Álan Crístoffer e Sousa

acristoffers@gmail.com

Curso Básico de MATLAB

Divinópolis

2020

Sumário

Capítulo 1

Introdução

MATLAB (Matrix Laboratory) é um ambiente de análise iterativo e uma linguagem de programação, ambos proprietários, desenvolvidos pela *MathWorks*. O *software* foi feito de forma modular, podendo-se adicionar ou remover módulos, denominados *toolboxes*. É comumente associado à ferramenta *Simulink*, um ambiente/linguagem de desenvolvimento gráfico usado para simulações.

A linguagem do MATLAB foi feita para expressar bem problemas matemáticos, sobretudo matrizes. É uma linguagem feita pensando primeiramente na matemática computacional, não sendo boa em outras áreas, como o desenvolvimento de aplicativos ou manipulação de textos. É basicamente procedural, com um suporte fraco à orientação a objetos. É interpretada, mas possui um compilador *JIT* (*Just In Time*) que compila porções do código para agilizar a execução.

Atualmente há uma grande lista de *toolboxes* disponíveis, com áreas como otimização, modelagem física, simulação em tempo real, automotiva, espacial, controle, biologia e finanças. As *toolboxes* adicionam novas funções à linguagem, novos aplicativos (interfaces gráficas com algum propósito) ao ambiente e/ou novos blocos ao *Simulink*. Sem nenhuma *toolbox*, já se tem acesso à várias funções de álgebra linear e cálculo numérico, mas algumas *toolboxes* são necessárias para ter acesso a funções que facilitam o trabalho, como a *Control System Toolbox*, que possui funções para criar e manipular modelos em função de transferência e espaço de estados, bem como *plotar* suas respostas temporais à sinais de entrada.

1.1 Interface

A interface do *MATLAB* pode ser vista na Figura 1.

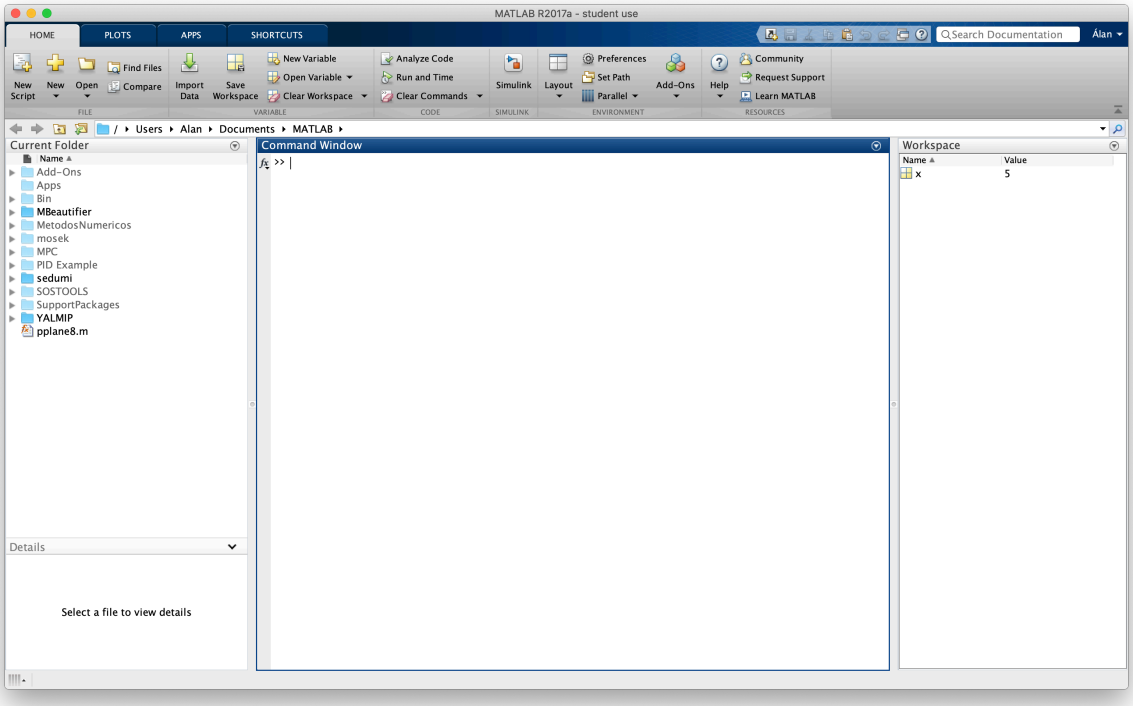

Figura 1 – Interface do MATLAB

Em *Command Window*, no centro, pode-se digitar os comandos que serão interpretados imediatamente. *Current Folder*, à esquerda, mostra os arquivos no diretório atual. Assim como linguagens de *shell*, o *MATLAB* mantém um diretório atual, que pode ser alterado através do comando `cd`. O comando `ls` lista os arquivos e diretórios no diretório atual e o comando `pwd` retorna o caminho completo do diretório atual.

À direita, em *Workspace*, temos uma lista de variáveis globais que foram definidas, bem como seus valores, caso seja possível exibi-lo. Clicando duas vezes em uma variável, uma aba é aberta que permite visualizar e, em alguns casos, editar os valores, como no caso de matrizes.

O painel na parte superior mostra várias ações. Caso sua interface não esteja organizada como acima, clique em *Layout* e depois em *Default* para retornar para o formato padrão.

O *MATLAB* também possui um editor de código integrado, que pode ser acessado

clicando duas vezes no arquivo que se deseja editar ou usando o comando `edit`, como `edit test`. Note que qualquer comando definido pode ser passado como argumento para o comando edit e, se este for um arquivo .m, será aberto. O editor também pode ser aberto clicando em *New* e depois em *Script* ou *Live Script*. A diferença é que *Live Script* mostra os resultados na janela do editor, à direita do código. Recomendo usar esse editor.

1.2 Sintaxe

A sintaxe da linguagem é simples e funciona como uma linguagem de *script*, como a *BASH* ou *PowerShell*. Todo arquivo de texto com extensão .m é considerado um comando, a não ser que defina uma função ou classe, e pode ser executado como se fosse uma função sem argumentos. Assim como nas linguagens citadas, os comando são buscados nos diretórios listados na variável especial *PATH*, que pode ser acessada pela interface do aplicativo (*Set Path*, na aba *Home*, seção *Environment*).

A linguagem é dinâmicamente tipada, significando que não é necessário definir o tipo das variáveis. Para definir uma variável basta digitar

```
x = 5
```

no *Workspace* e precionar *enter*. Note que o o nome da variável e seu valor são escritos na tela. Isso ocorre pois toda linha não terminada com o caractére *;* tem seu resultado escrito na tela. Isso ocorre mesmo dentro de funções e arquivos de *script*. Para silenciar o interpretador, basta digitar

```
x = 5;
```

que nada será escrito. Isso é útil para manter o console limpo, bem como para não gerar gargalos no código, já que escrever o resultado na tela tem um custo computacional elevado que se torna significativo dentro de *loops*.

A sintaxe para declarar matrizes e vetores é simples:

```
x = [1 2; 3 4]
```

Esse comando irá criar uma matriz 2×2 com 1 e 2 na primeira linha e 3 e 4 na segunda. Repare que os elementos de cada linha são separados por espaços (“1 2” e “3 4”) e as linhas são separadas utilizando *;*. Os elementos de uma linha também podem ser separados por vírgula, mas isso não é comum. Matrizes de qualquer dimensão podem ser

definidas dessa forma, e o interpretador interromperá a execução caso a matriz informada não tenha o numero correto de elementos, como neste caso:

```
x = [1 2; 3]
```

Para criar essa matriz:

$$\begin{bmatrix} -1 & 2 & 5 \\ 6 & -8 & 9 \end{bmatrix}$$

digitamos:

```
[−1 2 5; 6 −8 9]
```

Algumas funções também estão disponíveis para facilitar a criação de matrizes, como `zeros`, `ones`, `eye` e `diag`.

