Open Matlab

1 WYSWYG

If you haven’t already go ahead and open Matlab. You are invited to fiddle with it while I am talking.

Behold the power and the glory that is Matlab. There are a number of bits and pieces to this display, but we will be focusing on the box in the center.

The existence of this command window is one of the most important things about Matlab. Why? We can view partial work, and get instantaneous feedback. None of these statements are overly specific, so let’s start fiddling around. They will make more sense as we proceed.

So what can we do at the Command Window? We can execute commands. What are commands? A surprising number of things are commands. Suppose I wanted to compute $e^{1/\pi}$. Adjusting for matlab syntax I can do this. For those infront of Matlab terminal try typing in $\exp(1)/\pi$. The command window should show the following:

```matlab
>> \exp(1)/\pi
```

```matlab
ans =

0.8653
```

Neat, ain’t it? You can use Matlab as a calculator. For completeness, I am going to calculate a horrible number to illustrate the syntax of Matlab calculation.

```matlab
i+\exp(\sqrt{5^(0.2)})+\text{norm}(5,\text{inf})
```

```matlab
ans =

8.2369 + 1.0000i
```

So, what has been shown? We can do math in the command window. We have a glorified calculator. We can also define and assign values to variables.
Look at the top right corner. The box up there is called Workspace. It shows all variables currently in memory. It also shows the value of those variables. We will return to this box in time.

What if I don’t want to see what x is equal to? What I just want to assign x to a value and not be reminded of that fact? Well, add a semicolon.

Adding a semicolon at the end of a statement suppress output. It also allows you to have multiple commands on a single line.

x=40;2+2*exp(-1)

2.7358

Note that I have 2 variables in memory, "x" and "ans". If you do not assign a variable to a statement, Matlab stores the output of the statement to ans.

ans

2.7358

So we have variables. Some of you may be wondering what exactly the variables are. This brings us to a useful built in function "class".

class(x)

double

"x" is a double. "x" is a double precision floating point number, to be precise. A double is the basic data type you will deal with in your coursework. It is the highest default accuracy math type available.

Now, I’d like to pause, and say our Command Window is messy. I want all of that text to go away. Guess I have to restart..., or type "clc". Behold, my Command Window is pristine. But I still have too many variables. Time to clear things up. type "clear".

We have returned to our original state. But wait, I skipped something. I said that "x" has a type. Doesn’t that imply that there are different types? Well yes.
you can check here for the official list of variables.

There are two more that are of interest. Above, we showed that we can do mathematical operations. We can also perform logical operations. Behold, as I check some basic facts:

```matlab
>> 1==1
ans =
     1

>> 1~=2
ans =
     1

>> 1>0
ans =
     1

>> 1<=2
ans =
     1
```

Note that 1 is true, 0 is false.

The next one is pretty straightforward. We sometime need to print out to the command window informing the user that something has occurred. We have characters to do that:

```matlab
>> hello='Hello World'

hello =
Hello World

Now let’s try and add two strings together

