# 11.1 *MATLAB* Tutorial 6

*Introduction to Computational Science: Modeling and Simulation for the Sciences*

Angela B. Shiflet and George W. Shiflet
Wofford College

© 2006 by Princeton University Press

## Introduction

We recommend that you work through the introduction with a copy of *MATLAB*, answering all Quick Review Questions. *MATLAB* is available from MathWorks, Inc. (http://www.mathworks.com/).

The prerequisites to this tutorial are *MATLAB* Tutorials 1-5. Tutorial 6 introduces the following functions and options, which simulations of this chapter employ: joining arrays, size of an array, visualizing a rectangular array, multidimensional arrays, element-by-element comparisons of arrays, and finding where a condition is true. Module 11.3 on "Movement of Ants" uses the latter feature, but Module 11.2 on "Spreading of Fire" does not.

## Joining Arrays

By specifying sequences of rows and columns, we can obtain a subarray. Similarly, joining several arrays in brackets, we can get the **concatenation** or joining of arrays without changing any of the arguments. The following general form returns one array containing the elements from the arrays *array1*, *array2*, ... :

**[*array1*, *array2*, ... ]**

The following command concatenates three vectors:

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

Thus the resulting vector consists of eight numbers as follows:

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

**Quick Review Question 1** Start a new *M*-File. In opening comments, have "MATLAB Tutorial 6 Answers" and your name. Save the file under the name *MATLABTutorial6Ans.m*. In the file, preface this and all subsequent Quick Review Questions with a comment that has "QRQ" and the question number, such as follows:

```
% QRQ 1 a
```

Consider the following rectangular array *mat*:

```
mat = [1 2 3; 4 5 6; 7 8 9]
```

**a.** Give the command to return the last column. Thus, the returned column vector is [3; 6; 9].

**b.** Write a command to return a rectangular array that is equal to *mat* except the last column of *mat* appears as the first and last column of the new array. Thus, the new array has four columns, and its second, third, and fourth columns are equal to the columns of *mat*. In your command, do not type specific elements of *mat* but commands to take a subarray and join it to *mat*.

**Quick Review Question 2** Consider the array *mat* in Quick Review Question 1.

**a.** Give the command to return the last row. Thus, the returned vector is [7 8 9].

**b.** Write a command to return a rectangular array that is equal to *mat* except the last row of *mat* appears as the first and last row of the new array. Thus, the new array has four rows, and its second, third, and fourth rows are equal to the rows of *mat*. In your command, do not type specific elements of *mat* but commands to take a subarray and attach it to *mat*.

**Quick Review Question 3** Consider the array *mat* in Quick Review Question 1.

**a.** Write a statement to assign to *extendRows* the array with five rows: the last row of *mat*, the rows of *mat*, and the first row of *mat*. Thus, for *mat* having three rows, the new array has five rows.

**b.** Write a statement to assign to *extendCols* the array with five columns: the last column of *extendRows* from Part a, the columns of *extendRows*, and the first column of *extendRows*. Thus, for *extendRows* having three columns, the new array has five columns. For *mat*, the final result is the following 5-by-5 array:

```
extendCols =

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

## Size

For generality in functions, it is sometimes useful to obtain an array's size, or the number of elements, in a particular dimension. By specifying the array and the dimension, the ***size*** function returns the corresponding number of elements. The following form using ***size*** returns the number of elements array *ar* has in dimension *n*:

```
size(ar, n)
```

Thus, the length returned for the row vector below is 3:

```
size(['a' 'b' 'c'], 2)
```

**Quick Review Question 4** Write a statement to assign to *lst* an array of all zeros that has at random between 5 and 15 rows and at random between 1 and 4 columns.

**a.** Display the number of rows of *lst*.

**b.** Display the number of columns of *lst*.

## Grid Graphics

Frequently, simulations involve evolving rectangular arrays of numbers, and pictorial representations of these matrices can help scientists understand the information. The graphics function ***image*** represents such a grid with each cell being an index into a **color map**, which is an array of color values. The function ***colormap*** establishes the color map. We can use a built-in color map, such as ***gray*** for gray-scale, or define our own. The following command establishes a gray color map with five shades from black (index 1) to white (index 5):

```
colormap(gray(5))
```

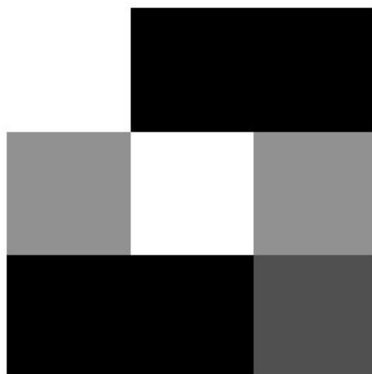
For example, the following segment generates a 3-by-3 array, *randGrid*, of random integer values from 1 through 5 and displays a visualization of the grid in black, white, and shades of gray with no axes and as a square:

```
randGrid = floor(rand(3) * 5 + 1)
image(randGrid)
axis off
axis square
```

With *randGrid* as follows, Figure 1 shows the resulting graphics with 1 as index to black and 5 as index to white:

```
5      1      1
3      5      3
1      1      2
```

**Figure 1** Visualization of grid in five shades, from black through shades of gray to white



To elicit a proper scientific visualization, it is often helpful to define our own color map. Consider the following command assigns to *matEx* a 4-by-4 array zeros and ones.

