MATLAB and OCTAVE are both programming environments for doing numerical mathematics and working with vectors. MATLAB is used in many scientific and engineering environments, and OCTAVE is an open-source version of MATLAB. You may use either tools for this course. To save some typing, I will use OCTAVE to refer to MATLAB or OCTAVE from here on out.

1 The Basics

OCTAVE is designed to do *interactive* mathematics. That is, you can type mathematics and immediately get results. Simple math operations work how you expect:

```
octave:1> 3*5
ans =  15
octave:2> 2*pi
ans =  6.2832
octave:3> 5^9
ans =  1953125
octave:4> sin(pi/3)
ans =  0.86603
```

Keep in mind, everything you do in OCTAVE will be *numerical* as opposed to *exact*. This makes OCTAVE great for experimenting, but after you've found an answer, you might want to algebraically reason it through.

OCTAVE can also input and manipulate vectors. To do so, use square brackets `[ ]` and list the components of your vector.

```
octave:5> [1; 2; 3]
ans =

   1
   2
   3

octave:6> dot([1;2;3], [4;5;6])
ans =  32
```

If you use '`;`' between the components of your vector, it will be a column vector. If you use '`,`' between the components of your vector, it will be a row vector. In this class, we will primarily use column vectors.

You can store values/vectors in variables with =.

```
octave:7> x = [1; 2]
x =

   1
   2

octave:8> y = [-2; 1]
y =

  -2
   1

octave:9> dot(x, y)
```

```
ans = 0
```

By default, OCTAVE will print out the contents of a variable after it is assigned. To suppress the printing, add a '`;`' to the end of the line.

```
octave:10> x = [1; 2];
octave:11> y = [-2; 1];
octave:12> dot(x, y)
ans = 0
```

You can access and set components of vectors by doing *vector*`(`*component*`)`.

```
octave:13> x=[5; -2; 7];
octave:14> x(1)
ans =  5
octave:15> x(2)
ans = -2
octave:16> x(2) = 14;
octave:17> x
x =

    5
   14
    7
```

2 Functions

OCTAVE comes with many built-in functions, including `sin`, `cos`, `tan`, and their inverses `asin`, `acos`, `atan`. It also comes with many vector-based functions, like `dot`. However, you will often want to define your own functions. If your function can be expressed in one line, the easiest way to define it is with the `@()` syntax. For example, if we wanted a function $f : \mathbb{R}^3 \to \mathbb{R}$ that summed the components of a vector, we might write:

```
octave:18> f = @(x) (x(1) + x(2) + x(3))
f =

@(x) (x (1) + x (2) + x (3))

octave:19> a=[1; 2; 3];
octave:20> f(a)
ans =  6
```

When we defined f, the x was a *dummy variable*. That is, it has no meaning outside the definition of the function and is only used as a name. Therefore, it is equivalent to say:

```
octave:18> f = @(t) (t(1) + t(2) + t(3))
f =

@(t) (t (1) + t (2) + t (3))

octave:19> a=[1; 2; 3];
octave:20> f(a)
ans =  6
```

2.1 Math Operations

You've probably noticed already that the operations `+`, `-`, `*`, `^`, and `/` are usable in OCTAVE. If you use them with single numbers, these operations do what you expect. However, they are actually *vector* operations, not *scalar* operations. The corresponding scalar operations are `.+`, `.-`, `.*`, `.^`, and `./` (they all have a '`.`' in front). When we learn matrix/vector operations, the distinction will make sense, but for now, just use a '`.`' in front of these operations (you can omit the '`.`' in front of `+` and `-` because vector addition and scalar addition are treated nearly identically in OCTAVE).

Putting this into practice, if we wanted to define $f(x) = x^2$, we should write

```
octave:21> f = @(x) (x.^2)
f =

@(x) (x .^ 2)

octave:22> f(5)
ans =  25
```

and not `f = @(x) (x^2)`.

3 Graphing

There are two types of things you might want to graph: sets of vectors/points and functions. Let's start with points.

The `plot` command can be used for graphing, though the syntax might not quite be what you are used to. `plot` takes a list of x's, y's and a *style*. For example,

```
octave:25> plot([0 1 2 3 4], [0 1 4 9 16], 'o')
```

will plot the points $(0, 0)$, $(1, 1)$, $(2, 4)$, $(3, 9)$, and $(4, 16)$ putting a little circle at each point. Replacing `'o'` with `'.'` would put a dot at each point and using `'-'` would connect each pair of consecutive points with a line segment.

Since OCTAVE plots (x, y) pairs in the usual way, if you've defined a function, you can plot it by evaluating the function on a set of x values.

```
octave:29> f = @(x) (x.^(1/2));
octave:30> xs = [0 1 2 3 4 5];
octave:31> plot(xs, f(xs), '.')
```

It can be tedious to type out x values, so there is a shortcut syntax: *start*`:`*step*`:`*stop*. So, to plot $f(x) = \sqrt{x}$ with points space 0.1 apart on the interval $[0, 5]$, you could run the command:

```
octave:29> f = @(x) (x.^(1/2));
octave:30> xs = 0:.1:5;
octave:31> plot(xs, f(xs), '.')
```

