Capítulo 7
Vetores de células (cell arrays)
e
estruturas

Vetores de células

- São vetores cujos conteúdos são endereços de memória. Nestes endereços de memória encontram-se os dados que, de fato, queremos manipular.
- Cada conteúdo, *apontado* por uma posição do vetor (denominada *célula*), pode armazenar qualquer tipo de dado do **MATLAB** (vetores numéricos, strings ou mesmo outro vetor de células, etc).
- *Em resumo:* Vetores de células são tabelas de apontadores para dados de tipos (potencialmente) diferentes.

Construção explícita

```
>> B = cell(2,3)
B =
     []     []     []
     []     []     []
```

Cria uma matriz B composta de células vazias, isto é, que não apontam para posição alguma de memória.

Veremos agora como criar uma matriz com algum conteúdo nas células.

Construção - *cell indexing*

```
>> A(1,1) = {reshape(1:9,3,3)};
>> A(1,2) = {2+3i};
>> A(2,1) = {'Uma string'};
>> A(2,2) = {12:-2:0}
A =
    [3x3 double]    [2.0000+ 3.0000i]
    'Uma string'          [1x7 double]
```

- *As chaves* do lado direito significam que os dados representam uma célula. O lado esquerdo, *com parêntesis*, diz que a célula estará na posição i, j do vetor de células A, como é feito com matrizes numéricas.
- Note que a definição de que A será um vetor de células se dá implicitamente, após a atribuição de $A(1,1)$, já que A não existia antes. Caso existisse, as atribuições simplesmente alterariam as células.

Construção - *content addressing*

```
>> A{1,1} = reshape(1:9,3,3);
>> A{1,2} = 2+3i;
>> A{2,1} = 'Uma string;
>> A{2,2} = {12:-2:0}
A =
    [3x3 double]    [2.0000+ 3.0000i]
    'Uma string'              1x1 cell
```

- As chaves, neste caso, estão do lado esquerdo, o que diz ao **MATLAB** que, em vez de se referir o conteúdo da posição (i, j) da matriz A se refira ao *conteúdo* apontado pela posição (i, j).
- Note que, na quarta linha, a célula é por si só um vetor de células.

cell indexing $\times$ *content adressing*

$$A(i,j) = \{x\} \qquad \text{e} \qquad A\{i,j\} = x$$

- Ambas as formas acima dizem ao **MATLAB** que armazene x no endereço de memória apontado pela posição (i,j).
- A primeira versão, $A(i,j)$ é chamada *cell indexing*.
- A segunda versão, $A\{i,j\}$, *content addressing*.

As chaves significam acesso ao conteúdo das células, e os parêntesis identificam as células sem olhar para o seu conteúdo.

Mais construção

- Podemos usar *chaves* para construir vetores de células analogamente à forma que usamos colchetes para construir vetores numéricos:
  - “,” ou “espaços” separam colunas; e
  - “;” ou “enter” separam colunas.

```
>> A = { [1 2], 'Texto'; 2+31 5
reshape(1:6,2,3), 111}

A =
    [1x2 double]    'Texto'
    [          33]  [    5]
    [2x3 double]    [  111]
```

Exibição dos conteúdos

- Considere o exemplo abaixo:

```
>> A = {reshape(1:9,3,3), 1; 'Texto', 1:2:10}

A =

    [3x3 double]    [          1]
    'Texto'         [1x5 double]
```

- O **MATLAB** mostra que A é um vetor de células, de tamanho 2×2, mas nem sempre exibe o conteúdo apontado pelas células. Basicamente, exibe o conteúdo quando for compacto.

Exibição dos conteúdos

- A função $celldisp(A)$ força o **MATLAB** a exibir os conteúdos apontados pelas células de A.

```
>> celldisp(A)

A{1,1} =
     1     4     7
     2     5     8
     3     6     9

A{2,1} =
Texto

A{1,2} =
     1

A{2,2} =

     1     3     5     7     9
```

Acesso aos conteúdos

- Uma forma simples de ter acesso ao conteúdo apontado por uma célula é usar *content addressing*. Note que isto é diferente de usar *cell indexing*.

```
>> A{2,2}                          >> A(2,2)
ans =                              ans =
    1    3    5    7    9             [1x5 double]

>> A{1,:}                          >> A(1,:)
ans =                              ans =
    1      4      7                 [3x3 double]   [1]
    2      5      8
    3      6      9

ans =
    1
```

- Veja que o conteúdo é genericamente atribuído à variável **ans** porque não há nomes específicos associados aos conteúdos.

