

Introdução à programação em MATLAB

João Cândido Magalhães

Vice-Presidente — SEG-Student Chapter Campinas.

Grupo de Geofísica Computacional (GGC)
DEP-FEM-Unicamp

29 e 31 de Outubro de 2019

SEG - Student Chapter Campinas em parceria com o SPE - Student Chapter Campinas

① Primeiro dia - manhã

- Introdução ao MATLAB
- Configuração do ambiente da área de trabalho
- Variáveis, Vetores e Matrizes
- Caracteres especiais, operadores aritméticos, relacionais e lógicos
- Programação em MATLAB - *scripts*

② Primeiro dia - tarde

- Controle de fluxo - *if* e *switch*
- Laço - *for* e *while*
- Funções

③ Segundo dia - manhã

- Entrada/Saída (I/O) - usuário e arquivos
- Introdução as ferramentas básicas de visualização — *plot*.

④ Segundo dia - tarde

- Projeto final
- Dúvidas

Configuração do ambiente da área de trabalho

- ① Sempre conferir o diretório atual em que o MATLAB está trabalhando.
- ② Comandos importantes:
 - help - Mostra ajuda no *prompt* de comando.
 - doc - Abre uma janela com mais detalhes.
 - cd - Troca para o diretório especificado.
 - ls - Lista o conteúdo do diretório atual.
 - mkdir - Cria um diretório.
 - pwd - Lista o caminho do diretório.
 - rm - Remove um arquivo.
 - rmdir - Remove um diretório.
 - clear - Limpa todo o *Workspace*.
 - clc - Limpa o *prompt* de comando.
 - CTRL + C - Para a execução do programa.

Declarando variáveis

>> a = 2 ;

Nome da variável

Atribuição de variável

Valor atribuído

Supressão de saída

- Onde = e ; são caracteres especiais.

O MATLAB possui regras para nomear variáveis:

- O nome deve começar com uma letra do alfabeto.
- O nome pode conter números e o caracter `_` — ex. `alpha_12`.
- O nome possui um limite de caracteres — `namelengthmax`.
- O nome é *case-sensitive* — ex. `Alpha` \neq `alpha`.
- Existem algumas palavras que são reservadas — `iskeyword`.

Além disso, seguem duas recomendações para nomear variáveis:

- Os nomes podem ter nomes de funções do MATLAB, mas essa prática deve ser evitada.
- Os nomes devem ser mnemônicos, i.e., ter nomes que auxiliam na descrição do propósito da variável.

mnemônico — Diz-se da técnica ou exercício que ajuda a desenvolver a memória e facilita a memorização.

Por exemplo, em um programa para calcular a área de um círculo, faz mais sentidos ter-se uma variável, que armazena o valor do raio, chamada raio, do que uma variável chamada a, b, x, ou porta. Tenha sempre em mente que o seu código deve maximizar a legibilidade, i.e., ao ler o código, entendemos exatamente o que ele faz.

Tipos de variáveis

MATLAB suporta vários tipos de variáveis e toda variável possui um tipo.

Podemos saber o tipo da variável com a função do MATLAB

```
class()
```

Começaremos com os dois tipos mais básicos

- Numéricos — números inteiros e ponto-flutuante,
- Caracteres e *strings* — texto em *arrays* de caracteres e *strings*.

Exercício: Utilize a função `class()` nos seguintes valores.

```
>> a = 1;  
>> a = 1.25;  
>> a = 'Curso2019';  
>> a = "Curso2019";
```

- Declara-se um *array* de caracteres com aspas simples ' '.
- Declara-se uma *string* com aspas duplas " ".

A principal diferença entre uma *string* e um *array* de caracteres é que a *string* é um objeto e um *array* de caracteres é um tipo mais primitivo de dados.

Operadores aritméticos - tipos numéricos

Os tipos numéricos (inteiro e ponto-flutuante) suportam os operadores aritméticos $+$, $-$, $/$ e $^$. Por exemplo, declaramos duas variáveis $a = 2$ e $b = 3$.

```
>> a = 3;  
>> b = 2;  
>> c = a + b  
c = 5;  
>> c = a^2  
c = 4  
>> c = a^b  
c = 8  
>> c = b/a  
c = 1.5
```

Operadores aritméticos - *strings* e *arrays* de caracteres

Aqui podemos explicitar, por meio de exemplos, a diferença entre uma *string* e um *array* de caracteres.

```
>> a = "Curso ";  
>> b = "MATLAB 2019";  
>> c = a + b;  
c = Curso MATLAB 2019  
c = 2*c  
c = Curso Curso
```

Experimente fazer o mesmo com um *array* de caracteres.

