More advanced structures and functions

1. Introduction

The purpose of this particular tutorial is to outline various more advanced structures and functions which would have been over-complex and out of place in earlier tutorials. In particular, cell arrays and so-called structs are valuable in offering neat ways of expressing certain types of data: these will be dealt with first. After that we consider anonymous functions and handles, which are respectively useful for introducing simple one-line functions and for operations such as graph plotting.

2. Cell arrays

In Tutorial 1 we considered vectors and matrices, which are basically 1-D and 2-D arrays of numbers or letters. In fact, vectors are merely special cases of matrices, and matrices can be considered as arrays which are restricted to homogeneous rectangular arrays of numbers or letters—‘homogeneous’ meaning that all elements have to be of identical type, i.e., either numbers or letters. In addition, arrays that are not rectangular, i.e., which do not have rows or columns containing equal numbers of elements, cannot be called matrices. This design enables matrices of equal size to be combined together without significant complexity or explanation. For example, \( A = B + C \) implies that the three matrices \( A, B \) and \( C \) have equal numbers of rows and equal numbers of columns. In fact, much of what we know about standard matrix (and vector) manipulation follows the conventions used in standard mathematics. Nevertheless, there is a need for some way of expressing situations when this is not true—in particular, when arrays have rows and/or columns with unequal numbers of elements—as in the following example:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
7 & 8 & 9 & 10 &   &   \\
11 & 12 &   &   &   &   \\
13 &   &   &   &   &   \\
14 & 15 & 16 & 17 & 18 & 19 & 20 \\
\end{array}
\]

The same idea is evident when we wish to construct arrays of letters, particularly when they form normal words of different lengths:

```
morning
afternoon
evening
night
```

All this was pointed out before, in Tutorial 2, which considered the properties of strings.

To deal with generalisations of matrices to cover such cases, the concept of cell arrays was invented. To put the idea in a nutshell, cell arrays differ from matrices in being defined using curly brackets instead of normal brackets. We can understand the
concept by considering the following examples, which define the two cases illustrated above:

```matlab
>> cellsofnumbers = { 1:6; 7:10; 11, 12; 13; 14:20 }
cellsofnumbers =
    [1x6 double]
    [1x4 double]
    [1x2 double]
    [1x1 double]
    [1x7 double]
```  

```matlab
>> cellsofwords = { 'morning'; 'afternoon'; 'evening'; 'night' }
cellsofwords =
    'morning'
    'afternoon'
    'evening'
    'night'
```

Notice that although the two cell arrays are correctly defined, their displays are constrained to show quite what sort of data is in each cell rather than the all-important internal data. To show the latter, we look at each cell and then list the data within it:

```matlab
>> cellsofnumbers( 2, 1 )
anst =
    [1x4 double]

>> cellsofnumbers{ 2, 1 }
anst =
    7 8 9 10
```

Similarly:

```matlab
>> cellsofletters( 3, 1 )
anst =
    'evening'

>> cellsofletters{ 3, 1 }
anst =
    evening
```

In other words, to access a cell of a cell array, we use normal brackets—in exactly the same way that we use normal brackets to access an element of a matrix—whereas to access the contents of a cell in a cell array we have to use curly brackets. Clearly, a cell of a cell array is merely a $1 \times 1$ component of an $n \times m$ cell array and is itself a cell array. To underline this, note that to extract a $1 \times 1$ cell array we follow the cell identifier with a curly bracket address; we can then address the individual matrix elements within the cell by using round brackets:
Above, we have only illustrated how cell arrays are defined and accessed in the two cases of numeric arrays and string arrays. However, they are far more general than this might indicate—not only in the use of digits and characters but also in the lengths and sizes of the arrays, and in the ways in which they are put together. For example, a mixed cell array of this type might be defined by:

```matlab
>> mixedcells{ 'a sequence of numbers' 1:2:6; ...
'a 3×4 matrix' [1:4; 5:8; 2:5]; 'the end' }
ans =
    'a sequence of numbers'  [1×3 double]
    'a 3×4 matrix''      [3×4 double]
    'the end'
>> mixedcells{2, 2}(2, 2)
ans =
   6
```

There is evidently a slight difficulty in seeing what is present in a cell array as a whole, as neither round or curly brackets suffice to do this. The `celldisp` function was designed specially to deal with this problem. We here apply it to the above example:

```matlab
>> celldisp(mixedcells)
mixedcells(1,1) =
a sequence of numbers

mixedcells(2,1) =
a 3×4 matrix

mixedcells(3,1) =
the end

mixedcells(1,2) =
    1  3  5

mixedcells(2,2) =
    1  2  3  4
    5  6  7  8
    2  3  4  5
```
When examining new structures such as cell arrays, it is important to have ways of interrogating the data in order to ascertain whether the particular structure has been defined properly. (We first met the need for this in Tutorial 2, Section 3 in relation to string functions such as isletter and isspace.) In fact, the iscellstr operator was designed to determine whether a cell array is composed solely of strings: if it is, it will return a logical 1; otherwise it will return a logical 0. In the above case, iscellstr(mixedcells) will return the value 0, as will iscellstr(cellsofnumbers), whereas iscellstr(cellsofwords) will return the value 1.

