Data containers

A data container is a mechanism for storing and working with data, typically an array. MATLAB has 4 main array types: numeric arrays, character arrays, cell arrays, and structures. Objects are also data containers that are more than simple arrays. All of these have a role to play in working with data.

Choice of the type of container to use is a design decision—a choice among alternatives. It can have a big impact on the code. The most important criteria for the selection are clearness of the code, reliability, and performance.

In this paper I discuss these array types and show some functions that can use them. There are more functions for working with these arrays than the examples I use here, but these should provide an idea of how to work with the arrays. There also can be more arguments for these functions than the ones I use in the examples. I have tried to use the most important ones.

Executive Summary

In general use numeric arrays whenever you can. Confine your use of character arrays to short strings unless you need more speed than cell arrays can provide. Confine your use of cell arrays to strings or non-uniform data sets. Use structures to bundle multiple data types or multiple arrays in a single variable. Consider objects for more ambitious projects.

Numeric arrays

MATLAB was founded on the use of 1 and 2D numerical matrices. These have subsequently been generalized for up to 10 dimensional arrays. Not surprisingly this type of array is fast and efficient for use.

Numeric arrays are easy to construct and easy to operate on. In MATLAB many operations can work with a single variable that is an array unlike most other languages that require looping through array indices.

In addition to mathematical operations, numeric arrays are particularly good for operations using indexing. For example to find the indices of all elements meeting some criteria and do something with them:

```
explicit = [1 2 3 5 7 11]
index = explicit>5
explicit(index) = 0

index =
     0     0     0     0     1     1


explicit =
     1     2     3     5     0     0
```

There are two important limitations of numeric arrays: All of the array elements must be numeric. Also numeric arrays must be rectangular, that is the number of elements in each column must be the same, and the number of elements in each row must be the same. When the array would not be rectangular, the usual approach is to pad with NaN:

```
oneRow = ones(1,4)
padded = [explicit; [oneRow nan nan]]

padded =
     1     2     3     5     0     0
     1     1     1     1   NaN   NaN
```

A numeric array can also be treated as a concatenation of independent or somewhat independent columns. Many of the functions that work with vectors will also by default work along the columns of a matrix.

```
vertical = [1 3 5 7; 1 3.6667 6.3333 9];

sum(vertical)

ans =
    2.0000    6.6667   11.3333   16.0000
```

Usually these functions can optionally work in the horizontal direction when specified by a direction argument:

```
mean(vertical, 2)

ans =
     4
     5
```

The values of a time series work well in a numeric array. They are usually arranged to be in a column.

One very handy way to work with series that have varying lengths is to use one column (typically) as an explicit or surrogate grouping variable. This gets around the need for numeric

arrays to be uniformly rectangular while avoiding padding with NaN (not a number). The Statistics Toolbox provides some functions with explicit support for use with grouping variables. It is also relatively easy to write such code yourself.

```
group1 =
     1     1
     3     1
     5     1
     7     1


group2 =
     1     2
     2     2
     3     2
     5     2
     0     2
     0     2


grouped =
     1     1
     3     1
     5     1
     7     1
     1     2
     2     2
     3     2
     5     2
     0     2
     0     2
```

Three and four dimensional arrays can also be useful for data, for example x, y, z or x, y, z, w. Many of the basic functions can also operate on these higher dimensional arrays. In working with them, you will want to become friends with the functions permute, reshape and squeeze.

Often when working with data in arrays, you will want to use associated arrays containing indices, scales or headers. For example you can have an associated cell array:

```
vertical
vertical =
    1.0000    3.0000    5.0000    7.0000
    1.0000    3.6667    6.3333    9.0000
colHead = {'Bob', 'Ted', 'Carol', 'Alice'}
rowHead = {'Test1'; 'Test2'}
colHead =
    'Bob'    'Ted'    'Carol'    'Alice'
rowHead =
    'Test1'
    'Test2'
```

Normally these associated arrays would be used for plot labeling. With a bit of work these can be displayed as a table. Note the use of tabs to avoid counting spaces for alignment.

```
disp(sprintf('\t\t%s',colHead{:}))
```

```
disp(sprintf('%s\t%5.2f\t%5.2f\t%5.2f\t%5.2f',rowHead{1},vertical(1,:)))
disp(sprintf('%s\t%5.2f\t%5.2f\t%5.2f\t%5.2f',rowHead{2},vertical(2,:)))
```

```
              Bob   Ted Carol Alice
   Test1      1.00  3.00  5.00  7.00
   Test2      1.00  3.67  6.33  9.00
```

An important array feature of MATLAB is that the basic arithmetic operations on 1D and 2D arrays are matrix operations. This makes no difference for addition and subtraction, but a big difference for multiplication and division. You may want to use .* or ./ for most computations so that the arrays are not treated as matrices. This dot grammar is also needed for 3D and higher arrays despite the fact that matrix operations are not defined for dimensions beyond 2.