- Na soma, você vai receber uma mensagem de erro do *prompt*. Pois você tentou efetuar uma soma de matriz com tamanhos diferentes.
- Na segunda operação a saída de *c* tem um resultado estranho.

A diferença consiste que o operador aritmético `+` está definido no objeto *string* como o operador de concatenação. Para o tipo *caracter* é o operador de soma aritmética.

Precedência de operadores

Os operadores aritméticos no MATLAB possuem ordem de precedência, i.e., a ordem em que a computação será efetuada em uma expressão. Por exemplo

```
>> a = (6 + 3)^0.5 + 3/2  
a = 4.5
```

O caracter especial `()` é o que possui maior ordem de precedência, i.e., a expressão dentro dos parênteses será avaliada antes de qualquer outra. Em seguida o operador que possui a segunda maior precedência é o de exponenciação, depois o de multiplicação/divisão e por último de soma e subtração.

Precedência	Operador
1	()
2	^
3	*, /
4	+, -

>> (6 + 3)^0.5 + 3/2

Pode ser lido como:

- 1 () — some 6 + 3,
- 2 ^ — eleve o resultado a 0.5 (raiz quadrada),
- 3 + — some 1.5. Pois a divisão tem precedência maior que soma e subtração.

Vetores e Matrizes

Um vetor de N elementos é uma matriz de apenas uma linha com N colunas.

Um vetor qualquer A pode ser representado matricialmente como uma matrix $A_{1 \times N}$.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \end{bmatrix}$$

Quando declaramos uma variável, no MATLAB, seu valor é gravado em uma região de memória. Podemos acessar o valor gravado por meio de seu nome associado — no exemplo abaixo de nome a .

1	2	3	4	5
---	---	---	---	---

 Região de memória

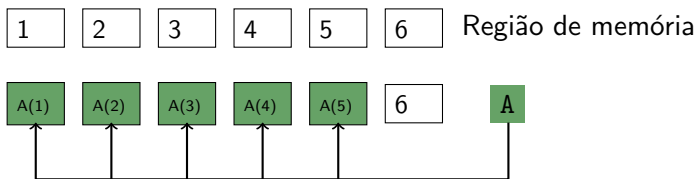
a	2	3	4	5
---	---	---	---	---

 >> $a = 3.2;$

Podemos gravar um conjunto de dados em uma região de memória e acessá-los por meio de um mesmo nome. Para isso, declaramos um vetor (*array*), com a sintaxe:

```
>> A = [3.2 5.6 -1 0 13];
```

A passa a ser, então, um vetor de inteiros.



Podemos acessar os elementos com a sintaxe

```
>> A(1)
ans = 3.2
>> A(4)
ans = 0
```

Declarando vetores

Podemos declarar vetores utilizando o caracter especial `[]`, e pode ser feito de duas formas

```
>> A = [e1 e2 ... en];
```

```
>> A = [e1,e2,...,en];
```

As duas formas são equivalentes, tanto usando vírgulas ou espaços como separadores.

Também podemos declarar vetores utilizando formas alternativas

```
>> A = Inicio:TamanhoDoSalto:Fim;
```

```
>> A = 1:2:6
```

```
A = 1 3 5
```

```
>> A = 1:4
```

```
A = 1 2 3 4
```

```
>> A = 0:0.25:0.5
```

```
A = 0 0.25 0.5
```


Ainda podemos declarar vetores utilizando as funções:

```
linspace()  
logspace()
```

Podemos calcular o número de elementos de um vetor com a função `length()`.

```
>> A = 1:200;  
>> n = length(A)  
      n = 200
```

Exercício: utilize os comandos `help` ou `doc` para acessar a documentação dessas funções.