Acesso aos conteúdos

```
>> A = {[1 2] 'Texto'; reshape(1:6,2,3) 10}
A =
    [1x2 double]    'Texto'
    [2x3 double]    [   10]

>> A(2,1)
ans =
    [2x3 double]

>> A(2)
ans =
    [2x3 double]
>> A{2,1}(2,:)
ans =
     2     4     6

>> A{2,1}{2,1}
??? Cell contents reference from a non-cell array object.
>> A(2,1)(2,1)
??? Error: () Indexing must appear last in an index expression.
```

```
>> A{2,1}
ans =
     1     3     5
     2     4     6
>> A{2,1}(2,1)
ans =
     2
>> A{2}(1,[1 2])
ans =
     1     3
```

Manipulação de vetores de células

- As técnicas para manipulação de vetores se aplicam para vetores de células. Isto é, manipulação de vetores de células é uma extensão da manipulação de vetores numéricos para um tipo diferente de vetores. Veja exemplos:

```
>> A(2,2) = {1}
A =
    []    []
    []    [1]

>> A{1,1} = [1 2 3]
A =
  [1x3 double]    []
    []    [1]
```

```
>> B = {10 'Texto'}
B =
  [10]    'Texto'

>> C = [A ; B]
C =
  [1x3 double]          []
    []    [     1]
  [      10]    'Texto'
```

Manipulação

```
>> D = C([1 3],:) % primeira e terceira linhas
D =
    [1x3 double]          []
    [         10]    'Texto'

>> D(:,2) = {30}
D =
    [1x3 double]    [30]
    [         10]    [30]

>> E = cell(2,3)
E =    []    []    []
       []    []    []

>> E = reshape(E,1,6)
E =     []    []    []    []    []    []


>> E = repmat(E,2,1)
E =     []    []    []    []    []    []
        []    []    []    []    []    []
```

Manipulação

```
>> A = ones(2,3); B = zeros(2,1); C = (3:4)';
>> F = {A B C} % cria um vetor de células
F =
    [2x3 double]    [2x1 double]    [2x1 double]

>> D = cat(2,F{:}) % = cat(2,A,B,C)
D =
     1     1     1     0     3
     1     1     1     0     4

>> E = cat(2,F(:))                    >> class(D)
E =                                   ans = double
    [2x3 double]
    [2x1 double]                      >> class(E)
    [2x1 double]                      ans = cell
```

Manipulação

```
>> A = ones(2,3); B = zeros(2,1); C = (3:4)';
>> F = {a b c};
>> [F{:}] % = [ F{1}, F{2}, F{3}]
ans =
     1     1     1     0     3
     1     1     1     0     4

>> [F{2:3}] % = [ F{2}, F{3} ]
ans =
     0     3
     0     4

>> iscell(F)
ans =
     1

>> isa(F,'cell')
ans =
     1
```

```
>> iscell(F{1})
ans =
     0

>> isa(F,'double')
ans =
     0
```

Funções úteis

- $deal$: função que permite extrair o conteúdo de várias células em variáveis separadas.

```
>> F ={ ones(2,3), zeros(2,1), (3:4)'};

>> [r,s,t] = deal(F{:}) % = (F{1},{2},{3})
r =
     1     1     1
     1     1     1
s =                       t =
     0                         3
     0                         4

>> [G{:}] = deal(F{:})
 ??? Error using ==> deal
 The number of outputs should match the number of inputs.

>> [G{1:3}] = deal(F{:})
G =
     [2x3 double]   [2x1 double]   [2x1 double]
```

