Vectors and Matrices

Leigh J. Little SUNY Brockport October 5, 2022

Contents

1 Introduction

To this point, we have been running programs that work with scalar data. Scalars are numbers. We have been doing calculations with them, but we have also written programs that read in a value, incorporate its contribution to an accumulation variable, then overwrite this value with a new value.

The homework problem where you needed to think about the best-fit line calculation hinted at the need to be able to save the values we input for later use. In the case of the best-fit line, we first need to compute the averages of the input data points, then use these averages to compute the slope, m. Because there was no way to save the input values, it is necessary to input these values twice. It would be great if there was an organized way to save large amounts of data for use in multiple calculations.

2 Vectors

A *vector variable* (also called an *array variable* or a *subscripted variable*) provides a way to store many values under a common variable name. You have used subscripted variables in your previous studies. For example, a common formula for the mean of a set of number is given by

$$x_{\text{mean}} = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

Here x is a set of n values. The set x might be $x = \{-1, 2, 5, -6, 7, 9\}$. For this set, $n = 6$. Each individual number (or scalar) that makes up x is called an *element* or *component* of x. These individual elements can be referenced using a subscript where the subscript indicates the location of the element in the vector. For the sample vector x above, we have

$$\begin{aligned} x(1) &= -1 \\ x(2) &= 2 \\ x(3) &= 5 \\ x(4) &= -6 \end{aligned}$$

$$\begin{aligned} x(5) &= 7 \\ x(6) &= 9. \end{aligned}$$

Note that the first element in the vector has index (or subscript) 1 and the last element has subscript 6. One of the oldest arguments in computing is 'should the first element of a vector have subscript 1 or subscript 0?' This argument has been going on for more than 50 years (despite everything that can be said about this issue was said 49 years ago).

If we need to reference a vector x of arbitrary length n, we can use the notation

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

Note that x refers to the entire vector whereas x_i refers to a specific element of the vector.

2.1 Row Vectors and Column Vectors

Linear algebra is the mathematical discipline that studies the properties of vectors and matrices. MATLAB was originally invented to allow for easy calculation with these quantities. As a result, it handles vectors and matrices differently than most other languages.

In most programming languages, a vector is just a list of numbers. MATLAB makes a distinction between *row* and *column* vectors. This means that vectors in MATLAB have an *orientation*. A row vector looks like

$$x = \begin{bmatrix} 1 & -3 & -1 & 0 & 5 & -4 & 9 \end{bmatrix}$$

while a column vector looks like

$$y = \begin{bmatrix} 1 \\ 0 \\ -2 \\ 3 \\ -4 \\ -1 \end{bmatrix}.$$

3 Matrices

A *matrix* is a set of scalars arranged into a rectangular grid. Examples of matrices are

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}; \qquad B = \begin{bmatrix} 3 & -2 \\ 0 & 5 \\ -1 & 9 \end{bmatrix}; \qquad C = \begin{bmatrix} 5 & 2 & -1 \\ 0 & 8 & -7 \end{bmatrix}.$$

Here, we say that the matrix A is 3×3 because it has 3 rows and three columns. Similarly, B is 3×2 and C is 2×3.

As with vectors, each individual element of a matrix can be referenced, but this is done using a double subscript. For example,

$$A_{1,3} = 3$$

because the element in row 1 and column 3 of A is 3. Similarly,

$$B_{2,2} = 5$$

and

$$C_{2,3} = -7.$$

Matrices can also be viewed as a series of column vectors placed side by side or a collection of row vectors stacked on top of one another. For example, the matrix B above can be viewed as the column vectors

$$\begin{bmatrix} 3 \\ 0 \\ -1 \end{bmatrix} \quad \begin{bmatrix} -2 \\ 5 \\ 9 \end{bmatrix}.$$

placed side-by-side or the row vectors

$$\begin{bmatrix} 3 & -2 \end{bmatrix}$$
$$\begin{bmatrix} 0 & 5 \end{bmatrix}$$
$$\begin{bmatrix} -1 & 9 \end{bmatrix}$$

stacked on top of each other.

