Engineering Problem Solving and Programming (CS 1133)

MATLAB Arrays

K. Ming Leung

mleung@duke.poly.edu

http://cis.poly.edu/~mleung

Department of Computer Science and Engineering

Polytechnic Institute of NYU

1 Scalar Variables

The variables that we have considered thus far can each contain only a single value. These are referred to as **scalar variables** in MATLAB. However very often we want to have variables each containing a collection of values. For example, to keep track of the exam score for one student one can introduce a scalar variable `score` and assign it a value: `score = 78`.

But the class has more than one student. It would be ridiculous to introduce a separate variable to store the score for each of the students:

```
score1 = 78;
score2 = 93;
score3 = 46;
score4 = 63;
```

The memory locations of these variables are generally scattered throughout memory space.

And to add up the scores we would have to write

```
totalScore = score1 + score2 + score3 + score4;
```

This approach would be very tedious. For a class of hundreds students this expression would take a couple of pages to write.

This approach is also highly inefficient since the computer would have to visit one at a time the memory location for each of the variables to retrieve its value.

2 One-Dimensional Arrays: Row and Column Vectors

To solve these problems, MATLAB (as well as other programming languages) has variables that can each store a collection of values. An index is used together with the variable name to access each value. These are referred to as array variables.

For example, the above problem can be handled by using a single array named `Scores` as follows:

```
Scores = [ 78 94 46 63 ];  % specify elements
totalScore = sum(score);    % sum the scores
```

where a MATLAB built-in function `sum` is used to add up all the scores stored in the array `Scores`. Each of the indexed variable is referred to as an element of the array. The value stored in the n-th element is accessed by `Scores(n)`. In MATLAB the array index starts at 1 and increases by 1 in going from one element to the next. Note the parentheses (round brackets) around the index n. Some other languages such as C and C++ use the square brackets instead.

Using an array variable for this problem makes the program easier to write.

The program is also easier to be modified. To add one extra student whose score is 37 to the class, all we need is to add an extra element to the array:

```
Scores = [ 78 94 46 63 37 ];  % specify elements
totalScore = sum(Scores);    % sum the scores
```

If we didn't use an array, we would have to do add another variable containing the extra score:

```
score5 = 37;
```

and add another term to the sum:

```
totalScore = score1 + score2 + score3 + ...
             score4 + score5;
```

The program is also more efficient since by design the values for the elements of an array are stored contiguous (adjacent to each other) in memory. When the program needs to access an array value for reading or writing, a block of values for the array elements is accessed. If we used individual variables for the scores, the computer would need to go to their memory locations, which are scattered in memory space, to read their values.

Next, we go back to the original problem. Suppose we want to reduce the score for the second student by a half as punishment for cheating on the exam, we can write

```
Scores(2) = Scores(2) / 2;
```

so individual elements of an array can be used like an individual variable.

The array `Scores` has only one index, and it is called a one-dimensional (1D) array. In linear algebra (a branch of mathematics which you will probably learn in your second year) this array is referred to as a row vector. The elements in `score` can be displayed using the statement `disp(Scores)`, and the resulting row of numbers are shown as

```
78     94     46     63
```

We could have created the array `Scores` as a column vector instead of a row vector and then sum up the elements to achieve the same result:

```
ScoresC = [ 78; 94; 46; 63 ];  % specify element

totalScoreC = sum(ScoresC);    % sum the scores
```

The semicolon separating the values of adjacent elements in the above array `ScoresC` acts as a new-line character so that the next value appears on the next row. The statement `disp(ScoresC)`, results in the following column of numbers

```
    78
    94
```

```
    46
    63
```

In MATLAB, it turns out that if the elements of a vector is generated one at a time without first specifying whether it is a row or column vector (we will learn how to do that later), the resulting array is a row vector. Try this:

```
ScoresR(1) = 78;
ScoresR(2) = 94;
ScoresR(3) = 46;
ScoresR(4) = 63;
disp(ScoresR)
```

and see the resulting display.

In linear algebra, there is an operation called the transpose which converts a row vector into a column vector and vice versa. MATLAB has a built-in function named `transpose` that perform this operation. Thus `transpose(Scores)` gives exactly the same vector as `ScoresC`.

