

# 15-385 Matlab Tutorial

---

## Questions?

1. Type `help <function>`, `doc <function>`, or `helpdesk`.
2. The location of most functions can be found by typing `which <functionname>`. Not sure which function you need? `lookfor <keyword>`. You can always open the `.m` file and see how it works. To print out the `.m` file to the screen, enter `type <functionname>`.
3. Still got questions? Email me, `yan+@cs.cmu.edu`

## First Steps

```
>> x = 5
>> s = 'dog'
>> who
>> whos
>> x = 5;
```

(semicolon hides the output)

Up and down arrow input at the prompt cycle through previous commands, now even in windows. Matlab also does completion - if you type `x` then press the up arrow, it will cycle to previous input that started with `'x'`).

Everything is a matrix. Indexing starts with 1, not 0, and 2D coords given as (row, col). Comments begin with the percent sign (%).

```
>> x(1,1)
>> x(1,2) = 6
>> x'
```

(the transpose operator. `help *`, `help -`, `help /`, etc all print out help for operators.)

```
>> ans*2
```

`ans` stores last value printed out. in this case it was `x'`. a multiplication (`*`) of a matrix by a scalar multiplies each element in the matrix.

## Basic Functions

```
>> x = [ 1 2 3; 4 5 6]
>> size(x)
>> length(x)
```

( `length(x) = max(size(x))` )

```
>> max(x)
>> min(x)
>> mean(x)
>> var(x)
>> sum(x)
>> prod(x)
```

these last six perform column-wise operations. So, to get the smallest element in each *row*, type `min(x, [], 2)` (min along the second dimension). Similarly, `mean(x, 2)` or `var(x')` will also operate on rows. Try these functions - if you don't know what they do, look up help.

```
>> x = [-2.5 1.2]
>> abs(x)
>> round(x)
>> fix(x)
>> ceil(x)
>> floor(x)
>> rem(x,2)
>> mod(x,2)
```

## Matrix subscripting, initializing, deleting...

```
>> x(:,2)
```

Colon by itself means all. So `x(:,2)` means all rows, second column.

```
>> x(:)
>> x(:, :)
>> x = [x; x]
>> y = [x' zeros(3,2)]
```

`zeros()`, `ones()`, `rand()`, `randn()` work in any number of dimensions. A single parameter implies a 2D square matrix (e.g. `zeros(4) = zeros(4,4)`)

```
>> y(2,:) = []
>> y = reshape(y,4,3)
>> z = rand(10)
```

Not sure how `rand()` and `randn()` differ? look up help.

```
>> help randn
```

Your best friend: the colon operator. It creates row vectors. Examples:

```
>> 1:10
>> -5:3:5
```

...useful as array indexing (but may also generate fractional values, `.1:.1:1`).

```
>> z(:, 1:2:end) = 1
```

('end' can be used to reference the end of the array)

## Plotting I: Fellowship of the Plotting

Now that you've done the basics, we can go fast...

```
>> x = -2:0.1:2;
>> y = sin(x);
>> plot(y)
>> plot(x,y)
```

Notice the difference? - (tighter axes but also...) Remember, typing all these repetitive commands is much faster if you type the first few letters, then scroll through past input with the up arrow.

```
>> grid on
>> plot(x,y,'-k');
>> hold on;
>> plot(x,-y,'.r');
>> hold off;
>> plot(x,y,'.-g');
>> subplot(2,2,3);
```

(Divide figure into 2-by-2 parts, ready to plot in 3rd one.) Colors are r,g,b,w,c,m,y,k and more.

```
>> plot(x,y,'k')
>> title('this is dope');
>> xlabel('i get the idea')
>> ylabel('can even throw LaTEx at it. wow');
>> text(0, .5, 'what a curve!')
>> subplot(2,2,2);
>> plot(x,-y,'r');
>> axis off
```

Look up `help axis` - these are all useful options. `xlim`, `ylim` also are often helpful; `legend` does what it promises; for other scales, see `semilogx`, `semilogy`, `loglog`.

