CHAPTER 3

Vectors and Arrays

Chapter Objectives

This chapter discusses the basic calculations involving rectangular collections of numbers in the form of arrays. For each of these collections, you will learn how to:

- Create them
- Manipulate them
- Access their elements
- Perform mathematical and logical operations on them

This study of arrays will introduce the first of many language characteristics that sets MATLAB apart from other languages: its ability to perform arithmetic and logical operations on collections of numbers as a whole. You need to understand how to create these collections, access the data in them, and manipulate the values in the collections with mathematical and logical operators. First, however, we need to understand the idea of functions built into the language.

3.1 Concept: Using Built-in Functions

We are familiar with the use of a trigonometric function like cos(θ) that consumes an angle in radians and produces the cosine of that angle. In general, a function is a named collection of instructions that operates on the data provided to produce a result according to the specifications of that function. In Chapter 5, we will see how to write our own functions. In this chapter, we will see the use of some of the functions built into MATLAB. At the end of each chapter that uses built-in functions, you will find a summary table listing the function specifications. For help on a specific function, you can either select the Help menu and look up the function or type the following in the Interactions window:

```
>> help <function name>
```

where `<function name>` is the name of a MATLAB function. This will produce a detailed discussion of the capabilities of that function.

3.2 Concept: Data Collections

Chapter 2 showed how to perform mathematical operations on single data items. This section considers the concept of grouping data items in general, and then specifically considers two very common ways to group data: in arrays and in vectors, which are a powerful subset of arrays.

3.2.1 Data Abstraction

It is frequently convenient to refer to groups of data collectively, for example, "all the temperature readings for May" or "all the purchases from Wal-Mart." This allows us not only to move these items around as a group, but also to consider mathematical or logical operations on these groups. For example, we could discuss the average, maximum, or minimum temperatures for a month, or that the cost of the Wal-Mart purchases had gone up 3%.

3.2.2 Homogeneous Collection

In Chapter 7, we will encounter more general collection implementations that allow items in a collection to be of different data types. The collections discussed in this chapter, however, will be constrained to accept only items of the same data type. Collections with this constraint are called **homogeneous collections.**

3.3 Vectors

A vector is an array with only one row of elements. It is the simplest means of grouping a collection of like data items. Initially we will consider vectors of numbers or logical values. Some languages refer to vectors as *linear arrays*

or *linear matrices*. As these names suggest, a vector is a one-dimensional grouping of data, as shown in Figure 3.1. Individual items in a vector are usually referred to as its **elements.** Vector elements have two separate and distinct attributes that make them unique in a specific vector: their *numerical value* and their *position* in that vector. For example, the individual number 66 is in the third position in the vector in Figure 3.1. Its value is 66 and its index is 3. There may be other items in the vector with the value of 66, but no other item will be located in this vector at position 3. Experienced programmers should note that due to its FORTRAN roots, indices in the MATLAB language start from 1 and not 0.

3.3.1 Creating a Vector

There are seven ways to create vectors that are directly analogous to the techniques for creating individual data items and fall into two broad categories:

- Creating vectors from constant values
- Producing new vectors with special-purpose functions

The following shows how you can create vectors from constant values:

- Entering the values directly, for example, `A = [2, 5, 7, 1, 3]` (the commas are optional and are frequently omitted)
- Entering the values as a range of numbers using the colon operator, for example, `B = 1:3:20`, where the first number is the starting value, the second number is the increment, and the third number is the ending value (you may omit the increment if the desired increment is 1)

The following introduces the most common MATLAB functions that create vectors from scratch:

- The `linspace(...)` function creates a fixed number of values between two limits, for example, `C = linspace (0, 20, 11)`, where the first parameter is the lower limit, the second parameter is the upper limit, and the third parameter is the number of values in the vector
- The functions `zeros(1,n), ones(1,n), rand(1,n)` (uniformly distributed random numbers), and `randn(1,n)` (random numbers with normal distribution) create vectors filled with `0, 1`, or random values between `0` and `1`.

Try working with vectors in Exercise 3.1.

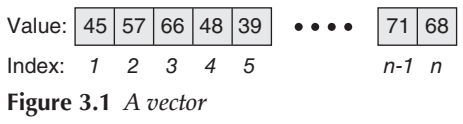

Figure 3.1 *A vector*

Exercise 3.1 Working with vectors

```
>> A = [2 5 7 1 3]
A =
     2 5 7 1 3
>> B = 1:3:20
B =
     1  4  7  10 13 16 19
>> C = linspace(0, 20, 11)
C =
     0  2  4   6  8 10 12 14 16 18 20
>> D = [4]
D =
     4
>> E = zeros(1,4)
E =
   0   0   0   0
```

Now, open the Variables tab and study the contents.

The Workspace window gives you three pieces of information about each of the variables you created: the name, the value, and the "class," which for now you can equate to "data type." Notice that if the size of the vector is small enough, the value field shows its actual contents; otherwise, you see a description of its attributes, like `<1 x 11 double>`. Exercise 3.1 deliberately created the vector `D` with only one element, and perhaps the result surprised you. `D` was presented in both the Interactions window and the Workspace window as if it were a scalar quantity. This is generally true in the MATLAB language—all scalar quantities are considered vectors of unit length.

3.3.2 Size of a Vector

A vector also has a specific attribute: its length (*n* in Figure 3.1). In most implementations, this length is fixed when the vector is created. However, as you will see shortly, the MATLAB language provides the ability to increase or decrease the size of a vector by inserting or selecting certain elements. MATLAB provides two functions to determine the size of arrays in general, and of vectors in particular. The function `size(V)` when applied to the vector V returns another vector containing two quantities: the number of rows in the vector (always `1`) and the number of columns (the length of the vector). The function `length(V)` returns the maximum value in the size of an array—for a vector, this is a number indicating its length.

3.3.3 Indexing a Vector

As mentioned earlier, each element in a vector has two attributes: its value and its position in the vector. You can access the elements in a vector in either of two ways: using a numerical vector or a logical vector. We refer to the process of accessing array elements by their position as "indexing."

Numerical Indexing The elements of a vector can be accessed individually or in groups by enclosing the index of zero or more elements in parentheses. Continuing Exercise 3.1, `A(3)` would return the third element of the vector `A, 7`. If you attempt to read beyond the length of the vector or below index 1, an error will result.

You can also change the values of a vector element by using an assignment statement where the left-hand side indexes that specific element (try Exercise 3.2).

A feature unique to the MATLAB language is its behavior when attempting to write beyond the bounds of a vector. While it is still illegal to write below the index 1, MATLAB will automatically extend the vector if you write beyond its current end. If there are missing elements between the current vector elements and the index at which you attempt to store a new value, MATLAB will zero-fill the missing elements. Try Exercise 3.3 to see how this works.

