig*.

```
» pole(G)
  ans =
   -0.5000 + 0.8660i
   -0.5000 - 0.8660i
```

zero

Retorna o valor do zero de um modelo *LTI*. Ainda pode retornar o valor do ganho se no vetor de saída tiver mais de duas variáveis.

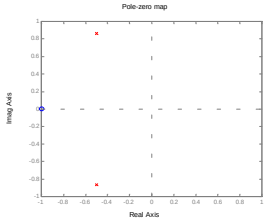
```
» [Z,K] = zero(G)
  Z =
   -1
  K =
   1
```

pzmap

Apresenta graficamente a localização dos zeros e dos pólos, permitindo uma análise visual sobre a estabilidade de um sistema, por exemplo. Este comando usa como argumento um objeto do modelo *LTI*.

```
» pzmap(G);
```

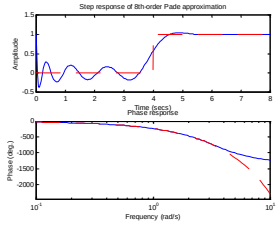
Resulta na seguinte figura:



pade

Faz a aproximação de Padè, muito utilizada para representar o tempo morto. Tem como argumento o valor do tempo morto e a ordem da aproximação, retornando uma função do tipo *NUM/DEN*. Quando executado sem os argumentos de retorno da esquerda, este comando compara a aproximação feita com a resposta exata do tempo morto de um sinal degrau e da defasagem. Para um tempo morto igual a 4, fazendo a aproximação de Padè de ordem 8, o resultado é o seguinte:

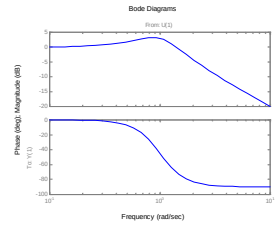
```
» pade(4,8);
```



bode

Apresenta o diagrama de Bode. Tem como argumento o objeto criado do modelo *LTI*. Neste gráfico é possível se obter muitas informações sobre o sistema referentes a sua estabilidade.

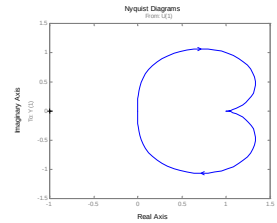
```
» bode(G)
```



nyquist

Faz o gráfico de Nyquist tomando como argumento o objeto criado do modelo *LTI*. A análise deste gráfico permite fazer inferências sobre a estabilidade do modelo em malha fechada a partir da observação da sua resposta em malha aberta.

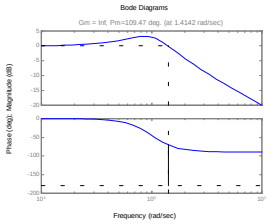
```
» nyquist(G);
```



margin

Este comando retorna a margem de ganho e a margem de fase de um sistema tendo como entrada o objeto gerado para o modelo *LTI*. Quando executados sem os argumentos de retorno dos valores à esquerda, ele apresenta graficamente como estes valores são obtidos através do diagrama de Bode.

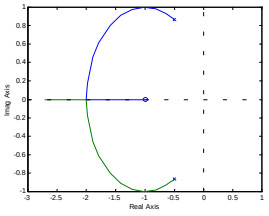
```
» [K,P]=margin(G)
    K =
      Inf
    P =
      109.4712
» [K,P]=margin(G)
```



rlocus

Retorna os valores e faz um gráfico com o lugar das raízes para um modelo SISO, tendo como argumento um objeto do modelo *LTI*. O gráfico dos lugares das raízes é utilizado para analisar o sistema em malha fechada, apresentando também as trajetórias dos pólos em malha fechada quando o ganho varia de zero a infinito.

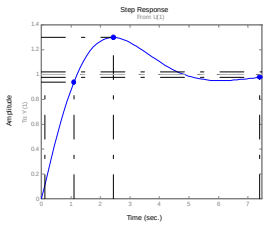
```
» rlocus(G);
```



ltiview

Abre o *LTI viewer*, o qual permite estudar as diversas análises apresentadas anteriormente em uma única janela. Ele utiliza como argumentos um modelo *LTI* criado previamente e o tipo de análise que se deseja realizar.

```
» ltiview('step',G);
```



Este gráfico foi obtido selecionand o estudo da resposta do sistema quando sugeito a uma entrada degrau, que pode ser selecionada clicando com o botão direito do *mouse* sobre a figura e escolhendo o tipo de gráfico (*plot type*). Pode-se ainda obter características do sistema, como tmpo de subida, estado estacionário, tempo de assentamento, etc. de forma interativa utilizando o *mouse*.