Operador *slice* — Podemos acessar um intervalo específico do vetor utilizando o caracter especial `:`. Por exemplo,

```
V = [e1 e2 e3 e4 ... en]
```

```
>> A(e4:e6)
```

```
ans = e4 e5 e6
```

```
>> A(e1:2:e7)
```

```
>> A(e3:end -1)
```

```
>> A(:)
```

Matrizes bidimensionais são declaradas por meio da sintaxe

```
>> A = [1,2,3;3,2,1]
```

O separador ; separa as linhas

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$$

Os elementos podem ser acessados da seguinte forma:

$A(\text{Linha}, \text{Coluna})$.

```
>> A(2,3)
```

```
ans = 1
```

O tamanho da matriz pode ser obtido com a função `size()`. Ela retorna um vetor com os valores do número de linhas e de colunas, respectivamente.

```
nA = size(A)
nA = 2 3
```

Podemos obter o tamanho de uma dimensão específica, se passarmos como argumento adicional o número da dimensão desejada.

Para o número de colunas:

```
size(A,2).
```

Para o número de linhas:

```
size(A,1).
```

O operador de *slice* (:) também pode ser utilizado da mesma maneira. Por exemplo,

`A(:,n)` - seleciona toda a n linha de A

`A(n,:)` - seleciona toda a n coluna de A

`A(1:n,1:n)` - seleciona o primeiro bloco n x n de A

Vetores e matrizes bidimensionais e multidimensionais podem ser decalaradas por meio das funções

`ones(dim1, dim2, dim3, ..., dimN)`

`zeros(dim1, dim2, dim3, ..., dimN)`

`rand(dim1, dim2, dim3, ..., dimN)`

`A = ones(1,30)` %vetor com 30 elementos

`A = zeros(4)` %matriz bidimensional 4x4

`A = zeros(3,3,3)` %matriz cubica 3x3x3

Operadores aritméticos para vetores e matrizes

Alguns operadores aritméticos, vistos anteriormente, mudam seu significado quando lidamos com vetores e matrizes.

- O operador + e - tornam-se soma e subtração matricial, respectivamente.
- O operador * torna-se o produto matricial.
- O operador / torna-se a solução do sistema linear do tipo $A*x = B \rightarrow A^{-1}A * x = BA^{-1} \equiv \gg x = B/A$.
- O operador ^ torna-se potenciação matricial.

Adicionalmente temos novos operadores aritméticos

- ' é o operador de matriz transposta complexo conjugado.
- .' é o operador de matriz transposta.
- .*, ./, .^, são os operadores de multiplicação, divisão e potenciação ponto-a-ponto.

Operadores relacionais

Os operadores relacionais comparam duas expressões e possuem apenas dois tipos de retorno: **Verdadeiro** (*True*) ou **Falso** (*False*), representados pelos valores 1 e 0, respectivamente.

Os operadores relacionais são

Símbolo	Significado
==	igual a
~=	diferente de
>	maior que
>=	maior ou igual a
<	menor que
<=	menor ou igual a

```
>> a = 3;  
>> b = 5;  
>> a == b  
ans = 0  
>> a ~= b  
ans = 1  
>> a > b  
?  
>> a >= b  
?
```


Operadores lógicos

De maneira similar, os operadores lógicos retornam 1 ou 0, se certas condições forem atendidas.

Os operadores lógicos são

Símbolo	Significado
&	e
	ou

```
>> a = 5;
```

```
>> b = 0;
```

```
a & b
```

```
ans = 0
```

```
a | b
```

```
ans = 1
```

- `a & b` — Avalia `a` e `b`, se pelo menos um deles for falso, o resultado será falso.
- `a | b` — Avalia `a` e `b`, se pelo menos um deles for verdadeiro, o resultado será verdadeiro.

```
>> a = 3;  
>> b = 2;  
>> c = 0;  
>> a & b  
ans = 1  
>> a & c  
ans = 0  
>> a | c  
ans = 1  
>> d = -3.2;  
>> d & b ?  
>> d & c ?  
>> d | a ?  
>> d | c ?  
  
>> ((a >= c) & (b < d)) | (d == c)
```

Até esse ponto, escrevemos todas as operações no *prompt* de comando.

Podemos escrever todos os comandos desejados em um *script* (um arquivo de texto) que será executado sequencialmente da primeira linha até a última. Os *scripts* em MATLAB são salvos com a extensão `.m`.

Um *script* existe com o objetivo de solucionar um problema definido. A solução deste problema é realizada por meio da execução de um conjunto de passos lógicos e finitos.

Antes de efetivamente escrevermos o *script*, sempre pensamos num **algoritmo** que solucionará o problema.

