Topic 16: Cell Arrays

Elementary data structures

- So far the only data structure we have seen is the array - a collection of identically typed data objects (typically numbers) stored sequentially in memory.
- Often one wants to work with a collection of data objects that are of different types.
- We may want to link character strings, numbers, or arrays of numbers together into data structures.
- We may also want to use mechanisms other than arrays for handling collections of data objects.

Cell Arrays (Text: Section 7.2.)

- A cell array is a special kind of array in Matlab.
- Instead of each location in the array holding a number, each element of a cell array is an address, or pointer, or reference, to another data structure.
- These "pointed to" data structures can be of any type.
- A cell array is like an address book that groups together information relevant to a problem.
An Example

• The addresses are the addresses in memory where the relevant data objects are stored. For example:

```
>> a = [1, 2 3, 4];  % A 2x2 matrix.
>> b = 'hello';      % A character string.
>> c = [7, 8, 9];    % A 1x3 matrix.
>> d = {a, b, c};    % d is a cell array that holds
                   % the addresses of, or "points to", a, b, and c.
```

• Notice curly brackets {}.

Cell arrays vs. normal arrays

• The key difference between cell arrays and normal arrays is that cell arrays contain pointers to data structures instead of data.

```
{1 2}
{3 4}

'dello'

[7 8 9]
```

Curly brackets

• Cell arrays use curly brackets ({})) for selecting and displaying contents of cells.

• Curly brackets indicate to Matlab that we want to refer to the data at the address stored in the cell array, rather than the address itself.

• Curly brackets display the contents of the data structure contained at that position in the cell array.

```
>> d{1}  % Access data stored at first address
        % in d using curly brackets.
ans =
   1  2
   3  4
```

Round brackets

• Round brackets display "what" is pointed to by the position in the cell array.

```
>> d{1}  % d{1} tells you that the first address
        % in d points to a 2x2 array of doubles.
        % You never have access to the actual
        % address itself.
ans =
   2x2 double
```

