Complex Data Types

Scientific Computing
Fall, 2019
Paul Gribble

In the notes on basic data types, operators & expressions we saw data types such as `double` and `char` which are used to represent individual values such as the number `1.234` or the character `'G'`. Here we will learn about a number of complex data types that MATLAB uses to store multiple values in one data structure. We will start with the *array* and *matrix*—and in fact a matrix is just a two-dimensional array. What's more, a scalar value (like `3.14`) is just an array with one row and one column. We will also cover *cell arrays* and *structures*, which are data types designed to hold different kinds of information together in a single type.

1 Arrays

Arrays are simply ordered lists of values, such as the list of five numbers: `1,2,3,4,5`. In MATLAB we can define this array using square brackets:

```
>> a = [1,2,3,4,5]

a =

     1     2     3     4     5

>> whos
  Name      Size            Bytes  Class     Attributes

  a         1x5                40  double
```

We can see that `a` is a `1x5` (1 row, 5 columns) array of `double` values.

We can also get the length of an array using the `length` function:

```
>> length(a)

ans =

     5
```

We can in fact leave out the commas if we want, when we construct the array—we can use spaces instead.

It's up to you to decide which is more readable.

```
>> a = [1 2 3 4 5]

a =

     1     2     3     4     5
```

MATLAB has a number of built-in functions and operators for creating arrays and matrices. We can create the above array using a colon (`:`) operator like so:

```
>> a = 1:5

a =

     1     2     3     4     5
```

We can create a list of only odd numbers from 1 to 10 like so, again using the colon operator:

```
>> b = 1:2:10

b =

     1     3     5     7     9
```

1.1 Array indexing

We can get the value of a specific item within an array by *indexing* into the array using round brackets `()`. For example to get the third value of the array `b`:

```
>> third_value_of_b = b(3)

third_value_of_b =

     5
```

To get the first three values of `b`:

```
>> b(1:3)

ans =

     1     3     5
```

We can get the 4th value onwards to the end by using the `end` keyword:

```
>> b(4:end)

ans =

     7     9
```

Remember, array indexing in MATLAB starts at 1. In other languages like C and Python, array indexing starts at 0. This can be the source of significant confusion when translating code from one language into another.

Another useful array construction built-in function in MATLAB is the `linspace` function:

```
>> c = linspace(0,1,11)

c =

  Columns 1 through 8

         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000

  Columns 9 through 11

    0.8000    0.9000    1.0000
```

By default arrays in MATLAB are defined as row arrays, like the array `a` above which is size `1x5`—one row and 5 columns. We can however define arrays as columns instead, if we need to. One way is to simply transpose our row array using the transpose operator `'`:

```
>> a2 = a'

a2 =

     1
     2
     3
     4
     5

>> size(a2)

ans =

     5     1
```

Now we can see `a2` is a `5x1` column array.

We can directly define column arrays using the semicolon `;` notation instead of commas or spaces, like so:

```
>> a2 = [1;2;3;4;5]

a2 =

     1
     2
     3
     4
     5
```

So in general, commas or spaces denote moving from one column to another, and semicolons denote moving from one row to another. This will become useful when we talk about matrices (otherwise know as two-dimensional arrays).

1.2 Array sorting

MATLAB has a built-in function called `sort()` to sort arrays (and other structures). The algorithm used by MATLAB under the hood is the quicksort algorithm. To sort an array of numbers is simple:

```
>> a = [5 3 2 0 8 1 4 8 5 6]

a =

     5     3     2     0     8     1     4     8     5     6

>> a_sorted = sort(a)

a_sorted =

     0     1     2     3     4     5     5     6     8     8
```

If you give the `sort` function two output variables then it also returns the indices corresponding to the sorted values of the input:

```
>> [aSorted, iSorted] = sort(a)

aSorted =

     0     1     2     3     4     5     5     6     8     8


iSorted =

     4     6     3     2     7     1     9    10     5     8
```

The `iSorted` array contains the indices into the original array `a`, in sorted order. So this tells us that the first value in the sorted array is the 4th value of the original array; the second value of the sorted array is the 6th value of the original array, and so on.

