Array Indicies, Whole Array Operations, Dot Operators

Leigh J. Little SUNY Brockport September 10, 2021

Contents

1 Introduction

Note: For the remainder of the semester, we will use the word *array* as a generic term for a vector or matrix.

We have seen that it is easy to create small vectors and matrices consisting of random elements in MATLAB using the array construction operators (`[`) and (`]`). However, often times the elements of an array are determined by a formula.

The most common type of vector that is needed in scientific computing is a vector that contains elements that are equally spaced over some range of values. A good example of this would be plotting a function. If you think back to when you took algebra, to plot the function $y = f(x)$ you might create a table of values that resembles the one shown in Table 1. Notice that the x column in Table 1 consists of a set of numbers

x	$y = f(x)$
0	0.67
1	1.45
2	2.62
3	3.87
⋮	⋮

Table 1: Table of values for some $y = f(x)$.

that are all 1 unit apart.

It turns out that working with tables forms the basis for a large part of scientific computing. We will see why as we go through the semester, but an analogy we can make right away is that a matrix is much like an Excel sheet. Like an Excel sheet, a matrix has columns and rows and the entries down a column or across a row represent different values of the same physical quantity.

Fortunately, MATLAB offers several techniques for quickly creating arrays of equally spaced elements over a range of values.

1.1 Array Indicies

An *array index* in MATLAB is a sequence of 3 numbers separated by colons (`:`). For example, enter the following:

```
>> x = (1:1:5)
x =
   1     2     3     4     5
```

Here, `(1:1:5)` is an array index. You should interpret this assignment as "x = the row vector from 1 to 5 in steps of 1".

An array index has the form

```
(starting value : step size : ending value)
```

The `( )` around the 3 values are optional, so you would get the same result if you typed

```
>> x = 1:1:5
x =
   1     2     3     4     5
```

Here are some other examples of array indicies

```
>> y = (1:2:10)
y =
     1     3     5     7     9
>> z = (0:0.2:1)
z =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
>> w = (-4:3:7)
w =
    -4    -1     2     5
```

What if you need a vector that counts backwards?

```
>> x = (5:1:1)
x =
   Empty matrix: 1-by-0
```

In this case, `x` is an empty matrix. This is a special type of matrix that MATLAB generates in situations like this. The reason that `x` is assigned this value is that it is impossible to start at 5 and end at 1 by stepping by 1. If you want to count backwards, you need a negative step size

```
>> x = (5:-1:1)
x =
     5     4     3     2     1
```

Finally, because a step size of 1 is the most common, you can omit the middle number if the step size is 1:

```
>> x = (1:6)
x =
     1     2     3     4     5     6
```

From these examples, we can make some observations:

- An array index is a row vector. You can add a transpose at the end if you need it to be a column vector.

  ```
  >> x = (7:-2:1)'
  x =
      7
      5
      3
      1
  ```

- The step can be anything, but this may result in an empty vector depending on the sign of the step and the starting/ending values.
- You don't have to pick the step size so that you land exactly at the ending value. The array will stop at the last value that is less than or equal to the ending value.
- If you want to step by 1, you can leave out the middle number.

1.2 `linspace` command

An alternative method for creating vectors of equally spaced elements is to use the `linspace` command. This is typically the way an equally spaced vector would be created in cases where you don't know the step size, but you do know how many elements the vector should have (this situation is quite common) or you have a non-integer step size. For example,

```
>> x = linspace(0,1,11)'
x  =
         0
    0.1000
    0.2000
    0.3000
    0.4000
    0.5000
    0.6000
    0.7000
    0.8000
    0.9000
    1.0000
```

In this case, `x` is a vector from 0 to 1 with 11 elements. In general, the `linspace` command has the format

```
>> x = linspace(a,b,m)
```

where `a` is the starting value, `b` is the ending value and `m` is the number of elements in the vector. The output will be a row vector unless you transpose it.

