Chapter 3

Arrays

An *array* is a collection of similar elements. You might be familiar with vectors or matrices, which are arrays of numbers. We will look at rows and columns of numbers as 1-D vectors and 2-D matrices. However, arrays can extend into higher dimensions.

Arrays are the fundamental data structure in MATLAB. Even single numbers are arrays with one row and one column!

In this chapter, we will look at how to create arrays and reference their elements, how to plot data using arrays, and some of the common array operations and functions.

3.1 Creating Matrices and Vectors

We create a *matrix* by enclosing rows and columns within square brackets. This can be done in a natural readable format:

```
>> A = [ 1  2  3
         4  5  6
         7  8  9 ]
```

or in a more compact form using a semicolon to separate rows:

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

A *row vector* is a one-row matrix:

```
>> r = [ 1 2 3 4 5 ]
```

and a *column vector* is a one-column matrix:

```
>> c = [ 1
         2
         3
         4
         5 ]
```

or

```
>> c = [ 1; 2; 3; 4; 5 ]
```

3.1.1 : notation

We can quickly create a row vector using the `:` operator in the format:

```
start : stop
```

which creates a vector with starting value `start` and ending value `stop`. For example:

```
>> r = 1:6
 r =   1    2    3    4    5    6
```

The default is to increment by 1. We can increment by a different step size using the format:

```
start : step : stop
```

For example:

```
>> r = 1 : 0.5 : 4
 r =   1.0000   1.5000   2.0000   2.5000   3.0000   3.5000   4.0000
```

We can generate a decreasing row vector with a negative increment:

```
>> r = 10 : -1 : 5
 r =   10     9     8     7     6     5
```

3.1.2 eye, rand, zeros, and ones

An $n \times n$ identity matrix can be created with `eye(n)`, e.g.,:

```
>> eye(4)
 ans =
    1   0   0   0
    0   1   0   0
    0   0   1   0
    0   0   0   1
```

We generate a random $m \times n$ matrix with `rand(m,n)`, e.g.,:

```
>> rand(3,5)
 ans =
        0.650308   0.697979   0.618013   0.019526   0.875914
        0.345039   0.857957   0.322563   0.184117   0.524573
        0.502943   0.049443   0.388005   0.721979   0.257603
```

where each element from `rand` is sampled from the uniform distribution between 0 and 1. To get a single random number we can use `rand()` with no input:

```
>> rand()
 ans = 0.8147
```

We generate an $m \times n$ array of zeros with `zeros(m,n)`, e.g.,:

```
>> zeros(2,5)
 ans =
    0     0     0     0     0
    0     0     0     0     0
```

We generate an $m \times n$ array of ones with `ones(m,n)`, e.g.,:

```
>> ones(4,12)
 ans =
    1      1      1      1      1      1      1      1      1      1      1      1
    1      1      1      1      1      1      1      1      1      1      1      1
    1      1      1      1      1      1      1      1      1      1      1      1
    1      1      1      1      1      1      1      1      1      1      1      1
```

3.1.3 size

The `size` function returns the size of an array in rows $\times$ columns.

```
>> A = rand(5,3);

>> size(A)
 ans =    5    3

>> arr = [1 2 3 4 5 6 7 8 9 10 11 12 13 14];

>> size(arr)
 ans =    1    14
```

Later, when we have large dynamic arrays that are updated as a program runs, or want to write loops over arrays, it will be very useful to be able to get the current size of an array.

3.1.4 Transpose '

The *transpose* of a matrix A, often written as A^T but written in MATLAB as `A'`, flips columns into rows. Given the following matrix `A`:

```
>> A = [ 1  2
         3  4
         5  6 ]
```

then the transpose of `A` is `A'`:

```
>> A'
 ans =
     1   3   5
     2   4   6
```

Note the first column of `A` becomes the first row in `A'`, and the second column of `A` becomes the second row in `A'`.

The transpose operation is used when performing linear algebra operations, but is also convenient for quickly changing row vectors to column vectors and vice versa.

