MATLAB bootcamp notes – Class 1

Written by Kyla Drushka, modified from notes by Darcy Ogden for course SIO113

Format of these notes:

Times font gives information and instructions. `Courier` is used to represent computer code; **`>> bold-faced courier`** represents text you type into MATLAB's command-line (no need to actually type in the >> , just the text that comes after).

I have included lots of examples in these notes. I encourage you to copy+paste them into your MATLAB command window so you can see for yourself how things work. Be careful with the quotes, though: copy+paste has the annoying tendency to convert single quotation marks ' into something fancier, but MATLAB only recognizes basic quotation marks. If you paste directly from these notes and receive an error (or if nothing at all happens), try replacing the pasted quotation marks by typing them in directly.

The MATLAB computing environment

MATLAB has a few different windows. The ones you will mostly use are:
(1) **command window**: where you can type commands and see the output of any commands or programs you run.
(2) **editor**: where you can edit programs and scripts (.m files)
(3) **figure windows**: figures are plotted in separate windows.

Navigating within the command window:

- The up arrow allows you to see your last command (you can then press return to run the command again)
- If you press the up arrow N times, you will see your N^{th}-ago command....
- Typing a letter (or two) and then the up arrow allows you to see the previous command that started with that letter
- Typing a letter (or two) and then TAB will give you suggestions of available variables, scripts, and functions starting with that letter.
- ESC key erases what's on the command line.
- CTRL+c interrupts the current command being run (useful if you are stuck in an endless loop, accidentally printing an enormous variable to the screen, or are otherwise panicking).

Getting help

If there are ever commands you need some help with, type
`>> help [commandname]`
into the command window, e.g. **`help figure`**

Variables

Typing into the command window

You can type commands directly into the command window. For example, you can evaluate simple arithmetical expressions by typing them in directly:

```
>> 1+3
```

Which will print to the screen:

```
ans =

     4
```

MATLAB has evaluated the expression 1+3 and stored the answer (4) in a temporary variable called `ans`.

Type another expression:

```
>> 9*9
```

This produces

```
ans =

    81
```

MATLAB has overwritten the variable `ans` with the value 81.

`ans` is the default variable name that MATLAB assigns. It is generally more useful to create your own variable names.

Creating variables

It is easy to create a MATLAB variable: simply type the name of the variable (e.g. **`x`**) and an equals sign, and the value you want to assign to it. (You do not have to declare variables before using them the way you do with some other languages). For example:

```
>> x=3
```

```
x =
     3
```

Now, you have a variable named **`x`**, which has a value of 3. Any time you type **`x`**, its value will be printed to the screen. Any time you use **`x`** in your code (e.g. in an equation), MATLAB will substitute its value (in this case, 3).

We can create as many variables as we want this way. E.g. type

```
>> x2=10
```

to create a new variable called **`x2`**, which has the value 10.

All of your variables are contained in your "workspace". To check which variables are contained in your workspace, use the command `who` or `whos`: e.g. typing

```
>> who
```

prints to the screen all the variables in your workspace.

You can also visualize and edit your workspace graphically using the workspace editor. Access this by typing

```
>> workspace
```

To remove variables from your workspace, use the command `clear`: e.g. **`clear x*`** clears all variables starting with the letter x; **`clear x y z`** clears the variables named x, y, and z, and **`clear`** (with no arguments) removes *all* of the variables in your workspace (so be careful with this one!)

If you assign a value to a variable that is already defined, the old value gets overwritten. E.g.

```
>> x2=1
>> x2=99
```

will first assign the value 1 to the variable **`x2`** , and will then overwrite it with the value 99.

Built-in variables

Matlab has a handful of build-in variables

e.g. type

```
>> pi
```

Others include `eps, inf, NaN or nan, i , j, nargin, nargout, realmin, realmax.` You don't need to know these all now, but it's generally good practice to avoid creating your own variables with these the names of built-in variables (and thus overwriting them).