You can compute the step size (`h`) that will be used in the `linspace` command by subtracting any two successive elements of `x`. For example

```
>> x = linspace(0,1,11)';
>> h = x(2)-x(1);
```

however this is not recommended because of round-off error effects.

A better way to compute `h` is to use the formula

$$h = \frac{b-a}{m-1}.$$

In the above example, `a` $= 0$, `b` $= 1$ and `m` $= 11$, so

$$h = \frac{1-0}{11-1} = \frac{1}{10} = 0.1.$$

1.3 Array Subscripts

One of the most powerful features of MATLAB is that a subscript is not limited to scalar values; a subscript can be an array. For example

```
>> x = (2:2:20)
x =
     2     4     6     8    10    12    14    16    18    20
```

We can access individual elements of `x` using a subscript

```
>> x(3)
ans =
     6
>> x(9)
ans =
     18
```

However, the subscript can also be a vector. Consider the example below

```
>> ii = [2 4 1 7]
ii =
     2   4   1    7
>> x(ii)
ans =
      4    8    2    14
```

In this case, the subscript `ii` is itself a vector. The answer returned is a vector consisting of `x(2)`, `x(4)`, `x(1)` and `x(7)`.

The subscript can also be an array index; for example

```
>> x(1:2:10)
ans =
     2     6    10    14    18
```

In this case, the array index `(1:2:10)` is equal to

```
>> (1:2:10)
ans =
     1     3     5     7     9
```

so `x(1:2:10)` extracts all of the elements of `x` whose subscript (or index) is odd. Similarly,

```
>> x(2:2:10)
ans =
     4     8    12    16    20
```

extracts all of the even subscript elements of `x`.

So far, we have used subscripts to extract elements of a vector, but we can use subscripts to assign values to elements of a vector. For example, suppose we wanted to set all of the even numbered elements of `x` to 0. Then we could do

```
>> x(2:2:10) = 0
ans =
      2     0     6     0    10     0    14     0    18     0
```

Array indicies and array subscripts can also be used with matrices.

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> A(2:3,2:3)
ans =
     5     6
     8     9
```

In this case, we extracted rows 2 through 3 and columns 2 through 3 of `A`.

```
>> A([1 3],[2 3])
ans =
     2     3
     8     9
>> A([3 1],[2 3])
ans =
     8     9
     2     3
```

In the first case, we extract rows 1 and 3 and columns 2 and 3. In the second calculation is similar, but we extract row 3 then row 1.

The notion of vector subscripts will be used throughout the semester.

2 Whole Array Operations

Enter the following (you can leave off the (`;`) at the end of the statements to see the answer):

```
>> x = linspace(0,1,11)';
>> y = 2*x+4;
```

What is the value of `y`? In this case, `x` is a vector from 0 to 1 in steps of 0.1. When we compute `y`, MATLAB will return a vector that has the same size and orientation as `x` and each element of `y` will be obtained by multiplying the corresponding element of `x` by 2 and adding 4 to it. In mathematical notation, we did the following:

$$y_i = 2x_i + 4 \quad \text{for} \quad i = 1, 2, \ldots, 11.$$

This is an example of a *whole array operation* and such calculations are frequently performed in MATLAB. A whole array operation is a calculation that is performed on an entire array. In most cases, the result of the calculation is also an array. Whole array operations are also known as *vectorized operations*.

One of the first things we can do with whole array operations is to quickly calculate tables of simple functions.

```
>> x = linspace(-pi,pi,21)';
>> y = sin(x);
>> [x y]
```

Here, we tabulated the sine function on the interval $[-\pi, \pi]$ using 21 equally spaced points.

2.1 The Dot (.) Operators

Suppose we wanted to create a table of $y = x^2$ for x on the interval $[-1, 1]$ with 11 equally spaced points.