2.1 Built-in Function: `length`

The number of elements in a given row or column array can be obtained using the MATLAB built-in function `length`. For example

```
len = length(ScoresR)
```

produces the result

```
len = 4
```

Thus the length of the vector `ScoresR` is `4` (it has `4` elements).

3 Two-Dimensional arrays: Matrices

A 1D array has a single index to specify the location of the elements within the array.

A two-dimensional (2D) array uses two indices to specify the location of its elements. In linear algebra, a 2D array is called a matrix. The elements of a matrix are displayed in a two-dimensional table form.

For example there are 4 students in the class and each took two midterms and a final exam. One can arrange the scores in a table form as follows

Name	1st midterm	2nd midterm	final
Tillary	78	66	91
Johnson	94	93	89
Lawrence	46	34	51
Jay	63	46	55

In MATLAB all the scores can be stored in an array SCORES by assigning the values of its elements as follows

```
SCORES = [ 78 66 91;
           94 93 89;
           46 34 51;
           63 46 55 ];
```

A space character is used here to separate the values on a given row. The semicolon separates the values from one row to the next row.

Since program inputs are in a free form in MATLAB, we can actually place all the values all on a single line.

A 1D array uses two indices to specify the location of its elements within the array. A 2D array uses two separate indices. The first index specifies the row the element is located in, and the second index specifies the column. These indices are separated by a comma and are enclosed in a pair of parentheses. Thus the ij-th element of the array SCORES is given by `SCORES(i,j)`.

For example, the final exam scores for the first two students were interchanged by mistake and have to be corrected. The following code will work

```
temp = SCORES(1,3);          % copy of score of 1
SCORES(1,3) = SCORES(2,3);   % replace it with th
SCORES(2,3) = temp;          % replace 2nd studen
```

Note that the use of a temporary variable is needed.

3.1 Built-in Function: `size`

The size of a 2D array is given by the number of rows and columns of the array. If it has m rows and n columns, the 2D array is called an m by n matrix. If $m = n$, it is a square matrix, otherwise it is a rectangular matrix.

One can find the size of an array using the built-in function `size`. There are several ways to use this function. For example,

```
Dim = size(SCORES)
```

results in a row vector variable `Dim` given by `[ 4 3 ]` because `SCORES` is a 4 by 3 matrix.

The statement

```
[ numRows numCols ] = size(SCORES)
```

gives `numRows = 4` and `numCols = 3`.

The `size function` has an optional second argument that can be used to return either the number of rows or columns. For example,

```
nRows = size(SCORES,1)
```

gives the number of rows (the first index), and

```
nCols = size(SCORES,2)
```

gives the number of columns (the second index).

For a row vector of length n, its size is 1 by n. Thus

```
Scores = [ 78 94 46 63 ];
Dim1 = size(Scores)
```

results in `Dim1 = [ 1 4 ]`.

On the other hand or a column vector of length n, its size is n by 1. Thus

```
ScoresC = [ 78; 94; 46; 63 ];
Dim2 = size(ScoresC)
```

results in `Dim2 = [ 4 1 ]`.

It is clear that a vector, either a row or column vector, is just a special case of a 2D array.

Similarly, a scalar such as `age = 18` can be considered as a 1 by 1 array. In fact MATLAB treats it exactly as such, as you can check using the `whos` command.

In the above example, we have placed the first, second and final exam scores for a given student into that given row of the 2D array.

We could have done it the other way round. That means that we place the first, second and final exam scores for a given student into that given column of the 2D array. The resulting 2D array is

```
SCORES = [ 78 66 91;
           94 93 89;
           46 34 51;
           63 46 55 ];

SCORES2 = [ 78 94 46 63;
            66 93 34 46;
            91 89 51 55 ];
```

These two arrays are clearly the transpose of each other as you can check as follows:

```
disp(transpose(SCORES));

disp(transpose(SCORES1));
```

The apostrophe can also be used to take the transpose of an array if its elements are all real quantities. Assuming that to be the case, then

`transpose(SCORES)` gives exactly the same array as `SCORES'`.