Other comments on variables:

Variables are case-sensitive, so `x1` and `X1` are completely different. Variable names can contain numbers and underscores (_) but they must start with a latter (e.g. `x2` works as a variable name, but not `2x`).

If you include a semicolon at the end of the line:

```
>> x=3;
```

this will still create the variable `x` having the value 3, but the echo (the output to the screen) will suppressed.

Basic math

MATLAB's basic arithmetical operators are `+ - * / ^`

The expression on the right-hand sign of a single equals sign is evaluated and stored in the variable on the left-hand side of the equals sign. For example:

```
>> x=3+1
```

MATLAB performs this calculation (3+1) and stores the answer in the variable `x`.

```
x =
     4
```

You can also use variables on the right-hand side of mathematical expressions, but only after they have been defined. For example, type

```
>> x=3;
>> y=2+x
```

this returns:

```
y =

     5
```

However,

```
>> y=2+k
```

results in an error because `k` has not yet been defined.

If you don't include a variable name on the left-hand side of an equation, MATLAB will evaluate the expression and store it in a variable called `ans`. `ans` is just like any variable: if you then assign another value to it, the previous value will be overwritten. E.g. typing in

```
>> 5+1
```

returns

```
ans =

     6
```

Then,

```
>> ans+7
```

returns

```
ans =

     13
```

…we have taken the first value of `ans` (6) and added 7 to it, giving a value of 13, which is then stored in `ans`.

Order of operations:

MATLAB uses a strict order of operations to evaluate mathematical expressions. You can use parentheses to control the order of operations. Spaces are ignored: you can include them or not. E.g.

```
>> 2+4 *4
```

gives 18, whereas

```
>> (2 + 4)* 4
```

gives 24.

Make sure to keep your parentheses in mind when you are entering exponentials and fractions:
`4^1/2` will give a different result than **`4^(1/2)`**

Built-in functions

MATLAB has a whole bunch of mathematical functions built into it. For example,

`>> log(x)` returns the natural log of the variable `x` ; similarly,
`>> log(9)` returns the natural log of the number 9.
`>> log10(9)` returns the log (base 10) of the number 9.
`>> sqrt(x)` returns the square root of `x`
`>> sin(x), cos(x)` return the sine and cosine of `x` (in radians)
`>> sind(x), cosd(x)` return the sine and cosine of `x` (in degrees)

Most of the built-in functions you will need probably already exist in MATLAB – it's just a question of figuring out what the function name and syntax is. Here are some tips for finding the name of a function you're after:

- do a google search
- start typing a likely function name and then press TAB to fill in the rest of the name
- invoke the `help` command for a similar function and look at the "see also" suggestions (e.g. **`help sin`** suggests looking at `sind`)

Matrices and arrays

A major strength of Matlab is that is allows you to store data in matrices, which makes it easy to manipulate.

Definitions:

Scalar: a single number. E.g.:

```
>> s=4
```

Vector**:** a one-dimensional set of values in a single “row” or “column”. Enter row-vectors by putting square brackets around a list of values:

```
>> r=[2, 3, 4, 5]    % r is a 1x4 vector.
```

Separating numbers with commas implies that the numbers are all in the same row. The commas are actually optional, but they do make things more clear.

```
>> r2=[2 3 4 5]       % this is equivalent to r=[2, 3, 4, 5]
```

Enter "column vectors" by separating the values with semicolons:

```
>> c=[1; -1; 4]        % c is a 3x1 vector
```

Matrix: a two-dimensional set of numbers. Enter matrices line-by-line, separating each new row with a semicolon:

```
>> m=[1 2 3; -3 3 9]    % m is a 2x3 matrix
```

Array: a multi-dimensional set of numbers. You can have "as many" dimensions as you want, though more dimensions uses a lot of memory. (Note that vectors and matrices are also types of arrays.)

