Vectorized Operations

Leigh J. Little SUNY Brockport October 10, 2022

Contents

1 Introduction **1**

2 Array Indicies **1**

3 Vector Subscripts **2**

4 Vectorized Calculations **3**
4.1 Other Vectorized Operations 4
4.2 The (.) Operators 5
4.2.1 Knowing Your Variables 5

1 Introduction

We have learned about vectors and how to work with subscripted variables. As we have seen, working with vectors often involves looping over individual elements and performing some operation on each element. In many cases, the same operation is performed on each element. It would be nice if there was a way we could tell MATALB 'perform this operation on each element' in a single programming statement rather than having to loop over individual elements. Fortunately, there is a way to do this. In this section we will learn about *vectorized* or *whole array* operations.

2 Array Indicies

We have seen that it is frequently necessary to generate vectors that have equally spaced elements. An example of this is tabulating a function. We generated a set of equally spaced x-coordinates on some interval of the x axis, then computed a corresponding y-coordinate for each x-coordinate.

Because this need arises so frequently, MATLAB has an easy way to do this. This is called an *array index*. You have actually be using array indicies whenever you used a `for` loop. Type the following into MATLAB (you don't need the parentheses but I like to put them in):

```
>> x = (1:10)
x =
     1     2     3     4     5     6     7     8     9    10
>> size(x)
ans =
     1    10
```

Notice that the output is a row vector of length 10 that contains the values 1 to 10 in steps of 1. When we wrote `for` loops

```
for i = 1:10
```

The `1:10` part of this statement was actually an array index. From before, the general syntax for an array index is

```
start_value : step : end_value
```

You can omit the `step` if the step size is 1.

Some other examples of array indicies are

```
>> x = (1:2:10)
x =
     1     3     5     7     9
>> x = (2:2:10)
x =
     2     4     6     8    10
>> x = (-4:3:12)
x =
    -4    -1     2     5     8    11
```

Note that an array index does not need to land exactly on the `end_value`. Also, the step size does not need to be an integer

```
>> x = (0:0.1:1)
x =
    0 0.1000 0.2000 0.3000 0.4000 0.5000 0.6000 0.7000 0.8000 0.9000 1.0000
```

As we saw with looping operations, if you want the array index to count backwards, you need to have a negative step size.

```
>> x = (10:-1:1)
x =
    10     9     8     7     6     5     4     3     2     1
```

What happens if you try to step from 10 to 1, but forget to include the negative step size?

```
>> x = (10:1)
x =
   1x0 empty double row vector
```

Here, MATLAB is telling you that `x` is an *empty vector*. Note that this is different than a zero vector (which would be a vector of zeros). In this case, MATLAB sees `x` as a vector with one row and zero columns. Sometimes it is necessary to define an empty vector. If you need to do this, you can do

```
>> x = []
x = []
>> size(x)
ans =
      0   0
```

3 Vector Subscripts

One of the reasons that MATLAB became a highly popular computing language in the sciences is that in MATLAB, a vector must be a positive integer but it doesn't need to be a scalar. A subscript can be a vector of positive integers. Consider the statements below:

```
>> x = [3 -6 1 0 -2 3 -5 2 2 4 -6]
x =
    3    -6     1     0    -2     3    -5     2     2     4    -6
>> ii = [3 5 8]
ii =
     3     5     8
>> x(ii)
ans =
     1    -2     2
```

Here, `x` is some vector of elements and `ii` is a vector of positive integers of length 3. What do we get when we type in `x(ii)`? We can see that this will generate the vector

```
>> x(ii)
ans =
     1    -2     2
```

This is a row vector because `x` was a row vector. If we examine the values in the `ii` vector, we can see that

```
x(ii) = [x(3) x(5) x(8)]
```

What we did was create a new vector whose elements are some subset of the original `x` vector.

Suppose we wanted to extract all elements of `x` that have an odd subscript. Then we could do

```
>> x(1:2:length(x))
ans =
      3     1    -2    -5     2    -6
```

This vector consists of the elements

```
[x(1) x(3) x(5) x(7) x(9) x(11)]
```

Similarly, if we needed only the even subscripted elements, we could do

```
>> x(2:2:length(x))
ans =
    -6     0     3     2     4
```

This vector consists of the elements

```
[x(2) x(4) x(6) x(8) x(10)]
```

The elements do not have to be extracted in any particular order, and you can also have duplicate subscripts

```
>> ii = [9 5 5 2 8 1 4]
ii =
     9     5     5     2     8     1     4
>> x(ii)
ans =
     2    -2    -2    -6     2     3     0
```

This vector is comprised of the elements

```
[x(9) x(5) x(5) x(2) x(8) x(1) x(4)]
```

4 Vectorized Calculations

If we return to the example where we were tabulating the sine function, we did this by first generating an equally spaced set of x-coordinates on the given interval in a loop, then evaluating the sine function for each x_i in x. Based on the discussion above, we should be able to do this without a loop by doing something like

```
clear
a = 0;
b = 2*pi;
n = input('Input n ');
h = (b-a)/n;
x = (a:h:b);
```

If you test this out, you can see that it works. x will be a row vector because we defined it in terms of an array index. It turns out that this is not the most convenient way to do this. A better way to do this is to use the `linspace` command.

```
clear
a = 0;
b = 2*pi;
n = input('Input n ');
x = linspace(a,b,n+1)';
```

Note that the last input to the `linspace` is the total number of points you want to have in x, and not the number of steps n. Also, because we want x to be a column vector, we have put an apostrophe at the end of the `linspace` command to transpose x.