O comando `help` mostra a documentação de uma função, basta chamar como:

```
help zeros
```

1.3 Palavras Chaves

Palavras chaves são nomes reservados pela linguagem e não podem ser usados como nome de variáveis, funções ou qualquer tipo de identificador. A Tabela 1 contém uma lista de palavras chaves.

Tabela 1 – Palavras Chaves

break	continue	for	otherwise	spmd
case	else	function	parfor	switch
catch	elseif	global	persistent	try
classdef	end	if	return	while

1.4 Identificadores

Identificadores são nomes dados a variáveis, classes, funções, métodos, etc.

- Eles podem conter letras minúsculas e maiúsculas, números e _.
- O primeiro caractére não pode ser um número.

- Palavras chaves não podem ser usadas como identificador.
- O limite no tamanho de um identificador pode ser obtido executando o comando `namelengthmax`.

Exemplos de identificadores válidos: MyClass, var_1, uma_funcao
Exemplos de identificadores inválidos: 1var, $myvar, user@host, ação

1.5 Indentação

A identação é irrelevante em *MATLAB*, mas é uma boa prática de programação manter o código devidamente identado. Versões mais novas possuem no editor uma função para identar automaticamente o código, acessada através do menu que se abre ao clicar com o botão direito no editor (opção *Smart Ident*).

A linguagem executa comandos linha a linha. Para quebrar uma linha muito longa, utiliza-se `...`.

```
x = [1 2 3; ...
     4 5 6  ...
    ]
```

1.6 Comentários

Comentários em *MATLAB* são feitos utilizando o símbolo *%*. Comentários em bloco utilizam o par *%{* e *%}*.

```
% Será executado:
x = 6;

% Não será executado:
% x = 5;

% Também não serão executados:
%{
x = 5;
y = 6;
x + y
%}
```

1.7 IO

Usados para se comunicar com o usuário ou outros programas. Apenas a comunicação com o usuário e os arquivos *MAT* serão mostrados. Note no entanto que é uma prática ruim escrever scripts que pedem o usuário para digitar valores de variáveis o tempo todo. É melhor definir as variávis no início do script com os valores necessários, já que isso acelera o processo de testar e alterar o código.

1.7.1 Console IO

O *MATLAB* fornece duas funções simples para interagir com o console: **input** e **disp**.

```
% você deve digitar aspas simples ao redor do seu nome
nome = input('Digite seu nome: ')
disp(['Seu nome é ' nome])

% aspas não são necessárias para números
idade = input('Digite sua idade: ')
disp(['Seu idade é ' int2str(idade)])

% qualquer código MATLAB válido pode ser digitado na função input.
% o código será executado e o valor retornado pela função.
```

1.7.2 File IO

O *MATLAB* possui um formato de arquivo próprio que permite salvar variáveis no disco e compartilhá-las. Isso pode ser feito através das funções `save` e `load`.

```
x = 5;
y = 3;

% cria um arquivo data.mat no diretório atual com todas as variáveis salvas
save data

% cria um arquivo data.mat no diretório atual com apenas a variável x
save('data', 'x')

x = 2;

% carrega as variáveis do arquivo data.mat; x volta a ser 5, valor que tinha
% quando o arquivo foi criado.
load data
```

1.8 Bibliotecas/Toolboxes

São apenas diretórios contendo arquivos .m. Qualquer arquivo disponível no *PATH* pode ser acessado. Não há uma forma padrão de distribuir bibliotecas, bastando baixá-las e adicioná-las ao path.

1.9 Operadores

Operadores são símbolos que representam operações, normalmente aritiméticas ou lógicas. Os seguintes símbolos representam operadores:

- Aritiméticos: `+,-,*,/,^,.*,./,.^`
- Comparação: `>,<,==, =,>=,<=`
- Lógicos: `&, |, ,~&&, ||`

1.10 Chamada de funções

O *MATLAB* não exige o uso de parênteses em todas as chamadas de funções. Dessa forma, as duas linhas são identicas:

```
clear
clear()
```

No entanto, os parênteses só são opcionais quando não há argumentos, ou quando o argumento é do tipo `char`. Para o segundo caso, há uma particularidade, já que tudo que for digitado após o nome da função será considerado texto:

```
display oi
% equivalente a
display('oi')

display hello world
% equivalente a
display('hello', 'world')

display 'hello world'
% equivalente a
display('hello world')
```

Capítulo 2

Variáveis

Variáveis são nomes que representam dados na memória do programa. Imagine uma tabela. Na célula da 3ª linha e 5ª coluna tem o valor 42. Observe o tanto de informação que eu tive que te dar para te falar o valor de uma célula. E se eu quiser o mudar valor agora? Vou ter que repetir todas as informações que localizam a célula. E se eu disser que essa célula tem um nome? Vamos chamá-la de "total". `total = 42`. Algumas modificações foram feitas, agora `total = 3`. Muito mais simples, não? "total" é uma variável e a tabela a memória do computador.

2.1 Tipagem

Em algumas linguagens variáveis tem tipo. *MATLAB*, por ser **dinâmicamente** tipada, não atribui um tipo à variável. No entanto o objeto referenciado por esta variável tem um tipo. No caso da variável "total" acima, o objeto referenciado é o número 3. 3 é do tipo `double`. Mesmo a variável não tendo um tipo, é comum dizermos que ela tem o mesmo tipo que o objeto que ela referencia. Neste caso "total" é um `double`.

Mas como a tipagem é dinâmica as variáveis podem receber outros valores, de outros tipos. Por exemplo, `total = 'zero'`. Agora "total" referencia um `char`. Podemos dizer que agora "total" é um `char`. Em linguagens como *C* e *Java*, que são estaticamente tipadas, isso não seria possível, já que não é permitido mudar o tipo da variável após sua declaração.

Outra característica das variáveis no *MATLAB* é que elas são **fracamente** tipadas. Isto quer dizer que a linguagem **vai** fazer suposições quanto a possíveis mudanças de tipo.

Execute `'1'+ 1` e veja o resultado. 50? O que aconteceu? Ao invés de retornar um erro por tipos incompatíveis, o interpretador primeiro transformou `'1'` em um `double`, utilizando a tabela *ASCII*, onde o caractére 1 é a entrada número 49 e depois somou. Em uma linguagem fortemente tipada um erro seria reportado.

2.2 Escopo

O *MATLAB* possui basicamente dois escopos: global e local. Diferente de outras linguagens, por padrão, tudo está no escopo global. Apenas funções criam escopos locais, e necessitam da palavra-chave `global` para acessar variáveis no escopo global. Como apenas funções criam um novo escopo, variáveis declaradas dentro de blocos como `if`, `for` e `while` são globais e podem ser acessadas de fora do bloco.

```
% escopo global
a = 1;

for i=1:10
    x=i;
end

% x está definida no escopo global e pode ter seu valor acessado normalmente.
x
```

```
% esse bloco de código deve estar em um arquivo chamado exemplo_de_escopo.m
function exemplo_de_escopo()

% apenas a variável x no escopo global pode ser acessada.
global x

% cria uma variável a no escopo local, e não afeta a variável a no escopo global
a = 3
```

2.3 Ciclo de vida

Ciclo de vida determina onde uma variável sai de escopo e tem seu valor removido da memória. A memória é liberada em pelo menos dois momentos: Ao utilizar a função `clear` para limpar o escopo atual, e quando uma função retorna. O *MATLAB* usa a técnica de contagem de referência para remover valores do tipo ponteiro (chamados de

handle). Variáveis normais são removidas imediatamente ao sair de escopo, ou quando são manualmente limpadas.