• But how the "what" is displayed depends on what it is!

```
>> d(1) = 24
>> d(1)
ans =
   [24]  % not [1x1 double]
Curly vs. round brackets (cont.)

>> d(2) % Access data at second address
    % stored in d.
ans =
    hello

>> d(2) % Display what is residing at the
    % second address of the cell array d.
ans =
    'hello'

• Matlab displays the data structures in each element of a cell
  array in a condensed form that limits each data structure to a
  single line.
  – If the entire data structure can be displayed on the single line, it is.
  – Otherwise, a summary is displayed.

Constructing cell arrays

• You can build cell arrays in the same way you build ordinary arrays,
  using curly brackets ({}) instead of square brackets ([]).

>> a = { [1 2], 'Jill'; [5 6 7; 8 9 10], 1 } % Construct a 2x2 cell array containing four
  % elements: a 1x2 array, a string, a 2x3 array,
  % and a scalar.

a =
    [1x2 double]    'Jill'
    [2x3 double]    [   1]

>> a{2, 1} % Access data at cell{2, 1}
ans =
    5     6     7
    8     9    10

Adding extra data to a cell array

• If you add extra data, Matlab will automatically resize the cell array to
  accommodate it.
• For example:

>> a{1, 3} = 'new data' % Add an element in
    % row 1, column 3.
a =
    [1x2 double]    'Jill'    'new data'
    [2x3 double]    [   1]            []

• To add the new string, Matlab needs to add a third column to the cell
  array.
• Cell arrays are themselves just arrays - they must remain rectangular.
• Note that the element {2, 3} in the cell array is set to the empty matrix.

What happens if brackets are wrong?

• If you use the wrong kind of addressing (the wrong kind of brackets), Matlab will
  warn you with a message like this:

>> a(1,3) = 'new data' % Attempt to set the address
    % stored in cell array
    % element(1,3) to 'new data'
??? Conversion to cell from char is not possible.
Extract data from a cell array

- We can extract individual elements from data stored in a cell array by adding the normal indexing commands to the cell indexing commands.

\[
\begin{align*}
&\text{>> } c(1,1)(2) \quad \% \text{ Access element 2 of the matrix referred to by cell array element } (1,1). \\
&\text{ans = } 2 \\
&\text{>> } c(2,1)(1:2, 2:3) \quad \% \text{ Access elements from the matrix referred to by cell array element } (2,1). \\
&\text{ans = } \begin{bmatrix} 6 & 7 \\ 9 & 10 \end{bmatrix}
\end{align*}
\]

View the contents of a cell array

- You can view the contents of a cell array using the \texttt{celldisp} and \texttt{cellplot} functions. For example:

\[
\begin{align*}
&\text{>> celldisp(a) } \% \text{ Prints out the contents of each cell in the cell array } a. \\
&a{1,1} = \begin{bmatrix} 1 & 2 \end{bmatrix} \\
&a{2,1} = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix} \\
&a{1,2} = \text{Jill} \\
&a{2,2} = 1 \\
&\text{etc...}
\end{align*}
\]

Graphical representation of the cell array

- The \texttt{cellplot} function displays a graphical representation of the cell array.

\[
\begin{align*}
&\text{>> cellplot(a) } \% \text{ Displays a graphical representation of the cell array } a.
\end{align*}
\]

Memory pre-allocation for a cell array

- You can pre-allocate memory for a cell array with the \texttt{cell} function.

\[
\begin{align*}
&\text{>> c = cell(1, 5) } \% \text{ Allocates memory for a 1x5 cell array.} \\
&\text{\% Note the use of round brackets - we are calling a function to create the cell array.} \\
&\text{\% Note that this call to cell only allocates space for the 5 addresses in memory - there is no data associated with these addresses yet (all we have is an empty address book with 5 blank entries).} \\
&c = \begin{bmatrix} [] & [] & [] & [] & [] \end{bmatrix} \\
&\text{>> whos} \\
&\text{Name: } c \quad \text{Size: } 1x5 \quad \text{Bytes: } 20 \quad \text{Class: cell array} \\
&\text{The cell array uses 20 bytes of memory - 4 bytes to store each address. This will be true on a 32 bit machine.}
\end{align*}
\]
Attach data to a cell array

- Having allocated space for a cell array, we can "attach" data to the array as we wish.
- For example:
  ```matlab
  >> c{4} = 'hello';
  >> d = [11 12 13 14]; % Construct a 1x4 matrix.
  >> c{1} = d         % Attach the new matrix to c{1}.
  c =                % Originally d is a 1x4 matrix.
    [1x4 double]    [ ]    [ ]    'hello'    [ ]
  - Actually it is a **copy** of d that is attached to c{1}.
  - If d is modified or even deleted, the contents of the data referenced by c{1} will be left intact (pass-by-value).

Pros and Cons

- Advantage: eliminates the possibility of nasty side effects that could inadvertently occur should the data that is attached to a cell array be modified - the cell array has its own private copy of the data.
- Disadvantage: extra time and space to create and store copy.
- Note that similar "attaching" operations in Java or C only result in the memory address being copied - not the whole data object.
- In Java or C, because there is only one actual data object, a change in the array d will result in a change being seen in c{1}; d and c{1} are simply two names for the same thing.

A copy of the data - *not* address

```matlab
>> c(2) = c(1)           % What does this do?
>> c =                  % Originally d is a 1x4 matrix.
    [1x4 double] [1x4 double] [ ]    'hello' [ ]
>> c{1}(1) = 99;         % Modify the first element in the element 1.
>> c{1}                  % Display data referenced by cell element 1.
  ans =                  % c(1) refers to the contents, whereas c{1} refers to the address (pointer, reference)
    99 12 13 14
>> c{2}                  % Display data referenced by cell array element 2. Note that it is unchanged - it is a copy of the original value of cell array element 1.
  ans =                  % Matlab doesn't let you access the address - it jumps in and says (to quote Hal from 2001) "I'm sorry Dave, I'm afraid I can't do that. I'll let you have a copy of the contents".
    11 12 13 14
```
Cell arrays of strings

- A major use of cell arrays is storing arrays of strings.
- Why?
- Matlab does not have a special String type as such - it just stores arrays of characters.

```matlab
>> 'PeterBell' == ['P' 'e' 't' 'e' 'r' 'B' 'e' 'l' 'l']
ans =
1 1 1 1 1 1 1 1 1
>> strcmp('PeterBell', ['P' 'e' 't' 'e' 'r' 'B' 'e' 'l' 'l'])
ans =
1
```

How to store a collection of strings?