Sparse arrays

In some cases the data in a 2D numeric array has a lot of zero or NaN values. When this happens the data can be stored more compactly in a sparse array containing only the array indices and values that are not zero or NaN, together with the dimensions of the corresponding full array. MATLAB supports many standard operations that use only these elements and not the zero elements for speed. The break even in storage requirements between a sparse and a full array is about 67% density. (The density of an array is the number of nonzero elements divided by the total number of elements.) The breakeven point for processing speed depends on the process and is probably a lower percentage.

The function sparse generates a sparse array.

```
i = [1 2 4];
j = [1 3 5];
data = [15 16 17];
s = sparse(i, j, data)

s =
   (1,1)        15
   (2,3)        16
   (4,5)        17
```

The function full makes a full array from a sparse one.

```
full(s)
ans =
    15     0     0     0     0
     0     0    16     0     0
     0     0     0     0     0
     0     0     0     0    17
```

A feature of the sparse-full approach is that if there is a duplicate index pair, the corresponding data values are added. In the past this method was used to quickly compute a 2D histogram. Sparse arrays are limited to 2D.

Other low density techniques

A different approach to converting the vectors (i, j, data) to a full 2D array without declaring a sparse array is

```
f=zeros(max(i),max(j));
f(sub2ind(size(f), i, j))=data
```

In the sub2ind approach, if there is a duplicate index pair, the last data value is used. This approach is generally better than reshape because it does not require that all elements of the full array be specified.

Another way to deal with low density 2D array data is to make an nx3 array, not declare it to be sparse, and work with it in that format. MATLAB will treat it as an ordinary numeric array, but you can write your own code to treat two of the columns as indices. The grouping variable approach is one version of this.

An advantage of these last two approaches over sparse is that they can be used with more than 2 dimensions.

Logical arrays

A logical array is similar to a numeric array, but its contents can only be true or false. Logical arrays are always displayed with contents 0 or 1, but in this context the values are not the same as the numeric 0 or 1.

The function logical will convert a numeric array to a logical array. Interestingly any number other than 0 is converted to true.

```
logical([-1 0 1.1])
ans =
     1     0     1
```

The functions true and false can generate logical arrays of a desired size.

```
true(2,3)
ans =
     1     1     1
     1     1     1
```

Character arrays

Characters can be stored in 1 or 2D plain arrays, one character per element. In principle character arrays can have more than 2 dimensions, but it's not clear where this would be useful. They almost always have each text string across a row.

```
disp('Hello')
Hello

disp(['Hello' ' ' 'World'])
Hello World

disp(['Hello'; 'World'])
Hello
World
```

Since the strings may not have the same length and character arrays must be rectangular, character arrays spanning more than one line usually involve padding with blanks at the ends of rows. These arrays can be created using the char, strcat, strvcat, or cellstr functions.

```
disp(char('Hi', 'World'))
Hi
World
```

Functions that use these arrays as input must be row aware. Some examples are: sortrows, strmatch, and unique(…'rows').

```
first = char('January', 'February', 'March')
first =

January
February
March

disp(sortrows(first))
February
January
March

disp(strmatch('March', first))
     3
```

A big use of character arrays is to display numbers with text, for example:

```
nIterations = 10;
message = ['There were ' int2str(nIterations) ' iterations.']
message =
There were 10 iterations.
```

Character arrays of date strings are of particular interest. Whenever possible, dates should be used as date number arrays except for display and character based indexing operations.

Character arrays are not as frequently used as the more flexible cell arrays.

Cell arrays

Cell arrays are flexible arrays with *numbered* cells. In MATLAB they are used primarily for character strings. Major examples are text labels for graphs and lists of strings for processing by functions. Usually cell arrays are 1D. Cell arrays can have a large number of dimensions and the cells can contain numeric arrays, character arrays, structures, and even cell arrays. In the case of more than 1D, the cell array itself must be rectangular.