Há um tipo especial de variável no *MATLAB* que só pode ser usado em funções. São variáveis persistentes. Elas são como variáveis de estado da função, que não são apagadas entre execuções. Elas apenas são removidas da memória quando a função é removida.

```
function [y] = fibseq()
    persistent a;
    if isempty(a)
        a = [1 1];
    end
    y = sum(a);
    a = [a(2) y];
end
```

Cada vez que a função `fibseq` for chamada, retornará o próximo número da sequência de Fibonacci. Para retornar ao início da sequência, a função deve ser removida da memória por comandos como `clear all`.

2.4 Passagem Por Referência

Passagem por referência ou por valor diz respeito a o que o interpretador fará quando uma variável for passada como argumento para uma função. Existem duas opções: criar uma cópia do valor e passar esta cópia para a função ou criar uma referência para o mesmo espaço de memória e passar esta referência para a função.

O *MATLAB* sempre passa por referência. No entanto, vale notar que todo valor no *MATLAB* sabe quando foi alterado. Assim, diferente de outras linguagens com passagem por referência pura, alterar um elemento de uma matriz cria uma cópia da matriz com o valor alterado, ao invés de alterar o valor original.

```
% Pouco menos de 1GB de memória
a = magic(1e4);
% Não ocupa mais memória, ela é compartilhada
b = a;
% Cria uma cópia de a, agora temos pouco mais de 2GB de memória sendo usados
% Em outra linguagem isso não causaria uma cópia e a(1,1) também seria alterado
b(1,1) = 1;
% Vale ressaltar que isso não irá gerar uma cópia, mas alterar o valor na
% memória, já que a não está compartilhando a memória com ninguém mais.
a(1,1) = 2
```

Capítulo 3

Estruturas de Dados

Ser uma linguagem desenvolvida primeiramente para matemática faz com que o *MATLAB* não ofereça as melhores opções em estrutras de dados. Ele possui várias estruturas, porém a criação e manipulação das mesmas não é tão simples quanto poderia ser.

3.1 Números

3.1.1 Tipos Numéricos

O tipo numérico padrão do *MATLAB* é o `double`. Todos os tipos numéricos são tratados de forma muito diferente dos seus análogos em outras linguagens, como resultado da implementação pensando na matemática. Todo tipo numérico no *MATLAB* é na verdade uma matriz. Um escalar é apenas uma matriz 1×1. Outros tipos presentes são[1] (todos possuem um dual complexo, por exemplo, **complex double**):

Tabela 2 – Tipos numéricos

double	Double-precision arrays
single	Single-precision arrays
int8	8-bit signed integer arrays
int16	16-bit signed integer arrays
int32	32-bit signed integer arrays
int64	64-bit signed integer arrays
uint8	8-bit unsigned integer arrays
uint16	16-bit unsigned integer arrays
uint32	32-bit unsigned integer arrays
uint64	64-bit unsigned integer arrays

[1] `https://www.mathworks.com/help/matlab/numeric-types.html`

Duas notações especiais são interessantes de serem ressaltadas.

1. **j**: a letra *j* representa a parte imaginária de números complexos. Deve ser adicionada imediatamente após o número. Sozinha representa uma variável. Por exemplo: `1+5j; 3j; 1+1j`. `1+j` é a some de 1 e a variável *j*.

2. **e**: a letra *e* representa uma potência de 10, e deve ser colocada entre números. $1 \cdot 10^{-16}$ é escrito como `1e-16`.

3.1.2 Conversão de tipos

Para converter entre tipos numéricos utiliza-se a função `cast`, que faz a conversão de memória necessária:

```
a = 5; % double
b = cast(a, 'int32'); % int32(5)

c = 1 + 2j; % complex double
d = cast(b, 'like', c); % complex double (5+0j)

e = cast(5.5, 'int8') % e = int8(6) (arredondamento)
```

Outra função existe, a `typecast`. Essa, porém, é como um cast de ponteiro (C: `int b = *((int*) &a);`). Ela não faz a reorganização de memória, apenas sua reinterpretação, sendo necessário que os dois tipos ocupem o mesmo espaço de memória:

```
typecast(5.5, 'int64') % 4617878467915022336
% em uma máquina 64 bits, tanto double quanto int64 ocupam 8 bytes
```

Nota sobre números complexos: `complex` no *MATLAB* funciona mais como um modificador do que como parte do tipo. Ele é carregado quando se manda alterar o tipo de uma variável. Se o original for real, o resultado será real, se for complexo, o resultado também será complexo.

3.1.3 Imprecisão Binária

```
1.1 + 2.2 == 3.3 # Falso!
```

Vale lembrar que todos os números são representados internamente de forma binária, o que pode gerar alguns resultados inesperados, como o acima. Isso ocorre pois o decimal é apenas aproximado do binário, e nem todo decimal tem uma representação binária

precisa. O resultado de $1.1 + 2.2$ é na verdade 3.3000000000000003 (arredondado) devido a essa limitação numérica.

Isso é algo que deve ser levado em conta ao resolver problemas com números muito pequenos. A função `realmin` retorna o menor número que pode ser representado no hardware atual (2.2251e-308 no meu computador 64bits). Números menores que isso ainda são possíveis, mas com penalidade de performance (podendo ser 100x mais lento), até `eps(0)` (4.9407e-324 no meu computador). Repare no entanto que na maioria dos campos de estudos números tão pequenos já são considerados zero. Costumamos (na área de Teoria de Controle) considerar como 0 qualquer valor na ordem de 10^{-16}, às vezes até 10^{-12} dependendo do contexto.

3.1.4 Símbolos e funções matemáticas

Como o *MATLAB* foi desenvolvido para resolver problemas matemáticos, e por funcionar como um *shell*, as constantes e funções matemáticas estão prontamente disponíveis:

```
pi
cos(pi) % ângulo em radianos
cosd(90) % ângulo em graus
exp(10) % número de Euler elevado à 10
log10(10) % logarítimo base 10 de 10
log(10) % logarítimo neperiano de 10
factorial(6)
```

Nota sobre `atan` e `atan2`: Essa diferenciação é puramente de implementação em software e não existe na matemática. A primeira recebe apenas um argumento, $\frac{x}{y}$ e a segunda recebe x e y separados. A segunda é geralmente mais interessante pois lida com o caso $y = 0$, que gera um erro na primeira, na maioria das linguagens. **Este no entanto não é um problema existente no *MATLAB***. No *MATLAB* a diferença é apenas no intervalo do retorno: `atan` retorna o resultado no intervalo $[-\frac{\pi}{2}, \frac{\pi}{2}]$, enquanto `atan2` retorna o resultado no intervalo $[-\pi, \pi]$.

Isso ocorre pois o *MATLAB* possui dois valores numéricos especiais: `NaN` (*Not a Number*) e `Inf`. Enquanto a divisão por zero é proibida em outras linguagens e gera excessões ou param a execução do programa, em *MATLAB* o resultado é `Inf`, que é corretamente interpretado por várias funções. `NaN` é geralmente fruto de erro, e toda operação realizada nele retorna ele mesmo, propagando o erro.

Exemplo de operações que produzem `NaN`:

```
0/0
0*inf
inf/inf
inf-inf
```

3.2 String

Uma `string` representa uma sequência de caractéres (letras, números e dígitos especiais). Todas strings no *MATLAB* são codificadas em Unicode (UTF-8), o que faz com que seja possível utilizar caractéres especiais, como acentos, símbolos e letras de outros alfabetos.

Há dois tipos básicos para representação textual no *MATLAB*: `string` e `char`. O tipo `string` foi inserido na versão R2017a. O tipo `char` guarda apenas um caractére, e uma sequência de `chars` faz o texto. O tipo `string` por sua vez guarda todo o texto em um único objeto e disponibiliza mais funções pra manipulação textual através de métodos.

