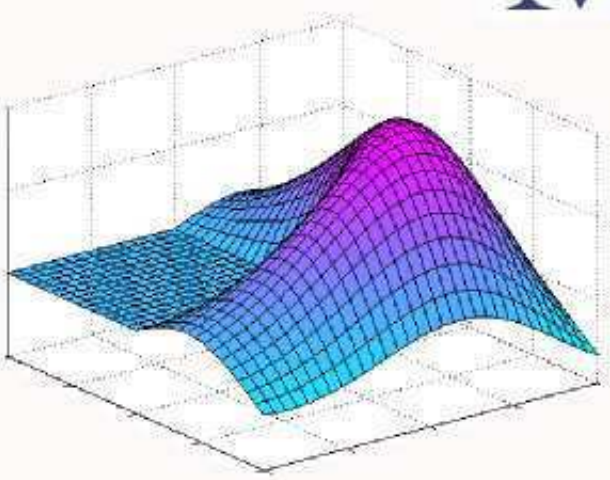
Introdução Ao MATLAB



Uma abordagem prática para desenvolver ferramentas essenciais para pesquisas e projetos.

Introdução Ao MATLAB

Uma abordagem prática para desenvolver ferramentas essenciais para pesquisas e projetos.

1ª Edição

Material distribuído por PET-ELÉTRICA UFF

Visite nosso site www.peteletrica.uff.br e participe dos nossos cursos e projetos.

Responsável: André Rangel Vieira

Dúvidas e sugestões: peteletrica@vm.uff.br

Setembro 2012

Sumário

Introdução

O MATLAB é um software interativo de alto desempenho para aplicação em pesquisas e no desenvolvimento técnico-científico de projetos. Ele atua como um ambiente interativo para a computação numérica, desenvolvimento de algoritmos e análise e visualização de dados. Ele possui compatibilidade com diversos formatos de arquivos, como o *xls* para Excel, que possibilitam a migração e o tratamento de informações vindas destes meios.

O ambiente do MATLAB é dividido em vários produtos nativos como o Simulink, que contém outras abordagens para desenvolvimento de simulações ou protótipos. O elemento fundamental de programação desse software é uma matriz que não requer dimensionamento ou definição por parte do usuário, conduzindo o desenvolvedor a um tratamento simplificado de operações envolvendo vetores e matrizes, uma vantagem que não existe em muitos outros softwares do mercado.

Em suma, estudar essa ferramenta se faz necessário para aprimorar qualquer pesquisa e projeto em todas as áreas de engenharia, ciências exatas, econômicas e para todos os níveis de estudantes. Por fim, o objetivo dessas notas de aula é de resumir o funcionamento e algumas características de programação para aqueles que desejam utilizar seus recursos para fundamentar seus trabalhos acadêmicos e profissionais.

1. Operações Básicas

As operações matemáticas ou lógicas básicas podem ser feitas dentro do *prompt* do MATLAB, que é o veículo de processamento de informações e impressão de dados nativos para este software. Ele se encontra no painel de entrada do programa e tem o nome de *Command Window.* É identificado pela seta dupla '>>'. Caso a janela não esteja evidente na interface, selecione *Desktop* na barra de menu e procure pela opção *Command Window* .

A primeira função, e uma das mais importantes para se usar, é a *help.* Através dela é possível encontrar e saber sobre todos os arquivos, funções nativas e *toolboxes.* Para muitos autores e profissionais, o *help* é o livro mais completo. Exemplo:

>> help nomedafunção (para saber sobre uma função qualquer)

>> help elfun (exibe todas as funções implementadas no software)

>> help help (exibe todas as opções de ajuda)

Para atribuir um valor a uma variável, utiliza-se a posição direita do sinal de igualdade para o valor e a esquerda para o nome da variável:

variável = valor

Exemplo 1.1:

```
>> meunum = 6

meunum =

   6

>> meunum = meunum + 1

meunum =

   7
```

Os nomes das variáveis devem seguir algumas regras:

- Começar por letra, pode conter números e '_';
- O MATLAB diferencia letras maiúsculas;
- Tomar cuidado com palavras-chaves;
- Nomes mnemônicos.

Alguns comandos úteis para obter dados no *Command Window:*

who e *whos :* Mostram variáveis que estão na memória do software;

clear: apaga as variáveis já definidas;

clc: apaga as atividades no prompt;

lookfor 'assunto': acha todos os tópicos relacionados ao assunto.

1.1. Tipos de Variáveis

Uma variável pode ser:

Número real - single ou double;

Número inteiro – int8, int16, int32 e int64;

Caractere ou string – char;

Binária – logical;

1.2. Operadores e constantes:

Operadores	
+ adição	== igualdade
- subtração	~= diferença
* multiplicação	< menor que
/ divisão	> maior que
^ exponenciação	<= menor ou igual
... continua em outra linha	>= maior ou igual
&& 'e'	~ negação
|| 'ou'	xor 'ou' exclusivo

Tabela 1: Operadores lógicos

Constantes:

pi = 3,14159...

i = -1

j = -1

inf = infinito.

NaN = não é um número.

1.3. Funções Úteis

O MATLAB contém muitas funções nativas que podem ajudar o usuário a fazer cálculos e encurtar o código do programador. Alguns exemplos a seguir:

fix(x)- arredonda para o inteiro mais próximo de zero;

floor(x)- arredonda na direção - x;

ceil(x)-arredonda na direção +x;

round(x)- arredonda para o inteiro mais próximo;

rem(x,y) - resto após a divisão de x/y;

sign(x) – retorna '1' se x>0, '0' para x = 0 e '-1' se x<0;

abs(x) – valor absoluto de x;

fator(x) – retorna um vetor contendo os fatores primos de x;

Para saber mais sobre elas, pesquise nos tópicos de ajuda do programa ou use o comando *help elfun.* O terminal listará todas as funções elementares.