2. GUIDE

Esta é uma mais uma das ferramentas poderosas do MATLAB, que permite a construção de uma interface amigável para o usuário e o que é melhor, de uma forma amigável, ou seja, utilizando uma interface gráfica, o *Graphical User Interface* (*GUI*), para criar outras.

Conceitos Básicos

O *Handle Graphics* é uma estrutura orientada para objeto que permite a criação e manipulação de gráficos e imagens. Os *Object Handles* podem ser considerados como indicadores únicos atribuídos pelo MATLAB para cada objeto, permitindo desta maneira referenciar estes objetos para manipulá-los. Existem 3 formas de se obter um *handle*:

a) na criação do objeto

b) através de comandos utilitários

c) utilizando o *findobj*

Para o caso de se desejar obter o *handle* do objeto na sua criação (a), basta armazená-lo um uma variável, pois os *handles* são retornados automaticamente pelas suas funções de criação.

```
» p = plot([1:10],sin(2*[1:10]));
```

Neste caso o *handle* do gráfico da função seno ficará armazenado na variável *p*, portanto se desejar-mos fazer qualquer modificação no gráfico utilizando seu *handle*, basta utilizar *p*.

A segunda forma de se obter um *handle* é via comandos utilitários (b):

b.1) *gcf* retorna o *handle* da figura corrente

b.2) *gca* retorna o *handle* dos eixos correntes

b.3) *gco* retorna o *handle* do objeto corrente

b.4) *gcbf* obtém *handle* para *CallBack Figure*

b.5) *gcbo* obtém *handle* para *CallBack Object*

A terceira forma é através do comando *findobj* (c), que procura na figura o objeto através de propriedades específicas do objeto de interesse (como o *TAG* do objeto, por exemplo) e retorna o seu *handle*.

Este comando irá procurar através de toda a hierarquia de objetos na figura até encontrar o objeto que possua as propriedades especificadas, porém, esta busca pode ser mais rápida se também for informado o objeto a partir do qual se deve iniciar a busca. A sua sintaxe é a seguinte:

var_handle = findobj(ponto_início,propriedade,valor)

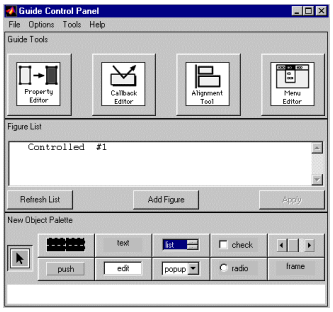
exemplo:

```
» plot([1:10]);
» h = findobj(gca, 'Color',[0 0 1]);
» set (h, 'Color', [1 0 0]);
```

Neste exemplo, o *handle* da linha do gráfico é identificado pela sua cor azul e armazenado em *h*, e em seguida, é atribuída a cor vermelha para a mesma.

Como visto no exemplo, uma das formas de manipular-mos objetos gráficos é através do comando *set*, enquanto que para pegarmos valores específicos de uma propriedade de um objeto, utiliza-se o comando *get*, em ambos os casos o *handle* funciona como "endereço" onde será feita a ação.

Uma maneira mais simples de trabalhar com os objetos gráficos é através do *Graphical User Interface* que é executado através do comando ***guide***.



Para entender melhor o seu funcionamento são apresentados alguns exemplos práticos para implementação de uma interface amigável.

Exemplos Simples de construção

Exemplo 1:

Apresentar a hora cada vez que um botão que apresenta a hora é pressionado.