3.2 Built-in Function: `sum`

The elements of am m by n 2D array can be summed using the built-in function `sum`, except now there are two ways to sum up the elements.

(1) One can add them up column by column. This means that for a given column (specified by a given second index) one adds up the elements on each of the rows (all possible choices for the first index). The result is a row vector of length n. The first element is the sum of the elements in the first column of the original array. The second element is the sum of the elements in the second column of the original array, etc.

In this example, the resulting vector gives the total score for each of the three exams:

```
SCORES = [ 78 66 91;
           94 93 89;
           46 34 51;
           63 46 55 ];
TotalExams = sum(SCORES,1);
```

which produces the result:

```
TotalExams =
   281   239   286
```

Summing over the first index like this is actually the default. This means that if you use the `sum` function without supplying a second argument (to specify which index to sum over), then MATLAB assumes that you want to sum over the first index.

(2) One can add them up row by row. This means that for a given row (specified by a given first index) one adds up the

elements on each of the columns (all possible choices for the second index). The result is a column vector of length m. The first element is the sum of the elements in the first row of the original array. The second element is the sum of the elements in the second row of the original array, etc.

In this example, the resulting vector gives the total score for each of the four students:

```
TotalStudents = sum(SCORES,2);
```

which produces the result

```
TotalStudents =
   235
   276
   131
   164
```

The built-in function `mean` can be used in the same way, to compute the mean (or the arithmetic average) of a given array.

4 Creating Special Arrays

There are a number of special arrays that are especially useful and can be created using built-in MATLAB functions.

```
zeros(m,n)
```

creates an n by m array whose elements are all zeros.

`zeros(size(SCORES))`

creates an array the same size as SCORES with elements all set to zeros.

`ones(m,n)`

creates an n by m array whose elements are all ones.

`ones(size(SCORES))`

creates an array the same size as SCORES with elements all set to ones.

`eye(n)`

create an n by n array with one along its main diagonal and zero everywhere else. Such a square matrix is called an n-dimension identity matrix.

5 Creating Vectors Using the colon operator

Especially in the case of plotting a graph, one often need to create a vector with more than a hundred points. Entering these elements explicitly is tedious. Fortunately MATLAB has easy

and efficient ways to generate a huge number of points in a vector.

Using the colon operator

Given three numbers `intendedFirst`, `increment`, and `intendedEnd`,

```
intendedFirst : increment : intendedEnd
```

is a row vector whose first element is `intendedFirst`, the second element is `intendedFirst + increment`, the third element is `intendedFirst + 2 * increment`, the fourth element is `intendedFirst + 3 * increment`, etc.

The last element of the resulting vector has a value closest to `intendedEnd` but does not go beyond it.

For example, `Vec1 = -5:3:7` gives the vector `[ -5 -2 1 4 7 ]`, but `Vec2 = -5:3:5` gives the vector `[ -5 -2 1 4 ]`.

The amount of increment specified by `increment` can be negative.

For example, `Vec1 = 5:-3:-1` gives the vector `[ 5 2 -1 ]`, and `Vec2 = 5:-3:-3` also gives the vector `[ 5 2 -1 ]`.

Notice that if `increment` is positive and `intendedEnd` is less than `intendedFirst`, then the resulting vector has no element. It is called an empty vector in MATLAB. It is an

array with no elements and is denoted by `[ ]`. Its size is 0 by 0 (sometimes its size is reported as 1 by 0. For example the vector `Vec3 = -5:3:-6` is empty.

Similarly, if `increment` is negative and `intendedEnd` is larger than `intendedFirst`, then the resulting vector is also empty. For example the vector `Vec4 = -5:-3:-4` is empty.

If needed you can use the transpose function to convert the resulting row vector into a column vector.

Note that the colon operator always generate equally spaced points.

Another way to generate a vector containing equally spaced points is to use the built-in function `linspace`.

The `linspace` function has three arguments `firstElement`, `lastElement` and `numberPoints`, and creates a row vector whose first and last elements are given exactly (to machine accuracy) by `firstElement` and `lastElement` respectively (unless `numberPoints` is less than 2).