## Plotting II: Return of the Plotting

Sometimes its nice to manually set locations of graphs in figures. Select figure 1:

```
>> f = figure(1)
```

This makes figure 1 the current figure, and returns the handle to the figure object, which is then stored in `f`. If figure `x` didn't exist, `figure(x)` will create a new one. You can get the handle of the current figure using `f = gcf`. Look at the object hierarchy (`doc figure`).

Let's start from scratch.

```
>> clf
>> leftaxes = axes('position',[0 .05 .5 .9]);
>> rightaxes = axes('position',[.5 .05 .5 .9]);
```

The object properties are usually defined as 'PropertyName', PropertyValue. Position values are given as arrays of [Xmin Ymin Xlength Ylength].

```
>> x = randn(100);
>> axes(leftaxes)
>> imagesc(x);
>> colormap gray;
```

Given a 2D matrix whose elements range 1-128, `image` will plot it as a bitmap, with the color indicating element value (using the current colormap). Built in colormaps are listed in the html help files. Ones I find most useful are `gray`, `jet`, `hsv`. `imagesc` automatically stretches the scale of `x` to fill the colormap.

```
>> axes(rightaxes)
>> y = sin(-2:.1:2)*cos(-2:.1:2);
>> mesh(y)
>> surf(y)
>> shading flat
>> shading interp
```

You can use the rotate view tool (selectable from the bar on the top of the window) to get a better look at the 3D surface. Figure, axes, labels, and lines can look however you want them to. Help on the load of visualization functions is browseable through `helpdesk`.

## Movie-related Funcs

The function `getframe` will take a snapshot of the current axis and return a Matlab “movie frame”. If you store successive frames in some array `M`, you can play them back using the function `movie`. Somehow, this is supposed to be more efficient than using a for-loop and `image`, but it’s obviously a hacky late addition to Matlab’s functionality, as it often behaves in strange ways.

## Element Opers, Trig Funcs

You’ll need to know the basic logical and comparison operators.

```
>> ~0
>> x = (1 == 0)
>> x = (1 ~= 0)
>> x = (1 & 0)
>> x = (1 | 0)
>> x = xor(1,0);
```

... those are “is equal to?”, “is not equal to?”, “and”, “or”, and “xor” if you’re unsure. They do great things on vectors and matrices! Speaking of exclamation point, ! passes the call to the OS, so you can do !ls, and eval(sprintf('!rm %s',filename)).

Other basic functions yet unmentioned: sqrt, exp, log, log2, log10 and of course the trig ones: sin, cos, tan, asin, acos, atan.

## Functions

Time to write your own functions. No more of that command line wiggidness. All you got to do is create a file that ends with .m, put it in the current directory, or add its directory to your path, path(path,<your\_dir>) then run it by typing the filename.

Example. A file called sphere\_data.m is created in the current directory and contains the following:

---

```
[X, C, evals] = function sphere_data(X)
% [X, C, evals] = function sphere_data(X)
%   Takes coordinates of points in 2D space, spheres the space
%   (rescales it so that covariance in all directions is 1)
%   and returns the original covariance matrix in C and its eigenvalues
%   in evals. In X, each row is an observation, each column a variable.

if (nargin < 1), % X is not given!
    fprintf('You have failed to provide any input.\n');
elseif (nargin < 2),
    str = 'This is here for demonstration purposes';
    fprintf('%s\n', str);
else,
    % won't get here - too many input args, error returned
end;

C = cov(X);           % compute the covariance matrix of X

[evects, evals] = eig(C); % eig returns the eigenvectors in the matrix evects,
                        % the eigenvalues in the diagonal elements of evects
evals = diag(evals);

X = X * evects * pinv(diag(sqrt(evals)))';
```

---

The function above begins with a comment. If a comment immediately follows the function declaration, it will be displayed as the help text. So, the first comment above will be displayed when the user types help sphere\_data.

Matlab routines are always call-by-value, so a function cannot modify its input arguments. So we could not get sphered data by writing a function with no output, []=function sphere\_data(X). Any variable update has to be explicitly stated, e.g. X = sphere\_data(X).

However, objects in Matlab are initially passed by reference, so unless you update it inside the function, passing a large matrix to the function will not cause Matlab to copy the whole data structure.