3.2.1 Criação de Strings

Texto do tipo `char` pode ser criado utilizando aspas simples e do tipo `string` utilizando aspas duplas:

```
nome = 'Álan' % char
nome = "Álan" % string
```

Também é possível utilizar o método `string` para converter alguns tipos de dados para texto. Por exemplo, `string(5)` transforma o `double` 5 em `string`.

Outra forma de se criar strings é utilizando a função `sprintf`. Assim como sua homônima em C, ela utiliza uma string com sequências especiais para formatar valores em texto. A saída pode ser `string` ou `char`, dependendo do primeiro argumento.

```
a = sprintf("Este é o número pi: %f", pi) % retorna string
b = sprintf('Este é o número pi: %f', pi) % retorna char
sprintf('Primeiras 10 casas decimais do pi: %.10f', pi)
sprintf('2 casas decimais, com 5 caractéres, adicionando 0 à esquerda: %05.2f', pi)
sprintf('2 casas decimais, com 5 caractéres, adicionando espaços à esquerda: %5.2f', pi)
% para mais informações:
doc sprintf
```

3.2.2 Concatenação de Strings

Para strings do tipo `char` a concatenação pode ser feita com a sintaxe de matrizes:

```
a = 'Nome:'
b = 'Álan'
[a ' ' b] % 'Nome: Álan'
```

Strings do tipo `string` são concatenadas utilizando a função `strcat`, que também funciona com `char`.

```
a = "Nome:"
b = "Álan"
strcat(a, " ", b) % "Nome: Álan"
```

3.2.3 Métodos Comuns

Strings do tipo `char` são vetores linha e podem ser indexadas. Strings do tipo `string` não podem ser indexadas da mesma forma. Para converter de `string` para `char` pode-se usar o método `char`:

```
msg_string = ”Hello World!”
msg_char = msg_string.char
msg_char(2) % e
msg_char(7) % W
msg_char(2:5) % ello
```

Alguns métodos mais utilizados são listados abaixo.

```
str = 'Curso de Matlab'
strlength(str) % 15

str = string('Curso de Matlab')
str.lower % 'curso de matlab'
str.upper % 'CURSO DE MATLAB'
str.split(' ') % [”Curso”; ”de”; ”Matlab”]
str.contains('de') % 1 (True)
str.replace('Matlab', 'MATLAB') % ”Curso de MATLAB”, nova string retornada!
```

3.3 Vetores/Matrizes

Basicamente tudo no *MATLAB* é uma matriz, sendo ela o tipo básico de listas na linguagem. Ao clicar duas vezes em uma variável no *Workspace*, repare que o tipo sempre contém a dimensão da matriz, mesmo que 1×1.

Assim, vamos retomar o básico de criação de matrizes:

```
% os elementos de uma linha são separados por espaços ou vírgulas:
a = [1 2 3] % vetor linha, 1x3
% as linhas são separadas por ponto-e-vírgula:
b = [1; 2; 3] % vetor coluna, 3x1
% uma matriz combina os dois:
c = [1 2 3; 4 5 6; 7 8 9] % matriz 3x3
% matrizes especiais podem ser criados com funções:
I = eye(3) % identidade 3x3
i = eye(3, 2) % identidade 3x2
j = diag(a) % matriz 3x3 com os elementos de a na diagonal
k = ones(3) % matriz 3x3 com todos os elementos 1
l = zeros(3) % matriz 3x3 com todos os elementos 0
m = ones(2, 5) % matriz 2x5 com todos os elementos 1. Reparou o padrão?
n = magic(5) % matriz 5x5 onde a soma das linhas e colunas são iguais.
o = n' % transposto conjugado de n
```

Elementos de matrizes podem ser acessados utilizando a sintaxe de partição:

```
a = magic(3)
a(1) % primeiro elemento da matriz
a(2) % segundo elemento da matriz (colunas primeiro)
a(1:3) % elementos 1 a 3 (inclusive) da matriz (colunas primeiro)
a(1:2,:) % linhas 1 a 2, todas as colunas (matriz 2x3)
a(1:2,1:2) % linhas 1 a 2, colunas 1 a 2 (matriz 2x2)
a(2:end,2:end) % linha 2 até o final, coluna 2 até o final (2x3)
a(1:end-1,1:end-1) % exclui a última linha e coluna (2x2)
a(:) % retorna todos os elementos de a em um vetor coluna
a([1 3], [2 3]) % Elementos das linhas 1 e 3 e colunas 2 e 3
% Observe que 1:30 gera um vetor sequencial, de 1 até 30 inclusive
% 1:2:30 gera um vetor de 1 até 30, de 2 em 2
```

Embora tudo seja matriz, uma matriz 1×1 é tratado como um escalar. Operações neles funcionam normalmente:

```
1+1
5-7
2*3
2/3
3^2
```

Já em matriz e vetores, os operadores tem a função esperada para operar nessas estruturas:

```
A = [-1 2; 2 -5]
x = [3; 1]
A*x % multiplicação matricial
A+x % soma elemento-por-elemento, repetindo quem for menor. Deve ter
    % dimensões compatíveis (mesmas dimensões, ou uma dimensão ser 1)
A-x % idem
A^2 % A*A
A^3 % A*A*A
A/x % "mesmo" que A*inv(x)
A\x % "mesmo" que inv(A)*x
```

Repare nas duas últimas expressões. A diferença entre `A/b` e `inv(A)*b` é que eles nem sempre resolvem o mesmo problema, o primeiro é numericamente mais eficiente e sempre retorna um resultado se possível. Isso ocorre pois caso A seja quadrada e regular (tenha inversa), os dois comandos são equivalentes. Caso A não seja quadrada ou seja singular, o primeiro caso resolve o problema $AX = b$ buscando um X que minimize $(AX - b)^2$. Se a equação $AX = b$ tiver infinitas soluções, uma solução com o menor número de zeros é retornada. O *MATLAB* recomenda sempre usar `A/b` e `Ab` ao invés da função `inv`.

Para matrizes de dimensões compatíveis, a multiplicação, divisão e potenciação podem ser feitas termo a termo. Multiplicação e divisão por escalares são sempre termo a termo.

```
A = magic(3)
B = rand(3)

A .* B
A ./ B
A .^ 2

2 * A
A / 2
```

Várias funções também são aplicadas termo a termo, como as trigonométricas, por exemplo.

```
A = magic(3)
sin(A)
cos(A)
exp(A)
log(A)
```

Funções matriciais importantes:

```
A = magic(3)
det(A) % determinante
inv(A) % inversa
trace(A) % traço
diag(A) % diagonal principal
eig(A) % autovalores
[eigenvectors, ~] = eig(A) % autovetores, ver help eig
expm(A) % exponenciação matricial
svd(A) % single value decomposition
rank(A) % posto
charpoly(A) % polinômio característico (coeficientes)
orth(A) % base orthonormal para A
null(A) % base orthonormal para o espaço nulo de A
length(A) % tamanho de A
size(A) % dimensões de A
```

Funções vetoriais importantes:

```
x = [1; 2; 3]
norm(x) % norma
sum(x) % somatório
prod(x) % produtório
orth(x) % normaliza x (x tem que ser vetor coluna, Nx1)
length(x) % tamanho de x
size(x) % dimensões de x
% os valores de x podem ser os coeficientes de um polinômio:
% [a b c] seria a*x^2 + b*x + c
% [a b c d] seria a*x^3 + b*x^2 + c*x + d
poly([-1 -2 -3]) % cria um vetor de coeficientes para um polinômio cujas raízes são -1, -2
    e -3
polyval(x, 5) % calcula o valor do polinômio para x=5, 1*5^2+2*5+3
conv([1 5], [3 6]) % convolução, mesmo que (x+5)*(3*x+6), retorna um vetor de coeficientes
deconv([3 21 30], [3 6]) % contrário de conv, (3*x^2+21*x+30)/(3*x+6). veja help deconv
roots(x) % raízes do polinômio com coeficientes x
```

Outra funcionalidade que merece destaque é a concatenação de matrizes. Alguns algorítmos requerem que matrizes sejam concatenadas para se aplicar operações, como a verificação de controlabilidade e observabilidade, ou para agrupar informações, como restrições de uma otimização. Você já deve ter feito o contrário, separar uma matriz em blocos, para resolver um sistema linear através de operações nas linhas.