4 Working with Vectors in MATLAB

It is easy to create vectors in MATLAB. To create the row vector

$$x = \begin{bmatrix} 1 & -3 & -1 & 0 & 5 & -4 & 9 \end{bmatrix}$$

in MATLAB, use the `[` and `]` *array construction* operators

```
>> x = [1 -3 -1 0 5 -4 9]
x =
     1    -3    -1     0     5    -4     9
```

You can put commas between the elements, but this is not necessary. We can always get information about the dimensions of a vector using the `size` command.

```
>> size(x)
ans =
     1     7
```

Here, MATLAB is letting you know that x is a vector with 1 row and 7 columns, which is equivalent to a row vector with 7 elements.

To create the column vector

$$y = \begin{bmatrix} 1 \\ 0 \\ -2 \\ 3 \\ -4 \\ -1 \end{bmatrix}$$

you can do

```
>> y = [1; 0; -2; 3; -4; 1]
y =
     1
     0
    -2
     3
    -4
     1
```

Note that to create column vector, put semicolons between the elements. Within a set of array constuctors, the semicolons mean 'go down to the next row.' If we ask for the size of y we get

```
>> size(y)
ans =
     6   1
```

Here, MATLAB is letting you know that y is a vector with 6 rows and 1 column, which is equivalent to a column vector with 6 elements.

When working with vectors, the `length` command is sometimes more convenient

```
>> length(x)
ans =
     7
>> length(y)
ans =
     6
```

Suppose we just had a scalar value $z = 1$. What happens when we apply the `size` and `length` commands to z?

```
>> z = 1;
>> size(z)
ans =
      1    1
>> length(z)
ans = 1
```

The `size` command is indicating that z has 1 row and 1 column (which is the same as a scalar). Similarly, the `length` command has a value of 1, also indicating that z is a scalar.

4.1 The Transpose Operation

The transpose operation is one of the most commonly needed operations when working with vectors. This operation transforms a row vector into a column vector and vice versa. For example, if you have defined the vectors x and y above, then

```
>> transpose(x)
ans =
      1
     -3
     -1
      0
      5
     -4
      9
>> transpose(y)
ans =
      1     0    -2     3    -4     1
```

Because the transpose operation is so common, there is a shortcut for it. This is the apostrophe (') operator.

```
>> x'
ans =
      1
     -3
     -1
      0
      5
     -4
      9
>> y'
ans =
      1     0    -2     3    -4     1
```

The transpose operation provides a slightly easier way to create the column vector y,

```
>> y = [1 0 -2 3 -4 1]'
y =
     1
     0
    -2
     3
    -4
     1
```

There is a slight technicality with the apostrophe operator. In addition to transposing a vector, it also takes the complex conjugate of the vector (this is called the *Hermitian* transpose). The reason for this is to make MATLAB operations behave like mathematical linear algebra operations. Because we don't work with complex numbers in this class, the apostrophe operator and the transpose operator behave the same way.

IMPORTANT: Many functions in MATLAB will output vectors as row vectors, however most of the time column vectors are needed (this is because all of the formulas that you would see in linear algebra assume that vectors are column vectors). It is important to be aware of the orientation of your vectors.

4.2 Adding Vectors and Vectorized Operations

Vectors can be added to each provided they have the same length and orientation. For example, suppose you have the two vectors

$$a = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}; \quad b = \begin{bmatrix} -4 \\ 5 \\ 2 \end{bmatrix}$$

and you wanted to compute the vector $c = a + b$. This can be done using a `for` loop.

```
a = [2 3 -1]';          % create a as a column vector
b = [-4 5 2]';          % create b as a column vector
for i = 1:length(a)     % use length(a) to get the loop limit
    c(i) = a(i) + b(i);
end
c
```

The output from this short script section would be

```
>> c =
      -2   8  1
```

Notice that we have created the vector c one element at a time by adding the corresponding elements of a and b. Mathematically, we have done

$$c_i = a_i + b_i, \quad i = 1, 2, 3.$$

Note also that c is a row vector even though a and b are column vectors. The reason for this is that we inserted the elements into c without telling MATLAB anything about the orientation of c. Whenever you do this, MATLAB will create a row vector by default.

Operations like this are very common, so there is a shortcut way to add two vectors. This is called a *whole-array* addition or a *vectorized* addition. To do this we can just do

```
>> a = [2 3 -1]';
>> b = [-4 5 2]';
>> c = a + b
```

Here, c will be a column vector the same length as a and b. We will make heavy use of vectorized operations. This feature is one of the reasons that MATLAB has become the popular programming language that it is.