3.1.5 Composing Arrays

Composing arrays involves combining arrays together as components of a bigger array. Given two or more arrays, we can compose them as long as the dimensions (lengths of rows and columns) are consistent. Given the following:

```
>> A = [ 1 2
         3 4
         5 6
         7 8  ]

>> B = [  9
         10
         11
         12 ]
```

where `A` is a 4×2 and `B` is a 4×1, we can join `A` and `B` into one larger array with:

```
>> C = [ A B ]
 C =
    1     2     9
    3     4    10
    5     6    11
    7     8    12
```

to get `C` which is a 4×3.

Here's an example of composing a matrix using `rand`, `zeros`, and `ones`:

```
>> M = [  rand(3,3)     zeros(3,1)
          zeros(1,3)    ones(1,1)  ]

 M =
     0.6116   0.9581   0.5784   0
     0.5061   0.6340   0.6745   0
     0.5061   0.6340   0.6745   0
     0        0        0        1
```

3.2 Plotting with Arrays

In Section 2.6 we saw how to plot one point at a time. That approach will be useful when we see how to animate plots, but when given a data set it is nice to plot all of the data at once.

Given a vector of x-values and a vector of y-values, the `plot` function plots the pairs of corresponding (x, y) coordinates.

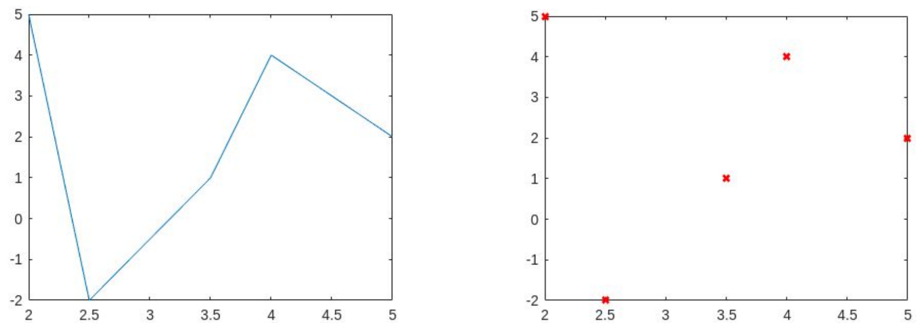
Consider again the data in this table:

x	2	2.5	3.5	4	5
y	5	-2	1	4	2

We can plot this data efficiently with:

```
>> x = [ 2, 2.5, 3.5, 4, 5 ];
>> y = [ 5,  -2,   1, 4, 2 ];
>> plot(x,y)
```

A plot will default to a line graph, but we can change this to a scatter plot by specifying the marker style as in Figure 3.1.



(a) `plot(x,y)` (b) `plot(x,y,'rx','Linewidth',2)`

Figure 3.1

3.3 Indexing and Slicing

The *index* of an element in an array refers to its position within the array. The first element is at index 1, the second element is at index 2, etc.

3.3.1 Indexing 1-D and 2-D Arrays

The vector below has six elements with values from 10 to 60. The first index is 1 and the last index is 6.

$$\texttt{vec =}\quad \begin{array}{|c|c|c|c|c|c|} \hline 10 & 20 & 30 & 40 & 50 & 60 \\ \hline \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} & \multicolumn{1}{c}{5} & \multicolumn{1}{c}{6} \end{array}$$

The value 40 in `vec` is at index 4. To reference this element in MATLAB, we use the name of the array with parentheses around the index:

```
>> vec = [ 10, 20, 30, 40, 50, 60 ];
>> vec(4)
 ans = 40
```

We can reference array values to print them, operate on them, or update them. For example,

```
>> vec = [ 10, 20, 30, 40, 50, 60 ];
>> 5 * vec(3)
 ans = 150
```

Indexing of a vector works the same regardless of whether it is a row vector or column vector.