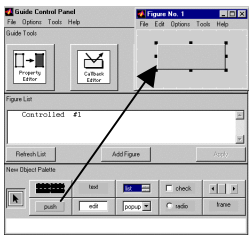
a) Abra o *Guide Control Panel* (*GCP*) digitando

```
» guide
```

b) Este comando já cria abre uma figura automaticamente caso não haja nenhuma figura aberta. No caso de já haver, gere uma nova figura, para tanto, basta digitar ***figure*** no *whorkspace* que uma figura nova se abrirá.

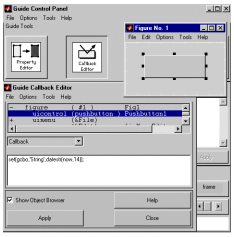
c) Se a figura ainda não estiver marcada como controlada pelo *GCP*, então selecione a figura desejada e pressione o botão *Apply* no *GCP*. Neste momento, a figura é apresentada com algumas marcas diferentes, permitindo a sua edição direta com o *mouse* adicionando, apagando, redimensionando e editando objetos.

d) Pressione o botão *push* no *GCP* e o coloque na figura clicando e dimensionando com o *mouse*.



e) Dê um duplo clic neste botão adicionado que se abrirá automaticamente a janela do *Property Editor*, que permite que você edite todas as propriedades de um objeto selecionado de uma dada figura. Procure pela propriedade *String* e a selecione. Depois de selecioná-la, escreva o texto que desejas que apareça no botão quando da execução do mesmo. Lembrando que o texto deverá estar entre aspas simples.

f) Ainda com o botão da figura selecionado, clique no botão *Callback Editor* do *GCP* e uma janela para edição de funções será aberta. Na sua porção editável, escreva o seguinte comando:
set(gcbo,'String',datestr(now,14));
Este comando fará com que toda vez que o botão da figura for pressionado, a hora atual seja apresentada no próprio botão, pois a sua propriedade *String* está agora recebendo o valor da hora atual.



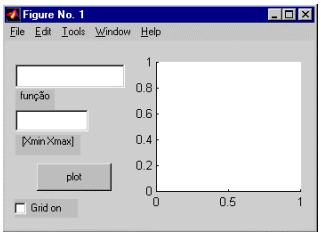
g) Após concluídas estas operações, salva-se a figura e no *GCP* torna a figura ativada da mesma maneira que ela foi tornada controlada anteriormente.

h) Para executá-la no MATLAB, basta digitar o seu nome no *workspace*.

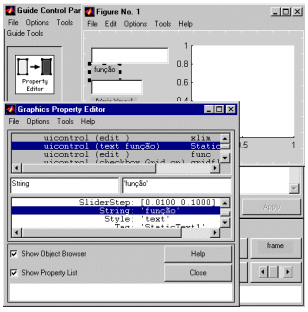
Exemplo 2:

Criar uma interface gráfica na qual pode-se entrar uma função e os limites da variável independente para gerar seu gráfico, permitindo a opção de apresentar ou não as linhas de grade.

a) Abrir uma figura e acrescentar um, um eixo para apresentação de um gráfico, duas caixa de texto, outras duas caixas editáveis e um *check box.*



b) Colocar os nomes de acordo com o esquema apresentado. Para tanto, basta selecionar o objeto, abrir o *Property Editor* e na propriedade *String* escrever os nomes correpondentes.



c) Abra o *Callback Editor* com o botão *plot* selecionado e digite as seguintes linhas de comando:

```
f=findobj(gcf,'style','edit','Tag','func');
xl=findobj(gcf,'style','edit','Tag','xlim');
xl =eval(get(xl,'String'));
f = get(f,'String');
fplot(f,xl);
```

d) A primeira linha de comando irá pegar o *handle* na caixa de texto editável que possui o *Tag func*, enquanto que a segunda irá atribuir a *xl* o *handle* da caixa de texto editável que possui o *Tag xlim*. Na linha seguinte, o valor que foi digitado nesta caixa, e que está armazenado na propriedade *String* é pego pelo comando *get* e em seguida convertido em um valor e armazenado em *xl*. A Quarta linha pega o valor da propriedade *String* da caixa de textos da função, que neste momento possui o identificador (*handle*) *f*, e armazena este valor na própria variável *f*. A última linha utiliza o comando *fplot* que faz um gráfico de uma função apenas tomando como argumento uma *string* da função e um vetor com os limites da variável independente.