```matlab
>> 'hello'+jello'
ans =
     210    202    216    216    222
```
What the heck? First of all, when you use a arithmetic operator, Matlab casts everything to a double. When you tell it to add two arrays of characters, it doesn’t know what you mean, so it converts them to their ascii values, and adds those. Second of all, EVERYTHING IS MATRIX.

As an aside, if you wish to delete a particular variable, you type ”clean ‘variblename””.

## 2 Everything is a Matrix

Matlab stands for Matrix Laboratory. So everything is Matrix. We can store things like vectors, matrices, and such easily using Matlab.

let’s try defining a row vector:

\[
x = [1 \ 2 \ 3]
\]

\[
x = \\
1 \quad 2 \quad 3
\]

let’s try defining a column vector:

\[
x = [1; \ 2; \ 3;]
\]

\[
x = \\
1 \\
2 \\
3
\]

Note that the usage of semicolons inside of brackets indicates the beginning of a new row. There exist a host of functions that generate vectors with special structures. One of particular interest is a mesh vector. Suppose I want a uniform mesh from point a to point b, with width h.

\[
a=0; b=1; h=.2; a:h:b
\]

\[
ans = \\
0 \quad 0.2000 \quad 0.4000 \quad 0.6000 \quad 0.8000 \\
1.0000
\]

Note, that if your \( b-a \ mod \ h \neq 0 \) the resultant vector will not provide a perfect mesh.

\[
h=.4; a:h:b
\]

\[
ans = \\
0 \quad 0.4000 \quad 0.8000
\]
This is useful, but might not be exactly what you desire. The vector \(a:h:b\) is often used in control structures, so you should be familiar with it. Another, similar, way of generating meshes is \texttt{linspace}.

\[
n\text{numpoints}=3; \text{linspace}(a, b, \text{numpoints})
\]

\[
\text{ans} =
\begin{bmatrix}
0 & 0.5000 & 1.0000
\end{bmatrix}
\]

\texttt{linspace} or linear spaced vector, takes a startpoint, an endpoint, and the desired number of points, and generates the uniform mesh.

Note: to get the conjugate transpose, you use ‘ for the regular transpose you use \texttt{transpose} or .’

\[
x=[1+i 2]
\]

\[
x =
\begin{bmatrix}
1.0000 + 1.0000i & 2.0000 + 0.0000i
\end{bmatrix}
\]

\[
>> x'
\]

\[
\text{ans} =
\begin{bmatrix}
1.0000 - 1.0000i & 2.0000 + 0.0000i
\end{bmatrix}
\]

\[
>> x.'
\]

\[
\text{ans} =
\begin{bmatrix}
1.0000 + 1.0000i & 2.0000 + 0.0000i
\end{bmatrix}
\]

\[
>> \text{transpose}(x)
\]

\[
\text{ans} =
\begin{bmatrix}
1.0000 + 1.0000i & 2.0000 + 0.0000i
\end{bmatrix}
\]

Matrix notation follows easily from this.

\[
A=\begin{bmatrix}
\end{bmatrix}
\]

\[
A =
\begin{bmatrix}
\end{bmatrix}
\]
We will talk about mathematical operations attached to matrices and vectors in a bit. Before that we should look at some indexing. Matlab is has a one-based indexing system. This means, that the first index in any vector, matrix, etc will be 1. This is contrary to other programing languages (such as c, c++) where 0 is the starting index.

```matlab
x=[4 5 6]; x(1)
ans =
   4
>> x(2)
ans =
   5
>> x(3)
ans =
   6
```

What if I want the first 2 values?
```
>> x(1:2)
ans =
   4   5
```
equivalently, we can pass in [1 3] or [2 3] or [1 1], etc. This allows us to access subvector or submatrices directly. This is extremely useful, and makes code far more compact and easier to read. Now for some fun.