1.4. Vetores e matrizes

Para criar vetores, podemos:

Assinalar variáveis do mesmo tipo entre colchetes, seguidas ou não de vírgulas:

Exemplo 1.2: Criação de vetores.

```
>> v1 = [1 2 3 4]

    v1 =

        1    2    3    4

>> v2 = [1; 2; 3]

v2 =

       1

       2

       3
```

Usar o operador ' : ':

```
>> v3 = 1:4

v3 =

   1    2    3    4

>> v4 = 1 : 2 : 8      (primeiro : passo : fim )
```

v4 =

1 3 5 7

>>v5 = linspace(1,10,5)

ans=

1.0000 3.2500 5.5000 7.7500 10.0000

Referência e modificação de elementos:

Em algumas linguagens como o C, inicia-se o primeiro elemento de um vetor com o índice 0. Entretanto, o MATLAB entende o primeiro elemento com índice '1' em relação à linha e coluna.

Dentro do vetor ou matriz, os elementos podem ser chamados da seguinte forma:

	(1)	(2)	(3)	(4)
Vetor =	Val1	Val2	Val3	Val4

Tabela 2: Representação dos índices de um vetor.

Por índice i do vetor: >> Vetor(i)

Por linha i e coluna j da matriz: >> Vetor(i , j)

Exemplo 1.3: Referência e mudança de elementos

>> v5 = 1:2:9

v5 =

1 3 5 7 9

>> v5(2) = 6

v5 =

1 6 5 7 9

Podemos fazer referência a mais de um elemento por vez:

>> b = v5([1 3 4]) (se refere ao 1º, 3º e 4º elemento)

b =

1 5 7

>> b1 = v5(1:4)

b1 =

1 6 5 7

Inicializando uma matriz:

```
>> v6 = [1 2; 3 4]

v6 =

    1    2

    3    4
```

O operador ‘:’ podem ser usados para referenciar uma linha ou coluna inteira:

```
>> v6( :, 1)

ans =

    1

    3
```

O apóstrofo é um operador para transposição de vetores e matrizes:

```
>> v1'

ans =

    1

    2

    3

    4
```

```
>> v5(end)   (comando para exibir o último elemento da fila )

ans =

   9
```

1.4.1. Geradores de Matrizes Especiais

Existe um grupo particular de matrizes e vetores que pode ser gerado através de funções nativas do software:

zeros – gera matriz ou vetor nulo;

ones – gera matriz ou vetor com elementos igual a 1;

rand – matriz com elementos aleatórios entre 0 e 1;

randi – matriz com elementos inteiros aleatórios

eye – matriz identidade.

1.4.2. Operações com Vetores

As operações entre variáveis multidimensionais, como vetores e matrizes, podem ser feitas dentro do prompt de comando sem que precise de algum código para executar iterações entre filas. Os detalhes podem encontrados com o comando *help arith.*

Exemplo 1.4: Operações com vetores.

```
>> a = [1 2 3]; b = [6 5 4];

>> a + b

ans =

   7    7    7

>> a - b

ans =

  -5   -3   -1

>> [a, b]

ans =

   1    2    3    6    5    4
```

As demais operações algébricas devem ser indicadas com o '. ' em frente ao sinal de operação para levar em consideração o elemento e não as regras de dimensões de vetores e matrizes. Contudo, deve haver correspondência de posição entre os elementos em cada matriz :

```
>> a.*b

ans =

   6   10   12

>> a.^b

ans =

   1   32   81

>> 3*a

ans =

   3    6    9

>> a/3

ans =
```

0.3333 0.6667 1.0000 (arredondamento)

Os comandos anteriores são usados nas operações com funções vetoriais:

Exemplo 1.5: Utilizando vetores em funções.

Seja $f(x) = x^3/(1+x^2)$, definida para $x \in \mathbf{R}$. Podemos aplicar vetores a esta função desde que usemos o operador ' . ' para escrever a função:

```
>> x = 0:.2:1;
>> y = x. ^3./(1 + x.^2);
```

1.5. Funções length, size e numel:

- *length* – Retorna o número de elementos de um vetor ou a maior dimensão de uma matriz;
- size – Retorna o número de linhas e colunas da matriz;
- numel – Retorna o total de números de elementos em vetores ou matrizes.

Exemplo 1.6: Dimensões de matrizes.

```
>> vec = -2:1
vec =
   -2   -1    0    1
>> length(vec)
ans =
    4
>> mat = [-2:1;0:3]
mat =
   -2   -1    0    1
    0    1    2    3
>> [lin col] = size(mat)
lin =
    2
col =
    4
>> numel(mat)
ans =
    8
```

É possível executar operações lógicas entre vetores e matrizes para achar elementos ou exibir aqueles que se encaixam em alguma propriedade.

Exemplos 1.7: Operações lógicas entre vetores.

```
>> A = [2 5; -2 3]; B = [1 -4; 6 -1];
>> A > 2
```

```
ans =
    0    1
    0    1
>> A > B
ans =
    1    1
    0    1
>> find(A < 1)
ans =
    2
>> sum(A)
ans =
    0    8
>> sum(sum(A))
ans =
    8
```

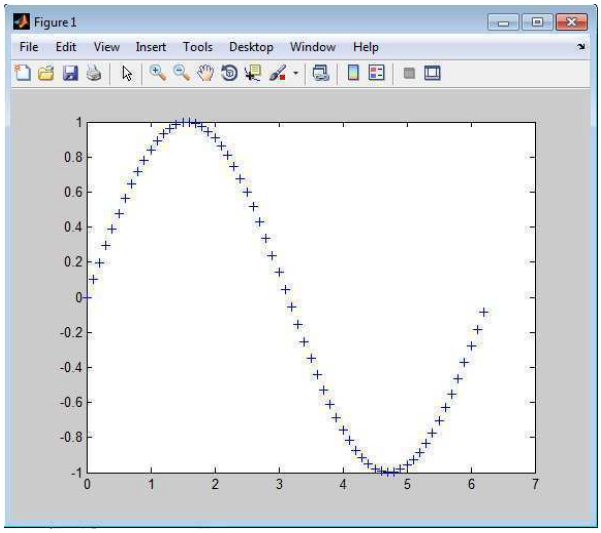


Figura 1: gráfico usando a função plot

1.6. Gráficos

O MATLAB possui muitas funções com propriedades que auxiliam a confecção de gráficos em 2D e 3D. A mais simples entre elas é a *plot.*

Sintaxe: *plot(x, f(x))*

Exemplo 1.8: Gráfico usando plot.

```
>> x = 0:.1:2*pi;
>> y = sin(x);
>> plot(x, y, '+b')
```

O terceiro argumento dessa função indica as características dos pontos que serão marcados no gráfico. Todos os detalhes de como utilizar essa função podem ser acessados com o comando *help plot*.