The default sort happens in ascending order. If we want to reverse this we can specify this as an option to the `sort()` function:

```
>> sort(a, 'descend')

ans =

     8     8     6     5     5     4     3     2     1     0
```

1.3 Searching arrays

We can use MATLAB's built-in function called `find()` to search arrays (or other structures) for particular values. So for example if we wanted to find all values of the above array `a` which are greater than 5, we could use:

```
>> ix = find(a > 5)

ix =

     5     8    10
```

This tells us that the 5th, 8th and 10th values of `a` are greater than 5. If we want to see what those values are, we index into `a` using those found indices `idx`:

```
>> a(ix)

ans =

     8     8     6
```

We could combine these two steps into one line of code like this:

```
>> a(find(a>5))

ans =

     8     8     6
```

We can also use a shorthand, like so:

```
>> a(a>5)

ans =

     8     8     6
```

I will leave it as an exercise for you to see how this works. Deconstruct the expression into its constituent parts and think about how they are combined.

1.4 Array arithmetic

One great feature of MATLAB is that arithmetic (and many other) operations can be carried out on an entire array at once—and what's more, under the hood MATLAB uses optimized, compiled code to carry out these so-called *vectorized* operations. Vectorized code is typically many times faster than the equivalent code organized in a naive way (for example using for-loops). We will talk about vectorized code and other ways to speed up computation later in the course.

We can multiply each element of the array `a2` by a scalar value:

```
>> a2 * 5

ans =

     5
    10
    15
    20
    25
```

We can perform a series of operations all at once:

```
>> a3 = (a2 * 5) + 2.5

a3 =

    7.5000
   12.5000
   17.5000
   22.5000
   27.5000
```

These mathematical operations are performed *elementwise*, meaning element-by-element.

We can also perform arithmetic operations between arrays. For example let's say we wanted to multiply two `1x5` arrays together to get a third:

```
>> a = [1,2,3,4,5];
>> b = [2,4,6,8,10];
>> c = a*b
Error using  *
Inner matrix dimensions must agree.
```

Oops! We get an error message. When you perform arithmetic operations between arrays in MATLAB, the default assumption is that you are doing matrix (or matrix-vector) algebra, not elementwise operations. To force elementwise operations in MATLAB we use *dot-notation*:

```
>> c = a.*b

c =

     2     8    18    32    50
```

Now the multiplication happens elementwise. Needless to say we still need the dimensions to agree. If we tried multiplying, elementwise, a `1x5` array with a `1x6` array we would get an error message:

```
>> d = [1,2,3,4,5,6];
>> e = c.*d
Error using  .*
Matrix dimensions must agree.

>> size(c)

ans =

     1     5

>> size(d)

ans =
     1     6
```

2 Matrices

In mathematics a matrix is generally considered to have two dimensions: a row dimension and a column dimension. We can define a matrix in MATLAB in the following way. Here we define a matrix A that has two rows and 5 columns:

```
>> A = [1,2,3,4,5; 1,4,6,8,10]

A =

     1     2     3     4     5
     1     4     6     8    10

>> size(A)

ans =

     2     5
```

We use commas (or we could have used spaces) to denote moving from column to column, and we use a semicolon to denote moving from the first row to the second row.

If we want a `5x2` matrix instead we can either just transpose our `2x5` matrix:

```
>> A2 = A'

A2 =

     1     1
     2     4
     3     6
     4     8
     5    10
```

Or we can define it directly:

```
>> A2 = [1,2; 2,4; 3,6; 4,8; 5,10]

A2 =

     1     2
     2     4
     3     6
     4     8
     5    10
```

There are other functions in MATLAB that we can use to generate a matrix. The `repmat` function in particular is useful when we want to repeat certain values and stick them into a matrix:

```
>> G = repmat([1,2,3],3,1)

G =

     1     2     3
     1     2     3
     1     2     3
```

This means repeat the row vector `[1,2,3]` three times down columns, and one time across rows. Here's another example:

```
>> H = repmat(G,1,3)

H =

     1     2     3     1     2     3     1     2     3
     1     2     3     1     2     3     1     2     3
     1     2     3     1     2     3     1     2     3
```

Now we've repeated the matrix `G` once down rows and three times across columns.