Funções úteis - *num2cell*

- Esta função tem como entrada um vetor arbitrário e cria um vetor de células preenchido com os elementos deste vetor.

```
>> A = rand(2,4)
A =    0.9501    0.6068    0.8913    0.4565
       0.2311    0.4860    0.7621    0.0185

>> num2cell(A)  % ans{i,j} = a(i,j)
ans =
   [0.9501]    [0.6068]    [0.8913]    [0.4565]
   [0.2311]    [0.4860]    [0.7621]    [0.0185]

>> num2cell(A,1)  % ans{i} = a(:,i)
ans =
[2x1 double] [2x1 double] [2x1 double] [2x1 double]

>> num2cell(A,2)  % ans{i} = a(i,:)
ans =
   [1x4 double]
   [1x4 double]
```

Funções úteis - $cellfun$

- Função que permite aplicar algumas funções a todas as células de um vetor de células de uma só vez.

```
>> A = {ones(2,3), zeros(2,1) 1+2i}
A =
   [2x3 double] [2x1 double] [1.0000+ 2.0000i]

>> cellfun('isreal',A)
ans =
     1     1     0

>> cellfun('length',A)
ans =
     3     2     1
```

- A variedade de uso é grande: olhar o help *on line*.

Structures (estruturas)

Estruturas

- É um tipo de dado que permite o armazenamento de elementos de tipos (potencialmente) diferentes.
- Cada elemento de uma estrutura é identificado por um nome (*campo*). Onde os vetores de células usam chaves, estruturas usam a notação *.campo*.
- São uma tabela de apontadores que apontam para regiões de memória onde elementos ficam efetivamente armazenados.
- Podem ter um número arbitrário de dimensões, mas os escalares e vetores unidimensionais são os mais comuns.

```
>> S = struct('Nome',{'Ricardo'},'Idade',{46})
S =
     Nome: 'Ricardo'
    Idade: 46
```

Construção de estruturas

- Por atribuição direta:

```
>> circle.radius = 2.5;
>> circle.center = [0 1];
>> circle.linestyle = '--';
>> circle.color = 'red'

circle =
        radius: 2.5000
        center: [0 1]
     linestyle: '--'
         color: 'red'

>> size(circle)
ans =
     1     1
```

- *circle* é uma estrutura escalar (1×1).
- Nomes de campos seguem mesmas regras de variáveis.

Construção de estruturas

```
>> circle(2).radius = 3;
>> circle(2).center = [-1 0];
>> circle(2).linestyle = '...';
>> circle(2).color = 'blue'

circle =
1x2 struct array with fields:
    radius
    center
    linestyle
    color

>> size(circle)
ans =
     1     2
```

- Agora, *circle* é uma estrutura com 2 elementos.

Construção de estruturas

- Cada campo é uma variável do **MATLAB** . Os mesmos campos, em registros diferentes de uma mesma estrutura, podem ser de tipos diferentes. Por exemplo:

```
>> circle(1).radius = 3;
>> circle(2).radius = 'tres';

>> class(circle(1).radius)
ans =
double

>> class(circle(2).radius)
ans =
char
```

Manipulação

- Acresentar um novo campo a uma estrutura segue a filosofia do **MATLAB** : acrescentar em um dos registros faz com que o **MATLAB** expanda para todos os registros.

```
>> circle(1).filled = 'yes'
circle =
1x2 struct array with fields:
    radius
    center
    linestyle
    color
    filled

>> circle.filled % exibição deste campo
% em todos os registros
ans =
yes

ans =
     []
```

Manipulação

- Podemos combinar e indexar estruturas de forma análoga a vetores numéricos e de células, ressalvando-se que, no caso de combinação de estruturas, os campos das estruturas combinadas devem ser exatamente os mesmos.

```
>> cad.name='Ricardo';
>> cad.age=46
cad =
     name: 'Ricardo'
      age: 46

>> cad1.name='Francisco';
>> cad1.age=1
cad1 =
      name: 'Francisco'
       age: 1

>> F = [ cad cad1 ]
F =
1x2 struct array with fields:
    name
    age

>> cad2.nome = 'Clara'; cad2.age = 0; F = [ F cad2 ]
??? Error using ==> horzcat
CAT arguments are not consistent in structure field names
```