```matlab
A=[[1 3];[2 4]];
A(1)
ans =
   1
>> A(2)
```

```matlab
1 2 3
3 4 6
7 8 9
```
ans =
  2

>> A(3)
ans =
  3

>> A(4)
ans =
  4

What happened? Matlab matrices, and matrices in general are stored on flat arrays. Any vector, matrix or tensor has an associated linear ordering. This is because really an $m \times n$, matrices are stored in memory by single vector of length $mn$

>> A(1,2)
ans =
  3

>> A(:,1)
ans =
  1
  2

>> A([2 2],:)
ans =
  2   4
  2   4

>> A(:)
ans =
  1
  2
  3
4
>> A(end)
ans =
   4

• (i,j) is ith row, jth column
• "":" indicates the entire row/column
• vector indicates which rows/columns you want
• "." vectorizes the matrix by column major
• end is a keyword that returns the last index’s value

A couple useful tools for interacting with matrices is the ability to compute the dimension of a variable, and the number of elements in a variable. To do so, we use size, and numel respectively.

Besides our direct instantiation of matrices, we have access to some pre-canned ones. Matlab has a variety of functions for construction of matrices with special structures.

• ones creates a matrix of ones
• eye creates an identity matrix
• rand creates a random matrix

There are many more, but these three by far the most used.

3 Power of the Matrix: Matrix operations

We have the command window, we have arithmetic, we have matrices, we have indexing. Now to show the workhorse of Matlab, the juicy juicy built-in matrix functionality.

We can do traditional operations +,-,*, etc along with elementwise versions of those operators by placing a "."," in front of the command.

Let’s construct a couple matrices to illustrate what I mean.

>> A=2*ones(2,2);B=eye(2,2);
>> A^2
ans =
   8   8
   8   8
A.ˆ2

ans =

    4   4
    4   4

A^2 calculates A * A where A.ˆ2 takes each element of A to the second power.

>> A*B

ans =

    2   2
    2   2

>> A.*B

ans =

    2   0
     0   2

A * B calculates the matrix multiplication, while A. * B does multiplication at an element level.

Of course, one of the most useful things about Matlab, is that it makes computing matrix inversions very, very easy.

To solve AX = B, simply type A\B

>> A\B

ans =

     1.0e+15 *

    2.2518  -2.2518
   -2.2518   2.2518

Wait, what? A is singular, why was this okay? Well, it isn’t. Matlab tends to prefer not to throw errors, but give warnings, and a best effort answer. In some cases it does is not able to detect an issue, even though there is one. Matlab can, when you use backslash, perform a number of different methods, to give you a reasonable answer. I only mention it because it can be dangerous to assume you know what is occurring. Read the documentation!

Okay, that aside, Let’s perturb A to show that it actually works.

>> A=A+[[0   1];[0   0]];
>> A\B
ans =

\[
\begin{pmatrix}
-0.5000 & 1.0000 \\
0.5000 & -0.5000 \\
\end{pmatrix}
\]

>> A*(A\B)−B
ans =

\[
\begin{pmatrix}
0 & 0 \\
0 & 0 \\
\end{pmatrix}
\]

We can also solve \( xA = B \) by using \( B/A \).

>> B/A
ans =

\[
\begin{pmatrix}
-0.5000 & 1.0000 \\
0.5000 & -0.5000 \\
\end{pmatrix}
\]

>> (B/A)∗A−B
ans =

\[
\begin{pmatrix}
0 & 0 \\
0 & 0 \\
\end{pmatrix}
\]

On top of Matrix Operations, we also have a plethora of matrix factorizations.

- LU
- Cholesky
- QR
- SVD
- Eigenvalue
- Schur

Probably the most useful in this tutorial is eig.

>> eig(A)
ans =
\[ V = \begin{bmatrix} 0.8165 & -0.8165 \\ 0.5774 & 0.5774 \end{bmatrix} \]

\[ D = \begin{bmatrix} 4.8284 & 0 \\ 0 & -0.8284 \end{bmatrix} \]

So we do basic arithmetic, declare variables, and we can solve linear system. Now what?

## 4 Scripts

Suppose you were asked to solve \( Ax = b \). You entered your variables, called backslash, and wrote down your answer. You close matlab and go about your daily life.

What happens if you want to solve that problem again, but with a different right hand side? So you open matlab, enter in your variables, call backslash, and write down your answer.

What happens if you want to solve that problem again, but with a different right hand side? So you open matlab, enter in your variables, call backslash, and write down your answer.

What happens if you want to solve that problem again, but with a different right hand side? So you open matlab, enter in your variables, call backslash, and write down your answer.

Clearly, this is unpleasant in the extreme. Once you set up a problem, you don’t want to go back and redo it every time you open matlab.

Enter, scripts.

What is a script? It is a series of commands contained in a file. When you type the name of the file, Matlab executes all of the commands in the file.

But first, a digression. Consider the Matlab gui.
On the left is a listing of all files and folders in your current folder or present working directory. Unless you fiddle with path variables, this is the scope of files available to you. So let’s navigate to a folder where we actually want to do work. Above your command window, is a bar that indicates your present working directory.

Now to create a script. Click new, and choose script, or type `ctrl+n`.

Now we need to name this script. For historical reasons, I will name mine `foo`. Note that the name of the file is what you must call, so no spaces in the name. A couple things to note. Any line that starts with the character `%` is a comment. It is ignored by Matlab, in that it is not executed. Comments are useful for anyone reading your code, including future you.
4.1 Control Structures

Matlab is not only a numerical linear algebra toolbox, but a programming language. It is not a very efficient programming language, but it incredibly useful for prototyping and testing ideas.

With this in mind, we have the usual slew of control structures that come with a programming language.

Control structures are commands that allow code to analyze variables and make decisions based upon that analysis. Think predicate logic.

The major ones of interest are "if","while", and, "for". below are the conceptual structures. We'll demo them after this.

```matlab
if
  logical
  dostuff
else
  dootherstuff
end
```
e.g.
```matlab
if (number_of_inputs == 0)
  disp('You must enter at least one input!');
else
  disp('Inputs entered');
end
  elseif
  if logical
    dostuff
  elseif logical
    dootherstuff
  else
    dootherotherstuff
end
```
e.g.
```matlab
x = 0;
if (x < 0)
  disp('x is negative');
elseif (x > 0)
  disp('x is positive');
else
  disp('x is zero');
end
```
```matlab
while
```
while logical
dostuff
end

e.g.
j = 0;
while (i < 10)
    if (isprime(j))
        i = i+1;
        disp(j)
    end
    j = j+1;
end

for
for values
dostuff
end

e.g
x = rand([2,10]);
for i = 1:10
    disp(norm(x(:,i)));
end

There is another control structure, switches, but they do not come up that often. If you find yourself writing a huge if-elseif chain, look up switches. Otherwise, the above are sufficient.

And now for the chunk of code illustrating Matlab syntax.

5 Functions

Now scripts are pretty nice and work well when you want to run a couple of commands or do something to the variables you have declared in the workspace, but sometimes you want to do something complicated, or maybe something without changing the variables that you have already declared. Enter the function

Functions are scripts that are self contained and take in a few arguments, and then output a few values. Lets make things a bit more concrete with an example.

Lets say that you are doing your younger brother’s geometry homework where she has to compute the area of a bunch of triangles given their height and base. You could compute the area by typing:

```matlab
>> b = 3;  h = 5;  a = .5*(b*h)
```
Now that works pretty well if you only have to compute the area of a few triangles, but if you had to compute the area of a bunch of triangles typing all that would get old fast. Let's make a function for computing the area of triangles to save ourselves some typing. Make a new script, except this time type this:

```matlab
function a = triarea(b, h)
a = 0.5*(b.*h);
```

Here the name of our function is triarea, and if you give it 2 inputs (b,h) then it outputs a value (a) which is the area of a triangle with base b and height h. Then save the file as 'triarea.m' and take it for a spin. Try typing:

```matlab
>> a = triarea(5,3)
a = 7.5000
```

We did the same computation as before, but this time we didn't have to type as much. An important thing to mention here is that unlike a script when you are running code inside a function MATLAB creates a new workspace and puts all other variables on the shelf until the function finishes.

Functions can also output a matrix. Suppose we want to write a function to compute the roots of a quadratic polynomial. It might look like this:

```matlab
function roots = quadroots(a, b, c)
x1 = (-b+sqrt(b^2-4*a*c))/(2*a);
x2 = (-b-sqrt(b^2-4*a*c))/(2*a);
roots = [x1; x2];
```

Then if we want to know the roots of the polynomial \(x^2 + 2x + 1 = 0\) then we just type:

```matlab
roots = quadroots(1, 2, 1)
```

```matlab
roots =

-1
-1
```

Easy peasy. These functions are pretty simple, but what if we wanted to something more complicated, like maybe take in a huge vector of values, compute their standard deviation? Of course MATLAB already has a function for doing this, but ours will be way better!

```matlab
function output = my_stdev(V)
n = length(V);
```

```matlab
total = 0;
for i = 1:n
    total = total + V(i);
end
average = total / n;
variance = 0;
for i = 1:n
    variance = variance + (V(i) - average)^2;
end
output = sqrt(variance / (n-1));

Now we can save ourselves a lot of time and typing if we want to compute these things later. There is no limit to the things that functions can do, and in practice they can be hundreds, or thousands of lines long and range from computing something very simple to running an entire simulation.

5.1 Inline Functions

Finally let’s talk about inline functions. Inline functions a little functions that are so small that they don’t deserve their own file. For example, let’s say that you wanted to express the mathematical function $f(x) = x^2$ in MATLAB. One thing that you could do it:

```matlab
function srq = f(x)
sqr = x^2
```

But as you can see, this is a lot of work to do for a little function. what’s worse, is that if you had a whole bunch of functions, it might be difficult to keep track of them all. For that reason there is something called an inline function, i.e. a function, but one that’s declared on a single line.

Try typing

```matlab
>> f = @(x) x^2
```

f =

```matlab
 @(x)x^2
```

By writing that line we defined a special variable called a function handle. A function handle is a special kind of variable, that denotes that the variable f is actually a function, and should be evaluated as such. Then we can do things like

```matlab
>> f(1)
ans =
```

1
If we want to describe a function of two variables, then we put more than one thing inside of the parenthesis.

```matlab
g = @(r, theta) r*exp(i*theta)
g = @(r, theta) r*exp(i*theta)
```

Remember! Any inline function could be rewritten as a proper function in it's own .m file, but inline functions are nice for little functions.

### 6 Plotting

One of the strengths of MATLAB is that it makes it very easy to data, and so you can quickly make sense of output. Out of all the different kinds a few of them are most useful, plot, scatter, hist, surf and imagesc.

`plot(x,f(x))` is the default function for plotting 1 dimension of data. For example,

```matlab
>> x = .1:.1:6*pi;
>> plot(x,x.*sin(x))
```

Gives us:
If we try and plot something else, then it will replace what we already have on the plot, so if we also want to play the function \( f(x) = x \), then we have to type 'hold all.' To differentiate the plots I can also add a legend.

```
>> x = .1:.1:6*pi;
>> plot(x,x.*sin(x))
>> hold all
>> plot(x,x)
>> plot(x,-x)
>> legend( 'x*sin(x)', 'x', '-x')
```
Notice that every time we plot a new function, it is plotted in a different color. If you want to plot something, but don’t want to plot over what you already have, just type ‘figure’ to open a new figure, so your old figure won’t be erased.