O comando *hold on* permite que vários gráficos sucessivos possam ser feitos nos mesmos pares de eixos. Já *hold off* cancela esta possibilidade.

A expressão *grid on* quadricula o seu gráfico.

Existem outros tipos de funções para gráficos em 2D, algumas delas são listadas a seguir:

- *hist(vec)* - produz histogramas com os dados do vetor *vec;*
- *stairs(vec)* - gráficos do tipo escada com os elementos de vec;
- *pie e pie3* – gráficos formados por setores circulares;
- *subplot* – comando que possibilita compor vários gráficos em uma mesma janela. Saiba mais através de *help subplot.*

- *line – funciona exatamente como a função plot, não é preciso usar o comando hold on* para expor vários gráficos na mesma janela.
- mesh(X,Y,Z) – cria uma superfície com um gráfico em 3D parametrizado por X,Y e Z. A cor é proporcional ao comprimento de Z da região.

O comando *clear figure* fecha todas as janelas abertas.

1.6.1. Eixos e Títulos

Os eixos de cada gráfico podem ser escolhidos pelo usuário através de *axis:*

Sintaxe: *axis([xmin xmax ymin ymax])*

Os títulos dos eixos são escritos através de *xlabel, ylabel e title:*

Sintaxe: *xlabel('texto') ylabel('texto') title('texto')*

help graph2d e help graph3d para mais informações.

1.7. Comando *save* e *load*

O comando *save* armazena variáveis do *workspace* no MATLAB. Caso nenhum formato seja especificado, o software salva no formato 'mat'.

Sintaxes:

save('arquivo') – armazena todas as variáveis do workspace no formato 'mat'.

save('arquivo', 'variaveis') – armazena apenas as variáveis especificadas.

save('arquivo', … , '-append') – acrescenta as variáveis relacionadas em um arquivo existente.

save('arquivo', … , formato) – armazena as variáveis em um formato específico: 'mat' ou '-ascii'.

save 'arquivo' … - outra forma de escrever a função.

O comando *load* também apresenta várias formas e pode armazenar dados de um arquivo em variáveis de vários tipos para manipulação e referência de dados.

Sintaxes:

load('arquivo') – armazena o arquivo citado no *workspace*.

load('arquivo','variáveis') – armazena apenas as variáveis citadas do arquivo.

load 'arquivo' – forma de referência sem parênteses.

2. Programando

O código em MATLAB pode ser feito dentro de seu editor ou de qualquer outro desenvolvedor de IDEs compatível. Caso o editor não esteja visível no painel inicial do software, ele é habilitado marcando o item *editor* dentro de *Desktop*, encontrado na barra de menu acima.

2.1. Programas e Scripts

O MATLAB e outros softwares similares possuem uma forma diferente de enxergar o código fonte proposto pelo desenvolvedor. Ele se encaixa na categoria de interpretadores e não de compiladores. Isso significa que o código fonte é lido e executado diretamente linha após linha pelo programa. Essa característica permite mais mobilidade para o desenvolvedor ao escrever o script, como os códigos são chamados, e também ao identificar erros existentes durante a sua execução.

Os tipos de terminação para os programas executados em MATLAB podem ser: m (scripts), dat (arquivos com elementos ASCII), txt (arquivos do tipo texto) e mat (arquivo binário para variáveis).

Exemplo 2.1: um script simples.

```
% calcula área do círculo

raio = 5;

area = pi *(raio^2);
```

O comando *type nomedoscript* mostra o conteúdo do script no *Command Window.*

2.2. Entrada e Saída de Dados (Parte 1)

A função mais simples para entrada de dados é *input:*

```
>> var = input('informação do programador');
```

Ela interage com o usuário através do terminal do software. Sua configuração padrão é para receber números reais. Ao receber vetores de caracteres (*strings*), é preciso completar com:

```
>> var = input('informação do programador', 's');
```

Outras funções são *fprintf* e *disp:*

```
>>fprintf('o valor de x e %d. Pode ter certeza. ', 4^3);

O valor de x e 64. Pode ter certeza.
```

Ao utilizar *fprintf* posso alterar o número de caracteres exibidos no terminal. Por exemplo:

%6.2f -> exibe um total de 6 caracteres de números reais com 2 espaços decimais.

% d inteiros
%f ponto flutuante
%c caracteres
%s strings
\n pular a linha

Tabela 3: argumentos da função fprintf.

O comando *disp* exibe um string ou variável no terminal:

Sintaxe: disp('Informacao do usuario');

Exemplo 2.2: *Script* simples.

```
% Este script calcula a área do círculo e pede ao usuário que entre com o raio.

raio = input(' Digite o raio em cm :  ');

% calculo da área.

area = pi * (raio ^2);

%imprime todas as variáveis.

fprintf( ' para um circulo de raio %.2f, a area e %.2f\n' , raio, area )
```

2.3. Funções (Parte 1)

Funções implementadas pelo usuário devem ser escritas com a palavra-chave *function* no cabeçalho.

Sintaxe: function arg_saida = nomefun (arg_entrada)

O desenvolvedor deve estar atento para que o nome do arquivo salvo seja igual ao nome da função implementada.