Manipulação

- Exemplos de endereçamentos:

```
>> circle
circle =
1x2 struct array ...
    radius
    center
    linestyle
    color
    filled

>> circle(3) = circle(2)
circle =
1x3 struct array ...
    radius
    center
    linestyle
    color
    filled

>> [circle(1:2) circle(3) circle(1)]
ans =
1x4 struct array with fields:
    radius
    center
    linestyle
    color
    filled
```

reshape e *repmat*

- $reshape$: Pode ser usada, mas em geral não é muito útil porque o mais comum é trabalhar com estruturas unidimensionais.
- $repmat$: É mais comumente usada para iniciar os campos de uma estrutura com algum valor padrão.

```
>> square.width = 1; square.height = 1
square =
     width: 1
    height: 1

>> S = repmat(square, 1, 5)
S =
1x5 struct array with fields:
    width
    height
```

Manipulação

- Já vimos que podemos ter acesso aos dados de uma estrutura fazendo referência ao registro e campo corretos.
- A idéia é hierárquica, como em programação tradicional: localizamos o nome da estrutura, o índice que segue localiza o registro, o nome que se segue é o campo ao qual queremos ter acesso. Se este for um vetor, prosseguimos da mesma forma hierárquica, até que cheguemos ao dado que queremos recuperar.

```
>> circle(1)                    >> circle(1).filled
ans =  radius: 2.5000           ans = yes
       center: [0 1]
    linestyle: '--'             >> circle(1).filled(1)
        color: 'red'            ans = y
       filled: 'yes'

>> circle(1).filled(2:end)
ans = es
```

Manipulação

- Exemplo com listas separadas por vírgulas.

```
>> circle.center
ans =                ans =                ans =
     0     1              -1     0                -1     0

>> cent = cat(1,circle.center)
% = cat(1, circle(1).center, circle(2).center,
% circle(3).center)

cent =
     0     1
    -1     0
    -1     0

>> cent = cat(2,circle.center)
% = cat(2, circle(1).center, circle(2).center,
% circle(3).center)
cent =
     0     1    -1     0    -1     0
```

Funções úteis

- $deal$: Usada para extrair o conteúdo de diversos registros em variáveis separadas.

```
>> [c1, c2, c3] = deal(circle.color)
c1 = red   c2 = blue   c3 = blue

>> [c1, c3] =deal(circle([1 3]).color)
c1 = red   c3 = blue

>> deal(circle([1 3]).color)
Error using ==> deal
The number of outputs should match
the number of inputs.

>> [circle.radius] = deal(1, 5, 10);
>> circle.radius
ans = 1    ans = 5   ans = 10
```

Funções úteis

- $fieldnames(S)$: Retorna os nomes dos campos da estrutura S em um vetor de células, onde cada célula é uma string.
- $S = rmfield(S, field)$: Remove o campo $field$ da estrutura S.
- $S = rmfield(S, FIELDS)$: Generalização do caso anterior. Supondo que FIELD seja um vetor de células, onde cada célula é uma string com nomes de campos válidos, ou um vetor de caracteres, remove os campos especificados neste vetor da estrutura S
- Há outras funções para manipulação de campos($setfield$ e $getfield$). Maiores detalhes veja *help on line* ou bibliografia indicada.

Exemplo de $rmfield$

```
>> S = struct('campo1', {1},'campo2', ...
{2}, 'campo3', {3}, 'campo4', {4})

>> S = rmfield(S, 'campo4')
S =
    campo1: 1
    campo2: 2
    campo3: 3

>> campos = {'campo1', 'campo2'}
campos =
    'campo1'    'campo2'
>> S = rmfield(S, campos)
S =
    campo3: 3
```

Funções úteis

- Há outras funções para manipulação de estruturas equivalentes às versões para manipulação de vetores de células:
  - $isstruct$
  - $isfield$
  - $cell2struc.$
  - $struct2cell.$
- Maiores detalhes: *help on line*.