MATLAB Basics

Leigh J. Little SUNY Brockport September 3, 2021

Contents

1 Introduction

MATLAB is a powerful software tool that allows you to do complex numerical simulations and view the results using a variety of graphical tools. MATLAB is easy to learn. It has its own programming language which allows you to create detailed functions and operates much like an interactive FORTRAN or C/C++ compiler.

MATLAB has its roots in linear algebra and as such, it interprets calculations in a linear algebra context. In order to make the best use of MATLAB, a cursory understanding of linear algebra concepts is required.

2 MATLAB Linear Algebra

The section details how MATLAB interprets the basic linear algebra type operations. In addition, the details of creating vectors and matrices of various types are explained.

2.1 General MATLAB Ideas

Before getting started, there are some important items to be aware of

1) MATLAB is case sensitive. This means that a variable named `B` and a variable named `b` are different quantities.

2) MATLAB uses double precision (*i.e.,* 15 or 16 decimal digits) in its calculations and storage of variables. There are special cases where this can be changed, but we will not use these in this course.

3) You can type `format compact` at the command line to remove extraneous blank lines.

4) You can bring up previously typed commands using the up-arrow key.

2.2 MATLAB Scalars

Scalars in MATLAB are the same as in mathematics, with one important difference; scalars are always interpreted as 1×1 matrices.

From the command line, type:

```
>> x = 6
```

The system will respond with

```
>>
x =
    6
```

If you type:

```
>> x = 6;
```

The system will respond with

```
>>
```

This is the most useful feature of MATLAB. Anytime you place a semicolon (`;`) after a command, the action is carried out, but the result of the computation is not echoed to the display. This is very useful when learning how to compute with MATLAB because you can immediately see the results of calculations. Also, it makes debugging your programs easier because you can remove the (`;`) from commands for which you want the results displayed during computation.

Now enter

```
>> size(x)
```

The system responds with

```
ans =
     1     1
```

The system is telling you that `x` is a matrix with 1 row and 1 column (*i.e.*, a scalar).

The `size` command is an example of a MATLAB function. One of the things that separates MATLAB from low level languages like C/C++/Fortran is that it has a large number of functions and commands that are available. We will use these functions extensively.

All of the usual mathematical operations are available For example:

```
>> x = 1
x =
   1
>> y = 2
y =
   2
>> x + y
ans =
     3
>> x - y
ans =
     -1
>> x*y
ans =
     2
>> x/y
ans =
     0.5000
>> y^5
ans =
```

```
     32
>> sqrt(y)
ans =
     1.4142
```

The trigonometric functions (sin, cos, *etc.*) as well as their inverses are also available.

There are some special predefined constants in MATLAB:

`pi` - this is a double precision approximation of π.

`i,j` - these are the imaginary units, $i, j = \sqrt{-1}$.

`Inf` - infinity (for example, the result of `1/0`).

`NaN` - stands for Not a Number (for example, the result of `0*Inf`)

MATLAB will automatically detect the presence of complex numbers and use complex arithmetic when needed.

```
>> x = -1
x =
   -1
>> y = 2 + 3i
y =
   2.0000 + 3.0000i
>> z = 4 - j
z =
   4.0000 - 1.0000j
>> sqrt(x)
x =
   0 + 1.0000i
>> y*z
ans =
     11.0000 + 10.0000i
```

2.3 Relational Operators

Relational (or logical) operators are used to make comparisons between variables. The result of a logical comparison is always either TRUE (= 1) or FALSE (= 0). The relational operators available in MATLAB are shown in Table 1. Here are some simple examples of logical comparisons:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	logical equivalence
&	logical AND
|	logical OR
~	logical NOT (negation)

Table 1: Relational Operators in MATLAB

```
>> x = -1
x =
   -1
>> y = 3
y =
   3
>> x < y
ans =
```

```
     1
>> x > y
ans =
     0
>> x == y
ans =
     0
>> (x < 0) & (y > 0)
ans =
     1
```

The first comparison returns 1 (TRUE) because `x < y` is true. Conversely, the next 2 comparisons return 0 (FALSE) because the comparisons `x > y` and `x == y` are false. Finally, the last comparison is called a compound relation. It is true because both `x < 0` AND `y > 0` are true.