In a 2-D matrix, we can reference an element by its row position and column position. Consider the matrix `arr` below:

$$\texttt{arr =}\quad \begin{array}{c|c|c|c|c|} \multicolumn{1}{c}{} & \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} \\ \cline{2-5} 1 & 10 & 20 & 30 & 40 \\ \cline{2-5} 2 & 50 & 60 & 70 & 80 \\ \cline{2-5} 3 & 90 & 100 & 110 & 120 \\ \cline{2-5} \end{array}$$

The element in `arr` with value 70 is at row 2, column 3. Consider that we want to update the value of 70 in `arr` to be 700. We can do this with:

```
>> arr = [ 10  20  30  40
           50  60  70  80
           90 100 110 120 ];

>> arr(2,3) = 700
arr =  10   20   30  40
       50   60  700  80
       90  100  110 120
```

Remember that we always state the row first when referencing rows and columns.

It is worth acknowledging two differences between MATLAB and several other programming languages.

1. MATLAB uses 1-based indexing where many other languages use 0-based.
2. MATLAB uses parentheses to reference elements of an array where many other languages use square brackets.

3.3.2 Multiple Indices

We can select multiple elements at once from an array by using a vector of indices:

```
>> nums = [5, -4, 8, 17, 6, 52]

>> nums([2, 3, 6])
ans =  -4   8   52
```

Note that the indices can arrange in any order (and even repeat), e.g.,

```
>> nums = [5, -4, 8, 17, 6, 52]

>> nums([6, 2, 6])
ans =  52   -4   52
```

3.3.3 Slicing

A *slice* is a subset of elements from an array. *Slicing* uses indices or index ranges specified with the : operator. For example, to select of the indices from 3 to 7 we can use:

```
>> nums = [72, 68, 3, 4, 55, 92, 61, 14, 399, 57, 86, 16, 12]
>> nums( 3:7 )
 ans =  3  4  55  92  61
```

If we want to reference indices all the way to the end without specifying the exact length, we can use the keyword `end`:

```
>> nums = [72, 68, 3, 4, 55, 92, 61, 14, 399, 57, 86, 16, 12]
>> nums( 3:end )
 ans =  3  4  55  92  61  14  399  57  86  16  12
```

Using the : operator with no indices simply returns all of the elements:

```
>> nums = [72, 68, 3, 4, 55, 92, 61, 14, 399, 57, 86, 16, 12]
>> nums( : )
        ans =  72  68  3  4  55  92  61  14  399  57  86  16  12
```

With a 2-D array, we can specify slices along both rows and columns, separated by a comma. This lets us grab a single row, single column, or any subset of rows and columns. For example,

```
>> M = [  1   2   3   4   5
          6   7   8   9  10
         11  12  13  14  15  ]
>> M(2,:)
 ans = 6  7  8  9  10

>> M(:,3)
 ans =   3
         8
        13

>> M(3, 1:3)
 ans =  11   12   13

>> M(2,4)
 ans = 9
```

In this example,

- `M(2,:)` grabs the 2nd row, every column
- `M(:,3)` grabs every row, 3rd column
- `M(3,1:3)` grabs the 3rd row, columns 1 through 3
- `M(2,4)` grabs the single element at row 2, column 4

3.4 Array Functions

We have already seen the `size` function that takes an array as input and returns the number of rows and columns of that array. Below is a small collection of functions that work on arrays:

`sum`, `prod`, `max`, `min`, `mean`, `median`, `mode`, `std`, `var`

You can read about each of these functions using the `help` command.

When working with a 1-D array, a function like `sum` finds the sum of the entire vector.

```
>> v = [ 97, 82, 76, 85, 78, 96, 100, 94 ]
>> sum(v)
 ans = 708
```

With a 2-D array, we might ask how a function like sum should work. Should it just sum every element from both dimensions? By default, arrays are treated as columns of data, where every column represents a different feature and every row is a sample or measurement.

Consider the tiny dataset of measurements of iris flowers in Figure 3.2. There are four features: sepal length, sepal width, petal length, and petal width. The five rows represent samples of five flowers that were measured.