Exemplo 2.3: Função implementada pelo usuário.

```
function [ f ] = userf2(x)
% função para ser testada em métodos numéricos.
f = x - 2*sin(x.^2);
```

Para acessar o conteúdo:

```
>> userf2(2)

ans =

   3.5136
```

Outra opção para enviar vários argumentos para uma função é utilizar o comando *feval:*

Sintaxe: *y = feval(fun, a, b, c ,d, ...);*

Os tipos de funções podem ser:

- Funções que calculam e retornam um valor;
- Funções que calculam e retornam mais de um valor;
- Funções que apenas cumprem com uma tarefa, sem retornar valor.

Exemplo 2.4: Corpo da função.

function [s1, s2, s3, ...]=nome(v1, v2, v3, ...)

```
% comentários e definições
Sequências de comandos;
```

Exemplo 2.5: Função para cálculo de área

```
function [area circ] = areacirc( rad )
% calcula a area e o perimetro
% de um circulo
area = pi * rad .* rad;
circ = 2 *pi* rad;
```

Exemplo 2.6: Chamando uma função dentro de um script.

```
% script calculareа.m que chama a função
% areacirc  para calcular area e o perimetro.
raio = input('raio do circulo:  ');
[ area per ] = areacirc( raio );
fprintf('Circulo de raio %.1f ,\n', raio )
fprintf(' a area e %.1f e perimetro %.1f ... \n', area, per)
```

Exemplo 2.7: Função que recebe o numero total de segundos e converte em horas, minutos e segundos.

```
function [hora min sec] = tempohms( seg )
% converte segundos em horas, minutos e
% segundos.
hora = floor( seg / 3600 );
restoseg = rem(seg, 3600 );
min = floor( restoseg/60 );
seg = rem(restoseg, 60);
```

Exemplo 2.8: Função que recebe dados e executa uma tarefa.

```
function imprime( a, b )
```

```
% exemplo de função que
% executa uma tarefa.
fprintf(‘ o primeiro e %.1f  e ...
o segundo e %.1f \n’, a, b)
```

Exemplo 2.9: Função que executa uma tarefa.

```
function imprimeqq ()
% Imprime um número
% aleatorio.
fprintf(‘ o numero e %.1f\n’,rand)
```

2.4. Subfunções

As Subfunções são funções que são implementadas dentro de outras ou abaixo delas. Seus nomes não participam do nome do arquivo salvo.

Exemplo 2.10: Subfunção dentro de uma função principal.

```
function [ saida1 ] =nome1(entrada1)
        % conteúdo
        function [ saida2 ] = nome2(entrada2)
                %conteúdo
        end
end
```

3. Instruções para Seleção e Iteração

Em MATLAB existem duas estruturas que permitem escolhas: *if e switch*. As funções para iteração são o *for* e *while.*

3.1. Instruções IF – Else

A instrução *if* é escrita da seguinte forma:

Exemplo 3.1: A instrução *if.*

```
If  condição
    códigos;
end
```

O argumento *condição* é uma sentença lógica que é considerada verdadeira se ela tem elementos não nulos (uma variável nula ou matriz nula é interpretada como falsa).

Exemplo 3.2:

```
>> num = -4;
>> if num < 0
    num = abs( num);
end
num =
        4
```

Exemplo 3.3: Script usando *if.*

```
% sqrtcomif.m
% Entrar com um número e sai a sua raiz quadrada.
 num =  input('digite um numero:  ');
 If num < 0
    num = abs(num);
  end
fprintf('A raiz do numero %.2f e %.2f \n',  num,sqrt(num))
```

Para escolher entre várias opções, podemos aninhar a instrução *if* e utilizá-la com o seu complemento *else:*

Exemplo 3.4: Instrução *if* aninhada e utilização do *else.*

```
 If condição              If condição
    ação 1;                  ação 1;
     if condição 2       else
        ação 2;              ação 2;
     end                 end
 end
```

Exemplo 3.5: Utilização de *if* para implementar uma função.

Seja *y = f(x)* tal que:

y = 1 ,para x < -1;
y = x² ,para -1 ≤ x ≤ -2;
y = 4 , para x > 2 .

```
If  x < -1
    y = 1;
end
 if  x > = -1 && x < = 2
       y = x^2;
 end
If  x > 2
   y = 4;
end
```

Exemplo 3.6: Calcula a área de um círculo

```
% checkraio.m
% Calcula a área de um circulo e verifica erros do usuário
raio = input('digite o raio: ');
If raio <= 0
    fprintf('Valor %.2f nao e um raio valido \n', raio)
else
    area = calculare a(raio);
    fprintf(' Para o raio %.2f, ' , raio)
    fprintf('a area e %.f \n', area)
end
```

3.1.1. Instrução *elseif*

Para escolher entre duas ou mais ações a instrução *elseif* pode ser usada para substituir estruturas aninhadas de *if-else*. Um detalhe importante é que esse comando analisa a condição e só executa se ela for verdadeira e todas as outras falsas. O comando *elseif* não precisa de um fechamento *end* como o *if.*

Exemplo 3.7*:* instrução *elseif.*

```
If condição 1
   ação 1;
elseif condição 2
    ação 2;
elseif condição 3
    ação 3;
% etc.
else
   ação n;
end
```

Exemplo 3.8: Estruturas aninhadas de *if* e *elseif.*

```
If  x < -1
   y = 1;
else
    % x precisa ser >= -1
    if   x >= -1 && x <= 2
        y = x^2;
    else
        % x precisa ser > 2
         y = 4;
     end
end
If  x < 1
   y = 1;
elseif  x >= -1 && x<= 2
   y = x^2;
else
   y = 4;
end
```

Exemplo 3.9: Função para encontrar tipo de dado.

```
function tiposaida = achatipo( entrada )
% Função que determina se o argumento é escalar, vetor ou matriz.
[lin col] = size(entrada);
If  lin == 1 && col == 1
     tiposaida = 'escalar';
elseif  lin == 1 || col == 1
      tiposaida = 'vetor';
else
      tiposaida = 'matriz';
end
```