The cells in a single array can hold different data types and these data do not need to be a consistent size. Unfortunately this flexibility comes at a price. Operations involving cell arrays can be slower than when using more simple arrays. In particular using a cell array is usually slower than using a character array with the same contents.

Cell arrays use curly brace notation, and it can take some time to get proficient with this form.

```
c = {'Hi'; 'World'}
c =
    'Hi'
    'World'
```

```
disp(c{1})
Hi
```

In a definition the braces can be on the left hand side

```
c{1} = 'Hello'
c =
    'Hello'
    'World'
```

Or on the right hand side

```
c(1) = {'Hello'}
c =
    'Hello'
    'World'
```

If you put braces on both sides, you will get cells within cells, which is usually not what you want.

Data accessed from a database or mixed input file may arrive in MATLAB in multiple cell arrays. Usually it will be appropriate to extract the non-character data from these cell arrays to numeric arrays for better performance and cleaner code.

Useful functions for working with cell arrays are cell2mat, char, cellstr, and cellfun

```
n = {1; 2; 3}
n =
    [1]
    [2]
    [3]

>> disp(cell2mat(n))
     1
     2
     3

asString = char(c)
asString =
Hello
World

d = cellstr(asString)
d =
    'Hello'
    'World'
```

Since loading or extracting the contents of a cell can be slow, it can be a good idea to avoid using a cell array in a loop if possible. The JIT (just in time) approach that MATLAB uses to speed loop execution does not play well with cell arrays.

Some MATLAB functions operate directly on cell arrays. This capability can be expanded by using functions with cellfun.

Cell arrays can also replace comma separated lists in function arguments and display operations. This can sometimes simplify notation. Cell arrays can be more convenient than structures if your application does not naturally fit with the use of fieldnames.

Structures

Structures are flexible arrays with *named* fields. They are very convenient for storing multiple data types in a single variable. A typical use is data with metadata.

```
s.data = magic(3);
s.comment = 'example';
s
s =
        data: [3x3 double]
    comment: 'example'
```

Mathematical functions and operators can often work on a structure field but usually not across fields or on an entire structure.

```
sum(s.data)
ans =
    15    15    15
```

It can be a good idea to avoid using a structure in a loop if possible. The JIT (just in time) approach that MATLAB uses to speed loop execution does not play well with structures.

The contents of fields do not need to be consistent. Fields can hold different things. For example:

```
s(1).a = [1 2];
>> s(2).a = 'Hi';
>> s
s =
1x2 struct array with fields:
    a
```

However this practice may not be so useful and should be avoided unless necessary.

What are good layouts for numeric data in a structure? Structure fields can be stacked, so it might be tempting to do something like

```
WorldData.country.sector.parameter.value
```

but this almost guarantees that multiple loops will be required for access. It would be more efficient to organize the data as a multidimensional numeric array with padding or grouping if necessary.

It is usually faster to extract the full contents of multiple fields and sort them as simple arrays rather than use multiple accesses within each field to sort within the structure. In the

```
WorldData.country.sector.parameter.value
```

example, it would usually be faster to copy the entire value array and extract from it rather than extract elements of value one at a time from the structure.

When it is not important that the data and related parameters be in a single variable, you can store the information in multiple arrays with meaningful names. These can share common indexing for convenient use. These variables can be lumped together in a single MAT file.

Structures can be very useful to simplify argument lists. For example a function definition or call with many arguments will tend to wrap lines and be somewhat difficult to read. Putting some or all of these arguments in a structure can increase readability. Also for functions with several arguments it is necessary to match the argument order between the definition and call. Structure fields used as arguments are not dependent on order.

Time series can work well in a structure. One structure can have fields for the series values, the time scale, and any metadata.

Structures are also the storage arrays that are used for object properties.

Objects

A MATLAB object is like a structure combined with a function library. It affords protection of the data in the structure by limiting how it can be accessed and manipulated. For example, an object can provide a reliable ways to update some data when other data changes. It determines what functions can be used on the data. It allows some design changes to the structure without affecting object usage. It can provide an automated way to communicate with other objects.

