Capítulo 10
Estruturas de controle de fluxo

Controle de fluxo

- O **MATLAB**, como toda linguagem de programação, possui estruturas que permitem o controle do fluxo de execução de comandos baseadas em tomadas de decisão;
- O **MATLAB** apresenta as seguintes estruturas de controle:
  - Estruturas de repetição: *for* e *while*;
  - Estruturas condicionais: *if-else-end* e *switch-case*; e
  - Blocos *try-catch*.
- Nos casos em que uma expressão lógica na estrutura tiver muitas subexpressões, apenas as subexpressões necessárias para determinar o valor lógico da expressão serão executadas. Por exemplo, se tivermos $(a \ \& \ b)$ e a for igual a $falso$ então b não será calculada.

Loop *for*

- Estrutura que permite que uma seqüência de comandos seja repetida um número fixo, predeterminado, de vezes.

Sintaxe:

```
for x = vetor
    ação
end
```

- A ação é composta por uma seqüência de um ou mais comandos.
- A variável x, variável de controle do *for*, na i-ésima iteração assume o valor da i-ésima coluna do *vetor*.

Loop *for* - exemplos

```
>> for n=1:10
       x(n)=sin(n*pi/10);
   end
>> x
x =
  Columns 1 through 5
   0.3090   0.5878   0.8090   0.9511   1.0000
  Columns 6 through 10
   0.9511   0.8090   0.5878   0.3090   0.0000

>> for n=10:-1:1
       x(n)=sin(n*pi/10);
   end
>> x
x =
  Columns 1 through 5
   0.3090   0.5878   0.8090   0.9511   1.0000
  Columns 6 through 10
   0.9511   0.8090   0.5878   0.3090   0.0000
```

Loop *for* - exemplos

- Nos exemplos anteriores 10 iterações são executadas porque o vetor de controle possui 10 colunas. A seguir usamos um vetor com apenas 1 coluna.

```
>> i=0;
>> for n=(1:10)'
       i=i+1;
       x(n) = sin(n*pi/10) % exibe x
   end
x =
  Columns 1 through 5
   0.3090   0.5878   0.8090   0.9511   1.0000
  Columns 6 through 10
   0.9511   0.8090   0.5878   0.3090   0.0000
>> disp(i)
     1
```

- O número de iterações é confirmado pelo valor final da variável i, incrementada dentro do *for*.

Loop *for*

- Reafirmando: a variável de controle do *for* recebe, a cada iteração, uma coluna do vetor da definição da estrutura, na ordem em que estas aparecem.

```
>> vetor=randperm(10)
vetor =
   8   2  10   7   4   3   6   9   5   1
>> for n = vetor
       x(n) = sin(n*pi/10);
   end
>> disp(x)
  Columns 1 through 5
   0.3090   0.5878   0.8090   0.9511   1.0000
  Columns 6 through 10
   0.9511   0.8090   0.5878   0.3090   0.0000
```

- Fazer uma atribuição, por exemplo $n = 10$, dentro do *for* para forçar a saída da estrutura não funciona (que bom!).

Loop *for* - exemplos

- A variável de controle não precisa ser um vetor linha ou coluna:

```
>> i = 1;
>> for x = rand(4,5)
       y(i) = sum(x);
       i = i+1;
   end

>> disp(y)
   1.3335   2.2560   1.6050   2.1366   2.8112

>> disp(i)
    6
```

Loop *for* - exemplos

- Podemos aninhar os comandos *for* da maneira usual.

```
>> for n=1:5
       for m= 5:-1:1
           A(m,n) = n*n+m*m;
       end
   end

>> disp(A)
     2     5    10    17    26
     5     8    13    20    29
    10    13    18    25    34
    17    20    25    32    41
    26    29    34    41    50
```

Loop *for* - observações

- O *for* deve ser evitado quando existe uma forma equivalente de resolver o mesmo problema que use vetores ou matrizes. Uma solução deste tipo é chamada de solução *vetorizada*. A solução vetorizada costuma ser mais rápida.

- Seguindo esta filosofia, o primeiro exemplo poderia ser substituído por

```
>> n = 1:10;
>> x = sin(n*pi/10);
```

Loop *for* - observações

- O último exemplo, embora menos intuitivo, poderia ser substituído por:

```
>> n = 1:5;m=1:5;
>> [nn,mm] = meshgrid(n,m)
nn =                              mm =
     1     2     3     4     5          1     1     1     1     1
     1     2     3     4     5          2     2     2     2     2
     1     2     3     4     5          3     3     3     3     3
     1     2     3     4     5          4     4     4     4     4
     1     2     3     4     5          5     5     5     5     5

>> A = nn.^2+ mm.^2
A  =
     2     5    10    17    26
     5     8    13    20    29
    10    13    18    25    34
    17    20    25    32    41
    26    29    34    41    50
```

Gerenciamento de memória

- Utilização eficiente de memória é importante para computação em geral; em **MATLAB** de forma ainda mais crítica.
- O **MATLAB** não executa nenhum gerenciamento explícito de memória. Alocação e liberação de memória recorrem às funções **C** padrão (*malloc*, *calloc*, *free*);
- Em **MATLAB**, o uso de mais memória não necessariamente implica em maior desempenho, mas, **MATLAB** opta sempre pelo uso de mais memória se isto significar melhoria no desempenho;
  - Assim, é importante que saibamos como a memória é alocada e o que pode ser feito para que não ocorram excessos de uso, nem fragmentação de memória.

Como o MATLAB aloca memória?