3.2. Switch

A instrução *switch* analisa uma variável entre várias opções dentro da expressão, esta variável pode ser numérica ou um vetor de caracteres. Ela deve ser escrita do seguinte modo:

Exemplo 3.10: A instrução *switch*

```
switch expressão
     case exp 1
         ação 1;
      case exp 2
          ação 2;
      otherwise
          ação n;
 end
```

3.2.1. A função 'menu'

A sentença *menu* é uma função que gera uma interface com uma lista de itens programados pelo usuário:

Exemplo 3.11: Função *menu* para escolher uma opção.

```
% pizza.m
% Escolhe o tipo de pizza que o usuário quer com
% a função menu
escolha = menu ( 'Escolha:', ' 4 queijos, 'Marguerita', 'Palmito' );
switch escolha
   case 1
      disp('Sai uma 4 queijos!')
   case 2
      disp('Sai uma marguerita!')
   case 3
      disp('Sai uma palmito!')
   otherwise
    disp('Sem pizza hoje!')
end
```

3.3. A Função 'is'

Existem muitas funções incluídas no MATLAB que testam a veracidade de uma variável e elas são identificadas pelo prefixo *is* em seu nome.

Exemplo 3.12: Funções do tipo *is*.

Retorna 1 para caracteres.

```
>> isletter('h')

ans =

      1
```

Retorna 1 para variável vazia.

```
>> evec = [];

>>isempty(evec)

ans =

      1
```

Pode ainda ser utilizada para verificar se o usuário digitou algo ou não.

```
>> caract = input(' Digite uma letra : ', 's');

Digite uma letra:

>> isempty(caract)

ans =

      1
```

3.4. Instrução *for*

O *for* inicia uma iteração com o número de repetições definidas pelo usuário. Deve ser declarada do seguinte modo:

Exemplo 3.13: A sentença *for.*

```
for contador = amplitude
      ação
end
```

Exemplo 3.14: (a) instrução de impressão com o *for;* (b) função somatório de 1 à n; (c) produto de elementos do vetor ;(d) forma eficiente de implementá-las.

(a)

```
for i = 1 : 5
   fprintf('%d\n', i )
end
```

(b)

```
%exemplo :  soma de 1 até n
function resultado = somatorio(n)
resultado = 0;
for i = 1 : n
   resultado = resultado + i;
end
```

(c)

```
%exemplo com produto
function output = vecprod(vetor)
output = 1;
for i = 1 : length(vetor)
  output = output*vec(i);
end
```

(d)

```
>> sum([ 5 4 9 ])
ans =
      18
>> prod([ 5 4 9 ])
ans =
      180
```

3.5. A Instrução *while*

A *while* repete um bloco de ações até que uma sentença lógica do seu argumento seja falsa. Caso a código entre em um ciclo infinito de iterações, aperte ctrl + c.

Exemplo 3.15:

while expressão

    ações;

end

É possível combinar essas ações com os comandos *break,* que termina a iteração antes da validação da condição, e *continue,* que interrompe a iteração e continua na próxima.

Exemplo 3.16:

```
 function facsaida = facmaior(num)
% função que acha fatorial maior
% que o valor dado.
i = 0;  % contador de iteração
fac = 1;
while fac < = num
      i = i + 1;
      fac = fac*i;
end
facsaida = fac;
```

Exemplo 3.17: (a) *script* para ler um arquivo e achar o valor;(b) modo eficiente de implementá-lo.

(a)

```
% achavalor.m
% Leitura de um arquivo e mostra um gráfico com os
% valores. Dados do arquivo: 3.1 11 5.2 8.9 -99 4.4
load exp.dat
i = 1;
while  exp( i ) ~= -99
       vec( i ) = exp( i );
       i = i + 1;
end
plot(vec, 'ko')
```

(b)

```
%achavalorfind.m
load exp.dat
onde = find( exp == -99);
vec = exp(1 : onde – 1);
plot( vec, 'ko' )
```

3.6. Pré-alocação de um Vetor

Para ilustrar, vamos abordar um exemplo com as funções *cumsum* e *cumprod.*

Exemplo 3.18: usando as funções *cumsum* e *cumprod.*

```
>> v = [ 5 9 4 ];
>> cumsum (  v )
ans =
      5   14   18
>> cumprod( v )
ans =
       5  45   180
```

Existem dois métodos de implementar essas funções: utilizar um vetor vazio para inicializar a variável ou um vetor nulo. Já que o seu algoritimo é parecido, vamos analisar *cumsum*.

Exemplo 3.19: (a) método utilizando um vetor vazio e ir acrescentando valores ao vetor durante a iteração;(b) inicializar as variáveis que serão usadas com uma matriz nula.

(a)

```
function vecsaida = vecsoma( v)
% exemplo: imitação de cumsum
vecsaida = [ ];
somavec = 0;
for i = 1 : length( v )
somavec = somavec + vec( i );
vecsaida = [ vecsaida somavec ];
end
```

(b)

```
function saida =  vecsomaii ( v )
% pre-alocação de memória
saida = zeros( size ( v ) );
somavec  =  0;
for j = 1 : length( v )
    somavec = somavec + v( j );
    saida( j ) = somavec;
end
```

Ao seguir a segunda maneira, o *script* usa menos recursos do editor para interpretar o código, visto que, a variável vazia muda de dimensões a cada iteração da instrução *for*. Além disso, a complexidade das ações executadas pelo editor se converte em mais tempo gasto para processar o código. Esse tempo pode ser medido com o comando *tic* e *toc.*

3.7. Verificação de Erros

Os tipos de erros podem ser:

- Sintaxe – falta de alguma letra ou pontuação;

- Execução – erros durante a execução do programa;
- Operações Lógicas – sentenças lógicas incompletas ou erradas.

Entre os tipos de erros citados, os mais difíceis de serem localizados são lógicos porque não existe indicação deles no terminal. Porém, MATLAB dispõe algumas funções que ajudam a identificar erros de diversos tipos.

echo – Exibe no terminal cada linha que é executada e o seu resultado seguinte.