In Exercise 3.3 we asked to store a value in the eighth element of a vector with length 5. Rather than complaining, MATLAB was able to complete the instruction by doing two things automatically. It extended the length to 8 and stored the value 0 in the as yet unassigned elements. In these simple examples, we used a single number as the index. However, in general, we can use a vector of index values to index another vector. Furthermore, the size of the index vector does not need to match the size of the vector being indexed—it can be either shorter or longer. However, all values in an index vector must be positive; and if they are being used to extract values

Notes:

1. The key word `end` in an indexing context represents the index of the last element in that vector.

2. The vector generated by the colon operator does not necessarily include the ending value. In this case, since there are 8 values in the vector, end takes the value 8, but since that is not odd, the index vector is `[1 3 5 7]`

Exercise 3.2 Changing elements of a vector

Extending the exercise above:

```
>> A(5) = 42
A =
     2     5     7     1     42
```

Exercise 3.3 Extending a vector

Again extending the exercises above:

```
>> A(8) = 3
A =
     2    5    7    1    42    0    0    3
```

from a vector, the values must not exceed the length of that vector. Again continuing from Exercise 3.3, if we asked for `B = A(1:2:end)`, we would see the value of `B` to be `[2 7 42 0]`, the values of `A` in odd index positions. Later, we will see how to find the elements in `A` that have odd values.

Logical Indexing So far, the only type of data we have used has been numerical values of type `double`. The result of a logical operation, however, is data of a different type, with values either `true` or `false`. Such data are called **Boolean** or **logical** values. Like numbers, logical values can be assembled into arrays by specifying `true` or `false` values. For example, we might enter the following line in MATLAB to specify the variable `mask`:

```
>> mask = [true false false true]
mask =
      t    f    f    t[1]
```

We can index any vector with a logical vector as follows:

```
>> A = [2 4 6 8 10];
>> A(mask)
ans =
       2 8
```

When indexing with a logical vector, the result will contain the elements of the original vector corresponding in position to the true values in the logical index vector. The logical index vector can be either shorter or longer than the source vector; but if it is longer, all the values beyond the length of the source vector must be `false`.

3.3.4 Shortening a Vector

There are times when we need to remove elements from a vector. For example, if we had a vector of measurements from an instrument, and it was known that the setup for the third reading was incorrect, we would want to remove that erroneous reading before processing the data. To accomplish this, we make a rather strange use of the empty vector, `[]`. The empty vector, as its name and symbol suggest, is a vector with no elements in it. When you assign the empty vector to an element in another vector—say, `A`—that element is removed from `A`, and `A` is shortened by one element. Try Exercise 3.4.

Exercise 3.4 Shortening a vector

Using the vector A from Exercise 3.3:

```
>> A(4) = []
A =
     2   5   7   42   0   0   3
```

[1]If you are using MATLAB, logical vectors are presented with values 0 or 1, but they are not numerical values and should not be used as such

As you can see, we asked for the fourth element to be removed from a vector initially with eight elements. The resulting vector has only seven elements, and the fourth element, originally with value 1, has been removed.

3.3.5 Operating on Vectors

The essential core of the MATLAB language is a rich collection of tools for manipulating vectors and arrays. This section first shows how these tools operate on vectors, and then generalizes to how they apply to arrays (multi-dimensional vectors) and, later, matrices. Three techniques extend directly from operations on scalar values:

- Arithmetic operations
- Logical operations
- Applying library functions

Two techniques are unique to arrays in general, and to vectors in particular:

- Concatenation
- Slicing (generalized indexing)

Arithmetic Operations Arithmetic operations can be performed collectively on the individual components of two vectors as long as both vectors are the same length, or one of the vectors is a scalar (i.e., a vector of length 1). Addition and subtraction have exactly the syntax you would expect, as illustrated in Exercise 3.5. Multiplication, division, and exponentiation, however, have a small syntactic idiosyncrasy related to the fact that these are element-by-element operations, not matrix operations. We will discuss matrix operations in Chapter 12. When the MATLAB language was designed, the ordinary symbols (`*`,`/`, and `^`) were reserved for matrix operations. However, since element-by-element multiplicative operations are fundamentally different from matrix operations, a new set of operators is required to specify these operations. The symbols `.*`, `./`, and `.^` (the dots are part of the operators, but the commas are not) are used respectively for element-by-element multiplication, division, and exponentiation. Note that because matrix and element-by-element addition and subtraction are identical, no special operation symbols are required for + and –.

Common Pitfalls 3.1

Shortening a vector is very rarely the right solution to a problem and can lead to logical difficulties. Wherever possible, you should use indexing to copy the elements you want to keep rather than using `[ ]` to erase elements you want to remove.

Here, we first see the addition and multiplication of a vector by a scalar quantity, and then element-by-element multiplication of A and B. The first error is generated because we omitted the '.' on the multiply

Exercise 3.5 Using vector mathematics

```
>> A = [2 5 7 1 3];
>> A + 5
ans =
     7     10     12     6     8
>> A .* 2
ans =
     4     10     14     2     6
>> B = -1:1:3
B =
    -1      0      1     2     3
>> A .* B % element-by-element multiplication
ans =
    -2      0      7     2     9
>> A * B % matrix multiplication!!
??? Error using ==> mtimes
Inner matrix dimensions must agree.
>> C = [1 2 3]
C =
     1     2     3
>> A .* C % A and C must have the same length
??? Error using ==> times
Matrix dimensions must agree.
```

symbol, thereby invoking matrix multiplication, which is improper with the vector A and B. The second error occurs because two vectors involved in arithmetic operations must have the same size. Notice, incidentally, the use of the % sign indicating that the rest of the line is a comment.

You can change the signs of all the values of a vector with the unary minus (-) operator.

Logical Operations In the earlier discussion about logical indexing, you might have wondered why you would ever use that. In this section, we will see that logical operations on vectors produce vectors of logical results. We can then use these logical result vectors to index vectors in a style that makes the logic of complex expressions very clear. As with arithmetic operations, logical operations can be performed element-by-element on two vectors as long as both vectors are the same length, or if one of the vectors is a scalar (i.e., a vector of length 1). The result will be a vector of logical values with the same length as the longer of the original vector(s).

Try Exercise 3.6 to see how vector logical expressions work.

First we built the vectors A and B, and then we performed two legal logical operations: finding where A is not less than 5, and where each

Exercise 3.6 Working with vector logical expressions

```
>> A = [2 5 7 1 3];
>> B = [0 6 5 3 2];
>> A >= 5
ans =
    0    1    1    0    0
>> A >= B
ans =
    1    0    1    0
>> C = [1 2 3]
>> A > C
??? Error using ==> gt
Matrix dimensions must agree.
```

Exercise 3.7 Working with logical vectors

```
>> A = [true true false false];
>> B = [true false true false];
>> A & B
ans =
    1    0    0    0
>> A | B
ans =
    1    1    1    0
>> C = [1 0 0]
>> A & C
??? Error using ==> and
Matrix dimensions must agree.
```

element of `A` is not less than the corresponding element of `B`. As with arithmetic operations, an error occurs if you attempt a logical operation with vectors of different sizes (neither size being 1).

Logical operators can be assembled into more complex operations using logical and `(&)` and or `(|)` operators. These operators actually come in two flavors: `&/|` and `&& / ||`. The single operators operate on logical arrays of matching size to perform element-wise matches of the individual logical values. The doubled operators combine individual logical results and are usually associated with conditional statements (see Chapter 4). Try Exercise 3.7 to see how logical operators work.

In Exercise 3.7, we combine two logical vectors of the same length successfully, but fail, as with arithmetic operations, to combine vectors of different lengths. If you need the indices in a vector where the elements of a logical vector are `true`, the function `find(...)` accomplishes this by consuming an array of logical values and producing a vector of the positions of the `true` elements.

Exercise 3.8 Using the find(…) function

```
>> A = [2 5 7 1 3];
>> A > 4
ans =
     0 1 1 0 0
>> find(A > 4)
ans =
     2 3
```

Try Exercise 3.8 to see how this function works.

