Introduction to Scientific Computing with Matlab

In the previous reading, we discussed the basics of vectors and matrices, including matrix addition and multiplication. In this class, we'll explore more array operations and properties that will be useful in the experiments to come.

More Array Operations. In addition to providing addition and multiplication, MATLAB supports certain array operations that can be very useful in scientific computing applications. Some of these operations are:

```
.*    ./    .^
```

The *dot* indicates that the operation is to act on the matrices in an element by element way. That is,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} .* \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} .^ 3 = \begin{bmatrix} 1 \\ 8 \\ 27 \\ 64 \end{bmatrix}$$

Array Indexing. We can refer to any particular element(s) in an array by indexing. To do so, we specify the desired row and column position in parentheses after the array name. For example, if

$$A = \begin{bmatrix} 2 & 4 & 65 & 3 \\ 5 & -1 & 13 & 17 \\ 7 & 9 & 22 & 6 \end{bmatrix},$$

then $A(2,3)$ is 13 and $A(3,2)$ is 9. If we want to refer to an entire block of entries, we can use the colon operator. For example, we indicate the entries contained in the first two rows and the first three columns of A by $A(1:2\ ,\ 1:3)$. That is,

$$A(1:2\ ,\ 1:3) = \begin{bmatrix} 2 & 4 & 65 \\ 5 & -1 & 13 \end{bmatrix}.$$

Initializing Vectors with Many Entries. Suppose we want to create a vector, `x`, containing the values $1, 2, \ldots, 100$. We could do this using a loop:

```
n = 100;
for i = 1:n
  x(i) = i;
end
```

In this example, when the code begins, how does MATLAB know `x` will be a vector of length 100? The answer to this question is: it does not know! But MATLAB has a very smart *memory manager* that creates space as needed. However, forcing the memory manager to work a lot can make codes very inefficient.

Fortunately, if we know how many entries `x` will have, then we can help out the memory manager by first allocating space using the `zeros` function. Here is how we might do it:

```
n = 100;
x = zeros(1,n);
for i = 1:n
  x(i) = i;
end
```

In general, the function `zeros(m,n)` creates an $m \times n$ array containing all zeros. Thus, in our case, `zeros(1,n)` creates a $1 \times n$ array, which is just a row vector with n entries.

Actually, there is a much easier, and better way, to initialize a simple vector like this using MATLAB's *vector operations*. This can be done as follows:

```
n = 100;
x = 1:n;
```

As you may have noticed, the colon operator is very useful! In the case just above, it provided a clean, fast way to create the vector **x**. In class, we'll see just how fast it is.

Let's see another example where we use the colon operator to boost efficiency. Suppose we want to create a vector, x, containing n entries equally spaced between $a = 0$ and $b = 1$. The distance between each of the equally spaced points is given by $h = \dfrac{b-a}{n-1} = \dfrac{1}{n-1}$, and the vector, x, should therefore contain the entries:

$$0, \quad 0+h, \quad 0+2*h, \quad \cdots, \quad (i-1)*h, \quad \cdots, \quad 1.$$

We can create a vector with these entries, using the colon operator, as follows:

```
n = 100;
h = 1 / (n-1);
x = 0:h:1;
```

We often want to create vectors like this in mathematical computations. Therefore, MATLAB provides a function for it, called `linspace`. In general, `linspace(a, b, n)` generates a vector of n equally spaced points between a and b.

The **moral** is: If we want to do something fairly standard, then chances are MATLAB provides an optimized function for it. To find out, we could use the `help` and/or `lookfor` commands.

Evaluating Functions and Plotting. Suppose we want to graph the function $\dfrac{\sin(3\pi x)}{x}$ on the interval $1 \leq x \leq 2$. What we learned about graphing a long time ago will still work in MATLAB today: we plot a bunch of points (x_i, y_i), and connect them with a smooth curve. We can do this in MATLAB by:

- creating a vector of x-coordinates
- creating a vector of y-coordinates, where $y(i) = \frac{\sin(3\pi x(i))}{x(i)}$.
- using the MATLAB command `plot(x,y)`

This can be done as follows:

```
n = 100;
x = linspace(1, 2, n);
y = zeros(1, n);
for i = 1:n
  y(i) = sin(3*pi*x(i)) / x(i);
end
plot(x, y)
```

Note that in this code we have

- used the `linspace` command to efficiently create the vector `x`,
- helped the memory manager by allocating space for `y` with the `zeros` command,
- used a loop to generate the entries of `y` one at at time,
- and used the `plot` command to draw the graph.