A sintaxe é a mesma da criação de matrizes, exceto que cada elemento é uma matriz ao invés de um número. As dimensões devem ser compatíveis.

```
A = eye(3);
```

```
B = [1; 2; 3];
C = [1 0 0];
D = 0
S = [A B; C D]
```

3.4 Células

Matrizes aceitam apenas elementos do mesmo tipo, células aceitam elementos de tipos diferentes. Matrizes são criadas com colchetes, células com chaves.

```
a = {1 2 'MATLAB'; inf "привет" [1 2 3]}
```

`a` é na verdade um vetor de células, ou seja, uma matriz onde cada elemento é uma célula. Isso fica evidente ao utilizarmos a partição.

```
a = {1 2 'MATLAB'; inf "привет" [1 2 3]}
a(1) % uma célula, não o valor 1
a{1} % o valor 1
```

A partição funciona normalmente ao utilizarmos parênteses, da mesma forma que com matrizes. No entanto o comportamento é diferente ao utilizar as chaves. Vamos ver essa diferença transformando os 2 primeiros elementos da primeira linha em um vetor.

```
a = {1 2 'MATLAB'; inf "привет" [1 2 3]}
b = cell2mat(a(1,1:2)) % como matriz é homogênea, todos os elementos devem
                       % ter o mesmo tipo
c = [a{1,1:2}]
```

Ok, `b` e `c` são iguais, mas por quê? Ao digitar `,1:2a1` temos vários `ans` com valores, e não uma matriz. Acontece que o *MATLAB* tem uma estrutura especial chamada "lista separada por vírgula". Você pode simplesmente digitar `1,2,3` no console pra ver uma lista dessas. Essas listas não são um tipo de dados, e não podem ser salvas em variáveis, mas são parte intrínseca da linguagem. Praticamente em qualquer lugar onde você digitaria uma lista de items, você pode usar algo que gere uma lista separada por vírgula. Pense na lista sendo expandida antes do restante da expressão ser compilada:

```
% comecemos com a variável
a = {1, 2}
% vamos executar o seguinte código passo-a-passo:
zeros(a{:})
% primeiro devemos "calcular" a{:}, que significa todos os elementos de a,
% em ordem. O mesmo que a{1}, a{2}, a{3}, ..., a{end}.
```

```
% substitua esse valor na função:
zeros(a{1}, a{2})
% substitua as variáveis pelos valores delas:
zeros(1, 2)
% agora pode executar a função que retornará um vetor 1x2
```

Repare que os valores da lista separada por vírgula se tornaram os argumentos da funcão, ou seja, é uma forma simples de aplicar argumentos variados a uma função.

```
a = {'A parte inteira de %.2f é %d' pi 3};
str = sprintf(a{:})
```

Assim, o que ocorreu aqui: `,1:2c = [a1]` foi que `,1:2a1` virou `,1,2a1, a1`, ou seja `,1,2c = [a1, a1]` que virou `c = [1, 2]`.

As funções `mat2cell` e `num2cell` convertem matrizes para células. A última torna cada elemento da matriz em uma célula.

3.5 Estruturas

Assim como células, estruturas também agrupam valores heterogêneos, porém com nomes. Há duas sintaxes para criar estruturas:

```
% através do método
s = struct('nome', "Álan", 'idade', 30)
s.nome % retorna o nome
s.idade % retorna a idade

% diretamente
a.nome = "Álan"
a.idade = 30
```

Estruturas também podem ser escalares ou matriciais, como quase tudo no *MATLAB*:

```
value = {'some text'; [10, 20, 30]; magic(5)};
s = struct('f', value)
s.f % lista separada por vírgula! Equivalente a s(1).f, s(2).f, s(3).f

% ou
a.f = 'some text'
b.f = [10, 20, 30]
c.f = magic(5)
d = [a; b; c]
d.f % lista separada por vírgula!
```

```
fields = fieldnames(d)
field = 'f'
d.(field)
```

Estruturas são comuns no *MATLAB*. Execute os códigos acima linha a linha, observando o painel *Workspace* e a variável para compreender melhor como elas funcionam. Elas são simples e não tem segredos.

Existem outras estruturas menos utilizadas no *MATLAB*. Se você achar que seus dados não se encaixam bem em nenhum destes casos, pequise. O *MATLAB* possui, por exemplo, classes para Tabelas e Dicionários (Map).

Capítulo 4

Controle de Fluxo

4.1 If

A estrutura **if** permite executar ou não algum código baseado em uma condição.

```
if expression
    statements
elseif expression
    statements
else
    statements
end
```

Por exemplo, para dizer se um número é par ou ímpar:

```
a = round(input('Digite um número: '))
if mod(a, 2) == 0
    disp(sprintf('%d é par', a))
else
    disp(sprintf('%d é impar', a))
end
```

A condição do `if` será convertida para o tipo `logical`, se não o for. Veja `help logical` para obter uma lista de coisas que podem e não podem ser convertidas. Basicamente, números complexos e `NaN` não podem ser convertidos. Outros valores virarão `logical(1)` e `0` vira `logical(0)`. Também existem as funções `true` e `false` que retornam `logical(1)` e `logical(0)` respectivamente.

Caso haja uma sequência de `&&` e/ou `||`, a expressão seguinte na cadeia não é executada se a informação anterior já for suficiente para determinar o resultado. Por exemplo, em `abs(5)|| norm(x)`, a expressão `norm(x)` não será executada, pois `abs(5)` é verdadeiro e

uma expressão verdadeira é suficiente pra fazer uma cadeia de operadores OU retornar verdadeiro.

4.2 For

A estrutura `for` é utilizada para iterar vetores. Se uma matriz for passada, será iterada coluna por coluna. É comumente utilizada com a notação compacta de geração de sequências.

```
    for i=1:100
        prime = 1;
        for j=2:ceil(sqrt(i))
            if mod(i,j) == 0
                prime = 0;
                break
            end
        end
        if prime
            fprintf('%d é primo\n', i) % como sprintf, mas escreve na tela
        end
    end
```

4.3 While

O loop `while` é bem parecido com o de outras linguagens.

```
    while condition
        statements
    end
```

As palavras-chave `break` e `continue` podem ser usadas tanto com loops `for` quanto `while`.

```
1  i = 0;
2  while i < 10
3      i = i + 1;
4      if i==5
5          continue % pula o número 5, que não será escrito na tela.
6      end
7      disp(i)
8  end
```

4.4 Switch

A estrutura `switch` serve para executar código caso uma igualdade seja verdadeira.

```
switch switch_expression
    case case_expression
        statements
    case case_expression
        statements
        ...
    otherwise
        statements
end
```

Por exemplo:

```
a = 'bar'
switch a
    case 'foo'
        disp('a is foo')
    case 'bar'
        disp('a is bar')
    otherwise
        disp('a is something else')
end
```

Note que, diferente de C, não é necessário usar `break`, já que apenas o primeiro `case` verdadeiro executará. Diferente de C, pode-se definir mais de um valor para um `case`:

```
a = 'bar'
switch a
    case {'foo' 'bar'}
        disp('a is foo or bar')
    otherwise
        disp('a is something else')
end
```

Capítulo 5

Funções

Funções em *MATLAB* são mais restritas em sua declaração mas mais versáteis em seu uso quando comparadas com várias outras linguagens. Elas tem regras estritas sobre onde podem ser declaradas:

- Em um arquivo próprio, com o mesmo nome da função;
- Após outra função;
- Dentro de outra função;
- No **final** de um arquivo de *script*.