2.4 MATLAB Vectors

MATLAB vectors can be either row or column vectors.

2.4.1 Creating Row Vectors

Row vectors can be created in many ways. The simplest is to specify each of the components. Enter the following command:

```
>> x = [1 3 4 -1 6]
x =
    1     3     4     -1     6
```

When you create vector (and matrices) in this manner, the start and end of the vector is indicated by the `[` and `]` characters. The command

```
>> x = [1, 3, 4, -1, 6]
```

would give the same result.

The vector `x` has 5 elements. These can be individually referenced if needed. For example, if you wanted to see (or use) the third element, you could do

```
>> x(3)
ans =
     4
```

You can use the `size` command to get the dimensions of `x`.

```
>> size(x)
ans =
     1     5
```

In this case, `x` is a vector of size 1×5.

2.4.2 Creating column vectors

Column vectors can also be generated in a similar way. To create a column vector vector element by element, enter:

```
>> y = [0; -1; 2; 9; 11]
y =

     0
    -1
     2
     9
```

```
    11
```

Notice that this is the same as creating a row vector, except that the elements are separated by semi-colons (;) and not spaces or commas. Notice also that the semi-colon (;) operator has different meanings based on where it appears. When the (;) appears within a set of square brackets, it means "go to the next row." When it appears at the end of a calculation, it means "don't display this result."

2.4.3 Transposing Vectors

The transpose operation is used frequently. There are two ways to transpose a vector. The first is using the `transpose` function.

```
>> x = [1 3 4 -1 6]
x =
   1     3     4    -1     6

>> y = transpose(x)
y =
    1
    3
    4
   -1
    6
```

A more common way to generate the transpose is to use to use the (') (apostrophe) operator:

```
>> x = [1 3 4 -1 6]
x =
   1     3     4    -1     6
>> y = x'
y =
    1
    3
    4
   -1
    6
```

However, it should be noted that the `transpose` function and the (') are not exactly the same. The (') operator corresponds to the Hermitian transpose and thus takes the complex conjugate of the elements in addition to performing a transpose. For example:

```
>> x = [i 2+i 3-2i]
x =
   0 + 1.0000i   2 + 1.0000i   3 - 2.0000i
>> y = x'
y =
   0 - 1.0000i
   2 - 1.0000i
   3 + 2.0000i
```

If the elements of the vector are real, then the `transpose` function and the (') operator are equivalent.

2.5 MATLAB Matrices

As in mathematics, matrices in MATLAB are generalizations of vectors and can be viewed as a collection of scalars, row vectors or column vectors.

2.5.1 Creating Matrices

As with vectors, matrices can be created by explicitly listing the elements. You enter the elements row by row and start a new row using the ; character.

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
   1  2  3
   4  5  6
   7  8  9
>> B = [ 3 -2; 0 5; -1 9]
B =
   3 -2
   0  5
  -1  9
>> C = [5 2 -1; 0 8 7]
C =
   5  2 -1
   0  8  7
```

In order to successfully create a matrix in this manner, the number of elements in each row must be the same. For example, the following command gives the error shown:

```
>> D = [1 2 3; 4 5]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

As with vectors, the individual elements of a matrix can be referenced using a subscript, though in this case, its a double subscript.

```
>> A(2,3)
ans =
     6
>> B(2,2)
ans =
     5
```

Sometimes it is necessary to extract not just a single element, but an entire row or column of a matrix. This can be accomplished using the (:) operator as one of the subscripts. For example

```
>> A(:,2)
ans =
     2
     5
     8
```

Here, the result is the second column of `A`. The (:) operator is interpreted to mean "all rows that go with column 2". Similarly, you can extract a row by doing

```
>> A(3,:)
ans =
     7     8     9
```

In this case, the (:) operator means "all columns that go with row 3"'.

2.5.2 Creating Matrices from Vectors

Matrices can be created from vectors by combining the steps we have already used. Suppose you wanted to recreate the matrix A from the row vectors:

```
>> x1 = [1 2 3];
>> x2 = [4 5 6];
>> x3 = [7 8 9];
```

From the vectors `x1`, `x2` and `x3`, it can be seen that to obtain `A`, these vectors need to be stacked in the order they are given. This can be accomplished as follows:

```
>> A = [x1; x2; x3]
```

Recall that with in the `[` and `]`, a semicolon (`;`) starts a new row.