You can invert the values of all elements of a logical vector (changing `true` to `false` and `false` to `true`) using the unary not operator, `~`. For example:

```
>> na = ~[true true false true]
na = 0 0 1 0
```

As you can see, each element of `na` is the logical inverse of the corresponding original element. As is usual with arithmetic and logical operations, the precedence of operators governs the order in which operations are performed. Table 3.1 shows the operator precedence in the MATLAB language. Operations listed on the same row of the table are performed from left to right. The normal precedence of operators can be overruled by enclosing preferred operations in parentheses: `(...)`.

Applying Library Functions The MATLAB language defines a rich collection of mathematical functions that cover mathematical, trigonometric, and statistics capabilities. A partial list is provided in Appendix A. For a complete

Table 3.1 Operator precedence

Operators	Description
`.', .^`	Scalar transpose and power
`', ^`	Matrix transpose and power
`+, -, ~`	Unary operators
`.*,./,.\,*,/,\`	Multiplication, division, left division
`+, -`	Addition and subtraction
`:`	Colon operator
`<, <=, >=, >, ==, ~=`	Comparison
`&`	Element-wise AND
`|`	Element-wise OR
`&&`	Logical AND
`||`	Logical OR

list of those implemented in MATLAB, refer to the Help menu option in the MATLAB tool bar. With few exceptions, all functions defined in the MATLAB language accept vectors of numbers rather than single values and return a vector of the same length. The following functions deserve special mention because they provide specific capabilities that are frequently useful:

- `sum(v)` and `mean(v)` consume a vector and return the sum and mean of all the elements of the vector respectively.
- `min(v)` and `max(v)` return two quantities: the minimum or maximum value in a vector, as well as the position in that vector where that value occurred. For example:

```
> [value where] = max([2 7 42 9 -4])
value = 42
where = 3
```

indicates that the largest value is `42`, and it occurs in the third element of the vector. You will see in Chapter 5 how to implement returning multiple results from a function.

- `round(v)`, `ceil(v)`, `floor(v)`, and `fix(v)` remove the fractional part of the numbers in a vector by conventional rounding, rounding up, rounding down, and rounding toward zero, respectively.

Concatenation In Section 3.3.1, we saw the technique for creating a vector by assembling numbers between square brackets:

```
A = [2 5 7 1 3]
```

This is in fact a special case of concatenation. The MATLAB language lets you construct a new vector by concatenating other vectors:

```
A = [B C D ... X Y Z]
```

where the individual items in the brackets may be any vector defined as a constant or variable, and the length of `A` will be the sum of the lengths of the individual vectors. The simple vector constructor in Section 3.3.1 is a special case of this rule because each number is implicitly a 1×1 vector. The result is therefore a $1 \times N$ vector, where `N` is the number of values in the brackets. Try concatenating the vectors in Exercise 3.9.

Exercise 3.9 Concatenating vectors

```
>> A = [2 5 7];
>> B = [1 3];
>> [A B]
ans =
2 5 7 1 3
```

Notice that the resulting vector is not nested like [[2 5 7], [1 3]] but is completely "flat."

Slicing is the name given to complex operations where elements are copied from specified locations in one vector to different locations in another vector. As we saw earlier, the basic operation of extracting and replacing the elements of a vector is called indexing. Furthermore, we saw that indexing is not confined to single elements in a vector; you can also use **vectors of indices.** These index vectors either can be the values of previously named variables, or they can be created anonymously as they are needed. When you index a single element in a vector—for example, `A(4)`—you are actually creating an anonymous `1 × 1` index vector, `4`, and then using it to access the specified element(s) from the array `A`.

Creating anonymous index vectors as needed makes some additional features of the colon operator available. The general form for generating a vector of numbers is: `<start> : <increment> : <end>`. We already know that by omitting the `<increment>` portion, the default increment is 1. When used anonymously while indexing a vector, the following features are also available:

- The key word `end` is defined as the length of the vector
- The operator `:` by itself is short for `1:end`

Finally, as you saw earlier, it is legal to index with a vector of logical values. For example, if `A` is defined as:

```
A = [2 5 7 1 3];
```

then `A([false true false true])` returns:

```
ans =
      5      1
```

yielding a new vector containing only those values of the original vector where the corresponding logical index is `true`. This is extremely useful, as you will see later in this chapter, for indexing items in a vector that match a specific test.

The general form of statements for slicing vectors is:

```
B(<rangeB>) = A(<rangeA>)
```

where `<rangeA>` and `<rangeB>` are both index vectors, `A` is an existing array, and `B` can be an existing array or a new array. The values in `B` at the indices in `rangeB` are assigned the values of `A` from `rangeA`. The rules for use of this template are as follows:

- Either the size of `rangeB` must be equal to the size of `rangeA` or `rangeA` must be of size 1
- If `B` did not exist before this statement was implemented, it is zero filled where assignments were not explicitly made

- If B did exist before this statement, the values not directly assigned in rangeB remain unchanged

Study the comments in Listing 3.1 and do Exercise 3.10.

Listing 3.1 Vector indexing script

```
A = [2 5 7 1 3 4];
odds = 1:2:length(A);
disp('odd values of A using predefined indices')
A(odds)
disp('odd values of A using anonymous indices')
A(1:2:end)
disp('put evens into odd values in a new array')
B(odds) = A(2:2:end)
disp('set the even values in B to 99')
B(2:2:end) = 99
disp('find the small values in A')
small = A < 4
disp('add 10 to the small values')
A(small) = A(small) + 10
disp('this can be done in one ugly operation')
A(A < 10) = A(A < 10) + 10
```

Exercise 3.10 Running the vector indexing script

Execute the script in Listing 3.1.You should see the following output:

```
odd values of A using predefined indices
ans =
     2     7     3
odd values of A using anonymous indices
ans =
     2     7     3
put even values into odd values in a new array
B =
     5     0     1     0     4
set the even values in B to 99
B =
     5    99     1    99     4
find the small values in A
small =
     1     0     0     1     1     0
add 10 to the small values
A =
    12     5     7    11    13     4
this can be done in one ugly operation
A =
    12     5     7    11    13     4
>>
```

In Listing 3.1:

Line 1: Creates a vector `A` with five elements.

Line 2: When predefining an index vector, if you want to refer to the size of a vector, you must use either the `length(...)` function or the `size(...)` function.

Line 3: The `disp(...)` function shows the contents of its parameter in the Interactions window, in this case: `'odd values of A using predefined indices'`. We use `disp(...)` rather than comments because comments are visible only in the script itself, not in the program output, which we need here.

Line 4: Using a predefined index vector to access elements in vector `A`. Since no assignment is made, the variable `ans` takes on the value of a three-element vector containing the odd-numbered elements of `A`. Notice that these are the odd-numbered elements, not the elements with odd values.

Line 6: The anonymous version of the command given in Line 4. Notice that the anonymous version allows you to use the word `end` within the vector meaning the index of its last element.

Line 8: Since `B` did not previously exist (a good reason to run the `clear` command at the beginning of a script is to be sure this is true), a new vector is created with five elements (the largest index assigned in `B`). Elements in `B` at positions less than five that were not assigned are zero filled.

Line 10: If you assign a scalar quantity to a range of indices in a vector, all values at those indices are assigned the scalar value.