3. Structured arrays

Real objects, be they people, vehicles or commodities of various sorts, are often complex entities, and are usually best described using a variety of properties rather than a single variable. It is convenient to collect all the relevant information about objects into fields, each describing one property, and to place them together under a single 'structure' name. For example, a vehicle structure V may have field descriptors type, make, wheels and capacity, each of which is indicated as linked to the structure name by a dot—in this case V.type, V.make, V.wheels, V.capacity. Once defined, fields will act as descriptors, and may be of any type, including string, numeric value, character or cell array. For the vehicle structure, typical examples will be:

```matlab
V.type = 'car';
V.make = 'BMW';
V.wheels = 4;
V.capacity = 1.2;
```

To summarise, structures are designed to provide ways of grouping sets of related parameters to make them easier to refer to: importantly, they are described using easy-to-understand linguistic terms—and this is particularly advantageous in reducing the possibility of transcription errors.

A structure may be defined by making lists of statements as above, but it is frequently more convenient to define it in a single statement, which in the above instance would be:

```matlab
V = struct('type', 'car', 'make', 'BMW', ...
           'wheels', 4, 'capacity', 1.2);
```

Thus, the operation struct automatically generates a structure containing the right fields and values.

Although we have defined structures in terms of a single structure name, it is natural to extend this to cover arrays of elements having the same structure name (though in this case V would have to be taken to mean vehicles (plural), and array subscripts would have to be included). For example, we might have:
V(1).type = 'car';
V(1).make = 'BMW';
V(1).wheels = 4;
V(1).capacity = 1.2;
V(2).type = 'bicycle';
V(2).make = 'Rayleigh';
V(2).wheels = 2;
V(3).type = 'van';
V(3).make = 'Rover';
V(3).wheels = 4;
V(3).capacity = 2.2;

Note that the same fields need not be relevant for all vehicles. However, all structs in a given struct array will have the same number of fields, with the same field names, and those that are not specified in the definitions will contain empty fields. In the above case, V(2).capacity is empty. Indeed, by writing V(2), we can obtain a list of all the fields for this vehicle:

```
>> V(2)
ans = struct with fields
    type: 'bicycle'
    make: 'Rayleigh'
    wheels: 2
    capacity: []
```

We can also display the contents of the whole array as follows:

```
>> for i = 1:length(V)
    disp(V(i))
end

    type: 'car'
    make: 'BMW'
    wheels: 4
    capacity: 1.2

    type: 'bicycle'
    make: 'Rayleigh'
    wheels: 2
    capacity: []

    type: 'van'
    make: 'Rover'
    wheels: 4
    capacity: 2.2
```

Once all definitions have been made, it is important to be able to interrogate structures to be sure that they are valid and that they contain the right data. The following functions can be used for this purpose. The `class` function will assert that a
given name is a struct. The `isstruct` function will return a value of 1 if its argument is a valid structure variable and 0 otherwise; similarly, the `isfield` function will return a value of 1 if a field name is a valid name for the stated struct and 0 otherwise. Finally, the `fieldnames` function will list the names of the fields for a stated struct:

```matlab
>> class(V)
ans =
    struct
>> isstruct(V)
ans =
    1
>> isfield(V, wheels)
ans =
    1
>> fieldnames(V)
ans =
    'type'
    'make'
    'wheels'
    'capacity'
```

4. **Anonymous functions and handles**

In Matlab each new function is normally defined in its own script or .m file. This is a rigorous practice which makes for clarity of programming, but in a simple case can be unduly heavy-handed. Thus, it may be useful to declare a so-called **anonymous function** 'on the fly'. To achieve this, the function must be referenced using a 'handle' embodying the '@' sign followed by a bracketed list of input arguments and then by the function body. For example, the following anonymous function \( f \) is defined by its handle:

\[
f = @(x) x .* \exp(-1 ./ x);
\]

Having defined the function, it is very simple to apply it. For example:

```matlab
g = f(2);
```

which gives \( 2*\exp(-1/2) \). Fortunately—as is standard with Matlab—this also works with input vectors, e.g.:

```matlab
g = f([1: 2: 6]);
```

which means that \( g \) ends up as the vector \([ f(1), f(3), f(5) ]\).

One of the most common uses of anonymous functions is their application in plotting. In particular, if `linspace` is used to define \( x \) as a discrete set of values, e.g., \( x = \text{linspace}(0, 1, 200) \)—which creates 200 equally spaced values from 0 to 1 inclusive—we can **plot** the resulting function simply by writing `plot(x, f(x))`. We will return to cover plotting in more detail in a later tutorial.
## 5. Index

<table>
<thead>
<tr>
<th>anonymous function, 1, 6</th>
<th>isfield, 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell array, 1</td>
<td>isstruct, 6</td>
</tr>
<tr>
<td>celldisp, 3</td>
<td>linspace, 6</td>
</tr>
<tr>
<td>class, 5</td>
<td>plot, 6</td>
</tr>
<tr>
<td>field, 4</td>
<td>struct, 1, 4</td>
</tr>
<tr>
<td>fieldnames, 6</td>
<td>structure, 4</td>
</tr>
<tr>
<td>handle, 1, 6</td>
<td>structured arrays, 4</td>
</tr>
<tr>
<td>iscellstr, 4</td>
<td></td>
</tr>
</tbody>
</table>