e) Selecione o *check box* e no *Callback Editor* digite as seguintes linhas:

```
Hgrid=findobj(gcbf,'Tag','gridflag');
if get(Hgrid,'Value')
grid on;
else
  grid off;
end
```

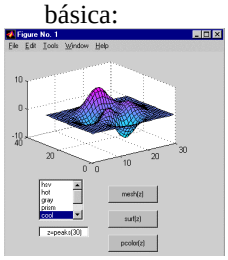
f) A primeira linha armazena em *Hgrid* o *handle* do objeto que possui como *Tag* o nome *gridflag*. Na Segunda linha, é avaliado se existe algum valor na propriedade deste objeto agora identificado por *Hgrid*, no caso afirmativo, *grid on* é executado, caso contrário é executado o *grid off*.

g) Salvar esta figura com um nome e ativá-la da mesma forma que no exemplo anterior.

Exemplo 3:

Criar uma interface gráfica que permita o usuário entrar uma função pré construída de três variáveis e esta seja apresentada de uma das diversas formas existente no MATLAB para gráficos 3D, e ainda com a opção de alterar suas cores.

a) Montar uma figura que tenha a seguinte estrutura básica:



b) Slecione os 3 botões *push* e no *Callback Editor* digite as seguintes linhas:

```
CommandString = get(gcbo,'String');
EditHandle = findobj(gcbf,'Tag','EditText1');
ZString = get(EditHandle,'String');
eval(ZString);
eval(CommandString);
```

A primeira linha pega a o que está na propriedade *String* do objeto corrente, enquanto que a Segunda linha armazena o *handle* do objeto que possui o *Tag EditText1*. A terceira linha pega o que está no campo *String* do objeto identificado na linha anterior e armazena em *ZString*, que é executado na linha seguinte e por fim, executa o comando que foi pego na primeira linha.

c) Para colocar elementos um abaixo do outro no *List box*, deve-se entrar no *workspace* um *cell array* com as opções desejadas. Neste caso, pode-se fazer o seguinte:

» cmaps={‘hsv’ ; ’hot’; ‘gray’; ‘cool’};

Com esta lista armazenada no *workspace*, seleciona-se o *List box* na figura e no *Property Editor* e no campo *String*, digita-se *cmaps* sem as aspas simples. Desta maneira, o *cell array* com as opções é armazenado em *string* para ser utilizado.

d) Ainda com o *List box* selecionado, escreva no Callback Editor as seguintes linhas de comando:

```
Value = get(gcbo,'Value');
String = get(gcbo,'String');
colormap(String{Value});
```

A primeira linha armazena em *Value* a posição referente às opções da lista, enquanto que na segunda linha é armazenado em *String* o *cell array* que está no campo *String* do objeto corrente. Na terceira linha é executado o comando *colormap* com a opção escolhida

e) Salvar a figura e torná-la ativa.

Lista de Exercícios

1. AULA 1

Comandos no Workspace

Arquivo: a1_ew

1. Criar uma matriz *A* 20x2 com os elementos da primeira coluna variando de 1 a 20, com um intervalo unitário, e dados aleatórios para os elementos da segunda coluna.
2. Atribuir ao vetor *X* os valores da primeira coluna da matriz *A* até a linha 15. Fazer o mesmo para a Segunda coluna, armazenando os dados no vetor Y.
3. Fazer um gráfico com os vetores (*X*,*Y*). Colocar um título e nomear os eixos.
4. Sobre o mesmo gráfico anterior, apresentar uma parábola (x^2 - 15x + 60) com linha tracejada vermelha.

---

2. AULA 2

Montar um script-file

Arquivo: a2_e1

O dióxido de titânio (TiO_2) é um pigmento branco muito utilizado na indústria de tintas.