Line 12: Logical operations on a vector produce a vector of Boolean results. This is not the same as typing `small = [1 0 0 1 1 0]`. If you want to create a logical vector, you must use `true` and `false`, for example:

```
small = [true false false true true false]
```

Line 14: This is actually performing the scalar arithmetic operation `+ 10` on an anonymous vector of three elements, and then assigning those values to the range of elements in `A`.

Line 16: Not only is this unnecessarily complex, but it is also less efficient because it is applying the logical operator to `A` twice. It is better to use the form in Line 14.

3.4 Engineering Example—Forces and Moments

Vectors are ideal representations of the concept of a vector used in physics. Consider two forces acting on an object at a point P, as shown in Figure 3.2. Calculate the resultant force at P, the unit vector in the

direction of that resultant, and the moment of that force about the point M. We can represent each of the vectors in this problem as a MATLAB vector with three components: the x, y, and z values of the vector. The solution to this problem for specific vectors is shown in Listing 3.2.

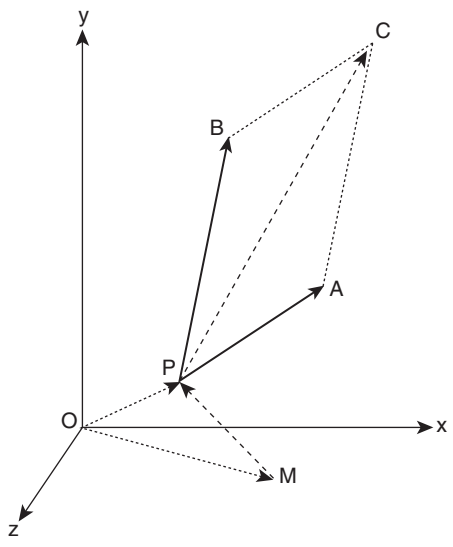

Figure 3.2 *Vector analysis problem*

Listing 3.2 Script to solve vector problems

```
1. PA = [0 1 1]
2. PB = [1 1 0]
3. P = [2 1 1]
4. M = [4 0 1]
   % find the resultant of PA and PB
5. PC = PA + PB
   % find the unit vector in the direction of PC
6. mag = sqrt(sum(PC.^2))
7. unit_vector = PC/mag
   % find the moment of the force PC about M
   % this is the cross product of MP and PC
8. MP = P - M
9. moment = cross( MP, PC )
```

Common Pitfalls 3.2

After any nontrivial computation, a good engineer will always perform a sanity check on the answers. When you run the code for this problem, the answers returned are:

```
PC = [  1   2   1]
unit_vector = [0.4082   0.8165   0.4082]
moment = [  1   2   -5]
```

To check the moment result, visualize the rotation of PC about M and apply the right-hand rule to find the axis of rotation of the moment. Roughly speaking, the right-hand rule states that the direction of the moment is the direction in which a normal, right-handed screw at point M would turn under the influence of this force. Without being too accurate, we can conclude that the axis of the moment is approximately along the negative z-axis, an estimate confirmed by the result shown.

In Listing 3.2:

Lines 1–4: Typical initial values for the problem.

Line 5: PC is the sum of the vectors PA and PB.

Lines 6–7: The unit vector along PC is PC divided by its magnitude. The magnitude is the square root of the sum of the squares of the individual components.

Line 8: The vector PM is the vector difference between P and M.

Line 9: There is a built-in function, `cross(..)`, to compute the cross product of two vectors.

3.5 Arrays

In Section 3.2, we saw that a vector is the simplest way to group a collection of similar data items. We will now extend these ideas to include arrays of multiple dimensions, initially confined to two dimensions. Each row will have the same number of columns, and each column will have the same number of rows.

At this point, we will refer to these collections as *arrays* to distinguish them from the *matrices* discussed in Chapter 12. While arrays and matrices are stored in the same way, they differ in their multiplication, division, and exponentiation operations. Figure 3.3 illustrates a typical two-dimensional array A with m rows and n columns, commonly referred to as an m × n array.

$$A_{(m \times n)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Figure 3.3 *An array*

3.5.1 Properties of an Array

As with vectors, individual items in an array are referred to as its *elements*. These elements also have the unique attributes combining their value and their position. In a two-dimensional array, the position will be the row and column (in that order) of the element. In general, in an n-dimensional array, the element position will be a vector of *n* index values.

When applied to an array `A` with `n` dimensions, the function `size(...)` will return the information in one of two forms.

- If called with a single return value like `sz = size(A)`, it will return a vector of length `n` containing the size of each dimension of the array.
- If called with multiple return values like `[rows, cols] = size(A)`, it returns the individual array dimension up to the number of values requested. To avoid erroneous results, you should always provide as many variables as there are dimensions of the array.

The `length(...)` function returns the maximum dimension of the array. So if we created an array `A` dimensioned `2 × 8 × 3`, `size(A)` would return `[2 8 3]` and `length(A)` would return 8.

The transpose of an `m × n` array, indicated by the apostrophe character (`'`) placed after the array identifier, returns an `n × m` array with the values in the rows and columns interchanged. Figure 3.4 shows a transposed array.

A number of special cases arise that are worthy of note:

- When a 2-D matrix has the same number of rows and columns, it is called **square.**
- When the only nonzero values in an array occur when the row and column indices are the same, the array is called **diagonal.**
- When there is only one row, the array is a row vector, or just a *vector* as you saw earlier.
- When there is only one column, the array is a **column vector,** the transpose of a row vector.

$$A'_{(n \times m)} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & & \ddots & \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}$$

Figure 3.4 *Transpose of an array*

3.5.2 Creating an Array

Arrays can be created either by entering values directly or by using one of a number of built-in functions that create arrays with specific characteristics.

- As with vectors, you can directly enter the values in an array using either a semicolon (`;`) or a new line to indicate the end of a row, for example: `A = [2, 5, 7; 1, 3, 42].`
- The functions `zeros(m, n)` and `ones(m, n)` create arrays with m rows and n columns filled with zeros and ones, respectively.
- The function `rand(m, n)` fills an array with random numbers in the range `0 .. 1.`
- The function `randn(m, n)` fills an array with random numbers normally distributed about `0` with a standard deviation of `1.`
- The function `diag(...)` takes several forms, the most useful of which are `diag(A)`, where `A` is an array, that returns its diagonal as a vector, and `diag(V)`, where `V` is a vector, that returns a square matrix with that diagonal. Type help diag in the Command window for a full description of the capabilities of `diag(...)`
- The MATLAB language also defines the function `magic(m)`, which fills a square matrix with the numbers `1` to m^2 organized in such a way that its rows, columns, and diagonals all add up to the same value.

Try Exercise 3.11 to practice working with arrays.

3.5.3 Accessing Elements of an Array

The elements of an array may be addressed by enclosing the indices of the required element in parentheses, with the first index being the row index and the second index the column index. Considering the values produced by Exercise 3.11, `A(2, 3)` would return the element in the second row, third column: `42.` If you were to attempt to read outside the length of the rows or columns, an error would result.