For example,

```
linspace(2,5,3)
```

gives

```
    2.0000    3.5000    5.0000
```

```
linspace(2,5,4)
```

gives

```
     2     3     4     5
```

```
linspace(2,5,5)
```

gives

```
    2.0000    2.7500    3.5000    4.2500    5.00
```

```
linspace(2,-5,5)
```

gives

```
    2.0000    0.2500   -1.5000   -3.2500   -5.00
```

The biggest difference between creating a vector of equally spaced points using the colon operator and using the `linspace` function is the former method controls the amount of increment, and the later method controls the total number of generated points.

5.1 Accessing Elements of an Array

Suppose we have the array

```
ARR = [
2.4    9.2    4.4    9.7    7.0    2.4
0.4    8.0    3.6    3.8    4.4    1.0
```

```
9.3    1.7    7.2    8.6    2.4    6.8
5.1    4.8    0.6    9.1    8.2    5.0
3.0    5.3    2.4    9.2    3.5    2.1]
```

One can retrieve any element by specifying the indices. For example its element at position $1, 2$:

```
theElement = ARR(2,3)
```

has the value 3.6.

One can retrieve multiple elements from a given array using vector indices.

For example,

```
MTX1 = ARR([ 2 5 ],3)

MTX1 =

    3.6000
    2.4000


MTX2 = ARR(2,2:4)

MTX2 =

    8.0000    3.6000    3.8000
```

The vector indices need not be in ascending or descending order:

```
MTX3 = ARR(4,[ 5 1 3 ])

MTX3 =

    8.2000    5.1000    0.6000
```

or do they need to be distinct:

```
MTX4 = ARR(2,[5 1 5])

MTX4 =

    4.4000    0.4000    4.4000
```

Both the first and the second indices can be vectors:

```
MTX5 = ARR([5 2],2:4)

MTX5 =

    5.3000    2.4000    9.2000
    8.0000    3.6000    3.8000
```

It make no difference whether the index vectors are row or column vectors:

```
MTX6 = ARR([5;2],2:4)
```

gives

```
MTX6 =

    5.3000    2.4000    9.2000
    8.0000    3.6000    3.8000
```

Notation: `end` gives the highest value that a given index can take. For example,

```
MTX7 = ARR(3,2:end)
```

gives

```
MTX7 =

    1.7    7.2    8.6    2.4    6.8
```

You can even do this

```
MTX8 = ARR(3:end,2:end)
```

and get

```
MTX8 =

    1.7    7.2    8.6    2.4    6.8
    4.8    0.6    9.1    8.2    5.0
    5.3    2.4    9.2    3.5    2.1
```

As the first index, `end` is 5, but as the second index it is 6. So its value depends on the context under which it is invoked.

As we will see later `end` also has other meanings that are also based on the context where it is being used.

When the colon appears singly as an index to an array, it means to take all possible values that index can have.

Thus

```
Row4 = ARR(4,:)
```

is the 4th row of ARR

```
Row4 =

    5.1    4.8    0.6    9.1    8.2    5.0
```

Of course you can replace the `:` by `1:end`, the result will be identical. But it is shorter just to use the colon.

5.2 Appending Elements to an Array

An element can be appended to an array. The size of the array will be expanded just enough to accommodate the extra element. The elements whose values are not yet specified are set to zero.

For example, we have the 3 by 4 array

```
MTX = [
2.4    9.2    4.4    9.7
0.4    8.0    3.6    3.8
9.3    1.7    7.2    8.6  ]
```

then the statement

```
MTX(4,6) = 5.1
```

appends an element with value 5.1 at position $4, 6$. Array MTX becomes the following 4 by 6 array

```
MTX = [
2.4    9.2    4.4    9.7    0.0    0.0
0.4    8.0    3.6    3.8    0.0    0.0
9.3    1.7    7.2    8.6    0.0    0.0
0.0    0.0    0.0    0.0    0.0    5.1 ]
```