We can actually shorten this code by replacing the loop with a single, *vector operation*. In general, scalar multiplication and common functions like `sin` can be used on arrays of entries. That is, if `z` is an $m \times n$ array containing entries z_{ij}, then `sin(5z)` is an $m \times n$ array containing the entries $\sin(5 \cdot z_{ij})$. Thus, we can do the above computations simply as:

```
n = 100;
x = linspace(1, 2, n);
y = sin(3*pi*x)./x;
plot(x,y)
```

If you can use array operations instead of loops, then you should do it. In general, **array operations are more efficient than using loops**.

Finally, we mention that for *easy* functions, we can use MATLAB's `inline` and `ezplot` commands. For example, to plot $f(x) = \dfrac{\sin(3\pi x)}{x}$ on the interval $1 \leq x \leq 2$, use:

```
f = inline('sin(3*pi*x)./x');
ezplot(f, [1, 2])
```

Function mfiles and Script mfiles. So far we have only discussed MATLAB as a sophisticated calculator. It is much more powerful than that, and should be thought of more as a computer language. Therefore, there should be some capability for writing sophisticated programs, which include functions, subroutines, structures, classes, objects, etc. MATLAB has all of these capabilities, but since this is not a MATLAB programming class, we cannot go into details about them.

However, we should be aware of at least of two types of programs that we can write: *scripts* and *functions*. Each kind of program is written using your favorite editor, and should be saved in a file with the *.m* extension. The difference between a script and a function is:

- A *script* is a file containing a collection of MATLAB commands which are executed when the name of the file is entered at the MATLAB prompt. This is very convenient if you have to enter a lot of commands. But you must be careful: variables in a script are global to the MATLAB session, and it can be easy to unintentionally change values in certain variables.

- A *function* is a file containing a collection of MATLAB commands that are executed when the function is called. The first line of a function must have the form:

  `function [`*output1, output2,...*`  ] = `*FunctionName(input1, input2, ...)*

  Any data or variables created within the function are private, and so there is less of a chance of accidentally changing a variable. You can have any number of inputs to the function. If you want to pass back results from a function, then you can specify any number of outputs. Functions can call other functions, and in this way you can

write sophisticated programs as in any powerful language such as C, Fortran and Java. The beauty of MATLAB, though, is that you do NOT need to declare variables as `int, double, double[]`, etc.

For additional information on functions, see other, more complete references.

Naming Functions and Scripts. MATLAB is case sensitive both in terms of variables, and with respect to the names of functions and script files. As mentioned above, these should all have names of the form:

FunctionName`.m` or *ScriptName*`.m`

MATLAB has **a lot** of built in functions available for your use. For example, if you want to find the roots of a polynomial, there is a function available for this purpose – you don't need to write your own. To find out more about this function, use the `help` command:

```
>> help roots
```

All MATLAB functions are named with lower case letters. If you write a function with the same name, MATLAB has a way of choosing which function to use. To be sure that you (or MATLAB) don't get confused over which one to use, you might include some upper case letters in your function name. For example, if you wanted to name a function roots, you might use `Roots.m` instead of `roots.m`.

Getting Help. MATLAB has two useful commands for getting help:

- The `help` command can be used to get some basic help on how to use a MATLAB function. For example, if we want to know how to use MATLAB's `plot` command, we can use:

  ```
  >> help plot
  ```

  The difficulty with this command is that we have to know that there is a function with the name plot. For example, suppose we try:

  ```
  >> help polynomial
  ```

  MATLAB will respond with a message saying there are no functions called `polynomial.m`. But MATLAB is a sophisticated package, so there must be some functions that can be used to manipulate polynomials. How do we find these functions?

- One way is to use the `lookfor` command, which searches for functions that reference key words. So we can try:

  ```
  >> lookfor polynomial
  ```

and see what is found.

That should get you started. Between creating matrices, performing arithmetic and array operations, writing efficient codes using `linspace` and the colon operator, and learning about functions, scripts, `help`, and `lookfor`, you're well on your way to solving some nice problems in MATLAB. In class, you'll get to practice your MATLAB skills on some problems based on today's reading. Then next time, you'll use your MATLAB skills plus some statistics, geometry, and gameroom-athletics to calculate a very good estimate for the number π.