Para que a função seja visível, e o arquivo seja considerado uma função e não um *script*, a declaração da função deve ser a primeira linha executável do arquivo (comentários podem vir primeiro, já que não são executáveis). Funções declaradas após a primeira função e aquelas declaradas em um *script* são funções *locais*. Funções também podem ser *aninhadas*, ou seja, declaradas uma dentro da outra. Uma função aninhada tem acesso às variáveis locais da função externa, funções locais não. Parece haver penalidade de performance em alguns casos quando se usa funções aninhadas, por isso, e por boas práticas de programação (separação de conceitos, encapsulamento, testabilidade), prefira usar funções locais.

Exemplo de declaração de função. Deve estar em um arquivo chamado **soma.m**.

```
function [z] = soma(x, y)
    z = x + y;
end
```

A forma de retornar valores no *MATLAB* é diferente da maioria das linguagens. A palavra chave `return` serve apenas para parar a execução da função ou *script* e retornar a execução para o script que o chamou, não para retornar um valor.

O *MATLAB* usa o conceito de variáveis de saída. As variáveis declaradas após a palavra chave `function`, dentro dos colchetes, são variáveis de saída, e é possível ter mais de uma delas.

```
function [media, desvio] = media(xs)
    media = mean(xs);
    desvio = std(xs);
end
```

Neste caso, tanto `media` quanto `desvio` são variáveis de saída. Se chamarmos essa função como

```
media(1:10)
```

apenas o valor da primeira variável é retornado. Se quisermos capturar o valor das duas variáveis de saída, devemos chamar como

```
[m,d] = media(1:10)
```

onde `m` e `d` (poderia ser quaisquer outras variáveis) terão os valores de `media` e `desvio`, respectivamente. Se quisermos apenas o desvio, podemos chamar como

```
[~, d] = media(1:10)
```

e o valor da média será ignorado. Isso é interessante pois algumas funções retornam valores diferentes dependendo do número de variáveis de saída que foram requisitadas.

Outra coisa interessante de ser notada é o suporte à documentação. Toda função interna do *MATLAB* e seus *toolboxes* possuem documentação acessível através das funções `help` e `doc`. Você também pode criar sua documentação que será mostrada pelo comando `help`. Para tal, basta definir um bloco de comentário especial logo após a declaração da função. Apenas funções publicamente acessíveis funcionam com o comando `help`.

```
function a()
%NOME_DA_FUNCAO Sumario, descricao resumida da funcao aqui
%   Explicacao detalhada aqui
%   podendo ocupar mais de uma linha.
```

5.1 Parâmetros

Parâmetros são os valores que uma função espera receber, as suas variáveis de entrada. Argumentos são os valores passados para a função na chamada. No entanto é comum ver os termos parâmetro e argumento serem utilizados indiscriminadamente.

No *MATLAB* há apenas dois tipos de argumentos: obrigatórios e variáveis.

5.1.1 Parâmetros Obrigatórios

Parâmetros obrigatórios terão seu valor definido pela sua posição na declaração da função. Repare que o exemplo abaixo é um *script*, por isso as funções devem vir no final.

```
1  soma(1, 2) % x=1, y=2
2  soma3(1, 2, 3) % x=1, y=2, z=3
3
4  function z = soma(x,y)
5      z = x + y;
6  end
7
8  function w = soma3(x,y,z)
9      w = x + y + z;
10 end
```

5.1.2 Parâmetros Variáveis

Parâmetros variáveis são utilizados para aumentar a quantidade de argumentos recebidos pela função indefinidamente. Ele deve ser o último parâmetro da função. É basicamente um nome especial, *varargin*, que será interpretado de forma diferente, agrupando todos os argumentos em uma célula. A função `nargin` contém o número de argumentos variáveis que foram passados.

```
somatorio(1, 2)
somatorio(1, 2, 3, 4, 5)

function z = somatorio(varargin)
    z = 0;
    for x=[varargin{:}]
        z = z + x;
    end
end
```

Não há sobrecarga de funções no *MATLAB*, e esse efeito é obtido através dos parâmetros variáveis ou verificando o valor passado para um argumento.

```
operacao(1)
operacao(1, 2)
operacao(1, '2')
operacao(1, 2, 3)

function z = operacao(varargin)
    %operacao Realiza uma operação baseado nos argumentos de entrada
    %   Caso 1 argumento seja passado: multiplica por 2
    %   Caso 2 argumentos sejam passados:
    %          Caso o segundo seja uma string: concatena
    %          Caso o segundo seja um número: soma
    %   Caso 3 argumentos sejam passados: produtório

    switch nargin
        case 1
            z = 2 * varargin{1};
        case 2
            if isstring(varargin{2}) || ischar(varargin{2})
                z = strcat(num2str(varargin{1}), varargin{2});
            else
                z = sum([varargin{:}]);
            end
        case 3
            z = prod([varargin{:}]);
    end
end
```

```
operacao
operacao(2)
operacao(2, 3)

function z = operacao(a, b)
    z = 1;
    if exist('a', 'var')
        if exist('b', 'var')
            z = a + b;
        else
            z = a;
        end
    end
end
```

5.1.3 Retorno Variável

Da mesma forma, existem `varargout` e `nargout`.

```
gen % 1
[a,b] = gen % a=1, b='2'
[c,~,e,~] = gen % c=struct(nome:Álan,idade:30), ignorado, e=30, ignorado

function varargout = gen()
    switch nargout
        case 0
        case 1
            varargout = {1};
        case 2
            varargout = {1, '2'};
        case 3
            varargout = num2cell(3:-1:1);
        case 4
            p.nome = "Álan";
            p.idade = 30;
            varargout = {p "Álan" 30 magic(5)};
    end
end
```

Algumas funções mudam completamente a saída de acordo com o número de variáveis de saída solicitados. Um exemplo é a função `eig`. Veja a documentação (`help eig`) para ver as várias opções.

5.2 First Class Citizens e High Order

Funções no *MATLAB* não são *first class citizens*, mas ponteiros (chamados *handle*) podem ser criados para qualquer função, permitindo que funções sejam passados como argumento em chamadas de funções.

```
somatorio = @sum % cria uma function_handle que aponta para a função sum
somatorio([1 2 3]) % mesmo que sum([1 2 3])
```

Algumas funções precisam de uma *function_handle*. Um exemplo (a função `integral`) será mostrado na próxima seção.

5.3 Lambdas

Lambdas são funções anônimas e curtas. Elas podem receber qualquer número de parâmetros e são uma única expressão. Geralmente são expressões que transformam um valor.

```
% integral de sin(x) de 0 a pi
integral(@(x) sin(x), 0, pi)
```

Neste caso, poderíamos simplesmente chamar `integral(@sin, 0, pi)`, já que nossa *lambda* (`@(x)sin(x)`) está apenas repassando os argumentos para o seno. Mas se quisermos a integral de $x^2 + 2x - 4$:

```
f = @(x) x.^2+2*x-4
integral(f, 0, pi)
```

Por questões de performance a função `integral` usa vetores, por isso é necessário utilizar a operações item a item.

5.4 Módulos/Pacotes/Bibliotecas

Excluindo as *toolboxes* desenvolvidas pela *MathWorks*, não há nenhum mecanismo de isolamento/distribuição de pacotes, módulos, bibliotecas, *namespaces*, nem nada do gênero. O *MATLAB* funciona como um interpretador *shell*, contendo uma variável **PATH** que controla os diretórios onde ele buscará por funções, classes e *scripts*.