Algoritmo — Conjunto de regras e operações e procedimentos, definidos e ordenados usados na solução de um problema, ou de classe de problemas, em um número finito de etapas.

Exemplo:

```
% areacirculo.m
% Esse programa calcula a area de um circulo
% e imprime na tela seu valor

raio = 2;
area = pi*raio^2;
disp(['Circulo de area: ' num2str(area)])
```

No *prompt*

```
>> areacirculo
```

Exercício:

- Implemente um algoritmo que calcula e imprime na tela o tamanho da diagonal de um retângulo.

① Primeiro dia - manhã

- Introdução ao MATLAB
- Configuração do ambiente da área de trabalho
- Variáveis, Vetores e Matrizes
- Caracteres especiais, operadores aritméticos, relacionais e lógicos
- Programação em MATLAB - *scripts*

② Primeiro dia - tarde

- Controle de fluxo - *if* e *switch*
- Laço - *for* e *while*
- Funções

③ Segundo dia - manhã

- Entrada/Saída (I/O) - usuário e arquivos
- Introdução as ferramentas básicas de visualização — *plot*.

④ Segundo dia - tarde

- Projeto final
- Dúvidas

É comum, em um algoritmo, a necessidade de se tomar uma decisão sobre quais conjuntos de operações serão executadas dependendo de uma ou mais condições.

Para exemplificar, vejamos o caso da função Heaviside (degrau).

$$H(t) = \begin{cases} 1, & \text{se } t > 0. \\ 0, & \text{se } t \leq 0. \end{cases}$$

A primeira cláusula de ramificação/seleção, que veremos é o *if*. O *if* avalia uma expressão, caso seja verdadeira, um bloco de comandos será executado.

No MATLAB possui sintaxe:

```
if expressao1
    bloco 1 de comandos a serem executados.
elseif expressao 2
    bloco 2 de comandos a serem executados.
elseif expressao 3
    bloco 3 de comandos a serem executados.
.
.
.
else
    bloco final de comandos a serem executados
    caso todas as expressoes sejam falsas

end -> palavra reservada indicando final da regioao de
ramificacao.
```

Como o *if* avalia logicamente uma expressão, é muito comum utilizar os operadores relacionais e lógicos nessas expressões. Voltando ao exemplo da função Heaviside,

$$H(t) = \begin{cases} 1, & \text{se } t > 0. \\ 0, & \text{se } t \leq 0. \end{cases}$$

Uma possível implementação dessa função em um script seria

```
%heaviside.m

t = 2;
if t > 0
    H = 1;
    disp('H(' num2str(t) ') = ' num2str(H))
else
    H = 0;
    disp('H(' num2str(t) ') = ' num2str(H))
end
```

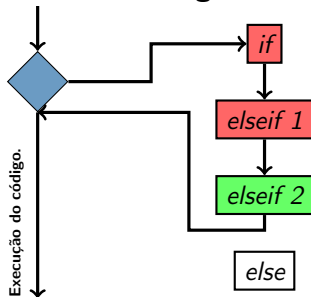
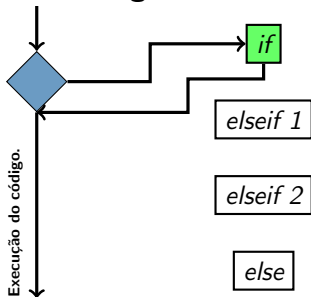

Perceba a duplicidade do comando

```
disp('H(' num2str(t) ') = ' num2str(H)).
```

Podemos calcular H baseado no valor de t e imprimir seu valor fora do bloco *if*.

Qual a diferença?

Vale destacar que quando utilizamos *if*, *elseif* e *else*, se a expressão avaliada é verdadeira, **apenas aquela expressão será executada e todas as seguintes, mesmo verdadeiras, serão ignoradas!**



Exercício: faça um *script* que compara duas variáveis numéricas, e imprime na tela a variável de maior e menor valor. Depois veja a solução para o problema de três variáveis (ifsol.m).