As another example, consider the array

```
RAY = [
1   4   6   3
2   5   5   2
3   6   4   1 ]
```

then after executing the statement

```
RAY(2,6) = 9
```

array RAY becomes

```
RAY = [
1   4   6   3   0   0
2   5   5   2   0   9
3   6   4   1   0   0 ]
```

6 Concatenating Arrays to form Larger Arrays

Given an n by m array A and an n by k array B, since they both have the same number of rows, they can be concatenated side by side to obtain an n by $m + k$ array as follows

```
ABH = [ A B ]
```

For example, if

```
A = [
1    4
2    5
3    6 ]

B = [
6    3    8
5    2    8
4    1    7 ]
```

then ABH is given by

```
ABH = [
1    4    6    3    8
2    5    5    2    8
3    6    4    1    7 ]
```

On the other hand, given an n by m array A and an k by m array C, since they both have the same number of columns, they can be concatenated by placing one on top of the other to obtain an $n + k$ by m array as follows

```
ACV = [ A; C ]
```

For example, if

```
A = [
1    4
2    5
3    6 ]

B = [
6    3
5    2 ]
```

then ACV is given by

```
ACV = [
1    4
2    5
3    6
6    3
5    2 ]
```

Notice that `[ A; B ]` and `[ A C ]` are illegal operations since the resulting arrays are not square or rectangular.

6.1 Built-in Function: `repmat`

Using the `repmat` Function to Replicate a Given Arrays

The built-in function `repmat` can be used to replicate a given array a certain number of times to produce larger array. Specifically given an array `A`, `repmat(A,m,n)` is an array obtained by replicating `A` m times vertically and n times horizontally.

For example if

```
A = [
1   3   5
2   4   6 ]
```

then

```
ABIG = repmat(A,2,3)
```

gives

```
ABIG =

     1     3     5     1     3     5     1     3
     2     4     6     2     4     6     2     4
     1     3     5     1     3     5     1     3
     2     4     6     2     4     6     2     4
```

7 Deleting Elements from an Array

Rows and columns can be deleted from an array by assigning them to an empty array. The remaining rows and columns are repackaged so that the original array will have a reduced size.

Let us take the array

```
ARR = [
2.4    9.2    4.4    9.7    7.0    2.4
0.4    8.0    3.6    3.8    4.4    1.0
9.3    1.7    7.2    8.6    2.4    6.8
5.1    4.8    0.6    9.1    8.2    5.0
3.0    5.3    2.4    9.2    3.5    2.1]
```

To delete the 4th row use the statement

```
ARR(4,:) = []
```

The result is

```
ARR =

    2.4    9.2    4.4    9.7    7.0    2.4
    0.4    8.0    3.6    3.8    4.4    1.0
    9.3    1.7    7.2    8.6    2.4    6.8
    3.0    5.3    2.4    9.2    3.5    2.1
```

From this new array we can delete the 3rd and 5th column using the statement

```
ARR(:,[3 5]) = []
```

to obtain

```
ARR =

    2.4    9.2    9.7    2.4
    0.4    8.0    3.8    1.0
    9.3    1.7    8.6    6.8
    3.0    5.3    9.2    2.1
```

Note that the original array is gone when elements are deleted from it. If you need the unmodified version of the array, you need to store another copy of it in another variable before deleting anything.

Note: after deleting elements from a 2D array, the result must be representable as an array, otherwise the operation is illegal. The following statement is not legal:

```
ARR(2,2:3) = [ ]
```

Because if you delete the second and third elements in row 2, the remaining elements do not form an array. Thus the result cannot represented in MATLAB using an array. (Need to use another MATLAB structure to do that).

Thus for 2D arrays, one can only delete an entire row or an entire column, the remaining elements can be re-packaged

into an array. However deleting a partial row or a partial column from a 2D array is illegal.

For a vector (a 1D array), one can delete any number of elements. The remaining elements can be re-packaged into another vector.

8 MATLAB Characters and Strings

Other than working with floating-point numbers, MATLAB can deal with characters in the ASCII character set which you can find in Appendix A of our textbook. Each character in the set has a character code (also known as its ASCII value) (see table on p.365) which is stored using 16-bits (2 bytes) of memory.

