Introduction to MATLAB

Basics

MATLAB is a high-level interpreted language, and uses a "read-evaluate-print" loop: it *reads* your command, *evaluates* it, then *prints* the answer. This means it works a lot like a calculator:

```
>> 1+2
ans =
    3
```

Here, it read the command `1+2`, evaluated it to `3` and printed that. It also stores the answer from that in the variable `ans` so that you can refer to it in the next command if you want.

```
>> 1.5^2 + 2.5^2
ans =
    8.5000

>> sqrt(ans)
ans =
    2.9155
```

Be careful: after executing the second line, `ans` now has a new value. This also shows the syntax for a power (a^b is evaluated using `a^b`) and square root using the `sqrt` function. Much of the syntax follows mathematical syntax that you would expect, such as `cos(angle)` to get the cosine of an angle, for example.

You can create your own variables, and assign them values using =

```
>> a = sqrt(8.5)
a =
    2.9155

>> a^2
ans =
    8.5000
```

All of your current variables, and their values, are listed in the **workspace** on the right hand side. If you want to clear out all of the variables, use `clear`.

If you want to change how many digits are output, or use scientific notation automatically, use the `format` command. The default is `format short`, and `format shortEng` will use engineering notation (scientific notation, where the exponent on 10 is a factor of 3). You can also use `format long` and `format longEng`.

Useful functions

The trigonometric functions are all at your disposal: `cos`, `sin`, `tan`, and their inverses `acos`, `asin`, and `atan`. There is also `atan2(y,x)` which takes two values, `y` and `x` being the height and base of a triangle (rather than `atan(y/x)`). That version gives you the angle in the correct quadrant. All of the trignometric functions use radians; if you want to give an argument in *degrees*, then use `cosd`, `sind`, `tand`, and so on.

```
>> atan2(1,1)
ans =
    0.7854

>> atan2d(1,1)
ans =
    45
```

You can also convert between radians and degrees using `deg2rad` and `rad2deg`. Finally, you can access the value of π by using `pi`. *Be careful:* MATLAB allows you to use `pi` as a variable name, which … could give you surprising results if you, e.g., `pi = 3`.

Keyboard shortcuts

Help. If you're not sure what a command does, type `help` *commandname*. If you can't remember if `cos` uses radians or degrees, then `help cos` will tell you. You can also search the documentation in the upper right hand corner.

Tab completion. If you're typing a command like `cos`, when you hit the TAB key, it will give you a list of commands that start with the letters `cos`.

Command history. You can use the up and down arrows to move through previous commands that you've entered. You can then press ENTER to rerun that command exactly, or move the cursor left and right in the line and make edits (e.g., if you made a mistake you need to correct). This is useful if you've made an error with a variable value and need to reevaluate an expression.

Vectors and matrices

MATLAB (MATrix LABoratory) is optimized for working with *vectors* and *matrices*. As such, it has a nice syntax for making vectors and matrices easily, using the `[]` syntax

```
>> A = [1 2]
A =
     1     2

>> B = [3,4]
```

```
B =
     3     4

>> M = [5 6 ; 7 8]
M =
     5     6
     7     8
```

You can separate entries in a vector using a space or a comma (and can mix and match: `[1 2,3]`), and you separate the rows in a matrix using a semicolon.

You can then access the values inside a vector (v_i) or matrix (M_{ij}) with `()`

```
>> B(1)
ans =
     3

>> M(1,1)
ans =
     5
```

The indices follow row-column order, so that M_{ij} is `M(i,j)`, and the indices begin at 1. In addition to *accessing entries*, you can also assign values.

```
>> M(2,1) = 10
M =
     5     6
    10     8
```

If you want a row or column vector out of a matrix, you use the `:` operator; then `M(1,:)` gives you the row M_{1j}, while `M(:,1)` gives you the column M_{i1}.

```
>> M(1,:)
ans =
     5     6

>> M(:,1)
ans =
     5
    10
```

You can do things like get the dot product of $\vec{a}$ and $\vec{b}$ with `dot(a,b)`; you can get the crossproduct $\vec{a} \times \vec{b}$ with `cross(a,b)`. You can get the transpose of a vector or matrix with the `'` operator

```
>> B'
ans =
     3
     4

>> M'
ans =
     5    10
     6     8

>> M * B'
ans =
    39
    62
```

Note: the transpose of a *row vector* (like `[1 2]`) is a *column vector* (like `[1;2]`). To right-multiply a vector times a matrix (like $M \cdot \vec{v}$), the vector needs to be a column vector. You can also use this to take dot-products if you want: if `A` and `B` are row vectors, then `dot(A,B)` is the same as `A*B'`.