Podemos utilizar a cláusula *if* dentro do escopo de outros *if*, *elseif* e *else*.

```
if expr 1
    if expr 11
        ...
    else if expr 12
        ...
    end
elseif expr 2
    bloco 2 de codigo
elseif expr 3
    if expr 31
        ...
    end
    bloco 3 de codigo
else
    bloco 4 de codigo
end
```

Cláusula *switch*

A cláusula *switch* é semelhante a cláusula *if*. Mas no caso do *switch*, apenas uma expressão será avaliada.

```
switch expressao
    case expressao do caso 1
        codigo a ser executado
    case expressao do caso 2
        codigo a ser executado
    .
    .
    .
    otherwise
        codigo a ser executado
end
```

```
a = 3;
b = 2;

switch a+b
    case 1
        disp('Soma de valor 1')
    case 2
        disp('Soma de valor 2')
    ...
    case 5
        disp('Soma de valor 5')
    otherwise
        disp('Erro')
end
```

O laço *for* é uma estrutura de repetição. Possui sintaxe

```
for indice = valores
    codigo a ser executado n vezes
end
```

Um exemplo simples seria o de imprimir na tela n vezes o valor do índice (contador).

```
for i = 0:10
    disp(['i =' num2str(i)])
end
```

Qual seria o efeito de `i = 0:2:10` ?

Exercício: implemente um laço que inverta as posições do vetor
 $A = 1:15;$

Vetorização de laço tipo *SIMD*

Laços do tipo **SIMD** - **Single Instruction Multiple Data**, podem ser otimizados em MATLAB utilizando vetorização.

O exercício acima poderia ser escrito como

```
A = 1:15;
```

```
A = A(end:1);.
```

Dado um problema onde precisamos calcular um vetor da média entre a posição corrente e seus vizinhos. A solução mais intuitiva seria fazer um laço do tipo

```
A = rand(1,10);
```

```
B = A;
```

```
for i = 2:size(A,2) - 1
```

```
    B(i) = (A(i-1) + A(i) + A(i+1)) /3;
```

```
end
```

Entretanto, se analisarmos o que está sendo feito no laço, identificamos que cada posição i do vetor B está executando a **mesma operação/instrução com dados diferentes (SIMD)**! Vetorizando o laço:

$$B(2:\text{end}-1) = (A(1:\text{end}-2) + A(2:\text{end}-1) + A(3:\text{end})) / 3;$$

Para vetores de tamanhos pequenos ou *scripts* com tarefas pequenas, o tempo de execução não chega a ser uma preocupação. Entretanto, se trabalharmos com um volume grande de dados, a redução em tempo de execução pode ser muito grande. Utilizando o exemplo acima com um vetor de 10^5 elementos, a velocidade de execução chega a ser $\approx 2 \sim 3\times$ mais rápido.

Laço *while*

O laço *while*, também é uma estrutura de repetição. A expressão é avaliada a cada iteração, caso a expressão seja falsa, o laço para de ser executado.

Exemplo 1

```
while (1)
    disp('Exemplo de laço infinito.')
end
```

```
while(0)
    %Exemplo de laço que nao executa.
end
```

```
i = 0;
while(i<10)
    i = i + 1;
    disp('Esse laço executa 10 vezes')
end
```

Laços aninhados

Podemos utilizar os laços aninhados.

Para ilustrar o efeito de dois *for aninhados* o *script* abaixo

```
for i = 1:2
    for j = 1:2
        disp(['iteracao i: ' num2str(i) ...
              ', iteracao j: ' num2str(j)])
    end
end
```

produz

```
iteracao i: 1, iteracao j: 1
iteracao i: 1, iteracao j: 2
iteracao i: 2, iteracao j: 1
iteracao i: 2, iteracao j: 2
```

Funções são pequenos programas que podem ou não receber argumentos e retornar algum tipo de saída.

Um exemplo que utilizamos até agora é a função `disp()`. Essa função recebe uma variável (argumento) e imprime no *prompt* de comando seu valor (saída). Um outro tipo de função é a função `sin()` que recebe um argumento (ângulo em radianos) e retorna seu valor.

Olhando a documentação podemos ver quais tipos de variáveis, na entrada e na saída, são suportadas pela função.

`Y = sin(X)`

X Input angle in radians

scalar | vector | matrix | multidimensional array

Data Types: single | double

Complex Number Support: Yes

Y Sine of input angle

scalar | vector | matrix | multidimensional array

Funções definidas pelo usuário

Podemos criar nossas próprias funções para resolvermos problemas específicos.