In the example you saw a simple if-else statement and some calls to `fprintf` (good for debugging and normal runtime output). The function `diag` was called twice with two different effects - when passed a matrix, it returns a column vector of the elements of the principal diagonal (the one that starts at position 1,1). When `diag` is passed a vector of length  $n$ , it returns a diagonal matrix of size  $n$  by  $n$ . You can also modify arbitrary diagonals of the matrix (see `help diag`).

Ever useful while developing Matlab code are the debugger tools. You can add a breakpoint by inserting `keyboard` somewhere in your code. The execution will pause there, and you will get the debugging prompt `K>>`. From here you can see what values are assigned to what variables, print them to the screen, plot them, check the sizes of your matrices, and continue step-by-step execution of whatever code you want. Useful (and mostly self explanatory) commands are `dbstep`, `dbcont`, `dbup`, `dbdown`, `dbstatus`, `dbstack`, `dbquit`.

Declaration of global variables is done with `global`. So, if a function declares `global x;`, then `x=3;`, and calls a different function, which also declares `global x;`, `x` will have the value 3 in the second function.

Final note about functions: you can include several in a single file, each beginning with its declaration. It's often nice to break off small plotting routines, and helps organize the code.

## Loops

While loops are just like the conditional statements above: `while(flag==true), ... end;` For-loop syntax is slightly different because of Matlab's vectorization. They look like `for(i=vec), ... end;` where `vec` is a previously initialized vector or a new vector construction. All these are valid:

```
>> vec = 5:10;
>> for(i=vec),      fprintf('%d',i); end;
>> for(i=(vec < 7), fprintf('%d',i); end;
>> for(i=find(vec)), fprintf('%d',i); end;
>> for(i=[1 3 4 5]), fprintf('%d',i); end;
```

## IO

`fprintf` and `sprintf`, previously mentioned, are used often to output to the screen or to a string, respectively. `fprintf` also writes to a file, hence its name. Look up help for details.

Single or multiple variables can be saved in Matlab's binary `.mat` format using `save`; and loaded with `load`. These can also handle ASCII (see `help save`). Other functions for file input are `textread` and `dlmread` for text files, `imread` for images.

## Matrix Guts

The key to Matlab's speed is the highly optimized library for matrix operations. Vectorize a for-loop so it becomes a matrix operation, and 10,000 calls will execute almost as fast as 10. You've seen that all low level operands work on matrices as well, so most of the time, explicit loops can be avoided.

Multidimensional arrays are possible: `zeros(3,4,2)`, and are indexed in familiar ways. `shiftim` is useful for reordering dimensions. Multidimensional arrays may contain singleton dimensions (a dimension along which the array is flat, i.e. `size(A,dim)==1`); use `squeeze` to get rid of these. By the way, Matlab **will** grow your arrays/matrices as necessary, so if you're filling in huge matrices as you go along, it's a good idea to initialize them to zeros so Matlab can allocate the right amount of space.

*Cells* are collections of arrays that can contain all sorts of objects, such as matrices, strings, and complex numbers. *Structs* are also useful, and are referenced in a familiar way. We can create a new struct and pass it around as a single variable:

```
>> DispOptions.plotf = true;
>> DispOptions.nfigs = 3;
>> DispOptions.figtitle = 'my plot';
>> plotData(X,DispOptions);
```

Some matrix operations have been mentioned already, I list some others below:

```
>> X = eye(3)
>> X = X' + floor(rand(3)*10)
>> diag(X)
>> X = X + diag(diag(X))
>> trace(X)
>> det(X)
>> [V D] = eig(X)
```

## Numbers

`find` returns the indices of an array that satisfy some condition; functions like `max` and `min`, called with two output parameters, will also return the indices of the maximum or minimum values.

Semantics particular to Matlab are `inf` (infinity, e.g.  $1/0$ ) and `nan` (non a number, e.g.  $0/0$ ). Useful functions related to these are `isinf`, `isnan`, `isfinite`, `isempty`, and `isreal`.

Complex numbers are handled easily, inline:

```
>> x = 1 + 3i
>> y = x + 2i
```

See also `complex`, `real`, `imag`, and `conj`.