There are also special functions `zeros()` and `ones()` to create arrays or matrices filled with zeros or ones:

```
>> I = ones(4,5)

I =

     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1

>> J = zeros(7,3)

J =

     0     0     0
     0     0     0
     0     0     0
     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

Of course we can fill a matrix with any value we want by multiplying a matrix of ones by a scalar:

```
>> P = ones(3,4) * pi

P =

    3.1416    3.1416    3.1416    3.1416
    3.1416    3.1416    3.1416    3.1416
    3.1416    3.1416    3.1416    3.1416
```

If we use `zeros` or `ones` with just a single input argument we end up with a square matrix (same number of rows and columns):

```
>> Q = ones(5)

Q =

     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
```

There is also a special MATLAB function called `eye` which will generate the identity matrix (a special matrix in linear algebra sort of equivalent to the number `1` in scalar arithmetic):

```
>> eye(3)

ans =

     1     0     0
     0     1     0
     0     0     1
```

2.1 Matrix indexing

We can *index* into a matrix using round brackets, just like with an array. Now however we need to specify both a row and a column index. So for example the entry in the matrix `A2` corresponding to the 3rd row and the second column is:

```
>> A2(3,2)

ans =

     6
```

To get the value in the last row, and the first column:

```
>> A2(end,1)

ans =

     5
```

We can also specify a range in our index values. So to get rows 1 through 3 and columns 1 through 2:

```
>> A2(1:3,1:2)

ans =

     1     2
     2     4
     3     6
```

We can use a shorthand for “all columns” (also works for all rows) using the colon operator:

```
>> A2(1:3,:)

ans =

     1     2
     2     4
     3     6
```

We can use indexing to replace parts of a matrix. For example to replace the first row of `A2` (which is presently `[1 2]` with `[99 99]` we could use this code:

```
>> A2(1,:) = [99 99]

A2 =

    99    99
     2     4
     3     6
     4     8
     5    10
```

To replace the second column of `A2` with 5 random numbers chosen from a gaussian normal distribution with mean zero and standard deviation one, we could use this code:

```
>> A2(:,2) = randn(5,1)

A2 =

   99.0000    0.5377
    2.0000    1.8339
    3.0000   -2.2588
    4.0000    0.8622
    5.0000    0.3188
```

Note the use of the `randn()` function to generate (pseudo)random deviates from a gaussian normal distribution.

2.2 Matrix reshaping

MATLAB has a built-in function called `reshape()` which is handy for reshaping matrices into new dimensions. For example let’s say we have a `4x3` matrix `M`:

```
>> M = [1,2,3; 4,5,6; 7,8,9; 10,11,12]

M =

     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

We can use `reshape()` to reshape `M` into a `6x2` matrix `Mr`:

```
>> Mr = reshape(M,6,2)

Mr =

     1     8
     4    11
     7     3
    10     6
     2     9
     5    12
```

Note that `reshape()` does its work by taking values *columnwise* from the original input matrix `M`. If we want to perform the reshaping in the other way—row-wise—we can do this by transposing the original matrix:

```
>> Mr2 = reshape(M',6,2)

Mr2 =

     1     7
     2     8
     3     9
     4    10
     5    11
     6    12
```

To reshape a matrix (or indeed any multi-dimensional array) into a column vector, there is a convenient shorthand in MATLAB, namely the colon operator `(:)`:

```
>> M_col = M(:)

M_col =

     1
     4
     7
    10
     2
     5
     8
    11
     3
     6
     9
    12
```

If we want a row vector instead we can just transpose the result:

```
>> M_row = M(:)'

M_row =

     1     4     7    10     2     5     8    11     3     6     9    12
```

2.3 Matrix arithmetic

In MATLAB as with arrays, matrix–scalar operations happen elementwise, whereas matrix–matrix operations are assumed to be based on the rules of matrix algebra. We won't go through matrix algebra in all its glory here, but you can see a reminder of the basic operations on this wikipedia page:

Matrix Basic Operations

I can recommend a great book on Linear Algebra by Gilbert Strang:

Introduction to Linear Algebra, 4th Edition by Gilbert Strang. Wellesley-Cambridge Press, 2009