`iris_data =`

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
Flower 1	5.1	3.5	1.4	0.2
Flower 2	4.9	3	1.4	0.2
Flower 3	5.4	3.9	1.7	0.4
Flower 4	4.6	3.1	1.5	0.2
Flower 5	4.6	3.4	1.4	0.3

Figure 3.2: A sample of five flowers from the iris dataset.

To see the behavior of some of our functions, let's put this data into MATLAB and run those functions to see some descriptive statistics about our dataset.

```
>> iris_data = [ 5.1   3.5   1.4   0.2
                 4.9   3     1.4   0.2
                 5.4   3.9   1.7   0.4
                 4.6   3.1   1.5   0.2
                 4.6   3.4   1.4   0.3  ];
>> min(iris_data)
 ans =  4.6000    3.0000    1.4000    0.2000

>> max(iris_data)
 ans =  5.4000    3.9000    1.7000    0.4000

>> median(iris_data)
 ans =  4.9000    3.4000    1.4000    0.2000

>> mean(iris_data)
 ans =  4.9200    3.3800    1.4800    0.2600

>> std(iris_data)
 ans =  0.3421    0.3564    0.1304    0.0894

>> var(iris_data)
 ans =  0.1170    0.1270    0.0170    0.0080
```

These functions operate on each column separately, similar to how one would calculate these statistics in a spreadsheet. The `std` function, for example, finds the standard deviation of each column's data.

3.5 Element-wise Operations

Some operations act independently on each single element of an array. We call these *element-wise* operations. A subset of those operations are element-wise by default, while others have element-wise versions.

When adding a constant (single number) to a matrix, the constant is added to each element of the matrix[1].

```
>> A = [ 0  1  2
         3  4  5 ]
>> A + 10
 ans =  10  11  12
        13  14  15
```

Matrix addition (adding two matrices together) is also an element-wise operation (5.1.1).

Other operations have a special syntax to disambiguate from their standard meaning. These include `.*`, `./`, and `.^`. These are element-wise multiply, divide, and power, respectively. Applying these, we can see their effects:

```
>> A = [ 0  1  2
         3  4  5 ]
>> B = [ 5  3  1
         6  4  2 ]
>> A .* B
 ans =   0     3     2
        18    16    10

>> A ./ B
 ans =     0         0.3333    2.0000
           0.5000    1.0000    2.5000

>> A .^ B
 ans =      0     1     2
          729   256    25
```

Each of these operations applies only between corresponding elements of `A` and `B`. The element-

[1]This is not a standard convention, but can be viewed as $A + b\mathbf{1}$ where $\mathbf{1}$ is the matrix of all ones.

wise power, for example, is calculating:

$$A\,.\hat{}\,B = \begin{bmatrix} 0^5 & 1^3 & 2^1 \\ 3^6 & 4^4 & 5^2 \end{bmatrix}$$

This is different from how these operations would behave if acting on entire matrices instead of pairs of elements. We will see more about matrix arithmetic in Section 5.1.

3.6 Sequences, Sums, and Series

3.6.1 Sequences

A *sequence* is an ordered list of numbers, which can be finitely or infinitely long. Here is a sequence with nine *terms*:

$$\mathbf{a} = 4, 7, 10, 13, 16, 19, 22, 25, 28$$

This particular sequence is an *arithmetic sequence*, one that has the same difference from each term to the next (in this case, a difference of 3). Using the : notation we saw in Section 3.1.1, we can easily generate arithmetic sequences. The code for this sequence is:

```
>> a = 4 : 3 : 28
 a =   4   7   10   13   16   19   22   25   28
```

Another useful type of sequence is a *geometric sequence*, one that has the same factor, or common ratio, from each term to the next. For example, the following sequence has a common ratio of 1.5:

$$\mathbf{b} = 4, 6, 9, 13.5, 20.25, 30.375, 45.5625, 68.3438$$

We can generate this sequence with:

```
>> b = 4 * 1.5 .^ (0:7)
 b = 4.0000   6.0000   9.0000  13.5000  20.2500  30.3750  45.5625  68.3438
```

Below are examples of several sequences. Try these, and also try to break them! What happens if you use a regular ^ instead of the element-wise .^? What other sequences can you make?