Next is ‘scatter(x,f(x))’ This function is just a scatter plot, meaning that it won’t attempt to connect all of the data with a solid line. This is useful if you want to view data that may not be in a useful order.

```matlab
>> y = normrnd (0,1,[1000,1]);
>> x = normrnd (0,1,[1000,1]);
>> scatter (x,y)
```
If I want to scatter plot to be in a different color, or use a different marker, then I can add additional options by including them in apostrophes. Kere ‘c*’ specifies that I want the output to be cyan, and use a star as a marker. Plot has these additional options too.

```matlab
>> figure;
>> y = normrnd(0,1,[1000,1]);
>> x = normrnd(0,1,[1000,1]);
>> scatter(x,y,'*c')
```

If I wanted to label the axis, and add a title, then I’d use these commands:

```matlab
>> xlabel('random x variable')
>> ylabel('random y variable')
>> title('Scatter plot of some random data')
```

As you can imagine, these labeling commands are the same if you are using plot, scatter, hist, etc... I can even specify that I want the title to be in red, at 18pt font.

```matlab
>> title('\fontsize{18}\color{red} Scatter plot of some random data')
```
It looks terrible, but you get the idea.

'hist' is the command that you use if you want to make any histograms.

```matlab
>> x = normrnd(0,1,[1000,1]);
>> hist(x)
```
By default it uses 10 buckets, but we can add more, like 100!

>> \texttt{hist(x,100)}
'surf' is the command for you if you want to plot a surface plot of a function of the form \( \text{surf}(x,y,f(x,y)) \). This is useful if the thing you are plotting depends on more than one variable.

\[
\begin{align*}
    &>> x = -2\pi : 0.5 : 2 \pi ; \\
    &>> y = -2\pi : 0.5 : 2 \pi ; \\
    &>> z = \sin(x) \cdot \sin(y) ; \\
    &>> \text{surf}(z) \\
    &>> \text{colorbar}
\end{align*}
\]

The nice thing about surf is that if you press the rotate 3-D button at the top of the figure, then you can rotate your view and look at the surface plot from different angles.

Finally we have 'imagesc', which is like surf, except you can't rotate your view. Although this seems worse, there are two good reasons why you might consider using imagesc over surf. First, imagesc usually loads much more quickly, especially for large amounts of data. Second, imagesc is best when you want to plot an actual image, e.g. in image processing.

\[
\begin{align*}
    &>> \text{load clown} \\
    &>> \text{imagesc}(X)
\end{align*}
\]
Although that clown is certainly creepy, we can make it slightly easier on the eyes by using a winter color map instead of the default.

```python
>> colormap gray
```
And before you ask, I have no idea why MathWorks decided to include a picture of a creepy clown as a default image.

7 Documentation

Finally there are lots and lots of built in functions that MATLAB has, and features that we don’t have time to cover. One of the main strengths of MATLAB is that is has literally thousands of built in functions which do everything from import images of different formats to conduct entire simulations (try typing toilet!). As a result, if you are considering doing something in MATLAB, chances are there is a function included in the standard distribution which does exactly what you wanted. From edge detection in images to wavelet transforms to symbolic calculus if you can think of something that would be of interest to researchers, chances are that MATLAB has a built in feature that does what you want. Then question is then how do you find them?

The answer to this is the answer to all questions. Google.

Seriously though, on of the most important skills any programmer can have is knowing how to search through documentation quickly.

8 Exercises and Examples

We are going to devote the rest of the session to having you all do some exercises, so that you can get some experience, and experiment a little. Try copying the
following code into a function, and giving it a try. The argument is an image, represented by a 2D matrix of doubles.

function detect_edge(x)
    horizontal_edge_filter = [[-1 0 1]; [-2 0 2]; [-1 0 1]];
    vertical_edge_filter = [[1 2 1]; [0 0 0]; [-1 -2 -1]];
    x_horiz = conv2(x,horizontal_edge_filter);
    x_vert = conv2(x,vertical_edge_filter);
    figure(1); imagesc(x); colormap gray; title('Original Image')
    figure(2); imagesc(abs(x_horiz)); colormap gray; title('Horizontal edges')
    figure(3); imagesc(abs(x_vert)); colormap gray; title('Vertical edges')
    figure(4); imagesc(sqrt(x_horiz.^2 + x_vert.^2)); colormap gray; title('Both edges')
end