For a matrix, you can access the determinant with `det(M)` and the trace (sum along the diagonal) with `trace(M)`

```
>> det(M)
ans =
   -20

>> trace(M)
ans =
    13
```

The eigenvalues and eigenvectors of a matrix can be computed using `eig`

```
>> Msq = [1 0.5 0.25 ; 0.5 -1 0.75 ; 0.25 0.75 0]
Msq =
    1.0000    0.5000    0.2500
    0.5000   -1.0000    0.7500
    0.2500    0.7500         0

>> eig(Msq)
ans =
   -1.4461
    0.1693
    1.2768
```

```
>> [V,D] = eig(Msq)
V =
    0.1370   -0.4514   -0.8817
   -0.8888    0.3369   -0.3106
    0.4373    0.8262   -0.3551
D =
   -1.4461         0         0
         0    0.1693         0
         0         0    1.2768
```

The first form, `eig(M)` just gives a vector listing the eigenvalues. The second, `[V,D] = eig(M)`, returns the eigenvalues in a diagonal matrix `D` (and you can get those entries using `diag(D)`), and the eigenvectors are the columns of `V`. Thus, the first eigenvector is `V(:,1)` and has eigenvalues `D(1,1)`; the second is `V(:,2)` with eigenvalue `D(2,2)`, and the third is `V(:,3)` with eigenvalue `D(3,3)`.

```
>> Msq*V(:,1)
ans =
   -0.1981
    1.2853
   -0.6324

>> D(1,1)*V(:,1)
ans =
   -0.1981
    1.2853
   -0.6324
```

Solving (linear) equations

We can use MATLAB to solve equations, including systems of equations. For our purposes, we will almost exclusively deal with linear equations. The first step is defining a set of *symbolic variables* using `syms`

```
>> syms Fx Fy Fz
```

With these, we can construct vectors of symbolic variables, including more complicated expressions

```
>> F = [Fx Fy Fz]
F =
[Fx, Fy, Fz]
```

but you could also have `F1 = [Fx, -2*Fy, 3+Fz]` as a valid expression, and you can mix and match with numeric vectors; so you can make the combination `F + [0 0 -900]` if you wanted to add the force `F` to a force $-900\hat{k}$.

From these expressions, we can either use `solve` or `linsolve`. The syntax is very similar, though each are slightly idiosyncratic. First, `solve` for a series of equations using == to indicate *equality* instead of *assignment*

```
>> struct2array(solve([ F == [0 0 -900] ], [Fx Fy Fz]))
ans =
[ 0, 0, -900]
```

The `solve([` *equations* `], [` *variables* `])` returns an object that contains symbolic solutions, while `struct2array` converts it into an array. *Note:* the symbolic expressions may be returned as *fractions*, and so you will need to use `double()` to convert the fraction into a floating point number. You can use `double(ans)` to convert the entire vector answer if you would like.

You can use more complicated expressions in your solve; for example, we use `cross` to get cross-products for moments. You can set an entire vector to zero by writing `F == 0`.

Alternatively, you can convert your linear problem to matrix form $Av = b$ and use `linsolve`

```
>> [A, B] = equationsToMatrix([ F == [0 0 -900] ], [Fx Fy Fz])
A =
[ 1, 0, 0]
[ 0, 1, 0]
[ 0, 0, 1]

B =
    0
    0
 -900

>> linsolve(A,B)
ans =
    0
    0
 -900
```

Numerical integration

MATLAB has the ability to integrate in one, two, or even three dimensions. However, you need to understand *anonymous functions* (a function that you cook up without giving it a name), and how to deal with some vector operations.