```
matEx = [0 0 1 0; 0 1 0 0; 1 0 0 0; 1 1 0 0]

matEx =

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

To visualize the grid with yellow representing 0 and forest green, 1, we first establish the color map as an array with two rows. Yellow has red-green-blue (**RGB**) values of 1, 1, and 0, respectively, on the first row; and forest green has the three values 0.1, 0.75, and 0.2 on the second row. After defining, we establish *map* as the color map, as follows:

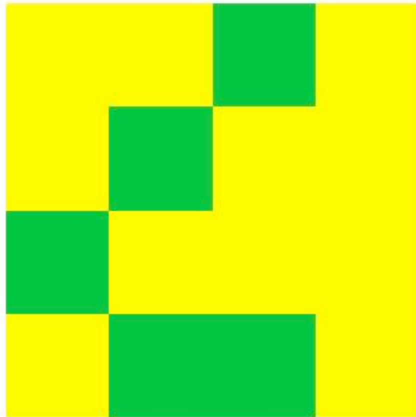
```
map = [1 1 0; 0.1 0.75 0.2];
colormap(map);
```

Because the row index begins with 1, not 0, we add 1 to every value of *matEx* and generate an image of the resulting grid of ones and twos, as follows:

```
image(matEx + 1)
axis off
axis square
```

Figure 2 displays the resulting visualization.

**Figure 2** Visualization of array of zeros and ones with color map having yellow and green, respectively



**Quick Review Question 5**

**a.** Generate, but do not display, a 10-by-10 matrix, *randMat*, of random integers between 1 and 5, inclusively.

**b.** Visualize *randMat* in 5 shades of gray as a square and with no axes.

The following parts create a color map with shades of red and visualize *mat* with this coloring scheme.

**c.** The color map, *map*, is to have the RGB values for five colors. Thus, establish *map* as an array of zeros of the appropriate size.

**d.** Using a *for* loop, assign to each row of *map* the RGB values for a shade of red with red values from 1/5 to 5/5 = 1. Thus, each row should have a red value of *i*/5 and blue and green values of zero.

**e,** Make *map* the color map.

**f.** Visualize *randMat* in 5 shades of red as a square with no axes.

**Quick Review Question 6**

**a.** Generate a 10-by-10 array, *mc*, of integers indicating the columns. For example, column four contains all fours.

**b.** Generate a color map of ten colors, where the first and second coordinates each varying from 0 to 0.9 in steps of 0.1 and with the third coordinate being 0. Visualize *mc* using this color map.

**Quick Review Question 7** Smooth out the display from Quick Review Question 6 with a 100-by-100 matrix *mc2* and appropriately smaller step sizes.

## Multidimensional Arrays

We have frequently employed one- and two-dimensional arrays. In this chapter, three- and even four-dimensional arrays, called **multidimensional arrays**, are useful for storing grids from multiple time steps. For example, suppose data for one time step is in a 20-by-20 array, or grid, of integers. Each time step of the simulation updates the grid values. To store all the grid information for later analysis, processing, or visualization, we can use a three-dimensional array, say *gridList*. The first index gives the row; the second, the column; and the third index specifies the **page**. The initial grid is the 20-by-20 array *gridList*(:, :, 1), and *gridList*(:, :, 2) stores the grid for time step 1.

**Quick Review Question 8**

**a.** Make *mcList* a 10-by-10-by-6 array of zeros.

**b.** Assign *mc* from Quick Review Question 6 to the first page of *mcList.*

**c.** Using a *for* loop with index *i* varying from 1 to 5, store the results of adding *i* to *mc* in subsequent pages of *mcList*. Thus, Page 2 of *mcList* will contain *mc* + 1, and Page 6 of *mcList* will contain *mc* + 5.

**d.** Generate color map of fifteen colors, where the first and second coordinates each varying from 0 to 14/15 in steps of 1/15 and the third coordinate is 0.

**e.** Produce an animation of the pages of *mcList* using this color map and a movie.

## Array Comparisons

We can employ the relational operators (<, >, <=, >=, ==, and ~=) to compare two numbers and for an element-by-element comparison of two arrays that have the same size. For the latter situation, the return value is a **logical array** of the same size with 1 appearing where a relationship is true and 0 where false. For example, consider the following vectors:

```
v1 = [3 6 5 9];
v2 = [2 8 7 4];
```

The expression *v1* < *v2* returns the vector [0 1 1 0] because only the middle two element-by-element inequalities, 6 < 8 and 5 < 7, are true.

If we compare an array with a number, *MATLAB* expands the number into an array of the same size. Thus, *v1* == 5 performs the comparison *v1* == [5 5 5 5], which returns [0 0 1 0] because 3 ≠ 5, 6 ≠ 5, 5 == 5, and 9 ≠ 5.

### Quick Review Question 9

**a.** Generate and display a 3-by-3 array, *ra*, of random floating point numbers between 0 and 1.

**b.** Perform a comparison that returns a 3-by-3 logical array indicating the elements that are less than 0.5.

## Position of a Pattern

The *MATLAB* function ***find*** identifies the locations in an array, *expr*, that are nonzero (*true*) with the following command form:

```
find(expr)
```

To find the locations of pattern, *pattern*, in an array, *expr*, we can employ the following command form:

```
find(expr == pattern)
```

The function returns a vector of indices. We can use these indices in further computations, such as finding corresponding values in two lists.

The following call to *find* returns a vector, [2 4 5], indicating the indices of 'b' in ['a' 'b' 'c' 'b' 'b']:

```
pos = find(['a' 'b' 'c' 'b' 'b'] == 'b')
```

**Quick Review Question 10** Assign to *ages* a vector of 10 random integers with values between 21 and 23, inclusively. For example, *ages* might be a list of the ages of 10 patients. Then, using *find*, give a command to assign to *pos21* a vector of the indices where 21 occurs.

**Quick Review Question 11** Assign to *weight* a vector of 10 random integers between 100 and 250. For example, *weight* might store the weights of 10 patients. Using *weight* and *pos21* from the previous Quick Review Question, write an expression to display the weights of the patients who are 21 years old.