For the column oriented version of the same problem, we can recreate `A` from the column vectors

```
>> y1 = [1; 4; 7];
>> y2 = [2; 5; 8];
>> y3 = [3; 6; 9];
```

by typing

```
>> A = [y1 y2 y3]
```

Since the vectors are separated by spaces within the `[` and `]`, these vectors are placed next to each other.

As a more elaborate example, suppose you wanted to recreate A from the components:

$$D = \begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix}; \quad w = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}; \quad y = \begin{pmatrix} 4 \\ 7 \end{pmatrix}.$$

Comparing the components with A, you can see that D forms the portion of lower right-hand portion of A, the transpose of w forms the first row of A and y makes up the remaining portion of column 1. To form A, all we need to keep in mind is that the matrix needs to be created row by row from top to bottom. The following sequence of commands will form A.

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

The last expression within the square brackets reads 'the first row of `A` is the transpose of `w` and the next rows are the vector `y` appended with the matrix `D`.'

Whenever you create matrices using this technique, you must be sure that each row has the same number of elements, otherwise, you will get an error. For example, if you forgot the transpose operator on the `w`, you would see

```
>> A = [w; y D];
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

2.6 MATLAB Vector-Vector Operations

Just like with scalars, vectors can be added, subtracted and multiplied, however there are some restrictions that govern when and how these operations can be performed.

2.6.1 Vector Addition

For this section, the following definitions are assumed:

```
>> x1 = [1 2 3]
x1 =
    1  2  3

>> x2 = [2 4 6]
x2 =
    2  4  6

>> y1 = [-1; 3; -5]
y1 =
    -1
```

```
     3
    -5

>> y2 = [2; -4; 6]
y2 =
     2
    -4
     6
```

We can add or subtract `x1` and `x2` in the usual way:

```
>> x3 = x1 + x2
x3 =
    3  6  9

>> x4 = x1 - x2
x4 =
    -1  -2  -3
```

The reason we can do this is because `x1` and `x2` have the same dimensions.

Really important note: In linear algebra, you can't add `x1` and `y1` because `x1` is a row vector and `y1` is a column vector. In previous versions of MATLAB, if you tried to do this, you would get an error message like the one below:

```
>> a1 = x1 + y1
??? Error using ==> +
Matrix dimensions must agree.
```

A few years ago, MATLAB changed this. You can add row vectors to column vectors and get the result below:

```
>> a1 = x1 + y1
a1 =
     0     1     2
     4     5     6
    -4    -3    -2
```

I **hate** that they did this because it greatly reduces the usability of MATLAB as a linear algebra teaching tool. It also leads to unexpected computational results and requires you to pay extra attention to whether vectors are row vectors or column vectors.

We can also add `y1` and `y2`:

```
>> y3 = y1 + y2
y3 =
     1
    -1
     1

>> y4 = y1 - y2
y4 =
     -3
      7
    -11
```

Also, we can add `x1` and `y1` if one of these is transposed prior to adding:

```
>> y5 = y1 + transpose(x1)
y5 =
```

```
     0
     5
    -2
```

Finally, we can use the (') operator instead of the `transpose` function:

```
>> y5 = y1 + x1'
y5 =
     0
     5
    -2
```

2.6.2 Vector Multiplication

Multiplication and division of vectors is much more complicated. The notion of vector (and matrix) multiplication usually introduced in a course in linear algebra states that 2 vectors (or matrices) can be multiplied only if their inner dimensions are the same. To see this, consider trying to form the product `y3 = y1 * y2`. If you try to do this, an error will be generated:

```
>> y3 = y1 * y2
??? Error using ==> *
Inner dimensions must agree.
```

To see why this will not work, write out the dimensions of the quantities being multiplied:

$$y1 * y2 => (3 \times 1) * (3 \times 1).$$

It can be seen that the 'inner dimensions' are 1 and 3. Since these are not equal, this product cannot be computed.

We can get a meaningful product by transposing either `y1` or `y2`. This leads to the notion of the dot and outer products.

2.6.3 The Dot Product

We previously saw that to compute the dot product of 2 vectors `y1` and `y2`, you multiply corresponding elements, then add up the resulting products. We also saw that if the vectors are column vectors, then the formulat for a dot product can be written as

$$\text{dot product} = y_1^T y_2$$

and that the result of a dot product is a scalar.

To get MATLAB to perform a dot product, `y1` needs to be transposed and then multiplied with `y2`:

```
>> y1' * y2
ans =
   -40
```

Let examine this in more detail. You can always tell the result of a vector product by looking at the dimensions. If the computation is examined, we see that

$$y1' * y2 => (1 \times 3) * (3 \times 1).$$

The inner dimensions 'cancel out' and the result is a 1×1 matrix (*i.e.*, a scalar).

2.6.4 The Outer Product

If we were to transpose `y2` instead of `y1`, a meaningful computation can be obtained, but the result would be a matrix and not a vector. To see this, look at the dimensions:

$$y1 * y2' => (3 \times 1) * (1 \times 3).$$

Here, the inner dimensions cancel out and the result is a 3×3 matrix. This is called an `outer product`. This type of vector product is rarely encountered, so we will only show the result for now:

```
>> y1 * y2'
ans =
     2     -4     6
     6    -12    18
   -10     20   -30
```

2.7 MATLAB Matrix-Vector Operations

The only possible operation with a matrix and a vector that can be performed is multiplication. As with vectors, in order for a matrix and a vector to be multiplied, the inner dimensions must match. Define the following quantities:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> x3 = [-1 2 -3];
>> y3 = [4; -5; 6];
```

Examine the product `A * y3`. This calculation can be performed because the inner dimensions match.

$$A * y3 => (3 \times 3) * (3 \times 1).$$

After canceling out the inner dimensions, the result will be a 3×1 matrix (*i.e.*, a column vector).

```
>> z = A * y3;
z =
   12
   27
   42
```

Note that the operation `y3 * A` cannot be performed because the dimensions do not match.

The matrix-vector product is a very common operation. We can compare the results of the calculation above with what we learned before and make use of the operations we have been using at the same time.

Recall that to compute the element `z(1)` of this matrix vector product, you would take the dot product of row 1 of `A` with the vector `y3`. However, we already have a mechanism in place to do this.

```
>> A(1,:)  * y3
ans =
    12
```

The dimensions of the above calculation match. `A(1,:)` is a row vector of length 3 and `y3` is a column vector of length 3, so the result is a dot product and is equal to `z(1)`.

Although less common, products of the form `x3 * A` also occur. Examining the dimensions, we see that

$$x3 * A => (1 \times 3) * (3 \times 3).$$

The inner dimensions match and the result is a 1×3 matrix (*i.e.*, a row vector).

```
>> v = x3 * A;
v =
   -14 -16 -18
```

Here, the second component of `v` (=`v(2)`) should be equal to the dot product of `x3` with the second column of `A`. To verify this, we can compute

```
>> x3 * A(:,2)
ans =
   -16
```

Again, the dimensions match. `x3` is a row vector of length 3 and `A(:,2)` is the second column of `A` and also has length 3. The result is a dot product.

Note that if you need a matrix vector product, you should just compute

```
>> z = A*x
```

2.8 MATLAB Matrix-Matrix Operations

As with vectors, 2 matrices can be added or subtracted only if they have the same number of rows or columns. Define the following quantities:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> B = [1 -2 3; -4 5 -6; 7 -8 9];
>> A + B
ans =
     2   0   6
     0  10   0
    14   0  18
```

To add 2 matrices, simply add corresponding matrix elements.

Multiplying matrices is similar to forming matrix-vector products. As both of these are 3×3 matrices, we can form the matrix products `A * B` and `B * A`.

```
>> A * B
C =
   14   -16    18
   26   -31    36
   38   -46    54

>> D = B * A
D =
   14    16    18
  -26   -31   -36
   38    46    54
```

An important fact to note is that matrix multiplication is *not* commutative (*i.e.*, `A * B` is not the same as `B * A`).

There are several ways to view a matrix-matrix product. Perhaps the most clear way is to generalize the idea of a matrix-vector product introduced in the previous section. Consider the product `A * B`. The matrix `B` can be thought of as a collection of column vectors.