One reason that it is critical to be aware of the orientation of your vectors is that vectorized calculations like the one above can produce unexpected results. For example, suppose you were to add a to the transpose of b.

```
>> a = [2 3 -1]';
>> b = [-4 5 2]';
>> c = a + b'
```

The output from this will be

```
>> c =
      -2     7     4
      -1     8     5
      -5     4     1
```

Here, the output is a 3×3 matrix. Can you see how this result was obtained?

5 Example 1 - Summing the Elements of a Vector

Eventually we will rely on vectorized operations and built-in functions as much as possible but there are cases where it is necessary to work with arrays at the elemental level. It is instructive to write some simple algorithms that loop over the elements, both to get more practice using loops and also to develop a feel for working with vector subscripts.

A common operation in computing is summing the elements of a vector. This is a modification of the accumulation loops we have been doing to this point. Type the code below into a script called `sum1.m`

```
clear
x = [5 -1 4 6 -1 -1 0 6 -9];

% Set the accumulation variable to 0
total = 0;

% Loop over the elements of x
for i = 1:length(x)
    total = total + x(i);
end
disp('Total')
total
```

This gives the output

```
>> sum1
Total
total =
      9
```

Note that instead of waiting for keyboard input of the values from the user, the loop simply gets the next value in the vector.

6 Example 2 - Finding the Maximum Value in a Vector

Another common operation that needs to be performed is searching through a vector for the maximum value. If we mimic the demonstration done in class, we arrive at the algorithm below (called `getmax.m`)

```
clear
x = [5 -1 4 6 -1 -1 0 6 -9];

% Put the first person in the chair.
xmax = x(1);

% Loop over the remaining people.  Note that this loop starts at 2 because
% we have already accounted for the first person
```

```
for i = 2:length(x)
%
% Compare the next person x(i) with the person in the chair and reset if necessary
    if(x(i) > xmax)
       xmax = x(i);
    end
end
disp('Max value')
xmax
```

This script will output

```
>> getmax
Max value
xmax =
       6
```

6.1 Where is the Maximum Located?

Sometimes the location of the maximum value is more important than the maximum value itself. It is easy to write a script that also keeps track of where the maximum is located.

```
clear
x = [5 -1 4 6 -1 -1 0 6 -9];

% Put the first person in the chair and record their position
xmax = x(1);
locmax = 1;

% Loop over the remaining people.  Note that this loop starts at 2 because
% we have already accounted for the first person
for i = 2:length(x)
%
% Compare the next person x(i) with the person in the chair and reset if necessary.
% Reset the location of the max to this position.
    if(x(i) > xmax)
       xmax = x(i);
       locmax = i;
    end
end
disp('Max value')
xmax
disp('Location of max')
locmax
```

This script will output

```
>> getmax2
Max value
xmax =
     6
Location of max
locmax =
     4
```

Notice that the maximum value of 6 appears twice (in positions 4 and 8) in the vector x. This algorithm will return the first location where the maximum is achieved. This is consistent with the way a search like this usually behaves, but you can change the behavior if you want (as you will see in a homework problem).

7 Tabulating a Function

Building tables of values is at the core of scientific computing. For example, when differential equations are solved, the solution is not in terms of a mathematical function. Rather, the solution is returned as a table of values. Similarly, a weather simulation will not produce a formula for tomorrow's weather. Instead, a three dimensional grid of points will be created. At each of the (x, y, z) coordinates in this grid, values of temperature, pressure, humidity, *etc.* will be computed.

7.1 Getting the x-coordinates

The generic statement of tabulating a known function is: Given some interval of the x-axis, $x \in [a, b]$, tabulate some known function $f(x)$ on this interval. The process for doing this computationally is similar to how you would do it by hand. Choose a set of points in the interval $[a, b]$, then evaluate $f(x)$ at each of these points. It turns out the choosing the points is the more difficult part of the process.