Se você deseja criar uma biblioteca, simplesmente coloque todos os arquivos num mesmo diretório e distribua o diretório. Basta que o usuário adicione o diretório ao **PATH** para que ele use suas funções. Lembre-se de evitar conflitos de nomes, principalmente com funções padrão do *MATLAB*.

Capítulo 6

Programação Orientada a Objetos

Classes, assim como funções, devem ser declaradas em seus próprios arquivos, que devem ter o mesmo nome da classe. Há dois tipos de classes no *MATLAB*: *value* e *handle*. Basicamente a diferença é se a passagem do objeto derivado será por valor ou por referência.

Como o objetivo dessa apostila não é ensinar os conceitos, apenas a linguagem, apenas as sintaxes serão apresentadas. Um leitor familiarizado com orientação a objetos identificará os pontos chaves no exemplo. Recomendo que aprenda por experimentação e lendo a documentação quando necessário.

```
% value class
classdef person
    %person Representa o registro de uma pessoa

    % propriedades privadas
    properties(GetAccess=private)
        DataNasc = datetime(1900, 1, 1)
    end

    % constantes
    properties(Constant)
        IsPerson = true
    end

    % calculadas no acesso. O valor não é guardado. Veja o método get.Idade
    properties(Dependent)
        Idade
    end

    % propriedades públicas
    properties
```

```
        Nome = "" % valor padrão
    end

    methods
        % construtor
        function this = person(nome, year, month, day)
            this.DataNasc = datetime(year, month, day);
            this.Nome = nome;
        end

        % getter, assim como no Python, o objeto é passado como primeiro argumento
        function Idade = get.Idade(this)
            Idade = duration(datetime - this.DataNasc, 'Format', 'y');
        end

        % setter
        function this = set.Nome(this, nome)
            if isstring(nome) || ischar(nome)
                this.Nome = string(nome);
            end
        end

        % ~ diz que o primeiro argumento deve ser ignorado. Se alguma
        % propriedade do objeto fosse usado, ele poderia ser nomeado e
        % utilizado para acessar a propriedade.
        function ret = HasHair(~)
            ret = "maybe"
        end
    end

    methods(Static)
        function p = alan()
            p = person('Álan', 1990, 2, 17);
        end
    end
end
```

Para mudar essa classe para o tipo *handle*, altere a definição para `classdef person < handle` e remova o retonro dos *setters*.

A diferença entre os dois tipos de classe é que, dado o código

```
p1 = person.alan
p2 = p1
p1.Nome = 'João'
```

caso a classe seja do tipo *value*, `p2.Nome` continua sendo "Álan". Caso seja do tipo

handle, será "João" também. Ou seja, *value* copia os dados e *handle* evita essa cópia. Repare que, basicamente, classes são construídas a partir de Estruturas (3.5), que são passadas e retornadas dos métodos e que herança simples é suportada.

6.0.1 Eventos

Uma particularidade do *MATLAB* são os eventos, presentes nas classes do tipo *handle*.

```
classdef person < handle
    events
        ChangeEvent
    end

    methods
        function do(this)
            notify(this, 'ChangeEvent')
        end
    end
end
```

```
p = person

addlistener(p, 'ChangeEvent', @(object, event_data) disp(object))
addlistener(p, 'ChangeEvent', @(object, event_data) disp(event_data))

p.do % as duas funções registradas serão executadas.
```

Capítulo 7

Matemática Computacional

A matemática computacional explora o poder de processamento dos computadores para realizar cálculos e resolver problemas que seriam difíceis ou inconvenientes de serem resolvidos à mão.

7.0.1 Métodos Numéricos

Várias funções estão prontamente disponíveis no *MATLAB* para resolver problemas de cálculo, finanças, álgebra, etc; como integrais e derivadas numéricas, *Fast Fourier Transform*, simular sistemas ou resolver sistemas lineares. Várias delas foram apresentadas em capítulos anteriores, e algumas outras serão listadas.

7.0.1.1 Integral

Conforme já apresentado anteriormente, a integral numérica pode ser calculada utilizando a função `integral`. Integrais duplas e triplas podem ser calculadas utilizando as funções `integral2` e `integral3`.

```
integral(@sin, 0, 2*pi) % o número minúsculo é um erro numérico, o resultado
                        % é claramente 0
integral2(@(x,~) x, 0, 1, 0, 2*pi) % círculo de raio 1
integral3(@(r,t,p) r.^2.*sin(t), 0, 1, 0, pi, 0, 2*pi) % esfera de raio 1
```

7.0.1.2 Derivada

Derivadas podem ser calculadas com a função diff.

```
x = linspace(0, 2*pi, 1000); % 1000 elementos linearmente espaçados entre 0 e 2*pi
y = sin(x);
```

```
dy = diff(y);
dt = 2*pi/1000; % distância entre cada elemento

cosY = dy/dt;
```

7.0.1.3 Sistema de Equações Lineares

Para resolver sistemas de equações lineares numericamente devemos escrever o sistema em sua forma matricial. Podemos então usar a inversa diretamente ou utilizar a função `linsolve`. A vantagem dessa função é que ela também funciona quando a matriz `A` não for quadrada.

```
A = [5 4 -1; 0 10 -3; 0 0 1];
b = [0; 11; 3];

A\b
linsolve(A, b)
```

7.0.1.4 Equações Diferenciais

Existem várias métodos para a resolução de EDO's numericamente. Procure por métodos com nomes como `ode23`, `ode45` e `ode113`. Cada método tem sua particularidade e forma de resolver o problema. Leia a documentação.

Resolve a EDO $\dot{y} = \cos(y)$ no intervalo 0 a 2π:

```
f = @(t,~) cos(t);
ts = linspace(0, 2*pi, 100);
[t,y] = ode45(f, ts, 0);
```

7.0.2 Expressões Simbólicas

Expressões simbólicas são compostas por números e variáveis matemáticas indefinidas. Elas são úteis para descrever e manipular algébricamente funções e expressões.

Para definir uma variável simbólica use a função syms. Algumas considerações podem ser feitas quanto às variáveis: *real*, *rational*, *integer*, ou *positive*.

```
syms a b x(t)
assume(t, 'real')
assume(t, 'positive')
```

7.0.2.1 Somatório

A função symsum faz somatórios.

```
syms x(t) n
symsum(x^n/factorial(n), n, 0, inf)
```

7.0.2.2 Limite

A função **limit** calcula limites.

```
syms x

limit(sin(x)/x, x, 0) % 1
limit(1/x, x, 0, 'Right') % inf
limit(1/x, x, 0, 'Left')  % -inf
limit(1/x, x, inf % 0
```

7.0.2.3 Derivada

A função diff calcula a n-ésima derivada.

```
syms x y f(x)

diff(sin(x), x) % cos(x)
diff(f(x), x, x, x) % diff(f(x), x, x, x) simbólico
diff(f(x), x, 3) % diff(f(x), x, x, x) simbólico
diff(sin(x)*cos(y), x, x, y, y) % sin(x)*cos(y)
```

7.0.2.4 Integral

A função integrate integra uma função.

```
syms a x y

int(x*y, x) % x^2*y/2
int(log(x), x) % x*(log(x) - 1)
int(log(x), x, 1, a) % a*(log(a) - 1) + 1
int(x) % x^2/2
```

7.0.2.5 Manipualação de Expressões

Os comandos `simplify`, `expand`, `factor`, `collect` e `partfrac` podem ser utilizados para manipular expressões algébricas.

```
syms x y

simplify(cos(x)^2 + sin(x)^2) % 1
expand((x + 2)*(x - 3) ) % x^2 - x - 6
factor(x^3 - x^2 + x - 1) % [x^2 + 1, x - 1]
collect(x^2 + 2*(x + y) - x*y, x) % x^2 + (2 - y)*x + 2*y
partfrac((3*x+5)/(1-2*x)^2) % 3/(2*(2*x - 1)) + 13/(2*(2*x - 1)^2)

exp = (x + 2)*(x - 3)
subs(exp, x, 3) % 0
double(root(exp, x)) % [-2; 3]
solve(exp, x) % [-2; 3] | resolves exp == 0
```