We can also store values that are elements of an array. For example, continuing Example 3.11, `A(2, 3) = 0` would result in this answer:

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

As with vectors, MATLAB will automatically extend the array if you write beyond its boundaries. If there are missing elements between the current array elements and the index at which you attempt to store a new value,

Exercise 3.11 Creating arrays

```
>> A = [2, 5, 7; 1, 3, 42]
A =
     2     5     7
     1     3    42
>> z = zeros(3,2)
z =
     0     0
     0     0
     0     0
>> [z ones(3, 4)]  % concatenating arrays
ans =
     0     0     1     1     1     1
     0     0     1     1     1     1
     0     0     1     1     1     1
>> rand(3,4)
ans =
    0.9501    0.4860    0.4565    0.4447
    0.2311    0.8913    0.0185    0.6154
    0.6068    0.7621    0.8214    0.7919
>> rand(size(A))
ans =
    0.9218    0.1763    0.9355
    0.7382    0.4057    0.9169
>> diag(A)
ans =
     2
     3
>> diag(diag(A))
ans =
     2     0
     0     3
>> magic(4)
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>>
```

the missing elements will be zero filled. For example, again continuing Example 3.11, `A(4, 1) = 3` would result in this answer:

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

3.5.4 Removing Elements of an Array

You can remove elements from arrays in the same way that you remove elements from a vector. However, since the arrays must remain rectangular,

elements have to be removed as complete rows or columns. For example, for the array `A` in the previous section, entering `A(3, :) = []` would remove all elements from the third row, and the result would be:

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

Similarly, if `A(:, 3) = []` was then entered, the result would be:

```
A =
           2      5
           1      3
           3      0
```

3.5.5 Operating on Arrays

This section discusses how array operations extend directly from vector operations: arithmetic and logical operations, the application of functions, concatenation, and slicing. This section will also discuss two topics peculiar to arrays: reshaping and linearizing arrays.

Common Pitfalls 3.3

Removing rows or columns from an array is very rarely the right solution to a problem and can lead to logical difficulties. Wherever possible, use indexing to copy the rows and columns you want to keep.

Common Pitfalls 3.4

Performing array multiplication, division, or exponentiation without appending a dot operator requests one of the specialized matrix operations that will be covered in Chapter 12. The error message when this occurs is quite obscure if you are not expecting it:

```
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Even more obscure is the case where the dimensions of the arrays happen to be consistent (when multiplying square arrays), but the results are not the scalar products of the two arrays.

Array Arithmetic Operations Arithmetic operations can be performed collectively on the individual components of two arrays as long as both arrays have the same dimensions or one of them is a scalar (i.e., has a vector of length 1). Addition and subtraction have exactly the syntax you would expect, as shown in Exercise 3.12. Multiplication, division, and exponentiation, however, *must* use the "dot operator" symbols: `.*`, `./`, and `.^` (the dot is part of the symbol, but the commas are not) for scalar multiplication, division, and exponentiation.

Array Logical Operations As with vectors, logical array operations can be performed collectively on the individual components of two arrays as long as both arrays have the same dimensions or one of the arrays is a scalar (i.e., has a vector of length 1). The result will be an array of logical values with the same size as the original array(s). Do Exercise 3.13 to see how array logical operations work. Here, we successfully compare the array A to a scalar value, and to the array B that has the same dimensions as A. However, comparing to the array C that has the same number of elements but the wrong shape produces an error.

Exercise 3.12 Working with array mathematics

```
>> A = [2 5 7
    1 3 2]
A =
     2     5     7
     1     3     2
>> A + 5
ans =
     7    10    12
     6     8     7
B = ones(2, 3)
B =
     1     1     1
     1     1     1
>> B = B * 2
B =
     2     2     2
     2     2     2
>> A.*B % scalar multiplication
ans =
     4    10    14
     2     6     4
>> A*B % matrix multiplication does not work here
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Exercise 3.13 Working with array logical operations

```
>> A = [2 5; 1 3]
A =
    2    5
    1    3
>> B = [0 6; 3 2];
>> A >= 4
ans =
    0    1
    0    0
>> A >= B
ans =
    1    0
    0    1
>> C = [1 2 3 4]
>> A > C
??? Error using ==> gt
Matrix dimensions must agree.
```

Applying Library Functions In addition to being able to consume vectors, most mathematical functions in the MATLAB language can consume an array of numbers and return an array of the same shape. The following

functions deserve special mention because they are exceptions to this rule and provide specific capabilities that are frequently useful:

- `sum(v)` and `mean(v)` when applied to a 2-D array return a row vector containing the sum and mean of each column of the array, respectively. If you want the sum of the whole array, use `sum(sum(v))`.
- `min(v)` and `max(v)` return two row vectors: the minimum or maximum value in each column and also the row in that column where that value occurred. For example:

```
>> [values rows] = max([2  7 42;
                        9 14  8;
                       10 12 -6])
values = [10 14 42]
rows = [3 2 1]
```

This indicates that the maximum values in each column are 10, 14, and 42, respectively, and they occur in rows 3, 2, and 1. If you really need the row and column containing, say, the maximum value of the whole array, continue the preceding example with the following lines:

```
>> [value col] = max(values)
value = 42
col = 3
```

This finds the maximum value in the whole array and determines that it occurs in column 3. So to determine the row in which that maximum occurred, we index the vector of row maximum locations, `rows`, with the column in which the maximum occurred.

```
>> row = rows(col)
row = 1
```

Therefore, we correctly conclude that the maximum number in this array is 42, and it occurs at row 1, column 3.

Array Concatenation The MATLAB language permits programmers to construct a new array by concatenating other arrays in the following ways:

- Horizontally, as long as each component has the same number of rows:

```
A = [B C D ... X Y Z]
```

- Vertically, as long as each has the same number of columns:

```
A = [B; C; D; ... X; Y; Z]
```

The result will be an array with that number of rows and a number of columns equaling the sum of the number of columns in each individual item.

Exercise 3.14 gives you the opportunity to concatenate an array.

Exercise 3.14 Concatenating an array

```
>> A = [2 5; 1 7];
>> B = [1 3]';   % makes a column vector
>> [A B]
ans =
2 5 1
1 7 3
```

Style Points 3.1

The MATLAB language does not encourage concatenating data of different classes. However, it tolerates such concatenation with sometimes odd results. If you really want to achieve this in an unambiguous manner, you should explicitly cast the data to the same class.

Slicing Arrays The general form of statements for moving sections of one array into sections of another is as follows:

```
B(<rangeBR>, <rangeBC>) =
A(<rangeAR>,<rangeAC>)
```

where each `<range..>` is an index vector, `A` is an existing array, and `B` can be an existing array or a new array. The values in `B` at the specified indices are all assigned the corresponding values copied from `A`. The rules for using this template are as follows:

- Either each dimension of each sliced array must be equal, or the size of the slice from `A` must be `1 × 1`.
- If `B` did not exist before this statement was implemented, it would be zero filled where assignments were not explicitly made.
- If `B` did exist before this statement, the values not directly assigned would remain unchanged.

Reshaping Arrays Occasionally, it is useful to take an array with one set of dimensions and reshape it to another set. The function `reshape( ... )` accomplishes this. The command `reshape(A, rows, cols, ...)` will take the array `A`, whatever its dimensions, and reform it into an array sized (`rows × cols × ...`) out to as many dimensions as desired. However, `reshape(...)` neither discards excess data nor pads the data to fill any empty space. The product of all the original dimensions of `A` must equal the product of the new dimensions. Try Exercise 3.15 to see how to reshape an array.

Here, we first take a `1 × 10` array, `A`, and attempt to reshape it to `4 × 3`. Since the element count does not match, an error results. When we concatenate two zeros to the array `A`, it has the right element count and the reshape succeeds.