Sintaxe: *echo nomedafunção on/off*

Try/catch – Inicia dois blocos com os códigos suspeitos de erros. O bloco *try* e outro bloco verificador do erro, iniciado por *catch*. O bloco *try* contém a parte do *script* que deve ser analisado. O editor passará a executar o bloco verificador de erros *catch* se for detectado algum erro *try.* O programador , por sua vez, faz um código com uma sequência de passos para identificar o erro e exibir no *prompt* o seu resultado com o comando *rethrow*.

Sintaxe:

```
try

  %código

catch me

% código para apontar o  erro

  rethrow(me)

end
```

Para mais detalhes e outras funções que identificam erros, veja *help debug.*

3.8. Variáveis Permanentes

Normalmente, uma variável fica alocada na memória enquanto um *script* é executado. Vamos tratar agora de variáveis que permanecem na memória após o fim de um programa. Podem ser *persistent* ou *global.*

Sintaxe : *persistent* var1 var2 var3

Definição: Variáveis que são *locais* para funções em que são declaradas. Elas continuam na memória até o arquivo no qual foram definidas ser apagado. Na primeira vez de sua definição, elas são inicializadas com matriz vazia.

Sintaxe: *global* var1 var2 var3

Definição: Variável que são de escopo global para todas as funções em que são definidas e tem o seu valor comum repassado para elas, permitindo a modificação. São inicializadas com a matriz vazia.

Todas essas variáveis devem ser definidas no começo do *script* e após a palavra *function* em funções.

4. Vetores de Caracteres

As sequências de caracteres podem interpretadas como uma variável ou um vetor em que cada posição é ocupada por um caractere. Estes vetores podem ser criados como:

Exemplo 4.1: declarações de *strings.*

```
>> palavra = 'gato';
```

Para ler um *string* digitado pelo usuário, usa-se o 's' como complemento da função *input:*

```
>> strvar = input('Digite uma palavra: ', 's ')
```

Operações básicas :

Operações básicas no prompt:

```
>> lentgh('cat')

ans =

     3

>>length(' ')

ans =

     1

>> length('')

ans =

     0

>> palavra = 'Oi';

>>palavra(1)

ans =

O

>> palavra'

ans=

O

i
```

As matrizes de *strings* podem ser criadas da mesma forma que os números, contudo, é preciso verificar a quantidade de caracteres dos elementos.

Exemplo 4.2: Matrizes de *strings.*

```
>> mat = [ ‘Olaa’ ; ‘ voce’ ]

mat=

Olaa

voce

>> size(mat)

ans =

    2    4      (matriz 2x4)

>> primeira = ‘ônibus’;

>> segunda = ‘carro’;

>> união = [primeira segunda]

ans =

onibuscarro
```

Função strcat e strvcar – A função strcat concatena *strings* horizontalmente e strvcat o faz verticalmente. Elas ajustam seus argumentos colocando ou tirando espaços em branco, deixados após os caracteres, para ajustar os strings.

Exemplo 4.3: Comparação entre *strings*

```
>> strcat( primeira, segunda )

ans =

onibuscarro

>> strvcat( primeira, segunda )

ans =

onibus

carro
```

A função *char* converte um vetor que contém inteiros não negativos em códigos ASCII ou concatena *strings* verticalmente. Do mesmo modo, a função *double* converte o *string* para o seu número representante ASCII.

Exemplo 4.4: Uso da função char.

```
>>  char( primeira, segunda )

ans =

onibus

carro
```

A função *blanks* cria um número *n* de caracteres estabelecidos pelo usuário.

Exemplo 4.5: função *blank.*

```
>> [primeiro blanks(4) segundo]

ans =

onibus    carro
```

O comando *disp(blanks(n))* cria n espaços do cursor no terminal. É uma ferramenta útil para usar em layout de impressão de dados.

A função *sprintf* funciona como a *fprintf* só que ela cria um *string* ao invés de apenas uma saída de dados.

Exemplo 4.6: função *sprintf.*

```
>> x = sprintf('meu nome e Andre')

x =

meu nome e Andre
```

Outra forma de aproveitar essas funções é rotulando gráficos:

Exemplo 4.7: Nomeando gráficos.

```
>> dado = [ 1 2.2 2.9 3.4 4.2 5.0 ];

>> save expdado dado – ascii
```

O script exp1.m usa esses dados para fazer um gráfico de dispersão.

```
%script exp1.m que lê dados de um experimento e faz o gráfico com um

%índice para ele.

load expdado.dat

expe = expedado(1);

dado = expedado(2:end);

 plot(dado, ‘ ko ’)

title( sprintf(‘experimento %d’), expe)
```

Associando *sprintf* junto com *input:*

Exemplo 4.8: uso de *input* e *sprintf.*

```
>> nome = input(‘Nome do usuário: ’, ‘s’);

Nome do usuário: Andre

>> perg = sprintf(‘%s, seu # de matricula : ’, nome);

>> matricula = input(perg);
```

Andre, seu # de matricula: 31138103

matricula =

31138103

O MATLAB tem duas funções que editam letras maiúsculas e minúsculas: *lower e upper* (verifique).

Comparação de strings: função *strcmp e strncmp*. A strncmp compara apenas as n primeiras letras e ignora o resto:

Sintaxe: strncmp(palavra1,palavra2,n)

Exemplo 4.9: uso de *strcmp.*

>> palavra1 = 'gato';

>>palavra2 = 'Gato';

>>strcmp(palavra1,palavra2)

ans =

0

Para comparação sem distinção de letras maiúsculas e minúsculas, use *strcmpi.*

4.2. Trocando e Separando Strings

findstr – recebe duas sequências de caracteres como argumento de entrada e acha o número de ocorrências da menor sequência na maior e retorna o índice desses caracteres (sem ordem de declaração).

Exemplo 4.10: uso de *findstr.*

>>findstr('abcde', 'd')

ans =

4

strrep – encontra todas as ocorrências de uma string menor em uma maior e troca por outra string.