He also has his course on MIT's open-courseware, complete with videos for all lectures here:

video lectures of Professor Gilbert Strang teaching 18.06 (Fall 1999)

The Mathworks has a web page with a matrix algebra "refresher" that might serve as a useful reminder for those who have had linear algebra in the past:

Matrix Algebra Refresher

Here is an example of a scalar–matrix operation on our matrix `A2` from above:

```
>> A2 * 10

ans =

    10    20
    20    40
    30    60
    40    80
    50   100
```

Let's say we wanted to multiply, elementwise, a `5x2` matrix `A3` by `A2`:

```
>> A3 = rand(5,2)

A3 =

    0.2785    0.9706
    0.5469    0.9572
    0.9575    0.4854
    0.9649    0.8003
    0.1576    0.1419
```

As with arrays, we can use dot notation to force elementwise multiplication:

```
>> A4 = A2 .* A3
```

```
A4 =

    0.2785    1.9412
    1.0938    3.8287
    2.8725    2.9123
    3.8596    6.4022
    0.7881    1.4189
```

Without the dot notation we would get an error message:

```
>> A4 = A2 * A3
Error using  *
Inner matrix dimensions must agree.

>> size(A2)

ans =

     5     2

>> size(A3)

ans =

     5     2
```

To perform matrix multiplication we need a right-hand-side that has legal dimensions, in other words the same number of rows as `A2` has columns. For example, if we have another matrix A5 sized `2x3`:

```
>> A5 = rand(2,3)

A5 =

    0.8147    0.1270    0.6324
    0.9058    0.9134    0.0975
```

then we can perform matrix multiplication:

```
>> A6 = A2 * A5

A6 =

    2.6263    1.9537    0.8274
    5.2526    3.9075    1.6549
    7.8789    5.8612    2.4823
   10.5052    7.8150    3.3098
   13.1315    9.7687    4.1372
```

Again, review your linear algebra if you have forgotten about the rules of matrix multiplication. Just remember that to force elementwise operations, use dot-notation.

Addition and subtraction are always elementwise.

2.4 Matrix Algebra

As we saw above, MATLAB assumes that matrix–matrix operations are not elementwise, but conform to the rules of linear algebra and matrix arithmetic. The exception is matrix addition and subtraction, which happen elementwise even in matrix algebra. Multiplication is special however, as we saw above.

What about division (`/`)? In MATLAB the so-called *slash operator* is the gateway to much complexity.

```
>> help slash
 Matrix division.
  \   Backslash or left division.
      A\B is the matrix division of A into B, which is roughly the
      same as INV(A)*B , except it is computed in a different way.
      If A is an N-by-N matrix and B is a column vector with N
      components, or a matrix with several such columns, then
      X = A\B is the solution to the equation A*X = B. A warning
      message is printed if A is badly scaled or nearly
      singular.  A\EYE(SIZE(A)) produces the inverse of A.

      If A is an M-by-N matrix with M < or > N and B is a column
      vector with M components, or a matrix with several such columns,
      then X = A\B is the solution in the least squares sense to the
      under- or overdetermined system of equations A*X = B. The
      effective rank, K, of A is determined from the QR decomposition
      with pivoting. A solution X is computed which has at most K
      nonzero components per column. If K < N this will usually not
      be the same solution as PINV(A)*B.  A\EYE(SIZE(A)) produces a
      generalized inverse of A.

  /   Slash or right division.
      B/A is the matrix division of A into B, which is roughly the
      same as B*INV(A) , except it is computed in a different way.
      More precisely, B/A = (A'\B')'. See \.

  ./  Array right division.
      B./A denotes element-by-element division.  A and B
      must have the same dimensions unless one is a scalar.
      A scalar can be divided with anything.

  .\  Array left division.
      A.\B. denotes element-by-element division.  A and B
      must have the same dimensions unless one is a scalar.
      A scalar can be divided with anything.
```

The backslash (left division) when used like this: `A\B` performs matrix division of `B` into `A`. As the documentation says, this is roughly like `INV(A)*B` but it's not computed this way under the hood. The typical way you will use the backslash matrix operator in MATLAB is to solve systems of linear equations. So for example `X = A\B` is the solution to the matrix equation `A*X=B`. For example, if `B` is a column representing measurements of a dependent variable, and `A` is a matrix representing measurements of several independent variables, then `X` is the vector of regression weights that minimize the sum of squared deviations between `B` and `A*X`. More on this later in the course.