`length` tells you the length of a vector:

```
>> length(r)
```

`size` tells you the dimensions of an array; the output is listed in the order: #rows, #columns, length of dimension 3, length of dimension 4, ...etc

```
>> size(m)
```

Other ways of creating arrays:

The colon operator generates a series of numbers from a min to a max with a certain increment using the syntax

```
min : increment : max
```

e.g.

```
>> r=1:2:10  % creates a 1x5 vector containing values
1,3,5,7,9
```

if the increment is omitted, Matlab assumes an increment of +1:

```
>> r=1:10   % creates a 1x10 vector of values from 1 to 10
```

The increment can also be negative:

```
>> r=20:-2:10  % produces a vector containing values
20,18,16,...,10
```

Other commands for creating arrays:

```
>> x=ones(3,4,2) % creates a 3x4x2 array of all ones

>> x2=4*ones(2,1) % creates a 2x1 vector of 4s

>> y2=zeros(2,2)  % creates a 2x2 matrix of zeros
```

```
>> y=zeros(2) % also creates a 2x2 matrix of zeros (i.e. when you only supply one argument, Matlab assumes you want a square matrix; to make a vector, you must specify that one of the dimensions is 1

>> z=rand(1,4) % creates a 1x4 vector of (roughly) random numbers from 0 to 1

>> n=nan(3,4)  % creates a 3x4 matrix of NaNs (not-a-number)
```

Combining arrays:

You can also combine multiple arrays to make bigger ones, using brackets and colons and semicolons the way as with numbers. Note that matrix dimensions must agree if you are to combine them:

```
>> a=[2 3 1; 4 9 1]  % a is 2x3
>> b=[4 0 9]         % b is 1x3
>> c=[a ; b]          % c is composed of a with b below it
>> bad=[a , b]        % this gives an error: you are trying to put a and b next to each other, but they don't have the same number of rows so it doesn't work.
```

Transpose a matrix with an apostrophe:

```
>> m=[1 2 3 4;5 5 5 5; 0 2 1 -2]; % m is a 3x4 matrix
>> m'    % m' is the transpose of m, a 4x3 matrix
```

Basic plotting

The basic command to plot is `plot`. There are a number of variations for the syntax; the most basic

```
plot(y)
```

or

```
plot(x,y)
```

where `y` is the name of the variable you want on the vertical axis, and `x` is the name of the variable plotted on the horizontal axis (which is optional; if not specified, MATLAB assumes that `x` has uniform spacing). Note that `x` and `y` don't have to be variable names; you can also use numbers.

Try this:

```
>> x1=1:10
```

This creates `x1`, a vector of numbers from 1 to 10; more on vectors tomorrow.

```
>> y1=x1*2
```

This creates `y1`, which is equal to twice `x1`.

```
>> plot(x1,y1)
```

This will open a figure window (by default, Figure 1) and plot the `y1` as a function of `x1` (as a blue line, by default).

Note that the names of the arguments you give to the plot command are completely arbitrary – regardless of the name, the first argument will be plotted as the "x" axis and the second argument as the "y" axis.

You can plot multiple lines using the same `plot` command by using additional arguments. E.g.

```
>> a=[3 4 5];
>> b=[1 1 1];
>> plot(x1,y1,a,b)
```

will show `y1` as a function of `x1`, and `b` as a function of `a`, on the same figure.

Plot styles

By default, MATLAB plots a line; you can specify if you want to plot markers or different types of lines instead:

```
>> plot(x1,y1,'o')
```

plots circular markers

```
>> plot(x1,y1,'--')
```

plots a dashed line

```
>> plot(x1,y1,'*-')
```

plots star-shaped markers and a solid line.

You can also specify the color of the line/markers that you plot:

```
>> plot(x1,y1,'r')
```

plots a red line

There are many, many options for controlling what your figures look like. We will talk about more options later in the class.