Em uma planta há dois tanques contendo dióxido de titânio misturado com resina. Estes tanques possuem diferentes concentrações de pigmento:

Identificação do tanque	TQ-101	TQ-102
Concentração (% peso/peso)	15	5

Uma certa quantidade das misturas contidas em cada um destes tanques, será adicionada em um terceiro tanque (TQ-103) para a produção de tintas:

Tarefas:

a) Para auxiliar o departamento de produção deve ser preparado um gráfico que relacione a concentração de pigmento (% p/p) no tanque TQ-103 com a razão entre a quantidade de massa adicionada proveniente do tanque TQ-101 e a quantidade de massa adicionada proveniente do tanque TQ-102. Traçar o gráfico entre as razões 0 e 10.

b) Estimar através do gráfico, a razão entre as massas adicionadas para a produção de uma tinta com concentração de pigmento branco de 12 % p/p.

Montar uma função (m-file)

Arquivo: a2_e2

Em um laboratório de pesquisas são preparadas periodicamente soluções de sulfato de alumínio para pesquisas sobre o tratamento de água potável. Os pedidos de solução são realizados ao laboratório de preparo de reagentes em concentrações sempre variáveis.

Tarefas:

a) Desenvolver um programa para automatizar os cálculos necessários para o preparo das soluções. O programa deve receber a quantidade de solução necessária (ml) e a concentração desejada (molaridade), gerando a quantidade de sal que deve ser utilizada (g). As soluções são preparadas a partir de água destilada e sulfato de alumínio hexadecahidratado ($Al_2(SO_4)_3.16H_2O$).

Massas molares: `Al=26,98 S=32.06 O=16 H=1`

Exemplo de Programação utilizando mais de uma função

Arquivos: a2_e3, funcao

A determinação da solução de uma equação algébrica é um problema muito comum nos cálculos de engenharia. Muitas vezes a solução pode ser obtida através de técnicas simples de manipulação algébrica, no entanto, em certas equações, este procedimento não pode ser aplicado, ou seja, não é possível "isolar" a incógnita do problema. Para resolver este problema, foram desenvolvidos métodos numéricos que permitem determinar a raiz de uma equação a partir de um algoritmo adequado. O Método da Bissecção é um dos representantes mais simples desta classe de métodos. Este método se baseia na definição inicial de um intervalo de busca da raiz e à medida que o método se desenvolve, este intervalo é seqüencialmente reduzido até atingir a precisão desejada.

Descrição do Método da Bissecção:

O problema original é determinar a solução da equação, isto é, determinar x_0 tal que:

$$f(x_0) = 0 \qquad (1)$$

O Método da Bissecção se inicia com a determinação de um intervalo [a,b] onde a raiz desejada esteja situada. Para que o método funcione adequadamente, deve existir apenas uma raiz no intervalo [a,b] e a função f(x) deve ser contínua no intervalo.

Sejam xi e xs respectivamente o limite inferior e superior do intervalo de busca inicial. Então:

$$xi \leftarrow a \quad (2)$$

$$xs \leftarrow b \quad (3)$$

O primeiro passo do algoritmo é determinar o ponto médio do intervalo de busca [xi,xs] :

$$xm = \frac{xi + xs}{2} \quad (4)$$

São formados então dois intervalos [xi,xm] e [xm,xs]. No próximo passo deve-se comparar os sinais dos valores da função nos extremos destes intervalos. Esta comparação permitirá indicar em qual dos intervalos está a raiz desejada.

Em função do resultado desta comparação, o algoritmo pode seguir duas alternativas possíveis:

Alternativa 1:

f(xi) e f(xm) possuem o mesmo sinal e f(xm) e f(xs) possuem sinais contrários.

Isto significa que a raiz está no intervalo [xm,xs]. O intervalo de busca é então estreitado da seguinte forma,

$$xi \leftarrow xm \quad (5)$$

Alternativa 2:

f(xi) e f(xm) possuem sinais contrários e f(xm) e f(xs) possuem o mesmo sinal.