- Quando uma variável é criada num comando de atribuição, **MATLAB** requisita um bloco *contíguo* de memória para armazenar a variável.
- Quando a atribuição de uma variável é alterada, a memória anterior é liberada e uma nova requisição de memória é feita.
- Este processo de liberação/alocação ocorre mesmo se a nova variável ocupar a mesma quantidade de memória que a variável original.
- Quando copiamos valores para posições existentes, não há liberação/alocação de memória. Por exemplo, fazer $V(2,2) = 3$ após $V = zeros(2)$ não libera ou aloca memória porque a posição $V(2,2)$ já existe. Em particular, os escalares estão neste caso também.

Pré-alocação de memória

- Se uma variável já existir (como V acima) e atribuirmos um valor para uma posição desta variável que aumente o seu tamanho, então ocorrerá liberação/alocação. Por exemplo, se fizermos $V(4,4) = 20$.
- No nosso exemplo do comando **for**:

```
>> for n=1:10
        x(n) = sin(n*pi/10);
   end
```

Temos liberação/alocação de memória a cada iteração.

- Uma solução para isso é fazermos *pré-alocação de memória*:

```
>> x = zeros(1,10);
>> for n=1:10
     x(n) = sin(n*pi/10);
   end
```

- Um outro recurso interessante do **MATLAB**: *delayed copy*.

Loop *while*

- Estrutura de repetição usada quando não se sabe, a priori, o número de iterações que serão executadas.

Sintaxe:

while *expressão*
ação
end

- A ação é composta por uma seqüência de um ou mais comandos e somente é executada enquanto **todos** os elementos de expressão forem verdadeiros.
- Em geral, a expressão é um escalar, mas vetores também são válidos.

Loop *while* - exemplo

- Uma forma de calcular o valor *eps*.

```
>> num = 0; EPS = 1;   %evitamos redifinir eps
>> while (1+EPS) > 1
          EPS = EPS/2;
          num = num+1;
   end

>> num
num =
    53

>> EPS=2*EPS
EPS =
   2.2204e-16
```

Estrutura *if-else-end*

- Estrutura condicional em que uma seqüência de comandos é executada dependendo do resultado de uma condição lógica;
- Há três variantes no **MATLAB** de estruturas *if-else-end*:

Sintaxe:
if *expressão*
  ação
end

Sintaxe:
if *expressão*
  ação_1
else
  ação_2
end

if *expressão_1*
  ação_1
elseif *expressão_2*
  ação_2
elseif *expressão_3*
  ação_3
  .
  .
  .
else
  ação
end

Break e *continue* - exemplo

- Exemplo do cálculo de *EPS* modificado.

```
>> EPS=1;
>> for num=1:1000
      EPS = EPS/2;
      if (1+EPS)¡=1
         EPS = EPS*2
         break
      end
   end

EPS =
   2.2204e-16

>> disp(num)
    53
```

```
>> EPS=1;
>> for num=1:1000
      EPS = EPS/2;
      if (1+EPS)>1
          continue
      end
      EPS = EPS*2
      break
   end

EPS =
   2.2204e-16

>> disp(num)
    53
```

- *Break*: vai para o próximo comando fora do loop.
- *Continue*: passa para a instrução final do loop, ignorando todos os comandos entre *continue* e o *end*.

Estrutura *switch-case*

- Estrutura condicional baseada em diferentes testes de igualdade de um mesmo argumento.

```
switch expressão
    case opção_1
            ação_1
    case {opção_21, . . ., opção_2k}
            ação_2
    . . .
    otherwise
            ação
end
```

- No máximo, uma das ações é executada;
- A expressão deve ser um escalar ou uma string;
- Se expressão for um escalar, então o teste feito é *expressão==opção*. Se expressão for uma string, então o teste feito é *strcmp(expressão,opção)*;
- Quando, para alguma ação, as opções estiverem em um vetor de células, basta que a igualdade seja atingida para uma das opções;
- O **otherwise** (opcional) é executado quando todos os testes anteriores foram falsos.

Switch-case - exemplo

- O exemplo está em um *M-file* de nome *ExSwitch.m*

ExSwitch.m

```
x = input('Digite um real: ');
unidades = input('Digite a unidade: ');
switch unidades
case {'polegadas','pol'}
   y=x*2.54;
case {'pes','p'}
   y = x*2.54/12;
case {'metros','m'}
   y = x/100;
case {'milimetros','mm'}
   y =x;
otherwise
   disp(['Unidade desconhecida: ' unidades])
   y = NaN;
end
```

Uma execução:

```
>> ExSwitch
Digite um real: 12.7
Digite a unidade: 'pol'

>> disp(y)
   6.8580
```

Blocos *try-catch*

- Utilizado para controlar a execução em casos de erros inesperados;

Sintaxe:

```
try
    ação_1
catch
    ação_2
end
```

- Quando encontra um bloco *try-catch*, o **MATLAB** tenta executar os comandos que estão em *ação_1*.
- Caso algum erro seja gerado, o **MATLAB** executa os comandos que estão em *ação_2*. Além disso, a função *lasterr* retorna a string gerada pelo erro encontrado no bloco *try*.
- Se nenhum erro for encontrado, apenas os comandos do bloco *try* são executados.

Try-catch - exemplo

- Suponha o seguinte *M-file* de nome *ExTry.m*

ExTry.m

```
x= ones(4,2);
n = input('Dimensão da identidade: ');
y = 4*eye(n);
try
    z = x*y;
catch
    z = NaN;
    disp('x e y não são compatíveis')
    disp(lasterr)
end
z
```

Executando:

```
>> ExTry
z =
     4     4
     4     4
     4     4
     4     4

>> ExTry
Dimensão da identidade: 3
x e y não são compatíveis
Error using ==>  *
Inner matrix dimensions must agree.

z =
   NaN
```