Only the characters with character codes between 32 and 126 are printable (they are displayed as visible characters). The rest of the characters are non-printable control characters.

For example, character 'A' has an ASCII value of 65. It is followed by character 'B' which has an ASCII value of 66, etc. until the character 'Z' with an ASCII value of 90. So the uppercase letters are grouped together in the table.

The lowercase letters are also grouped together in the table. Character 'a' has ASCII value of 97, followed by 'b' which has an ASCII value of 98, etc. until 'z' which has an ASCII value of 122.

Similarly the digits also appear as a group in the ASCII table.

There is no need to remember these ASCII values. All one needs is to know is that the uppercase letters are grouped one after the other in the table. The same is true for the lowercase letters. And then the uppercase letters are listed first before the lowercase ones. For the digits they are listed in this order: 0, 1, ..., 9.

Note that do not confuse the digits '0', '1', etc. with the numbers 0, 1, etc. although they may look exactly the same in an output.

In MATLAB a printable character is represented by enclosing the character between two single quotes. The single quote on the left of the character is the same as the one on the right. On the keyboard the single quote character is located just to the left of the "enter" key.

A string is a squence of characters. In MATLAB a string is represented by a sequence of characters enclosed by two single quotes. This sequence is treated as a row vector of characters. So everything that we learn about arrays can be used to work with strings.

For example the following is a string variable containing the full name of our president:

```
FullName = 'Barack Hussein Obama'
```

Note that this string contains a total of two blank spaces (ASCII code of 32).

His last name is

```
LastName = FullName((end-4):end)
```

Note that from the last character of his full name, one has to move back 4 characters to get to the 'O'.

This changes the president's last name to the first name of an evil person:

```
LastName(2)= 's'

EvilName = [ LastName ' ' 'bin Laden' ]
```

Noted that a blank space has to be inserted between the first and the last name.

Appending a character to a string work almost exactly as appending a value to an array, except that unspecified characters are assumed to be blanks.

```
EvilName(19) = 'I'
disp(EvilName)
```

Finally try the following code. What is the result? (I apologize if I offended any one.)

```
Name = 'SCHMIDT'

Name(2:2:end) = []
```

9 Displaying Values using the function `disp`

The value of a variable can be displayed in the command screen using the built-in function `disp` and the name of the variable as its argument.

The syntax is:

```
disp(X)
```

where X is either a single numeric or string variable. The output is like the statement X without the semicolon at the end, except that `disp` does not display the name of the variable.

For example,

`disp(pi)` will display the number `3.1416`. Normally only 4 decimal places are displayed but one can alter the display format using the `format` statement.

To display values stored in more than one numeric arrays together, one must concatenate them in some way into a single array before using `disp`.

It is possible to display in a single `disp` statement a character string together with numerical data. The built-in function `num2str` must first be used to convert the numerical data into a string. This string is then concatenate with the character string into a single string (an array of characters) which is then displayed.

For example,

```
disp(['The value of pi is: ' num2str(pi,15)])
```

results in the display

```
The value of pi is: 3.14159265358979
```

The first argument of the function `num2str` is the name of a numeric array, and the optional second argument is used to specify the desire number of significant figures. The default number is 4.

10 Entering Data from the Keyboard

The built-in function `input` can be used to enter data from the keyboard.

For numerical input, `R = input('How many apples')` gives the user the prompt in the text string and then waits for input from the keyboard. The input can be any MATLAB expression, which is evaluated, using the variables in the current workspace, and the result returned in R. If the user presses the return key without entering anything, `input` returns an empty matrix.

For example,

```
numberApples = input('How many apples: ');
```

Note that with the addition of the colon and a blank space, this prompt is better than the one given previously.

One can input even an array, however to do that you need to use the square brackets, blanks or commas to separate numbers within the same row, and the semicolon or the carriage-return to enter values for elements in the next row.

The function `input` has an optional argument that can be used to input from the keyboard a character string. For example, `R = input('What is your name','s')` gives the prompt in the text string and waits for character string input. The typed input is not evaluated; the characters are simply returned as a MATLAB string.