Exemplo 4.11: uso de *strrep*

>> strrep('qwert', 'r', 'x')

ans =

qwext

strtok - quebra a sequência de caracteres em duas partes. A primeira parte é até o primeiro espaço em branco, não incluindo ele. A segunda parte inclui o espaço e segue até o fim.

Sintaxe:[token resto] = strtok(string)

>> sentenca = ‘como vai’;

>> [token resto] = strtok(sentena)

token =

como

resto =

vai

Prática: Como contar os espaços em branco em strings?

4.3. Função ‘is’ para Strings

Exemplo 4.12: uso das funções (a)*islette, (b) isspace e (c)ischar.*

(a)
isletter- retorna 1 para letras do alfabeto.

```
>> isletter(‘a’)
ans =
      1
>> isletter(‘EK3DG’)
ans =
      1   1   0   1   1
```

(b)
isspace- retorna 1 para espaço em branco.

```
>>isspace(‘a b’)
ans =
      0   1   0
```

(c)
ischar- retorna 1 para vetor de caracteres.

4.4. Conversão Entre Números e Caracteres.

- *int2str*- Conversão de inteiros para caracteres.
- *num2str*- Números reais para string.
- *str2num*- Caracteres para um double.

Exemplo 4.13: Conversão entre números e caracteres.

>> str2num(‘123.456’)

```
ans =
      123.4560
>>num = str2num('1  2  33')
num =
       1    2   33
```

5. Estrutura de Dados

São variáveis que armazenam mais de um tipo de dado. O MATLAB apresenta dois tipos de estruturas:

- Células (*cell array*) – tipo de estrutura de dados que armazena diferentes tipos de dados. Podem ser vetores ou matrizes.
- Estruturas (*struct*) - agrupam valores que possuem uma relação lógica, mas não necessariamente do mesmo tipo. Os valores são armazenados em diferentes campos (*fields*) da estrutura. São geralmente usadas para formar um *banco de dados*.

5.1. Células (Cell Array)

Formas para se criar células:

Exemplo 5.1: Células.

```
>>  celula = { 23,  'a',  1:2:9,  'oi' }

celula =

     [ 23 ]   'a'  [1x5 double]  'oi'

>> celula2 = {23;  'a';  1:2:9;  'oi' }

celula =

    [     23]

     'a'

    [1x5 double]

    'oi'
```

Outro método é formar uma variável com espaços de memória pré-definidos e depois assinalar dados:

Exemplo 5.2: Uso da função *cell.*

```
>> celula3= cell (2,2)

celula3 =

    [ ]   [ ]

    [ ]   [ ]
```

Para referenciar e mostrar os elementos :

```
>> celula { 2 }     ( para vetores )
```

```
ans =

a

>> celula3{ 1, 2 }  = 23   ( matrizes )

celula3 =

    [ ]  [23]

    [ ]  [ ]

>> celula{3}

celula =

     1   3    5   7   9

>> celula { 3 } (4)

ans =

      7

>> celula{ 2:3 }

ans =

a

ans =

      1   3   5   7   9
```

A função *celldisp* exibe todos os elementos de uma célula. Além disso, funções como *size* e *lenght* podem ser usadas.

Uma boa aplicação para células é armazenar strings de diferentes tamanhos:

Exemplo 5.3: Aplicação para células.

```
>> nomes = { ‘Andre’,  ‘Vinicius’, ‘ Caio’ }

nomes =

     ‘Andre’     ‘Vinicius’     ‘Caio’
```

As funções *cellstr* e *char* convertem uma célula com strings em um vetor de caracteres e vice-versa:

```
>> carcteres= char(‘Oi’,  ’ate logo’);

>> cellcaracter = cellstr(caracteres)

cellcaracter =

     ‘Oi’

      ‘ate logo’
```

A função iscellstr retorna 1 para células e 0 para qualquer outro tipo.

5.2. Estruturas (Struct)

Assim como em células, elas podem ser assinaladas em variáveis (campos) ou através da função *struct.*

Supondo a situação de um supermercado, cada produto precisa: Número, preço e código;

Exemplo 5.4: Uso de *struct.*

```
>> produto = struct( 'item' , 123, ... 'custo',  19.99, 'codigo', 'g' )

produto =

    item : 123

   custo:  19.99

 codigo:  'g'
```

O comando *disp* exibe a variável *struct* no *prompt.*

5.3. Vetores de Estruturas

Uma maneira de utilizar estruturas com grande aplicação é através dos vetores de *structs*. A informação pode ser distribuída em elementos de um vetor e cada elemento tem acesso à campos com variáveis relacionadas.

Exemplo 5.4: Vetores de *structs.*

```
>> produto = struct('item', 123, 'custo', ... 19);

>> produto(2) = struct('item', 124, 'custo',... 20);

>>produto(3) = struct('item', 125, 'custo', ...

21);
```

Outra maneira de inicializar esse vetor é usando a função *repmat.*

```
>>produto = repmat(struct('item',... 126,'custo',11.11) , 1,3);
```

Através do operador ponto, é possível manipular os campos, do mesmo modo anterior:

```
>>produto(1).item

ans =

      123
```

Aplicação: banco de dados de uma escola.

5.4. Estruturas Aninhadas

São estruturas em que pelo menos um de seus elementos é uma outra estrutura.

Exemplo 5.5: Quadro de dados para estruturas.

Seguimento	
Ponto 1	Ponto 2
X Y	X Y
2 4	1 6

Tabela 4: dados de um seguimento.

Para expressar essa tabela em forma de estrutura, poderemos inicializar 'ponto1' e 'ponto 2' como estruturas dentro de uma estrutura 'seguimento':

>> seguimento = struct('ponto_1', struct('x',2,'y',4) , 'ponto_2', struct('x', 1, 'y', 6));

5.5. Entrada e Saída de Dados (Parte 2)

O MATLAB tem funções que armazenam dados em vários tipos de arquivos. Porém, para utilizar essas funções, precisamos:

- Abrir um arquivo;
- Escrever dados;
- Fechar um arquivo.