Primeiro criamos um arquivo `.m` com o nome da função, por exemplo `minhafunc.m`. A primeira linha do arquivo é reservada ao cabeçalho da função, que deve ter a forma

```
function [argumentos de saida] = nome_da_funcao(argumentos de entrada)
```

em seguida vem o código a ser executado e ao final da função colocamos a palavra reservada `end`.

Nesse exemplo `minhafunc.m` não tem argumentos de entrada nem de saída, apenas imprime na tela a *string*: Essa é a minha função.

```
function minhafunc()  
    disp('Essa e a minha funcao')  
end
```


Podemos utilizar essa função desde que ela esteja **no mesmo diretório que o MATLAB está trabalhando**.

Basta digitar no *prompt* ou *script* seu nome seguido dos caracteres especiais ().

```
>> minhafunc()
```

Uma função que não recebe argumentos e apenas imprime na tela uma string não é muito útil.

Funções são ferramentas muito poderosas, pois nos habilita a escrever códigos modulares, que são de fácil manutenção. É difícil visualizar o valor das funções neste ponto, pois não escrevemos nenhum *script* grande (linhas > 1000).

Quando o código cresce muito em tamanho e produzimos um erro, identificar sua origem e solução geralmente é uma tarefa muito laboriosa.

Funções permitem escrevermos pequenos blocos funcionais de código. Apesar de executarem tarefas específicas e distintas (desacopladas), habilitam a solução um problema específico.

Um detalhe importante sobre funções é que os valores são passados para a função como cópia, i.e., quando a função é chamada, são criadas variáveis automáticas temporárias onde os valores dos argumentos são copiados.

O MATLAB não faz cópia dos argumentos se **não forem modificados dentro do escopo da função**. Esse tipo de otimização reduz o tempo de execução da função, pois a cópia de dados pode ser uma operação lenta — imagine copiar uma matriz que ocupa 2 GB de memória.

```
function [Y] = media(X)
% Exemplo de funcao que nao modifica seus argumentos
% de entrada.
% media.m
Y = 0;
for i = 1:length(X)
    Y = Y + X(i);
end
Y = Y /length(X);
end
```

```
function [Y] = mediacp(X)
% Exemplo de funcao que modifica seus argumentos
% de entrada.
% mediacp.m
    Y = 0;
    X = X/length(X); %<--- Modifica o vetor X
    for i = 1:length(X)
        Y = Y + X(i);
    end
end
```

Exercício: monte um script que retorna a menor distância entre os pontos de uma matriz de posições x e y e um ponto definido pelo usuário.

$$D = \sqrt{(x - x_0)^2 + (y - y_0)^2}$$

Funções úteis do MATLAB:

- `sqrt()` - raiz quadrada
- `min()` - retorna o valor mínimo de um *array*

① Primeiro dia - **manhã**

- Introdução ao MATLAB
- Configuração do ambiente da área de trabalho
- Variáveis, Vetores e Matrizes
- Caracteres especiais, operadores aritméticos, relacionais e lógicos
- Programação em MATLAB - *scripts*

② Primeiro dia - **tarde**

- Controle de fluxo - *if* e *switch*
- Laço - *for* e *while*
- Funções

③ Segundo dia - **manhã**

- Entrada/Saída (I/O) - usuário e arquivos
- Introdução as ferramentas básicas de visualização — *plot*.

④ Segundo dia - **tarde**

- Projeto final
- Dúvidas

Para receber um valor do usuário usamos a função `input()`. Essa função recebe um argumento obrigatório e uma *flag* opcional que sinaliza se o tipo da entrada é texto (*string*).

Possui sintaxe

```
A = input('Digite um numero')  
B = input('Digite uma string', 's')
```

Exercício: implemente um *script* que pede por três valores e imprime na tela o maior e o menor.

A função `load` é uma função de alto nível que lê arquivos da extensão `.mat` ou tipo ASCII (*American Standard Code for Information Interchange*)

```
load(nome_arquivo,variaveis)
load(nome_arquivo,'-ascii')
```


Entrada baixo nível - arquivos

Para lermos arquivos de texto (que contém caracteres) primeiro abrimos o arquivo com a função `fopen()`.

```
fileID = fopen(nome_arquivo,permissao)
```

Essa função aceita pelo menos dois argumentos, uma *string* que contém o nome do arquivo e uma *flag* indicando a permissão do arquivo.