MATLAB has many, many built-in (and compiled and optimized) functions for matrix algebra and matrix algorithms of all sorts. After all, the origins of MATLAB are “Matrix Laboratory”, and so from the start the emphasis has been on matrix computation.

In their web documentation, the MathWorks has a listing of some linear algebra algorithms implemented in MATLAB:

Linear Algebra

These include algorithms for solving linear equations, for matrix decomposition, for finding eigenvalues and singular values, for matrix analysis and so on.

3 Multidimensional arrays

We have seen one-dimensional (row or column) arrays and we have seen (two dimensional) matrices. In MATLAB you can create arrays with more than two dimensions. Here is an example of a three dimensional array:

```
>> A = ones([2,3,4])

A(:,:,1) =

     1     1     1
     1     1     1


A(:,:,2) =

     1     1     1
     1     1     1


A(:,:,3) =

     1     1     1
     1     1     1


A(:,:,4) =

     1     1     1
     1     1     1
```

We have created a three-dimensional array `A` that is size `2x3x4`:

```
>> size(A)

ans =

     2     3     4
```

You can think of it like this: `A` is a `2x3` matrix that is repeated `4` times (in a third dimension). Perhaps the third dimension is time. Perhaps it is something else (e.g. spatial third dimension).

We can even create 4-dimensional arrays:

```
>> B = rand(256,256,10,50);
>> whos
  Name      Size                Bytes  Class     Attributes

  B         4-D             262144000  double

>> size(B)

ans =

   256   256    10    50
```

Here is a 4-dimensional array `B`. You can think of it as a `256x256x128` dimensional 3D array repeated 50 times in a fourth dimension. Note how large the array is (262,144,000 bytes, about 250 megabytes). Depending on how much RAM you have in your computer, you can potentially have MATLAB work with very large data structures indeed.

A quick example, we could take the mean across the 4th dimension like so:

```
>> Bm = mean(B,4);
>> whos
  Name        Size                     Bytes  Class     Attributes

  B           4-D                  262144000  double
  Bm        256x256x10               5242880  double
```

Another useful function to know about is the `squeeze()` function in MATLAB. This will remove any singleton dimensions—that is, dimensions that have size 1. So for example consider the following three-dimensional array:

```
>> A = reshape(1:12,[3,1,4])

A(:,:,1) =

     1
     2
     3


A(:,:,2) =

     4
     5
     6


A(:,:,3) =

     7
     8
     9

```

```
A(:,:,4) =

    10
    11
    12

>> size(A)

ans =

     3     1     4
```

The second (middle) dimension is size 1, so we can use `squeeze` to reshape this three-dimensional array into a two-dimensional array of size `[3x4]`:

```
>> As = squeeze(A)

As =

     1     4     7    10
     2     5     8    11
     3     6     9    12

>> size(As)

ans =

     3     4
```

4 Cell arrays

Arrays (single-dimensional vectors as well as two-dimensional matrices and multi-dimensional arrays) must contain values of the same type. MATLAB has another data structure called a *cell array* that allows one to store data of different types in a structed similar to an array—namely it's an indexed data container containing *cells*, and each cell can contain different data types. Cell arrays use curly brackets instead of square brackets.

The MathWorks online documentation has a page devoted to cell arrays here:

Cell Arrays

So for example we can create a cell array called myCell that contains 5 cells:

```
>> myCell = {1, 2, 3, 'hello', rand(1,10)}

myCell =

    [1]    [2]    [3]    'hello'    [1x10 double]
```

Cells 1 through 3 are numeric, cell 4 is a character string, and cell 5 is a `[1x10]` array of `double` values. We can index into a cell array just like a regular array:

```
>> myCell{4}

ans =

hello

>> myCell{5}

ans =

  Columns 1 through 8

    0.7577    0.7431    0.3922    0.6555    0.1712    0.7060    0.0318    0.2769

  Columns 9 through 10

    0.0462    0.0971
```