5.5.1. Abrindo e Fechando um Arquivo

A função *fopen* é usada para abertura de um documento:

Sintaxe: var1 = fopen('nome', 'protocolo')

- O nome é o nome do arquivo;
- O protocolo é um string que concede permissão para fazer outras ações como leitura ('r' que é o *default*), escrever ('w') e acrescentar ('a').

Para fechar:

Sintaxe: var2 = fclose(var1) ou var2 = fclose('all') (Para fechar todos os arquivos)

5.5.2. Leitura de Dados

Duas funções que leem dados são *fgetl* e *fgets*. Elas leem uma linha por vês do arquivo e gravam em uma *string*, por isso, os dados podem ser manipulados.

A função *feof* retorna 1 quando o arquivo termina.

Um algoritmo simples para abrir e ler um arquivo:

- Abrir um arquivo e verificar se ele foi aberto;
- Iniciar uma iteração para ler o arquivo até ele terminar (verificar);
- Para cada linha: transformar o dado em string e manipulá-lo;
- Fechar e verificar o fechamento.

Exemplo 5.6: *Script* representando um algoritmo para abertura de um arquivo.

```
fid = fopen('nome');
if fid == –1
    disp('File open not successful')
else
    while feof(fid) == 0
      %ler cada linha
       linha = fgetl('nome');
       % manipular
    end
    end
    fecha = fclose('nome');
    if fecha == 0
      disp('fechado!')
    else
      disp('mal fechado!')
    end
end
```

6. Funções (Parte 2)

Agora vamos tratar de algumas funções e técnicas mais rebuscadas para programação, entre elas estão: funções anônimas, alça-função (*function handle*) e funções de funções.

6.1. Funções Anônimas

São funções que são escritas com uma linha e que não precisam ser armazenadas em um arquivo. São ferramentas que pode simplificar bastante o código.

Sintaxe: f_alca = @ (argumentos) corpo_da_função;

A variável f_alca armazena e gerencia o conteúdo da função. O operador @ associa uma alça *(handle*) para chamar a função através da variável dada.

Exemplo 6.1: Uso do operador @.

```
>> area = @ (raio) pi* raio .^2;

>>area(4)

ans =

      50.2655

>> prtran = @ () fprintf('%.2f\n' , rand);

>> prtran( )

ans =

0.95
```

A função pode ser salva em um arquivo .m:

```
>> area = @ (raio) pi*raio.^2;

>> area

area =

   @(raio)pi*raio.^2

>> save anonimo area

>> clear

>> load anonimo

>> who

Your variables are:

area
```

6.2. Function Handle

São variáveis que armazenam o conteúdo d e funções e gerenciam as propriedades delas com o operador @:

```
>> fatorial = @factorial;
```

Um motivo para criar essas variáveis é que se torna possível assinalar funções no argumento de outras funções.

Exemplo 6.2: Função como argumento de outra função.

```
function fnexemplo( fn )

x = 1: .25 :6;

y = fn(x);

plot( x, y, 'ko' )


 >> fnexemplo(@tan)
```

Com esse exemplo, vemos que *fn* pode ser qualquer função que receba dados e que possibilite a impressão deles em um gráfico.

6.3. Variável Número de Argumentos

Até agora vimos funções com um número definido de argumentos. Contudo, é possível fazer uma função com um número não definido de argumentos tanto de entrada como de saída, usando as variáveis *varargin* e *varargout.* Essas variáveis são células para armazenar vários tipos de argumentos.

Exemplo 6.3: Uso de *varargin.*

```
function area = calcarea(varargin)

% Calcula e retorna a área de um círculo

% em cm ou mm.

% o raio é o primeiro argumento e o

% segundo nos diz qual a unidade.

n = nargin; % número de argumentos

raio = varargin{1}; % raio em cm

If n == 2

    unidade = varargin{2};

     if unidade == 'mm'

         raio = raio*10;

     end
```

```
end

area = pi * raio ^2;
```

Exemplo 6.4: Uso de varargout.

```
% Exemplo usando varargout

function [ tipo , varargout ] = tipovar(entrada)

[ lin col ] = size(entrada);

If lin= 1 && col ==1

   tipo = 's';

elseif lin==1 || col==1

   tipo = 'v';

   varargout{1} = length(entrada);

else

   tipo = 'm';

   varargout{1} = lin;

   varargout{2} = col;

end
```

6.4. Funções Aninhadas

Funções são aninhadas quando eu escrevo elas dentro de outra função. Para isso, cada função deve terminar com *end.*

Exemplo 6.5: Função aninhada.

```
function s1 = nome1(dado1)

% corpo da função com códigos...

     function s2 = nome2(dado2)

     % corpo de argumentos

    end

end
```

Qualquer variável definida no corpo da função externa pode ser usada em funções internas.

```
function saivol = vol(comp, tam, alt)

% demonstra uma função aninhada
```

```
saivol = base * alt;

    function saibase = base

    saibase = comp*tam;

    end

end
```

6.5. Recursão

Uma função se torna recursiva quando ela a chama dentro do seu corpo.

Exemplo 6.6: Função recursiva.

```
function factn = fact ( n)

% Calcula fatorial de forma recursiva

if n == 1

   fact = 1;

else

   fact = n * fact(n - 1);

end
```

Outro exemplo bem comum é uma função que recebe um *string* e imprime em uma ordem reversa.

O seu algorítimo é, por exemplo:

- Recebe o string ;
- Usa *strtok* para quebrar o *string* na primeira palavra e no resto da sentença;
- Se o resto da sentença não for vazio, chama ela mesma para passar a sentença quebrada;
- Imprime a palavra.

Exemplo 6.7: Uso de função recursiva.

```
function printpal(frase)

[ palavra, resto] = strtok(frase);

If ~isempty(resto)

    printpal( resto );

end

disp( palavra )
```

7. Bibliografia

ATTAWAY,Stormy. *MATLAB: A practical introduction to Programming and Problem Solving.2* ed.Burlington: Elsevier,Inc,2009.