Consider the problem below. We want to step from x = 0 ft to x = 5 ft with a step length of 1 ft.

```
*         *         *         *         *         *
0.0       1.0       2.0       3.0       4.0       5.0
```

In the process of doing this, we will take a total of 5 steps of 1 ft each, but during the entire process, we will occupy a total of 6 different x-coordinates, $x = \{0, 1, 2, 3, 4, 5\}$. Now suppose that we want to step from x = 0 ft to x = 5 ft with a step length of 0.5 ft.

```
*    *    *    *    *    *    *    *    *    *    *
0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0
```

In this case, it takes 10 steps to complete the process and a total of 11 points x-coordinates are occupied.

What we need is a way to take the interval $[a, b]$ and subdivide it into a number of equal pieces. Fortunately, there are some easy formulas for this. Let n be the number of steps you want to take. From the above discussion, it can be seen that there will be a total of $n + 1$ points in the table. The step size h can be computed from

$$h = \frac{b - a}{n}.$$

For the first example, $a = 0, b = 5, n = 5$, so

$$h = \frac{5 - 0}{5} = 1.$$

For the second example, $a = 0, b = 5, n = 10$, so

$$h = \frac{5 - 0}{10} = 0.5.$$

We want to store the x-coordinates in a vector because these are likely to be used in many different calculations. A simple code to do this would be

```
clear
a = 0;
b = 5;
n = input('Input n')
h = (b-a)/n;
x(1) = a    % Set first element of x vector to a.
for i = 2:n+1
    x(i) = x(i-1) + h;   % To get the next point in the vector,
                         % add h to the previous point.
end
```

This works in principle, but there is are two huge problems with this procedure. In a large scale simulation, the vector x could very easily have 10,000,000 points in it. Type the code below into a script file (call it `genxc.m`) and run it for several values of n.

```
clear

% Set values of a and b
a = 0;
b = 5;

% Set value of n and compute h
n = input('Input n ');
h = (b-a)/n;

% Set first element in the vector x
x(1) = a;

% Main loop to get remaining elements.
for i = 2:n+1
    % Get the next element by adding h to the previous element
    x(i) = x(i-1) + h;
end

% The last element in x should be equal to b.  Compute the relative error.
relerr = (x(n+1)-b)/b
```

If you run this code for larger and larger values of n, you should observe that the relative error increases. This is because the last element in x contains the effect of n accumulated round-off errors. In addition, the round-off error in the first portion of x is smaller than the round-off error in the latter portion of x. Because the x-coordinates are a fundamental building block of a simulation, it is important that these be both accurate and have uniform error throughout the vector.

In order to obtain more accurate x-coordinates, we need to re-think how we are getting from one end of the interval $[a, b]$ to the other. In the first example, we got from a to b the same way we would if we walked. Instead, we need to think in terms of 'hopscotching.' Consider the alternative below

```
*           *           *           *           *           *
0.0
            1.0 = 0 + h
                        2.0 = 0 + 2h
                                    3.0 = 0 + 3h
                                                4.0 = 0 + 4h
                                                            5.0 = 0 + 5h
```

To get to the next point, we go back to the start of the interval $x = a$ and then jump from here to the next desired point. Instead of taking 5 steps of equal size, we take 5 leaps of varying size. The code below (`getxc2.m`) will implement this new process.

```
clear

% Set values of a and b
a = 0;
b = 5;

% Set value of n and compute h
n = input('Input n ');
h = (b-a)/n;
```

```
% Main loop to get remaining elements.
for i = 1:n+1
    % Get the next element by jumping a multiple of h from a.
    x(i) = a + (i-1)*h;
end

% The last element in x should be equal to b.  Compute the relative error.
relerr = (x(n+1)-b)/b
```

In this algorithm, each x-coordinate has at most two round-off errors so each coordinate has about the same size error.

7.2 Building the Table

Now that we have the x-coordinates, getting the corresponding function values is easy. Suppose we want to tabulate $f(x) = \sin(x)$ on the interval $[0, 2\pi]$. The code below (`tablef.m`) will do this

```
clear

% Set values of a and b
a = 0;
b = 2*pi;

% Set value of n and compute h
n = input('Input n ');
h = (b-a)/n;

% Main loop to get remaining elements.
for i = 1:n+1
    % Get the next element by jumping a multiple of h.
    x(i) = a + (i-1)*h;
    % Get the corresponding $y$-coordinate
    y(i) = sin(x(i));
end
% Because we just started stuffing elements into the vectors x and y, these will be
% row vectors.  Transpose them to make them column vectors instead.
x = x';
y = y';
```