Vector of Ten 2s

```
>> 2*ones(1,10)
 ans =   2   2   2   2   2   2   2   2   2   2
```

Powers of 2

```
>> (2*ones(1,10)) .^ [1:10]
 ans =    2    4    8   16    32    64   128    256    512   1024
```

Notice the term that needed to be wrapped in parentheses to be evaluated before the exponent.

Perfect Squares

```
>> [1:10] .^ 2
 ans =    1     4     9     16     25     36     49     64     81    100
```

Reciprocals of Perfect Squares

```
>> 1 ./ ([1:8] .^ 2)
 ans = 1.0000    0.2500    0.1111    0.0625    0.0400    0.0278    0.0204    0.0156
```

Powers of 2 modulo 5

```
>> mod( (2*ones(1,10)) .^ [1:10] , 5)
ans =    2    4    3    1    2    4    3    1    2    4
```

We have a cycle!

3.6.2 Summation

A *summation* is the addition of all the terms in a sequence. For example, the summation of the arithmetic sequence from above is:

$$4 + 7 + 10 + 13 + 16 + 19 + 22 + 25 + 28 = 144$$

The explicit formula for that sequence is

$$a_n = 1 + 3n, \qquad \text{for } n \text{ from 1 to 9}$$

so that

$$\begin{aligned} a_1 &= 1 + 3(1) = 4 \\ a_2 &= 1 + 3(2) = 7 \\ a_3 &= 1 + 3(3) = 10 \\ &\vdots \end{aligned}$$

We could write the summation using Sigma notation as:

$$\sum_{n=1}^{9} 1 + 3n = (1 + 3(1)) + (1 + 3(2)) + (1 + 3(3)) + \cdots + (1 + 3(9)) = 144$$

To find the summation of this sequence in MATLAB, we can use the `sum` function:

```
>> a = 4 : 3 : 28
 a =   4     7    10    13    16    19    22    25    28

>> sum( a )
 ans = 144
```

3.6.3 Series

A *series* is a summation of an infinite sequence. While it is sometimes possible to find such a summation analytically, it is also practical to inspect or approximate partial sums of a series.

One famous series is the infinite summation of reciprocal powers of 2,

$$\sum_{k=1}^{\infty} \frac{1}{2^k} = \frac{1}{2^1} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \dots$$

Although this series sums an infinite number of terms, it converges (gets closer and closer to a single number). We can approximate this series with the partial sum of the first 5, 10, or 200 terms. Mathematically we would write these as:

$$\sum_{k=1}^{5} \frac{1}{2^k} = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5}$$

$$\sum_{k=1}^{10} \frac{1}{2^k} = \frac{1}{2^1} + \frac{1}{2^2} + \cdots + \frac{1}{2^{10}}$$

$$\sum_{k=1}^{200} \frac{1}{2^k} = \frac{1}{2^1} + \frac{1}{2^2} + \cdots + \frac{1}{2^{200}}$$

In MATLAB, we can write the first 5 terms of the reciprocal powers of 2 with `1./2.^[1:5]` where the expression `[1:5]` is the array of five numbers `[1 2 3 4 5]`. Here are the results for the three partial sums above:

```
>> sum( 1 ./ 2.^[1:5] )
 ans = 0.9688

>> sum( 1 ./ 2.^[1:10] )
 ans = 0.9990

>> sum( 1 ./ 2.^[1:200] )
 ans = 1
```

Notice that the partial sum of the first 200 terms is so close to `1` that it appears (incorrectly) as the exact (correct) answer. When we come across such an answer, is it really correct? It is! But it sometimes isn't. But this one is! Or is it?