<i>flag</i>	Permissão
'r'	Abre arquivo para leitura.
'w'	Abre ou cria arquivo para escrita.
'a'	Abre ou cria arquivo para escrita. Escreve ao final do arquivo.
'w+'	Abre ou cria arquivo para leitura e escrita.

Em seguida utilizamos um laço *while* com a função `fgetl(idArquivo)` e fechamos o arquivo com a função `fclose()`.

```
%arqtxt.m
fid = fopen('meuarquivo.txt', 'r');
linha = 0;
while(~feof(fid))
    linha = fgetl(fid);
    disp(linha)
end

fclose(fid);
```

Para ler arquivos no formato binário, só precisamos utilizar a função `fread()`

```
A = fread(idArquivo,tamanhoA,precisao,pulo)
```

```
%arqbin.m
```

```
fid = fopen('meuarquivo.bin', 'r');
```

```
A = fread(fid,[3,3], 'double')
```

Estivemos utilizando a função `disp(X)` até agora que é a função de alto nível de saída para o usuário.

A função `save(nome_arquivo,variaveis,fmt)`, é uma função de alto nível semelhante à função `load()`. `save()` salva as variáveis da área de trabalho em um arquivo com a extensão `.mat` ou no formato ASCII.

```
A = rand(3);  
b = 'Curso MATLAB 2019';  
save('saida.mat');  
save('saidaascii.txt','-ascii');  
save('saida1.mat','A');
```

Saída baixo nível - arquivos tipo binário

Para escrevermos arquivos do tipo binário utilizamos a função `fwrite(idArquivo,A,precisao,pulo)`. Lembrando que precisamos abrir um arquivo com a permissão de escrita.

```
%saidabinario.m  
fid = fopen('saidabin.bin','w');  
A = rand(3);  
fwrite(fid,A,'double');  
fclose(fid);
```

A função `plot(X,Y,LineSpec)` cria um gráfico 2D da função $Y(X)$ em uma nova janela.

```
t = -pi:0.001:pi;  
f = sin(t);  
plot(t,f);  
%plot(t,sin(t)); <-- maneira alternativa
```

Os gráficos possuem várias características, como: título, etiqueta dos eixos, tamanho dos eixos, cor da linha, espessura da linha, tipo da linha e etc. Veja a documentação da função para mais detalhes.

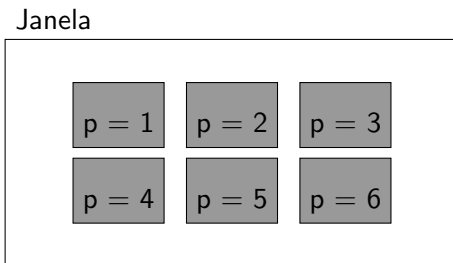
```
t = -pi:0.001:pi;
f = sin(t);
plot(t,f,'r');           %'r' -> cor da linha (red)
title('sen(t)')          %titulo
xlabel('t(rad)')          %etiqueta do eixo x
ylabel('Amplitude')      %etiqueta do eixo y
```

Cada chamada da função plot, abrirá uma nova janela e desenhará o gráfico nela. Podemos criar uma janela ou seleccionar com o comando `figure(i)`. Por exemplo, o *script* abaixo

```
figure(2)
t = -pi:0.001:pi;
f = sin(t);
plot(t,f,'r');           %'r' -> cor da linha (red)
title('sen(t)')          %titulo
xlabel('t(rad)')          %etiqueta do eixo x
ylabel('Amplitude')      %etiqueta do eixo y
```