Linearized Arrays A discussion of arrays would not be complete without revealing an infamous secret of the MATLAB language: multi-dimensional arrays are not stored in some nice, rectangular chunk of memory. Like all other blocks of memory, the block allocated for an array is sequential, and the array is stored in that space in column order. Normally, if MATLAB behaved as we "have a right to expect," we would not care how an array is

Exercise 3.15 Reshaping an array

```
>> A = 1:10
A =
    1     2     3     4    5     6     7    8     9     10
>> reshape(A, 4, 3)
??? Error using ==> reshape
To RESHAPE the number of elements must not change.
>> reshape([A 0 0], 4, 3)
ans =
      1    5     9
      2    6    10
      3    7     0
      4    8     0
```

stored. However, there are circumstances under which the designers of MATLAB needed to expose this secret. The primary situation in which array linearization becomes evident is the mechanization of the `find(...)` function. If we perform a logical operation on an array, the result is an array of logical values of the same size as the original array. In general, the `true` values would be scattered randomly about that result array. If we wanted to convert this to a collection of indices, what would we expect to see? The `find(...)` function has two modes of operation: we can give it separate variables in which to store the rows and columns by saying `[rows cols] = find(...)` or we can receive back just one result by calling `ndx = find(...)`. Indexing with this result exposes the linearized nature of arrays. The way this feature manifests itself is shown in Exercise 3.16.

Here, we build a `4 × 3` array `A` and calculate the logical array where `A` is greater than `5`. When we save the result of finding these locations in the variable `ix`, we see that this is a vector of values. If we count down the columns from the top left, we see that the second, seventh, eighth, and eleventh values in the linearized version of `A` are indeed `true`. We also see that it is legal to use this linearized index vector to access the values in the original array—in this case, to add 3 to each one. Finally, we would expect a loud complaint when trying to reference the eleventh element of an array with only three rows. In fact MATLAB "unwinds" the storage of the array, counts down to the eleventh entry—3 for column 1, 3 for column 2, and 3 for column 3—and then extracts the second element of column 4.

To understand all these array manipulation ideas fully, you should work carefully through the script in Listing 3.3, study the explanatory notes that follow, and do Exercise 3.17.

Style Points 3.2

1. It is best not to expose the detailed steps of finding logical results in arrays, but to use an integrated approach:

```
A(A>5) = A(A>5) + 3
```

This produces the expected answers without exposing the nasty secrets underneath.

2. Never use an array linearization as part of your program logic. It makes the code hideous to look at and/or understand, and it is never the "only way to do" anything.

Exercise 3.16 Linearizing an array

```
>> A = [2 5 7 3
        8 0 9 42
        1 3 4 2]
A =
     2    5     7     3
     8    0     9    42
     1    3     4     2
>> A > 5
ans =
     0    0    1     0
     1    0    1     1
     0    0    0     0
>> ix = find(A > 5)
ix =
     2    7     8    11
>> A(ix) = A(ix) + 3
A =
     2    5    10     3
    11    0    12    45
     1    3     4     2
>> A(11)
ans =
    42          % (sigh!)
```

Listing 3.3 Array manipulation script

```
 1. A = [2 5 7 3
 2.      1 3 4 2]
 3. [rows, cols] = size(A)
 4. odds = 1:2:cols
 5. disp('odd columns of A using predefined indices')
 6. A(:, odds)
 7. disp('odd columns of A using anonymous indices')
 8. A(end, 1:2:end)
 9. disp('put evens into odd values in a new array')
10. B(:, odds) = A(:, 2:2:end)
11. disp('set the even values in B to 99')
12. B(1, 2:2:end) = 99
13. disp('find the small values in A')
14. small = A < 4
15. disp('add 10 to the small values')
16. A(small) = A(small) + 10
17. disp('this can be done in one ugly operation')
18. A(A < 4) = A(A < 4) + 10
19. small_index = find(small)
20. A(small_index) = A(small_index) + 100
```

In Listing 3.3:

Lines 1 and 2: Create a 2×4 array A.

Line 3: Determines the number of rows and columns.

Exercise 3.17 Running the array manipulation script

Run the script in Listing 3.3 and observe the results:

```
odds =
      1      3
odd columns of A using predefined indices
ans = 2      7
      1      4
odd columns of A using anonymous indices
ans =
      1      4
put evens into odd values in a new array
B =
      5      0     3
      3      0     2
set the even values in B to 99
B =
      5     99     3
      3      0     2
find the small values in A
small =
      1 0 0 1
      1 1 0 1
add 10 to the small values
A =
     12     5    7   13
     11    13    4   12
this can be done in one ugly operation
A =
     12     5    7   13
     11    13    4   12
do the same thing with indices
small_index =
      1
      2
      4
      7
      8
A =
    112     5    7   113
    111   113    4   112
```

Line 4: Builds a vector `odds` containing the indices of the odd numbered columns.

Line 6: Uses `odds` to access the columns in `A`. The `:` specifies that this is using all the rows.

Line 8: The anonymous version of the command in Line 6. Notice that you can use the keyword `end` in any dimension of the array to represent the last index on that dimension.

Line 10: Because `B` did not previously exist (a good reason to have `clear` at the beginning of the script to be sure this is true), a new array is created. Elements in `B` that were not assigned are zero filled.

Style Points 3.3

1. Do not forget to begin all scripts with the two commands `clear` and `clc`.

a. `clear` empties the current Workspace window of all variables and prevents the values of old variables from causing strange behavior in this script.

b. `clc` clears the Command window to prevent confusion about whether a display was caused by this script or some earlier activity.

2. It is better to enter a few lines at a time and run each version of the script incrementally, rather than editing one huge script and running the whole thing for the first time. When you have added only a few lines to a previously working script, it is easy to locate the source of logic problems that arise.

3. It is very tempting to build large, complex vector operation expressions that solve messy problems "in one line of code." While this might be an interesting mental exercise, the code is much more maintainable if the solution is expressed one step at a time using intermediate variables.

Line 12: Puts `99` into selected locations in `B`.

Line 14: Logical operations on arrays produce an array of logical results.

Line 16: Adds `10` to the values in `A` that are small.

Line 18: Not only is this unnecessarily complex, but it is also less efficient because it is applying the logical operator to `A` twice.

Line 19: The function `find(...)` actually returns a column vector of the index values in the linearized version of the original array, as shown in Exercise 3.16

Line 20: As illustrated in Line 18, it is not necessary to use `find(...)` before indexing an array. However, this command does work.

Notice that all the results are consistent with our expectations.

3.6 Engineering Example—Computing Soil Volume

When digging the foundations for a building, it is necessary to estimate the amount of soil that must be removed. The first step is to survey the land on which the building is to be built, which results in a rectangular grid defining the height of each grid point as shown in Figure 3.5.