Anonymous functions. There are two places where we may deal with anonymous functions: the integrand, and the limits of an integral. For example,

$$\int_0^{\pi} \cos^2 x \sin^2 x \; dx$$

we'll need to be able to define the function $\cos^2 x \sin^2 x$. Another example, if we integrate

$$\int_{-R}^{R} dx \int_{-\sqrt{R^2-x^2}}^{\sqrt{R^2-x^2}} y^2 \; dy$$

then we will also want to define the functions $-\sqrt{R^2 - x^2}$ and $\sqrt{R^2 - x^2}$ which define the lower and upper bounds of integration for y. We do this by using the `@` operator. For example, the first case would be `@(x) (cos(x)).^2 .* (sin(x)).^2`. The first part, `@(x)`, tells us its a function of one variable x, and the rest, `(cos(x)).^2 .* (sin(x)).^2`, is the expression to evaluate given x. So if we want to evaluate that integral, we'd do

```
>> integral(@(x) (cos(x)).^2 .* (sin(x)).^2, 0, pi)
ans =
    0.3927
```

This also shows how `integral` works: it takes a function, and integrates it over a range. You can use `Inf` (or `-Inf`) to get ∞ (or $-\infty$) as an endpoint on the range, too. For our second example it's a bit longer:

```
>> R = 1.
R =
     1

>> integral2(@(x,y) y.^2, -R, R, @(x) -sqrt(R.^2-x.^2), ...
@(x) sqrt(R.^2-x.^2))
ans =
    0.7854
```

The two-dimensional `integral2` takes in x_{min}, x_{max}, y_{min}, and y_{max} where the y limits can be functions of x. You'll also notice that `integral2` takes in a function of two variables now: x and y.

You can, if you wish, name your anonymous function:

```
>> fupper = @(x) sqrt(R.^2-x.^2)
fupper =
    @(x)sqrt(R.^2-x.^2)

>> fupper(0)
ans =
    1
```

you can then also pass them to functions like `integral`:

```
>> flower = @(x) -sqrt(R.^2-x.^2)
flower =
    @(x)-sqrt(R.^2-x.^2)

>> integral2(@(x,y) y.^2, -R, R, flower, fupper)
ans =
    0.7854
```

Vector operations. You probably noticed that we used `.^` to raise to the second power, rather than `^` and `.*` to do multiplication rather than `*`. This may look odd, but the reason is that `integral` (or `integral2`) is going to construct a *vector* of x values and call the integrand for the entire *vector*; it expects to get back a *vector* of answers. This is to make the evaluation of `integral` efficient: it finds a grid of x (or x and y values for `integral2`) and then passes them to the function. So we need to be able to call our functions like

```
>> fupper([-1, -0.5, 0, 0.5, 1])
ans =
         0    0.8660    1.0000    0.8660         0
```

and get a vector back. But if you try to use `^`, you'll get

```
>> [-1, -0.5, 0, 0.5, 1]^2
Error using  ^
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.
```

which is fixed like this

```
>> [-1, -0.5, 0, 0.5, 1].^2
ans =
    1.0000    0.2500         0    0.2500    1.0000
```

So: using `.^` tells MATLAB to apply `^` to each element in the vector or matrix individually. Similarly, you use `.*` to do an element-by-element multiplication. Many built-in functions, like `cos`, `sin`, `exp`, `sqrt` and so on *already work on vectors element-by-element*, so there's nothing you have to do differently.

Finally, this leads to probably *the most confusing* piece of MATLAB code you will encounter. Suppose you wanted to integrate an *area*

$$\int_{-R}^{R} dx \int_{-\sqrt{R^2-x^2}}^{\sqrt{R^2-x^2}} dy$$

(the area of a circle of radius R in this case). The integrand would be 1, so it would seem like you could just do

```
>> R = 1.
R =
     1

>> integral2(@(x,y) 1., -R, R, @(x) -sqrt(R.^2-x.^2), ...
@(x) sqrt(R.^2-x.^2))
```

But instead of getting π, you get the error

```
Error using integral2Calc>integral2t/tensor (line 241)
Integrand output size does not match the input size.

Error in integral2Calc>integral2t (line 55)
[Qsub,esub] = tensor(thetaL,thetaR,phiB,phiT);

Error in integral2Calc (line 9)
    [q,errbnd] = integral2t(fun,xmin,xmax,ymin,ymax,optionstruct);

Error in integral2 (line 106)
    Q = integral2Calc(fun,xmin,xmax,yminfun,ymaxfun,opstruct);
```

The reason is that `1` is *not* an array of the same size as the array of x and y values that were passed. So you need to do something a little different:

```
>> integral2(@(x,y) ones(size(x)), -R, R, @(x) -sqrt(R.^2-x.^2), ...
@(x) sqrt(R.^2-x.^2))
ans =
    3.1416
```

The function `ones` returns an array of 1's of a given size, and `size(x)` finds the size of the input array x. It looks a little odd, but it does what you need.