The MATLAB object approach is flexible. However performance questions have been raised by some users who report that working with objects can be slower than working with equivalent structures. Because object property access can involve more work than structure field access (including listener notification, set-functions, and different levels of access), the worst-case time for property access will always be slower than for structure fields.

There is an inherent tension with objects in that the fastest access to data in the object depends on knowing the arrangement of the structure but object encapsulation is intended to hide this arrangement from the user so that it can be changed without impact to usage. There is also a strong MATLAB tradition of accessing data using indexing, but this is opposed to the object-oriented value of data hiding.

MATLAB supplies predefined object types or classes for data, and especially time series, analysis: the time series object, the dataset object in the Statistics Toolbox.

Time series object

The MATLAB time series object is generally oriented to GUI manipulations, which can be handy for data exploration but may be awkward for automation. Typically it contains one time scale and a matrix of values that are usually treated as columns. The intent is that all values have the same unit. It may be best to limit usage of the timeseries object to true time series data.

The time series object properties and methods (73) are more extensive than those of the dataset, and they can be extended by using the time series object as a superclass. It is easier to add properties and methods to a subclass of timeseries than to a subclass of dataset. This can produce a useful object for manipulation and computation that is more than a simple data container. Some functions have been overloaded to accept time series objects, and it would be possible to write more overloads and new functions, achieving increased capability.

Someone regrettably decided to also allow the time series in the object to be in rows as well as columns. This makes it more difficult than necessary to write cross time series functions.

The MATLAB time series collection object is a container for time series objects. It seems to be intended to group time series with the same time scale but different value units. Its only interesting methods are resample, plot, and in and out for time series.

Table

A table is a MATLAB® container for storing column-oriented variables that have the same number of rows. Unlike numerical or character arrays, the columns can have different data types. The table entries can be addressed using row and column names or numbers. The concepts of object oriented programming are helpful in understanding tables.

Tables were introduced in R2013b.

- A table is most useful for storing column-oriented data where the columns (variables) have different data types.
- It is a class with properties and methods.
- Its public properties are the data variables and a structure unfortunately named Properties.
- The Properties structure has non-extendable fields that are useful for metadata. One of the fields of Properties is a structure named UserData that you can use for arbitrary contents.
- Most of its methods support database and logical set operations. It also has overloads for many computational functions.
- It supports normal indexing for data extraction or replacement.
- There is an import function `readtable` that is useful but a bit limited. You can also use the Import Data Wizard.

Dataset array

Starting with R2013b, the MathWorks recommends use of tables rather than datasets.

Dataset in the Statistics Toolbox is lighter weight and more general purpose object with less functionality than the time series object. A dataset object can contain one or more variables. If there is more than one variable, they must be columns with the same number of rows. It can contain both character and numeric data, so it is useful for non-ordinal data. The available metadata properties are: `Description, Units, DimNames, UserData, ObsNames, VarNames.` The data analysis methods are: `grpstats and summary.`

For most purposes these properties and methods (25) are quite limited, so the dataset object as it stands would only serve as a simple data container. Some functions have been overloaded to accept dataset arrays, and it would be possible to write more, achieving increased capability. It would also be possible to write new functions that work with dataset arrays. Since the dataset class is defined using the new classdef approach, it can be used as a superclass.

Dataset does have the advantage that it does not assume the data is a time series, so it could be useful for more general types of data.

Date and time

Date and time data types, such as datetime and duration, support efficient computations, comparisons, and formatted display of dates and times. Work with these arrays in the same way that you work with numeric arrays. For example, add, subtract, sort, compare, concatenate, and plot date and time values.

Custom object

It is relatively easy in MATLAB to write code for basic classes. Solid object-oriented design and implementation is, of course, more challenging. In general an object should be more than a bare data container. That is, it should have methods.

Usage guidelines

In general use the simplest array type that meets the need. The overall needs are clear code and good performance. Addressing both of these needs will sometimes require moving data among array types. This translation may involve some pesky details and should usually be performed by functions or methods.

In general use numeric arrays whenever you can. Confine your use of character arrays to short strings unless you need more speed than cell arrays can provide. Confine your use of cell arrays primarily to strings. Cell arrays are also good for non-uniform size data sets and cases where indexing by number is clear enough and easer to implement than use of structure fieldnames. Use structures to bundle multiple data types or multiple arrays in a single variable. Consider objects for more ambitious projects that involve limiting access and manipulation.