7.1 Gráficos

A função `plot` é usada para exibir gráficos. As funções `title` e `legend` alteram o título e legendas, respectivamente. Outras funções podem ser usadas para gerar diferentes tipos de gráficos. A *MathWorks* tem uma galeria no link `https://www.mathworks.com/products/matlab/plot-gallery.html`.

```
x = linspace(-pi, pi, 256);
c = cos(x);
s = sin(x);

plot(x, c, 'b-')
hold on
plot(x, s, 'r-')
legend('Coseno', 'Seno')
title('Seno e Coseno')

saveas(gca, 'exemplo.eps', 'epsc')
```

A função `plot` recebe como argumentos os pontos x, y e a formatação da linha. A Tabela 3 (retirada da documentação) mostra as formas de linhas disponíveis. O terceiro argumento da função `plot` pode ser uma combinação dos elementos das 3 colunas (um de cada coluna).

Tabela 3 – Formatação de linhas

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	–	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

Capítulo 8

Otimização

Primeira regra da otimização: não otimize.

1. Faça certo.
2. Teste se está certo.
3. *Profile* se estiver lento.
4. Otimize.
5. Repita do 2.

Dito isso,

- Use funções no lugar de scripts;
- Use funções locais ao invés de aninhadas;
- Modularize o código (evite arquivos muito grandes);
- Pre-aloque as variáveis, ao invés de expandir dentro de loops;
- Vetorize;
- Evite mudar o tipo de uma variável, crie uma nova;
- Use operadores de curto circuito (&& e ||);
- Evite variáveis globais (casa com evitar scripts);
- Não sobrecarrege funções do matlab (*built-ins*);

- Evite gerar dados que podem ser guardados em arquivos MAT;
- Evite limpar desnecessariamente (especialmente `clear all`);
- Evite funções de intro-inspeção (`inputname, which, whos, exist(var), dbstack`);
- Evite `cd, addpath, rmpath`. Mudar o **PATH** causa a recompilação de funções;
- Acesse dados em colunas ao invés de linhas (*CPU cache*);
- Não crie variáveis desnecessariamente.

8.1 Transferência de Conhecimentos

MATLAB não é C. Nem Perl. Nem C++. Nem Java. Não assuma que coisas que funcionam em uma linguagem funcionará em *MATLAB*. Ou em qualquer outra linguagem. Por exemplo, em C/C++ você pode declarar quantas variáveis quiser e usar numa chamada de função, que o código compilado será extamente o mesmo que passar os parâmetros diretamente pra função. Isso não é verdade no *MATLAB*, onde todas as variáveis serão criadas e destruídas pelo interpretador.

```
1 a = 1
2 b = 2
3 c = a + b # Em C, isso é o mesmo que c = 1 + 2, e as variáveis a e b nunca serão criadas.
4           # Na verdade, isso é o mesmo que c = 3 em C, já que o compilador fará essa conta.
5           # Já no MATLAB todas as variáveis serão criadas.
```

8.2 Vetorização

Vetorizar é transformar um código (normalmente um *loop*) em uma expressão que envolva operações matriciais. Isso faz uso do de instruções SIMD (*Single Instruction Multiple Data*) do processador e melhora a performance do programa. O *MATLAB* foi feito para trabalhar com matrizes, e é otimizado para operações em matrizes e matriciais.

```
a = 1:10;

% somatória dos quadrados com loop:
s = 0;
for i=a
```

```
    s = s + i^2;
end

% ou
s = sum(a.^2)

% versão vetorizada
s = a*a'
```

Capítulo 9

Análise de Sistemas e Teoria de Controle

O *toolbox Control System* provê várias funções para lidar com análise e controle de sistemas lineares. As principais funções serão apresentadas nesse capítulo.

9.0.1 Sistemas/Modelos

As principais funções para criação de sistemas são `tf`, para função de transferência, `ss` para espaço de estados e `zpk` para função de transferência a partir de zeros, polos e ganhos. Essa última é interessante pois evidencia essas informações quando a variável é exibida na tela.

```
G = tf([5], [1 3]) % tf(numerador, denominador), lembra dos vetores de coeficientes?
G = zpk([], [−3], [5]) % zpk(zeros, polos, ganhos)
G = ss(−3, 2, 2.5, 0) % ss(A,B,C,D), colchetes são desnecessários em matrizes 1x1
```

É possível converter entre tipos utilizando as mesmas funções.

```
G = tf([5], [1 3])
zpk(G)
ss(G)
```

Se mais um argumento for passado, será o tempo de amostragem e o modelo será discreto. Também é possível especificar atraso de transporte.

```
G = zpk([], [0.98], 2, 5) % 5 segundos de tempo de amostragem
G = zpk([], [−3], 2, 'InputDelay', 1) % contínuo
G = zpk([], [0.98], 2, 5, 'InputDelay', 1) % discreto
```

Pode-se criar modelos de malha fechada utilizando o comando `feedback`. O primeiro argumento é o ramo direto, o segundo o ramo da realimentação.

```
G = zpk([], [0.98], 2, 5);
```

```
Gcl = feedback(G, 1) % realimentação unitária sem controlador
C = pid(1, 0.1, 0.1, 0, 5); % Kp, Ki, Kd, Ts, tempo de amostragem
Gcl = feedback(C*G, 1) % realimentação com controlador no ramo direto
Gcl = feedback(G, C) % realimentação com controlador no ramo inverso
G = ss(G) % funciona com qualquer um dos modelos
Gcl = feedback(C*G, 1) % realimentação com controlador no ramo direto
Gcl = feedback(G, C) % realimentação com controlador no ramo inverso
```

9.0.2 Gráficos

As funções de gráficos mais importantes são:

9.0.2.1 Step

Exibe um gráfico das saídas do modelo em resposta à uma entrada do tipo degrau. A amplitude do degrau é controlado multiplicando-se o modelo pelo valor desejado, fazendo com que o padrão seja um degrau unitário.

```
G = zpk([], [0.98], 2, 5);
step(G) % degrau unitário
step(5*G) % degrau de amplitude 5
```

9.0.2.2 Impulse

Exibe um gráfico das saídas do modelo em resposta à uma entrada do tipo impulso.

```
G = zpk([], [0.98], 2, 5);
impulse(G)
```

9.0.2.3 Lugar das Raízes

Exibe um gráfico do lugar das raízes do sistema. Preste atenção na ferramenta *Data Cursor*, que retorna várias informações pertinentes à sintese de controlador.

```
G = zpk([], [0.98], 2, 5);
rlocus(G)
```

9.0.2.4 Nyquist

Exibe um gráfico de Nyquist do sistema.

```
G = zpk([], [0.98], 2, 5);
nyquist(G)
```

9.0.2.5 Bode

Exibe um gráfico de Bode do sistema.

```
G = zpk([], [0.98], 2, 5);
bode(G)
```

9.0.3 Outras funções

Outras funções estão disponíveis para analisar sistemas lineares e sintetizar controladores, além dos gráficos.

```
G = zpk([], [0.98], 2, 5);

% analysis
pole(G) % polos
zero(G) % zeros
dcgain(G) % ganho
stepinfo(G) % várias informações numéricas sobre a resposta ao degrau
isstable(G)

% synthesis
pidtool(G)
G = ss(G)
acker(G.A, G.B, [-5]) % formula de Ackermann
place(G.A, G.B, [-5]) % Como acker, mas mais robusta
```