Isto significa que a raiz está no intervalo [xi,xm]. O intervalo de busca é então estreitado da seguinte forma,

$$xs \leftarrow xm \quad (6)$$

A motivação dos procedimentos associados às duas alternativas pode ser melhor compreendida através da visualização do gráfico da próxima figura:

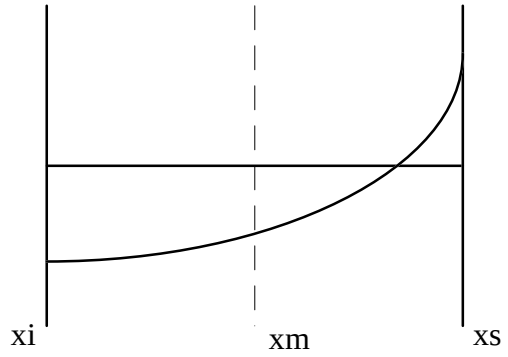


Figura - Procedimento de eliminação de intervalo

Observando o gráfico, torna-se óbvio que a raiz está no intervalo [xm,xs]. Neste intervalo a função corta o eixo y = 0, ou seja, neste intervalo a função muda de sinal e conseqüentemente f(xm) e f(xs) apresentam sinais contrários. O procedimento deve então seguir a alternativa 1, com a eliminação do intervalo [xi,xm], onde já se sabe, a raiz não está presente.

Após a eliminação do intervalo, segundo a alternativa 1 ou 2, o algoritmo deve recalcular o ponto médio e repetir a seqüência de procedimentos. É importante observar que à medida que o processo avança, o intervalo de busca vai se fechando em torno da raiz. Quando este intervalo torna-se menor que a tolerância desejada, o processo é interrompido e a solução é o ponto médio do último intervalo.

Observação:

É claro que se durante o algoritmo f(xm) = 0, o procedimento é imediatamente interrompido pois a raiz é o próprio valor de xm.

Tarefas:

a) Descrever o algoritmo do Método da Bissecção passo a passo, montando o fluxograma.

b) Criar uma rotina computacional para a aplicação do método.

c) Um trocador de calor é um equipamento destinado a permitir a transferência de calor entre dois fluidos. Em um determinado modelo de trocador de calor, água deve ser aquecida utilizando-se para isto uma corrente de óleo. A equação abaixo relaciona a carga térmica (taxa de transferência de calor entre os fluidos) com as temperaturas de entrada e saída do óleo e da água no equipamento:

$$Q - UA\frac{(Toleo_1 - Tagua_1) - (Toleo_2 - Tagua_2)}{ln(\frac{Toleo_1 - Tagua_1}{Toleo_2 - Tagua_2})} = 0$$

onde,

Q - Carga térmica do trocador, 160.10^3 W

U - Coeficiente global de transferência de calor, 320 W/m^{2}K

A - Área do trocador de calor, 16 m^2

$Toleo_1$ - Temperatura de entrada do óleo no trocador, 110 °C

$Toleo_2$ - Temperatura de saída do óleo no trocador, 80 °C

$Tagua_1$ - Temperatura de entrada da água no trocador, 35 °C

$Tagua_2$ - Temperatura de saída da água no trocador

c.1) Fazer o gráfico do valor numérico da equação em relação a temperatura de saída da água ($Tagua_2$). O gráfico deve conter o título e as identificações dos eixos. Verifique através do gráfico, se o Método da Bissecção pode ser aplicado para a resolução da equação.

c.2) Determinar a temperatura de saída da água do equipamento ($Tagua_2$), utilizando como região de busca inicial o intervalo 35,1 °C até 79,9 °C e tolerância de 0,1 °C.

---

3. AULA 3

Agenda de endereços usando estruturas

Arquivo: a3_e1

Montar uma lista de endereços simples armazenando os dados em uma estrutura. Este programa deve permitir a localização de telefones apenas dando o nome da pessoa.

---

4. AULA 4

Exemplo de S-function, máscaras e resolução de sist. De equações dif. no simulink

Arquivo: a4_e1.mdl, prexpre (s-function)

Devido ao fato de grande parte dos biorreatores operarem em regime de batelada ou de semi-batelada sua operação é inerentemente transiente, isto é todas suas variáveis [concentrações de células, de produtos(s) e de substrato(s); a temperatura (caso o biorreator for não isotérmico) mudam com o tempo. Os métodos de análise dinâmica de processos, geralmente utilizados em controle de processos, serão de extrema utilidade nestas apresentações, em especial serão apresentados conceitos iniciais de dinâmica não linear que justificam vários fenômenos inerentes a este tipo de processo. Como ilustração de modelos dinâmicos, o exemplo clássico é o modelo dinâmico da *presa-predador.*
O modelo de Lotka-Volterra, desenvolvido no final da década de 1920!] que é constituído por um sistema de duas equações diferenciais ordinárias fundamentadas em:

" o modelo descreve a interação entre duas espécies selvagens (sendo : N_1 o número de componentes da espécie 1 - *presa* - N_2 o número de componentes da espécie 2 - *predador*), a presa é herbívora e o predador carnívoro. Estes dois animais coabitam em uma região onde a presa tem um suprimento ilimitado de vegetação natural para sua alimentação e os predadores tem como exclusivo suprimento alimentar as indefesas presas. Originalmente o modelo foi desenvolvido por Lotka-Volterra para uma situação mais complexa e realista, isto é: mais de duas espécies coexistindo e onde os predadores alimentam-se de mais de uma espécie de presa. Este caso simplificado tem seu modelo matemático desenvolvido a partir das informações:
(a) Na ausência do predador, a presa tem uma *taxa natural* de nascimento b e uma *taxa natural* de falecimento d. Uma vez que o suprimento de alimentos para esta espécie é ilimitado, a taxa de nascimento é maior que a de falecimento [b > d] , portanto a *taxa específica de crescimento* de presa é positiva, isto é:

$$\frac{1}{N_1}\frac{dN_1}{dt} = b - d = \alpha > 0 ;$$

(b) Na presença do predador a presa é consumida a uma taxa específica proporcional ao número de predadores presentes, isto é:

$$\frac{1}{N_1}\frac{dN_1}{dt} = \alpha - \beta \cdot N_2 ;$$

(c) Na ausência da presa o predador tem uma taxa específica de crescimento negativa, uma vez que a inevitável conseqüência de tal situação é a inanição dos predadores, assim:

$$\frac{1}{N_2}\frac{dN_2}{dt} = -\gamma < 0 ;$$

(d) Na existência da presa, o predador tem o suprimento de alimento que lhe permite sobreviver e reproduzir a uma taxa específica proporcional a quantidade de presa, isto é:

$$\frac{1}{N_2}\frac{dN_2}{dt} = -\gamma + \delta \cdot N_1$$

Deste modo o modelo matemático deste problema é descrito pelas equações diferenciais ordinárias:

$$\begin{cases} \dfrac{dN_1}{dt} = \alpha \cdot N_1 - \beta \cdot N_1 \cdot N_2 \\ \dfrac{dN_1}{dt} = -\gamma \cdot N_2 + \delta \cdot N_1 \cdot N_2 \end{cases}$$

Além do conhecimento dos valores numéricos dos *parâmetros* do modelo (os valores de α, β, γ e δ), para se resolver este sistema de equações diferenciais ordinárias deve-se conhecer os valores de N_1 e N_2 em um determinado tempo (estas são as *condições iniciais* do problema).

Tarefa: Construir no SIMULINK (*S-function*) para o modelo de Lotka-Volterra e rodar simulações para várias condições iniciais utilizando os parâmetros:
α = 0,3/ano
β= 1/90 [1/(ano)/(número de espécies do predador];γ= 0,2106/ano
δ= 0,00026325[1/(ano)/(número de espécies de presa];

---

5. AULA 5

Utilizando o GUIDE

Arquivo: a5_e1

Tarefa:
Montar um programa que determine os coeficientes de um polinômio de grau *n* para um conjunto de dados experimentais. Este programa deve ter uma interface gráfica amigável.

Bibliografia recomendada

Manuais do MATLAB (podem ser encontrados na rede interna do DEQUI, no diretório:

p:\winapps\matlab2\hel\pdfdocs)

http://www.mathworks.com/

http://www.rpi.edu/~bequeb/Process_Dynamics/MATLAB/MATLAB_index.html

http://www.owlnet.rice.edu/~ceng303/Matlab/MatCont.html

http://www.indiana.edu/~statmath/smdoc/Matlab.html

http://www.math.utah.edu/lab/ms/matlab/matlab.html

http://www.math.ttu.edu/~gilliam/m5399-matlab.html

http://www.geodyn.psu.edu/GEOSC203/tutorial.html

http://www.math.ufl.edu/help/matlab-tutorial/

http://www.owlnet.rice.edu/~ceng301/toc.html