3.1 Graphing Vectors

What if we have a bunch of vectors that we'd like to plot? Let's say we define a function $f : \mathbb{R} \to \mathbb{R}^2$ by

$$f(t) = \begin{bmatrix} \sin t \\ \cos t \end{bmatrix}.$$

In order to plot such a function, we would need to get a hold of the x and y coordinates of the output. Let's first examine the initial setup.

```
octave:36> f = @(t) ([sin(t); cos(t)]);
octave:37> ts = 0:.25:pi;
octave:38> f(ts)
ans =

   0.00000   0.24740   0.47943   0.68164   0.84147   0.94898   0.99749
        0.98399   0.90930   0.77807   0.59847   0.38166   0.14112
   1.00000   0.96891   0.87758   0.73169   0.54030   0.31532   0.07074
     -0.17825  -0.41615  -0.62817  -0.80114  -0.92430  -0.98999
```

Here, the output of `f(ts)` is actually a list of vectors. We'd like to get a hold of the of the x and y coordinates of vectors in the list. This can be done with the *vectors*`(1,:)` and *vectors*`(2,:)` syntax.

```
octave:37> f = @(t) ([sin(t); cos(t)]);
octave:38> ts = 0:.25:pi;
octave:39> vecs = f(ts);
octave:40> vecs(1,:)
ans =

   0.00000   0.24740   0.47943   0.68164   0.84147   0.94898   0.99749
        0.98399   0.90930   0.77807   0.59847   0.38166   0.14112

octave:41> vecs(2,:)
ans =

   1.000000   0.968912   0.877583   0.731689   0.540302   0.315322
     0.070737  -0.178246  -0.416147  -0.628174  -0.801144  -0.924302
     -0.989992
```

We can see that *vectors*`(1,:)` picked off the x coordinates and *vectors*`(2,:)` picked off the y coordinates. (What really happened was that we indexed a matrix using OCTAVE *slice notation*. That is, the `1` in `(1, :)` says take the first row and the `:` says take all numbers in that row. Can you guess how to grab only the first three numbers of the first row?)

Now we can plot without issue:

```
octave:37> f = @(t) ([sin(t); cos(t)]);
octave:38> ts = 0:.25:pi;
octave:39> vecs = f(ts);
octave:40> plot(vecs(1,:), vecs(2,:), '.')
```

3.2 Multiple Graphs

There may be times when you'd like multiple graphs to show up on the same plot. For that, you can use the `hold on` and `hold off` commands. The command `hold on` prevents OCTAVE from creating a new plot, and therefore everything you plot will get pushed to the existing plot. The command `hold off` stops this behaviour. For example, to plot both $f(t) = (\sin t, \cos t)$ and $g(t) = (t, t^2)$ we could do:

```
octave:37> f = @(t) ([sin(t); cos(t)]);
octave:38> g = @(t) ([t; t.^2]);
```

```
octave:39> ts = 0:.25:pi;
octave:40> vec_f = f(ts);
octave:41> vec_g = f(ts);
octave:42> plot(vec_f(1,:), vec_f(2,:), '-')
octave:43> hold on
octave:44> plot(vec_g(1,:), vec_g(2,:), '-')
octave:45> hold off
```

4 Flow Control and Loops

Flow control is programming-speak for piecewise functions. That is, functions that have "if" statements in them. Loops are ways to repeat a procedure without copying and pasting.

We'll start with loops. A loop in OCTAVE is written with the

```
for i=1:n,
    ⟨loop contents⟩
end
```

syntax.

For example, if we want the variable x to be an accumulated sum of squares from 1^2 to 15^2 (that is $x = 1^2 + 2^2 + \cdots + 15^2$), we could write

```
x=0;
for i=1:15,
    x = x + i.^2;
end
```

After executing this code x will be the value 1240. The code works as follows: first we assign the value 0 to x. Then we enter the loop. In the first iteration, when i is 1, the new value of x will be the current value, 0, plus 1^2. The next time through the loop, i is 2 and so we assign x to be the current value, 1^2, plus 2^2, etc.. Once we have looped through with the value of i being 15, we stop.

If statements are written with the

```
if ⟨variable⟩ == ⟨value⟩,
    ⟨if contents⟩
end
```

[style=Matlab-Pyglike,escapechar=‘] syntax (note the double equals).

For example, if we wanted x to be the sum of the squares of only the even numbers between 1 and 15, we might write:

```
x = 0;
for i = 1:15,
    if round(i/2) == i/2,
        x = x + i.^2;
    end
end
```

Here, `round(i/2)==i/2` is true when `i/2` has no decimal places and so it is even. Thus, when that happens (and only when that happens) we add the value i^2 to x.

4.1 Accumulating Vectors

Often times, you'll want to use a loop to create a list of vectors. For example, maybe we want to create a list of vectors on the unit circle, one for each degree. One way to do this is to do a loop and *append* each vector created to a list.

The syntax for appending looks a lot like nesting matrices. Observe the following:

```
octave:51> x=[1; 2];
octave:52> y=[2; 7];
octave:53> z=[-1; -1];
octave:54> [x y]
ans =

   1   2
   2   7

octave:55> [x y z]
ans =

   1   2  -1
   2   7  -1
```

We can use this syntax to accumulate a list of vectors.

```
octave:51> x=[1; 2];
octave:52> y=[2; 7];
octave:53> z=[-1; -1];
octave:59> r = [x]
r =

   1
   2

octave:60> r = [r y]
r =

   1   2
   2   7

octave:61> r = [r z]
r =

   1   2  -1
   2   7  -1
```

Notice that one item gets added to `r` each time. This can be combined with a `for` loop to generate the list of vectors we want.

```
r = [];
for i=0:360,
    v = [cos(2*pi*i/360); sin(2*pi*i/360)];
    r = [r v];
end
```

Can you think of how you might get a list of vectors on the unit circle whose x coordinate is less than 0.6643?