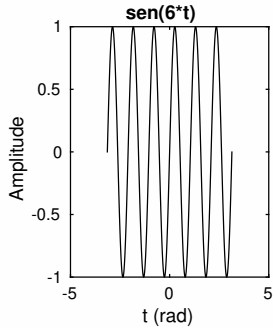
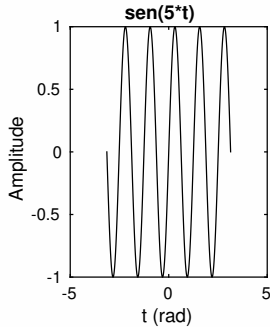
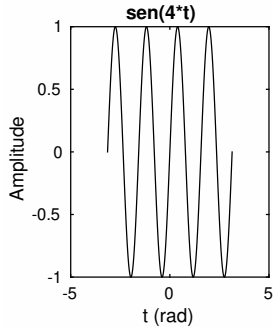
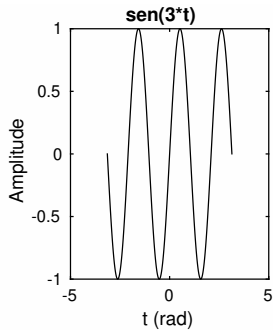
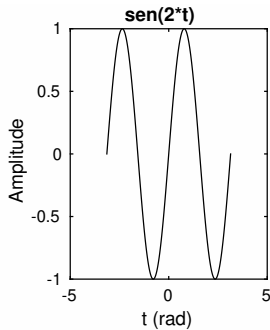
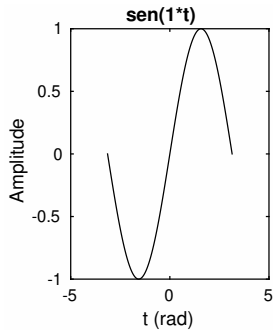
cria um gráfico na janela de título Figure 2.

Também é possível criar vários gráficos em uma mesma janela, com a função `subplot(M,N,p)`, onde M , N e p são o número de linhas, colunas e a posição do gráfico na matriz, respectivamente. Seja $M = 2$, $N = 3$, as posições p são da forma











































































A posição p pode ser obtida com a equação $p = (m - 1) * N + n$, onde m e n são os índices do gráfico na matriz $M \times N$.

```
t = -pi:0.001:pi;
figure(1)
for i=1:6
    f = sin(i*t);
    subplot(2,3,i)
    plot(t,f,'r')
    title(['sen(' num2str(i) '*t)'])
    xlabel('t(rad)')
    ylabel('Amplitude')
end
```



Existem várias outras funções para confecção de gráficos.

Line Plots	Data Distribution Plots	Discrete Data Plots	Geographic Plots	Polar Plots	Contour Plots	Vector Fields	Surface and Mesh Plots	Volume Visualization	Animation
									
									
									
									
									
									
									
									
									
									
									
									

https://www.mathworks.com/help/matlab/creating_plots/types-of-matlab-plots.html

Saída de gráficos em arquivos de imagem.

Para salvarmos a janela com o gráfico em um arquivo de imagem utilizamos a função `saveas(fig,filename,formattype)`.

No exemplo anterior:

```
t = -pi:0.001:pi;
figure(1)
for i=1:6
    f = sin(i*t);
    subplot(2,3,i)
    plot(t,f,'r')
    title(['sen(' num2str(i) '*t)'])
    xlabel('t(rad)')
    ylabel('Amplitude')
end
saveas(figure(1),'senos','jpeg')
```

Podemos usar a função `gcf` como o primeiro argumento da função `saveas()`, no lugar de `figure(1)`. A função `gcf` retorna um objeto que contém as informações da janela atual (nome, posição, tamanho e etc.).

Projeto Final

Suponha que você recebeu um arquivo binário de um sensor. Além disso você recebeu algumas informações extras sobre o sensor e o experimento:

Número de amostras	6283
Tipo de variável	double
Taxa de amostragem	0.5 ms

Além disso foi fornecida a informação que o sensor falhou em algumas medidas, e as medidas não estão amostradas em um intervalo de tempo constante.

Por algum motivo, você precisa calcular a derivada numérica desse sinal e gerar uma imagem com o dado e sua derivada. Pense em um algoritmo que leia esse arquivo, interpole os dados em uma malha regular e compute sua derivada numérica. Após isso, escreva um *script* que execute esta tarefa.

Dicas para o projeto final

A equação que gera o sinal

$$S(t) = 0.72 \cos(6(t - 1.5))^2 e^{-2\pi^2(t-1.5)^2}$$

$$\frac{dS(t)}{dt} = -(9 \cos(6(t - 1.5)) \sin(6(t - 1.5)) - 4\pi^2(t - 1.5))e^{-2\pi^2(t-1.5)^2}$$

Interpolação linear

$$S(t) = S(t_0) \left(1 - \frac{t - t_0}{t_1 - t_0}\right) + S(t_1) \left(\frac{t - t_0}{t_1 - t_0}\right)$$

Derivada numérica por diferenças finitas centradas

$$\frac{dS(t)}{dt} \approx \frac{1}{2} \frac{S(t + \Delta t) - S(t - \Delta t)}{\Delta t}$$