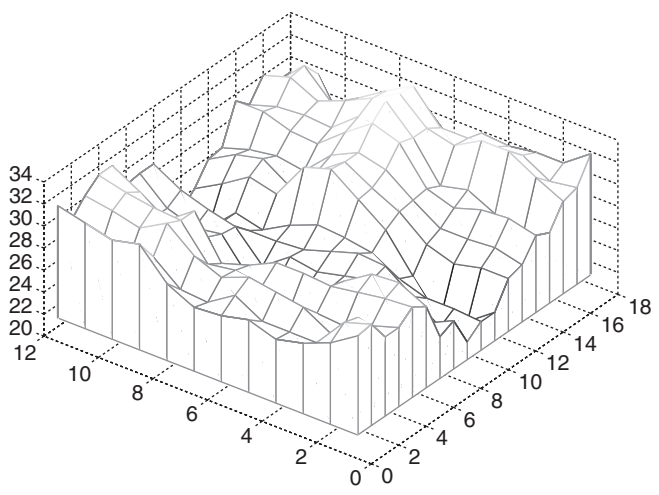

Figure 3.5 *Landscape survey*

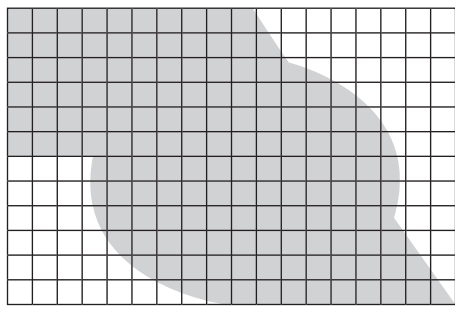

Figure 3.6 *Calculating soil volume*

The next step is to consider an architectural drawing of the basement of the building as shown in Figure 3.6. The shaded areas indicate those places where the soil really must be removed to make the building foundation. We can estimate from this figure the fraction of each surveyed square (for our purposes, a number between 0 and 1) where the soil must actually be removed.

The total amount of soil to move is then the sum of the individual square depths multiplied by the area in each square to be removed. The code in Listing 3.4 solves this problem.

Listing 3.4 Script to compute total soil

```
% soil depth data for each square produced by the survey
 1. dpth = [8  8  9  8  8  8  8  8  7  8  7  7  7  7  8  8  8  7
 2.         8  8  8  8  8  8  8  7  7  7  7  7  8  7  8  8  8  7
 3.         8  8  8  8  7  7  8  7  8  8  8  8  8  7  8  8  8  8
 4.         7  7  7  8  7  8  8  8  8  8  8  8  7  6  7  7  7  7
 5.         8  8  8  8  8  8  8  8  7  7  7  7  7  6  6  7  7  8
 6.         8  7  7  8  7  7  8  7  7  7  7  7  7  7  7  7  7  8
 7.         9  8  8  9  8  7  8  7  7  7  7  7  6  7  6  7  7  8
 8.         8  8  8  9  9  8  8  8  7  6  6  6  6  7  7  8  7  8
 9.         9  8  8  7  7  7  7  7  7  6  6  7  7  7  8  8  7  8
10.         9  8  8  7  7  7  6  7  7  6  6  8  8  8  9  9  7  8
11.         9  9  8  8  8  8  7  7  7  7  7  8  8  9  9  9  8  8
12.        9  8  8  7  7  8  7  7  7  7  8  8  9  9  9  8  7  8];
% estimated proportion of each square that should be excavated
13. area = [1  1  1  1  1  1  1  1  1  1 .3  0  0  0  0  0  0  0
14.         1  1  1  1  1  1  1  1  1  1 .7  0  0  0  0  0  0  0
15.         1  1  1  1  1  1  1  1  1  1  1 .8 .4  0  0  0  0  0
16.         1  1  1  1  1  1  1  1  1  1  1  1  1 .8 .3  0  0  0
17.         1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 .7 .2  0  0
18.         1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 .6  0  0
19.         0  0  0 .7  1  1  1  1  1  1  1  1  1  1  1  1 .8  0  0
20.         0  0  0 .7  1  1  1  1  1  1  1  1  1  1  1  1 .7  0  0
21.         0  0  0 .4  1  1  1  1  1  1  1  1  1  1  1  1 .6  0  0
22.         0  0  0 .1 .8  1  1  1  1  1  1  1  1  1  1  1  1 .4  0
23.         0  0  0  0 .2 .7  1  1  1  1  1  1  1  1  1  1  1 .9 .1
24.        0  0  0  0  0  0 .4 .8 .9  1  1  1  1  1  1  1  1  1 .6];
25. square_volume = dpth .* area;
26. total_soil = sum(sum(square_volume))
```

When you run this script, it produces the answer: 1,117.5 cubic units.

Common Pitfalls 3.5

The code in Listing 3.4 produces an answer around 1,120, and we should ask whether this is reasonable. There are 12×18 squares, each with area 1 unit, about 80% of which are to be excavated, giving a surface area of about 180 square units. The average depth of soil is about 7 units, so the answer ought to be about $180 \times 7 \cong 1{,}300$ cubic units. This is reasonably close to the computed result.

Chapter Summary

This chapter introduced you to vectors and arrays. For each collection, you saw how to:

- Create a vectors and arrays by concatenation and a variety of special-purpose functions
- Access and remove elements, rows, or columns
- Perform mathematical and logical operations on them
- Apply library functions, including those that summarize whole columns or rows
- Move arbitrary selected rows and columns from one array to another
- Reshape and linearize arrays

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`[ ]`	The empty vector	3.3.4
`[ . . . ]`	Concatenates data, vectors, and arrays	3.2.1
`:`	Specifies a vector as `from:incr:to`	3.2.1
`:`	Used in slicing vectors and arrays	3.3.5
`( )`	Used with an array name to identify specific elements	3.3.3
`'`	Transposes an array	3.5.1
`;`	Separates rows in an array definition	3.5.2
`+`	Scalar and array addition	3.3.5
`–`	Scalar and array subtraction or unary negation	3.3.5

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`.*`	Array multiplication	3.3.5
`./`	Array division	3.3.5
`.^`	Array exponentiation	3.3.5
`<`	Less than	3.3.5
`<=`	Less than or equal to	3.3.5
`>`	Greater than	3.3.5
`>=`	Greater than or equal to	3.3.5
`==`	Equal to	3.3.5
`≅`	Not equal to	3.3.5
`&`	Element-wise logical AND (vectors)	3.3.5
`&&`	Short-circuit logical AND (scalar)	3.3.5
`|`	Element-wise logical OR (vectors)	3.3.5
`||`	Short-circuit logical OR (scalar)	3.3.5
`~`	Unary not	3.3.5
`end`	Last element in a vector	3.3.5
`false`	Logical false	3.2.2
`true`	Logical true	3.2.2
`ceil(x)`	Rounds `x` to the nearest integer toward positive infinity	3.3.5
`cross(a, b)`	Vector cross product	3.3
`diag(a)`	Extracts the diagonal from an array or, if provided with a vector, constructs an array with the given diagonal	3.5.2
`disp(value)`	Displays an array or text	3.3.5
`find()`	Computes a vector of the locations of the true values in a logical array	3.3.5, 3.5.5
`[rows cols] = find()`	Computes vectors of row and column locations of the true values in a logical array	3.5.5
`fix(x)`	Rounds `x` to the nearest integer toward zero	3.3.5
`floor(x)`	Rounds `x` to the nearest integer toward minus infinity	3.3.5
`length(a)`	Determines the largest dimension of an array	3.2.2, 3.5.1
`linspace(fr,to,n)`	Defines a linearly spaced vector	3.2.1

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`magic(n)`	Generates a magic square	3.5.2
`[v,in] = max(a)`	Finds the maximum value and its position in `a`	3.3.5
`mean(a)`	Computes the average of the elements in `a`	3.3.5
`[v,in] = min(a)`	Finds the minimum value and its position in `a`	3.3.5
`ones(r, c)`	Generates an array filled with the value `1`	3.2.1
`rand(r, c)`	Calculates an `r` × `c` array of evenly distributed random numbers in the range 0…1	3.2.1
`randn(r, c)`	Calculates an `r` × `c` array of normally distributed random numbers	3.2.1
`round(x)`	Rounds `x` to the nearest integer	3.3.5
`size(a)`	Determines the dimensions of an array	3.2.2, 3.5.1
`sum(a)`	Totals the values in `a`	3.3.5

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. A homogeneous collection must consist entirely of numbers.
2. The function `linspace(...)` can create only vectors, whereas the functions `zeros(...)`, `ones(...)`, and `rand(...)` produce either vectors or arrays of any dimension.
3. The `length(...)` function applied to a column vector gives you the number of rows.
4. You can access any element(s) of an array of any dimension using a single index vector.
5. Mathematical or logical operators are allowed only between two arrays of the same shape (rows and columns).
6. You can access data in a vector `A` with an index vector that is longer than `A`.
7. You can access data in a vector `A` with a logical vector that is longer than `A`.