By default, the plot command will first delete any existing plot. If you don't want the current plot to disappear when you invoke the next `plot` command, use **`hold on`**

```
>> plot(x1,y1)
>> hold on
>> plot(4,4,'r*')
```

If you do want the current plot to be overwritten by the next plot, you can use **`hold off`**

Saving figures

The **`print`** command is used to save figures. The default format is a black+white postscript; you can use additional arguments to change the format. For example:

`>> print figname`
stores the current figure as postscript file called `figname.ps`

`>> print figname -dpdf`
stores the current figure as PDF file (`figname.pdf`)

`>> print figname -dpsc`
stores the current figure as color PS file (`figname.ps`)

`>> print figname -djpeg`
stores the current figure as color JPG (`figname.jpg`)

Figures and subplots

By default, when you use the **`plot`** command the plot is made in a new window titled Figure 1. You can specify a different figure if you like, using the command **`figure`** :

`>> figure`
opens a new figure; by default, the figures are opened sequentially, so if you already have a Figure 1 open, `figure` will open a Figure 2 window.

`>> figure(6)`
opens a figure window called Figure 6. If Figure 6 already exists, it will cause the next plot to be made in the Figure 6 window.

`>> close`
closes the current figure

`>> close figure(6)`
closes Figure 6

`>> close all`
closes all open figures

To clear everything that it plotted on a figure without closing it, use the command **`clf`**. It is often helpful to use **`clf`** every time you plot a new figure, just to make sure you aren't adding plots to a previously created figure.

You can put multiple small panels in a given figure window using the **`subplot`** command, which has the syntax **`subplot(m,n,p)`**, where `m` represents the number of rows of subplots, `n` represents the number of columns of subplots, and `p` represents the number of the current subplot: e.g.
`>> subplot(3,1,2)` makes a subplot in the second row of a 3-row, 1-column figure.

Saving your code: scripts

So far, we have just been entering commands directly into the command window. This works fine, but what if we want to do a whole bunch of commands? What if we want to run the code again later? You can write your code in the Editor window and store it in a text file called a *script*. You then tell MATLAB to follow all the commands you have written in the script. MATLAB needs to know where to look for these scripts, so change into the directory/folder where you will store your scripts for this class using the `cd` command, e.g.
`>> cd ~/class/` (on a Mac or Unix machine)
(You can also change folders using the command window interface)

Open the editor window by typing the command **`edit`** into the command window. The MATLAB editor is just a basic text editor. Type the following into the editor:

```
x=0:pi/100:2*pi;
y=sin(x);
plot(x,y,'k-')
title('sin(x)')
```

save this file as `mysine.m` in the directory you chose above. Note that MATLAB scripts must have the suffix `.m`

You can either run the script from the editor by pressing F5 or by typing the name of the script into the command window (note that you don't have to include the `.m`)

```
>> mysine
```

MATLAB reads scripts line-by-line, from start to finish. If there are any errors, the script will exit and an error message will display in red on the command window. You can use this message to figure out where your mistake was. For example, in your `mysine.m` script, change the second line to read:

```
y=sin(x
```

Now re-save the script and try running it. The second parenthesis in the `sin` command is missing, so MATLAB doesn't know how to execute the command and it returns an error, which tells you the exact location in the script where things have gone wrong:

```
Error: File: mysine.m Line: 2 Column: 8
Expression or statement is incorrect--possibly unbalanced (,
{, or [.
```

Commenting

It is often useful to add comments to your scripts in order to document what the code does. Including comments is a super important habit to get into. It may seem like a lot of effort, but you will thank yourself (and anyone who reads your code) in the future if you include detailed comments throughout the script as well as in the header.

Add comments by putting a `%` in front of any text you want MATLAB to ignore. E.g. here is my commented version of `mysine.m` :

```
% This script calculates the sine from x=0 to 2pi, and plots
it with a black line.
% Created by kyla d. on sept 14, 2013
x=0:pi/100:2*pi;       % define x from 0 to 2pi
y=sin(x);      % calculate sin (note, it's in radians!)
plot(x,y,'k-')
title('sin(x)')
```