You can create an empty cell array like this:

```
>> emptyCell = {}

emptyCell =

     {}
```

Or a cell array with a certain structure that is empty, like this:

```
>> emptyCell = cell(3,5)

emptyCell =

    []    []    []    []    []
    []    []    []    []    []
    []    []    []    []    []

>> emptyCell{2,3} = 'hello'

emptyCell =

    []    []         []    []    []
    []    []    'hello'    []    []
    []    []         []    []    []
```

5 Structures

MATLAB has another data type called a *structure* that is similar to what you might have seen in other languages like Python, and is called a Dictionary. In MATLAB a structure is an array with *named fields* that can contain any data type.

The MathWorks online documentation has a page devoted to structures here:

Structures

Let's create a structure called subject1 that contains a character string corresponding to their name, a numeric value corresponding to their age, a numeric value correponding to their height, a value corresponding to the date the experiment was run, and an array corresponding to some recorded empirical data during an experiment:

```
>> subject1.name = 'Mr. T';
>> subject1.age = 63;
>> subject1.height = 1.78;
>> subject1.date = datetime(2015,08,12);
>> subject1.data = rand(100,2);
>> subject1.catchphrase = 'I pity the fool!';
>> subject1

subject1  =

           name: 'Mr. T'
            age: 63
         height: 1.7800
           date: [1x1 datetime]
           data: [100x2 double]
    catchphrase: 'I pity the fool!'
```

As you can see we use dot-notation to denote a *field* of a structure. As soon as you introduce dot notation into a variable, it becomes a structure type. If a field with the given name does not exist, it is created. Note the use of the `datetime()` function which is a built-in function in MATLAB that handles dates and times.

We can access a field of a structure using dot-notation:

```
>> subject1.name

ans =

Mr. T

>> subject1.date

ans =

   12-Aug-2015
```

5.1 Arrays of structures

We can of course form arrays of structures. An array can hold any data type as long as each element is the same.

```
subject1.name = 'Mr. T';
subject1.age = 63;
subject1.height = 1.78;
subject1.date = datetime(2015,08,12);
subject1.data = rand(100,2);
subject1.catchphrase = 'I pity the fool!';
```

```
subject2.name = 'Polly Holliday';
subject2.age = 78;
subject2.height = [];
subject2.date = datetime(2015,08,12);
subject2.data = rand(100,2);
subject2.catchphrase = 'Kiss my grits!';

subject3.name = 'Leonard Nimoy';
subject3.age = 83;
subject3.height = [];
subject3.date = datetime(2015,02,26);
subject3.data = rand(100,2);
subject3.catchphrase = 'Live long and prosper';

allSubjects = [subject1, subject2, subject3];

>> allSubjects

allSubjects =

1x3 struct array with fields:

    name
    age
    height
    date
    data
    catchphrase
```

Now we can do convenient things like look at all of the age fields across the whole array:

```
>> allSubjects.age

ans =

    63


ans =

    78


ans =

    83
```

We can collect these into an array:

```
>> allAges = [allSubjects.age]

all_ages =
    63    78    83
```

Or we could collect all the `data` fields together into a three-dimensional array, and then average across subjects:

```
>> allData = [allSubjects.data];
>> size(allData)

ans =

   100     6

>> allData = reshape(allData,100,2,3);
>> size(allData)

ans =

   100     2     3

>> allDataMean = mean(allData,3);
>> size(allDataMean)

ans =

   100     2
```

Note that if all elements of an array are not the same identical structure (i.e. do not have the same fields defined) then we get an error:

```
>> subject4.name = 'me'

subject4 =

    name: 'me'

>> allSubjects = [subject1, subject2, subject3, subject4];
Error using horzcat
Number of fields in structure arrays being concatenated do not match.
Concatenation of structure arrays requires that these arrays have the
same set of fields.
```

The solution here would be to use a cell array instead of a plain array—remember, in a cell array the cells do not have to be the same type:

```
>> allSubjects = {subject1, subject2, subject3, subject4}

allSubjects =

    [1x1 struct]    [1x1 struct]    [1x1 struct]    [1x1 struct]
```