We have an easy way to get out set of x-coordinates, but what about the y-coordinates? Is there an easy way to get these? It turns out that to get the y-coordinates, we can just do

```
clear
a = 0;
b = 2*pi;
n = input('Input n ');
x = linspace(a,b,n+1)';
y = sin(x);
```

Here, the calculation for the y-coordinates is called a *vectorized* or *whole array* calculation. In this case, we have computed the entire vector y in a single statement. The vector y will have the same orientation as the vector x. This calculation is feasible because we want to perform the same operation on each element of x.

4.1 Other Vectorized Operations

Suppose we wanted to tabulate $y = x^2$. We try this

```
clear
a = 0;
b = 2*pi;
n = input('Input n ');
x = linspace(a,b,n+1)';
y = x^2;    % or x*x
```

In this case, MATLAB complains. It gives the error

```
Error using  ^
Incorrect dimensions for raising a matrix to a power. Check that the matrix
is square and the power is a scalar. To perform elementwise matrix powers,
use '.^'.
```

The reason for this error is somewhat technical, but is based in the fact that MATLAB interprets many operations on vectors in a linear algebra sense (because that is what it was originally designed for). However, the calculation we want to do does not exist in linear algebra. It is impossible to assign a meaning to the multiplication of one column vector with another.

However, we don't want to interpret the operation we are doing in a linear algebra sense. We just want to take every element of x and square it. We need a way to tell MATLAB that we want certain operations to be performed on each element (also called *elementwise operations*) rather than interpret the calculation in a linear algebra sense. The solution to this are the `dot` operators. Instead of doing

```
y = x^2;     % or x*x
```

we can do

```
y = x.^2;    % or x*x
```

The (`.^`) tells MATLAB to perform the indicated operation on each element.

With the (`.^`) operator, we can now easily compute something like

$$y = x^4 - 3x^2 + 5x - 8$$

for some vector $.x$

```
y = x.^4 - 3*x.^2 + 5*x - 8;
```

This is equivalent to what we have been doing in loops

```
for i = 1:n+1
    x(i) = a + (i-1)*h
    y(i) = x(i)^4 - 3*x(i)^2 + 5*x(i) - 8;
end
```

4.2 The (.) Operators

There are three (.) operators in MATLAB.

```
.*   -> Elementwise multiplication
./   -> Elementwise division
.^   -> Elementwise exponentiation
```

You use these when you want to apply an operation to the individual elements of a vector (or matrix) rather than interpret these in the linear algebra sense. Make sure you don't have a space between the (.) and the operation you want to perform. Some other examples are shown in Table 1.

$y = \cos\left(x^2\right)$	`y = cos(x.^2)`
$y = \ln\left(\frac{x-y}{y}\right)$	`y = log((x-y)./y)`
$z = \cos(x)\sin(y)$	`z = cos(x).*sin(y)`
$z = e^{x^2-y^2}$	`z = exp(x.^2 - y.^2)`

Table 1: Using the dot operator in formulas. All variables are assumed to be column vectors.

For example, suppose you wanted to divide the elements of one vector by another. You can do this using

```
>> x = [1 2 3];
>> y = [4 5 6];
>> z = x./y
z =
      0.2500    0.4000    0.5000
```

Here, `x` and `y` are row vectors so `z` will be a row vector. Each element of `z` is the quotient of the corresponding elements of `x` and `y`.

4.2.1 Knowing Your Variables

The above example provides an opportunity to demonstrate the importance of knowing what your variables are (scalar, vector or matrix) and (if they are vectors) their orientation. For example, suppose you left off the dot operator in the example above:

```
>> x = [1 2 3];
>> y = [4 5 6];
>> z = x/y
z =
    0.4156
```

MATLAB will produce an answer to this calculation, but it is not clear what this value means (actually, I have no idea how MATLAB arrives at the value given).

Suppose instead that x and y in the example above are column vectors instead of row vectors. In this case, a completely different z is computed

```
>> x = [1 2 3]';
>> y = [4 5 6]';
>> z = x/y
z =
       0         0    0.1667
       0         0    0.3333
       0         0    0.5000
```

As in the previous example, it is not clear why this answer is given.