- What if you want to store a collection of strings? Eg:
  - the names and addresses in my address book
  - the CDs in my record collection
  - the parameters to a function (such as the settings for a plot)
- Our first thought might be to store them in an array...

```matlab
>> cds = ['Boss Scaggs', 'U2', 'INXS', 'Oasis']
cds =
Boss Scaggs U2 INXS Oasis
```

- To Matlab this is just concatenating an array of characters, much like

```matlab
>> [ [1 2 3], [4 5], 6 ]
ans =
1 2 3 4 5 6
```

Would vertical array help?

- Next we might try to put them in a “vertical” array so that they don’t get concatenated:

```matlab
>> cds = ['Boss Scaggs'; 'U2'; 'INXS'; 'Oasis']
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

- Matlab complains since the horizontal rows are not the same length. This would be the same as:

```matlab
>> [ [1 2 3]; [4 5]; 6 ]
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

A solution?

- Thus, if you want to store character strings in a normal array, every string has to be “padded out” with spaces so that they all have the same length.

```matlab
>> cds = ['Boss Scaggs' ; 'U2' ; 'INXS' ; 'Oasis']
cds =
Boss Scaggs
U2
INXS
Oasis
```

- The spaces are all stored (though you can’t see them in the printout) as its simply a 2D array of characters.
- Why is this the answer?

```matlab
>> strcmp(cds(1), 'Boss Scaggs')
an =
0
```
What if?

• What if you now want to add ‘Red Hot Chilli Peppers’?
• All the previous “strings” must now be padded to the new length.
• Clearly this is impractical.
• Cell arrays provide the solution.

\[\text{cds} = \{\text{’Boss Scaggs’} \quad \text{’U2’} \quad \text{’INXS’} \quad \text{’Oasis’}\}\]

Solution – using cell arrays

• The cell array just points to the individual strings (character arrays) so there is no need for them to be padded to the same length.

\[\text{>> \quad \text{strcmp(cds{1},’Boss Scaggs’)}}\]
\[\text{ans} = \quad 1\]

• Adding strings is easy...

\[\text{>> \quad \text{cds{5} = ’Red Hot Chilli Peppers’}}\]
\[\text{cds} = \quad \begin{array}{c} \text{’Boss Scaggs’} \quad \text{’U2’} \quad \text{’INXS’} \quad \text{’Oasis’} \quad \text{[1x22 char]} \end{array}\]

The cell array in the switch statement

• You may recognise the cell array from the switch statement...

\[\text{switch (str)  \quad \% \quad \text{The variable str is the variable \}}\]
\[\text{\quad \% \quad \text{that controls the switching.}}\]
\[\text{\quad \% \quad \text{str can be a scalar or a string,}}\]
\[\text{\quad \% \quad \text{but not an array.}}\]
\[\text{\quad \% \quad \text{If str matches one of these items}}\]
\[\text{\quad \text{\quad \text{\quad case \{’dog’, ’cat’, -3\}}}}\]
\[\text{\quad \text{\quad \quad \% \quad \text{then this code block will be evaluated.}}}
\[\text{\quad \text{\quad \quad \text{disp(’A pet or -3’));}}\]
\[\text{\quad \text{\quad case (1, 2, 3, 4, 5)}}\]
\[\text{\quad \text{\quad \quad \text{\quad \quad \text{\quad \text{disp(’An integer in the range 1 to 5’));}}}}\]
\[\text{\quad \text{\quad end}}\]