8. When moving a block of data in the form of specified rows and columns from array A to array B, the shape of the block in A must match the shape of the block in B.

Fill in the Blanks

1. Vector elements have two attributes that make them unique: their _____ and their _____.
2. Vectors can be created using the colon operator, for example, `B = 1:3:20`, where the first number is the _____, the second number is the _____, and the third number is the _____.
3. When indexing a source vector with a logical vector, the result will contain the _____of the source vector corresponding in position to the _____ in the logical vector.
4. The normal precedence of operators can be overruled by the use of _____.
5. Arithmetic operations can be performed collectively on the individual components of two arrays as long as both arrays _____or one of them is _____.
6. To remove elements from arrays, you write _____in _____.
7. Removing rows or columns from an array is _____, and can lead to _____. Wherever possible, use _____ to _____.

Programming Projects

1. For these exercises, do not use the direct entry method to construct the vectors. Write a script that does the following:
   a. Construct a vector containing all of the even numbers between 6 and 33, inclusive of the end points. Store your answer in the variable `evens`. (*Note:* 33 is not an even number)
   b. Construct a vector, `threes`, containing every third number starting with 8 and ending at 38.
   c. Construct a vector, `reverse`, containing numbers starting at 20 and counting backward by 1 to 10.
   d. Construct a vector, `theta`, containing 100 evenly spaced values between 0 and 2π.
   e. Construct a vector, `myZeros`, containing 15 elements, all of which are zeros.
   f. Construct a vector, `random`, containing 15 randomly generated numbers between 1 and 12.

2. Write a script that performs the following exercises on vectors:
   a. You are given a vector `vec`, defined as: `vec = [45 8 2 6 98 55 45 -48 75]`. You decide that you don't want the numbers with even values. Write as script to remove all of the even numbers (i.e., 8, 2, 6, 98, and −48) from `vec`. You should alter the vector `vec` rather than storing your answer in a new variable. Since your commands must work for any vector of any length, you must not use direct entry.
   b. Create a variable called `vLength` that holds the length of the vector `vec` modified in part a. You should use a built-in function to calculate the value based on the vector itself.
   c. Create a variable called `vSum` that holds the sum of the elements in vector `vec`. Do not just enter the value. You should use a built-in function to calculate the value based on the vector itself.
   d. Calculate the average of the values in the vector `vec` two ways. First, use a built-in function to find the average of `vec`. Then, use the results from parts b and c to calculate the average of `vec`.
   e. Create a variable called `vProd` that holds the product of the elements in vector `vec`. You should use a built-in function to calculate the value based on the vector itself.

3. Write a script to solve the following problems using only vector operations:
   a. Assume that you have two vectors named `A1` and `B1` of equal length, and create a vector `C1` that combines `A1` and `B1` such that `C1 = [A1(1) B1(1) A1(2) B1(2) ... . A1(end) B1(end)]`. For example, if `A1 = [2, 4, 8]` and `B1 = [3, 9, 27]`, `C1` should contain `[2, 3, 4, 9, 8, 27]`
   b. Assume that you have two vectors named `A2` and `B2` of different lengths. Create a vector `C2` that combines `A2` and `B2` in a manner similar to part a. However, if you run out of elements in one of the vectors, `C2` also contains the elements remaining from the longer vector. For example, if `A2 = [1, 2, 3, 4, 5, 6]` and `B2 = [10, 20, 30]`, then `C2 = [1, 10, 2, 20, 3, 30, 4, 5, 6]`; if `A2 = [1, 2, 3]` and `B2 = [10, 20, 30, 40, 50]`, then `C2 = [1, 10, 2, 20, 3, 30, 40, 50]`

4. Write a script that, when given a vector of numbers, `nums`, creates a vector `newNums` containing every other element of the original vector, starting with the first element. For example, if `nums = [6 3 56 7 8 9 445 6 7 437 357 5 4 3]`, `newNums` should be `[6 56 8 445 7 357 4]`. *Note:* You must not simply hard-code the numbers into your answer; your script should work with any vector of numbers.

5. You are given a vector, `tests`, of test scores and wish to normalize these scores by computing a new vector, `normTests`, that will contain the test scores on linear scale from 0 to 100. A zero still corresponds to a zero, and the highest test score will correspond to 100. For example, if `tests = [90 45 76 21 85 97 91 84 79 67 76 72 89 95 55]`, `normTests` should be

```
[92.78 46.39 78.35 21.65 87.63 100 93.81 86.6 ...
81.44 69.07 78.35 74.23 91.75 97.94 56.7];
```

6. Write a script that takes a vector of numbers, `A`, and return a new vector `B`, containing the cubes of the positive numbers in `A`. If a particular entry is negative, replace its cube with 0. For example, if `A = [1 2 -1 5 6 7 -4 3 -2 0]`, `B` should be `[1 8 0 125 216 343 0 27 0 0]`

7. Great news! You have just been selected to appear on Jeopardy this fall. You decide that it might be to your advantage to generate an array representing the values of the questions on the board.
   a. Write a script to generate the matrix `jeopardy` that consists of six columns and five rows. The columns are all identical, but the values of the rows range from 200 to 1,000 in equal increments.
   b. Next, generate the matrix `doubleJeopardy`, which has the same dimensions as `jeopardy` but whose values range from 400 to 2,000.
   c. You've decided to go even one step further and practice for a round that doesn't even exist yet. Generate the matrix `squaredJeopardy` that contains each entry of the original `jeopardy` matrix squared.

8. Write a script named `arrayCollide` that will combine two arrays, sort them, and then return a new array of a specified size. Your script should process the following data:
   - `A`: a 2-D array of any size
   - `B`: another 2-D array that may be a different size from `A`
   - `N`: a number specifying the number of rows for the new array
   - `M`: a number specifying the number of columns for the new array.

   Your script should produce an array, `res`, of size `N` × `M` that contains the first `N` × `M` elements of `A` and `B` and is sorted columnwise. If `N` × `M` is larger than the total number of elements in `A` and `B`, you should fill empty spots with 0.

   Test this script by writing another script that repeatedly sets the values of `A`, `B`, `M`, and `N` and then invokes your `arrayCollide` script.

You can then create as many test cases as you wish. For example, if `A = [1 2 3; 5 4 6],B = [7 8; 9 10; 12 11],N = 3` and `M = 4,res` will be

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

Change `N` to 4, and `res` will be

```
[1 5  9 0
 2 6 10 0
 3 7 11 0
 4 8 12 0]
```