```
>> B = [b1 b2 b3];
```

Where `b1`, `b2` and `b3` are the column vectors

$$b1 = \begin{pmatrix} 1 \\ -4 \\ 7 \end{pmatrix}; \quad b2 = \begin{pmatrix} -2 \\ 5 \\ -8 \end{pmatrix}; \quad b3 = \begin{pmatrix} 3 \\ -6 \\ 9 \end{pmatrix}.$$

Thus, the first column of the matrix-matrix product `A * B` is, in equation form, `A * b1`. Similarly, the second and third columns of `A * B` are `A * b2` and `A * b3` respectively. This same idea can be applied to the matrix product `B * A`.

In general, to generate the element in row `i` and column `j` of a matrix-matrix product, form the dot product of row `i` of the first matrix with column `j` of the second matrix. For example, if we define C = A*B, then

```
C(i,j) = A(i,:) * B(:,j)
```

2.9 Vector and Matrix Norms

Vector and matrix norms are also common calculations. Fortunately, MATLAB has built-in commands for these operations. Type in the following:

```
>> help norm
```

This will bring up the basic usage of the `norm` command. Notice that the `norm` command can take different forms based on what norm you want to compute. For example,

```
>> f = norm(x)
```

will compute the 2-norm of the vector `x`, but

```
>> f = norm(x,1)
```

will compute the 1-norm of the vector `x`. The same commands would be used to compute the corresponding matrix norms.

This illustrates two important aspects about MATLAB

1) If you want help on a command whose name you know, you can just type `help command` to get information on what it does and how to use it.
2) The `norm` command, along with many other MATLAB commands, uses a *variable number of inputs*. If you want the 2-norm, you only need to give the name of the vector you want the 2-norm of. If you need any other norm, you need to specifically state which norm you want.

Finally, be *very* careful when computing the norm of a matrix. The formula for the 2-norm of a matrix ($||A||_2$) was not given previously because this is beyond the scope of this class. The 2-norm of a matrix is a very complex, time consuming calculation if the matrix is large.

3 Math Functions

This section summarizes the most common math functions used in MATLAB. See `help command` for more information on each command.

- `sqrt(x)` - Computes the square root of x.
- `exp(x)` - Computes e^x. Note that e is not a built in variable. If you need the value of e, you need to compute `exp(1)`. Also, do not do something like

  ```
  >>  e = 2.718
  ```

  to define a value for e. This only has 4 digits of accuracy and will adversely affect your calculations.
- `log(x)` - Computes the natural logarithm of x.
- `log10(x)` - Computes the base-10 logarithm of x.
- `sin(x)`, `cos(x)`, `tan(x)` - Compute the sine, cosine and tangent functions of x. The input angle is assumed to be in radians.
- `asin(x)`, `acos(x)`, `atan(x)` - Compute the inverse sine, inverse cosine and inverse tangent functions of x. The angle is returned in radians. Note that because the `sin`, `cos` and `tan` functions are not one-to-one, the values returned by these inverse functions are range-restricted in the usual mathematical way:
  - `asin(x)` - returns an angle in the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.
  - `acos(x)` - returns an angle in the range $[0, \pi]$.
  - `atan(x)` - returns an angle in the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.
- `atan2(y,x)` - Computes the 4 quadrant inverse tangent of $\frac{y}{x}$. The angle is output in radians and will be in the range $[-\pi, \pi]$.
- `sinh(x)`, `cosh(x)`, `tanh(x)` - Compute the hyperbolic sine, cosine and tangent functions of x. Note that we call these the hyperbolic trigonometric functions because they obey a set of identities that are very similar to the trigonometric functions. However, the inputs to these function are *not* angles.
- `asinh(x)`, `acosh(x)`, `atanh(x)` - Compute the inverse hyperbolic sine, cosine and tangent functions. Because these are not trigonometric functions, the outputs from these functions are not angles.
- `nthroot(x,n)` - compute the nth root of a number. For example, if you need to compute the cube root of x, you can do

  ```
  >> y = nthroot(x,3)
  ```

Finally, throughout the course, all angles are assumed to be given in radians unless specifically indicated otherwise.