```
>> x = linspace(-1,1,11)';
>> y = x^2
```

If you type this in you get an error. Why do you get an error?

Remember that MATLAB uses the rules of linear algebra when multiplying vectors and matrices. `x` is a column vector of length 11, so

$$x^2 = x \cdot x = (11 \times 1) \cdot (11 \times 1)$$

The inner dimensions don't match, so this quantity can't be computed.

This creates something of a problem. It would not be unreasonable to need to compute x^2, x^3 or $x^{-3.2}$ for some vector x in a scientific calculation, however, such calculations don't agree with what we have learned in linear algebra. Moreover, there is no linear algebra interpretation for the calculation that we are attempting to perform here. What we want is a vector (y) the same size and orientation as x whose elements y_i are equal to x_i^2.

Fortunately, there is a way of performing whole array operation calculations on vectors and matrices in situations where we don't want these operations interpreted in a linear algebra sense. These are called the *dot operators*. There are three (`.`) operators

- the multiplication operator (`.*`)
- the division operator (`./`)
- the exponentiation operator (`.^`)

These operators are used in situations where you need to use a multiply, divide or power operation, but you don't want the operation to be interpreted in a linear algebra sense. Instead, the dot operators will apply the requested operation to each element individually.

For example, consider

```
>> x = linspace(-1,1,11)';
>> y = x.^2;
```

Now MATLAB doesn't complain about the calculation and in fact, we get the result that we wanted. Here, the (`.^`) tells MATLAB that rather than attempt to multiply `x` by itself (which would be undefined in a linear algebra sense), you want to take each element of `x` and square it.

Using this idea we can easily compute something like $y = 4x^3 - 2x^2 + 5x - 10$.

```
>> x = linspace(-1,1,11)';
>> y = 4*x.^3 - 2*x.^2 + 5*x - 10;
```

Similarly, suppose we wanted to tabulate $y = \cos(x)\sin(x)$ on the interval $[-\pi, \pi]$. If we try to do

```
>> x = linspace(-pi,pi,101)';
>> y = cos(x)*sin(x);
```

an error will be generated because `cos(x)` and `sin(x)` are both column vectors of length 101. The `*` operator is being interpreted as a linear algebra multiply operation and the inner dimensions of these 2 column vectors don't agree. However, we can do

```
>> x = linspace(-pi,pi,101)';
>> y = cos(x).*sin(x);
```

In this case, `y` is a vector the same size and orientation as `x` and each element of `y` is computed by multiplying the sine and cosine of the corresponding element of `x`.

The division dot operator (`./`) is not used as often as the (`.*`) and (`.^`) operators. This operator will divide the individual elements of its operands. For example

```
>> x = [1 2 3 4 5]
x =
     1     2     3     4     5
>> y = [2 3 4 5 6]
y =
     2     3     4     5     6
>> z = [1 0 2 0 3]
z =
     1     0     2     0     3
>> x./y
ans =
    0.5000    0.6667    0.7500    0.8000    0.8333
>> x./z
ans =
    1.0000       Inf    1.5000       Inf    1.6667
```

Be careful when using the (`/`) or the (`./`) operators. Division of vectors and matrices is not defined in a linear algebra sense. For example if you do

```
>> x = [1 2 3 4 5]
x =
     1     2     3     4     5
>> y = [2 3 4 5 6]
y =
     2     3     4     5     6
>> x/y
ans =
    0.7778
```

the operation `x/y` returns a value of `0.7778`. In the past, this operation would have flagged an error (which it should because this operation makes no sense). However, the "new and improved" MATLAB will give you a value for this.

Finally, the spacing is important when using the dot operators

```
>> x. * y
 x. * y
    |
Error: Unexpected MATLAB operator.
>> x. *y
 x. *y
    |
Error: Unexpected MATLAB operator.
>> x.*y
ans =
     2     6    12    20    30
```

```
>> x .* y
ans =
     2     6    12    20    30
```

Don't put a space between the (`.`) and the operation (`*`, `^`, or `/`).