Introduction to Matlab

The purpose of this intro is to show some of Matlab's basic capabilities.

Nir Gavish, 2.07

Contents

Getting help

Any intro should start with how to get help. Matlab's documentation is accesable by pressing F1 in Matlab or via the net at

`http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html`

In addition, for specific command type

```
help sin
```

```
 SIN    Sine of argument in radians.
    SIN(X) is the sine of the elements of X.

    See also ASIN, SIND.
```

```
    Overloaded functions or methods (ones with the same name in other directories)
       help sym/sin.m

    Reference page in Help browser
       doc sin
```

or for a graphical reference of the same help

```
doc  sin
```

Matlab development enviroment

Matlab includes a full development enviroment which is composed of

- *Command Window:* A line mode interface window for entering Matlab commands and seeing Matlab's output. Note, we will usually write Matlab commands in scripts (see editor) and not trough the command window.
- *Workspace:* The inventory of all the variables we are using. See

  http://www.mathworks.com/products/demos/shipping/matlab/workspace.html?product=ML

  for more details. Also serves an interface to the array editor, see

  http://www.mathworks.com/products/demos/shipping/matlab/arrayeditor.html?product=ML

  for more details.
- *Editor*: Editor for Matlab scripts (M-files) . To save & run the m-file type 'F5'. To open the editor with a new or old m-file use the command

```
edit mfileName
```

- *Current Directory:* A listing of the files in the current diretory. Doubleclick an m-file to open it in the editor.

See this movie for more features of Matlab's development enviroment

http://www.mathworks.com/products/demos/shipping/matlab/WhatsNew_1DevEnviro_viewlet_swf.

Variable definitions

Matlab variables are defined by assigment. There is no need to declare in advance the variables that we want to use or their type.

```
% Define the scalar variable x
x=1
% Now a (row) vector
y=[1 2 3]
% and a column vector
z=[1;2;3]
% Finally, we define a 3x3 matrix
A=[1 2 3;4 5 6;7 8 9]
% List of  the variables defined
whos
```

```
x =

     1


y =

     1     2     3


z =

     1
     2
     3


A =

     1     2     3
     4     5     6
     7     8     9

  Name      Size             Bytes  Class     Attributes

  A         3x3                 72  double
  ans       1x64               128  char
  x         1x1                  8  double
  y         1x3                 24  double
  z         3x1                 24  double
```

Mathematical operations

The basic mathematical operators of Matlab work with scalar, vector and matrices. Any combination works, as long as it is mathematically possible.

```
% Adding one to a scalar
result1=x+1
% Multiply a vector by a scalar
result_x_times_y=x*y
% Vector multiplication, same in syntax as any other multiplication
result_y_times_z=y*z
% Notice that vector multiplication is not commutative
result_z_times_y=z*y
% More mathematical operators
result=(x+1)/2-3*x^2
```

```
result1 =

     2


result_x_times_y =

     1     2     3


result_y_times_z =

    14


result_z_times_y =

     1     2     3
     2     4     6
     3     6     9


result =

    -2
```

Term by term operations

As noted, mathematical operators such as multiplication (*), division (/) or power (^) work between vectors or matrices. In many cases, however, we would like to preform element-wise operations between the two operands. For example, raise to the power of two *each term* of a matrix, as opposed to multiplying the matrix by itself. This is implemented by 'term by term operations' in Matlab - '.*', './', '.^'.

```
original_matrix=A
% Here we use the classical power operator ^ - which multiplies the matrix
% by itself
classical_power_operator=A^2
% Now we use the term-by-term power operator .^ (notice the dot) - which
% multiplies _each term_ of the matrix by itself
term_by_term_power_operator=A.^2
```

```
original_matrix =

     1     2     3
     4     5     6
     7     8     9


classical_power_operator =

    30    36    42
    66    81    96
   102   126   150


term_by_term_power_operator =

     1     4     9
    16    25    36
    49    64    81
```

More complicated vector definitions - the semicolon operator

Clearly, the definition of vectors by explicitly stating its terms is impractical for vectors with more than a few terms. A better approach is to use the semicolon (:) operator which defines a range of values. Notice that for long enough vector it is recommended to suppress the output to the command window by using ';' .

```
% Define vector [1 2 3 4 5]
x=1:5
% Define spacing different than one
x=1:0.125:5
% Suppress output
x=1:0.125:5;
```

```
x =

     1     2     3     4     5


x =

  Columns 1 through 9

    1.0000    1.1250    1.2500    1.3750    1.5000    1.6250    1.7500    1.8750    2.00

  Columns 10 through 18

    2.1250    2.2500    2.3750    2.5000    2.6250    2.7500    2.8750    3.0000    3.12

  Columns 19 through 27

    3.2500    3.3750    3.5000    3.6250    3.7500    3.8750    4.0000    4.1250    4.25

  Columns 28 through 33

    4.3750    4.5000    4.6250    4.7500    4.8750    5.0000
```

Vector functions and operators

Here are some Matlab operations and functions defined for vectors. Notice that many of these functions can be implemented by a simple loop in our program. It is, however, significantly faster to use Matlab's vector function than to use loops in Matlab. The technique of converting a loop in a Matlab program to vector operations is called *'vectorization'* and is fundamental in preformance improvement in Matlab.

```
% Transpose of a vector\matrix
y_transpose=y'
% Accessing a specific term of a vector (first term is indexed one, not zero)
y2=y(2)
```

```
% A partial list of vector functions
% sum(y)  = sum all the values of the vector y
res_sum =  sum(y)
 % prod(y) = multiply all the values of the vector y
res_prod =  prod(y)
% diff(y) = [y(2)-y(1),y(3)-y(2), ..., y(n)-y(n-1)]
res_diff =  diff(y)
```

```
y_transpose =

     1
     2
     3


y2 =

     2


res_sum =

     6


res_prod =

     6


res_diff =

     1     1
```

Matlab ("continous") functions

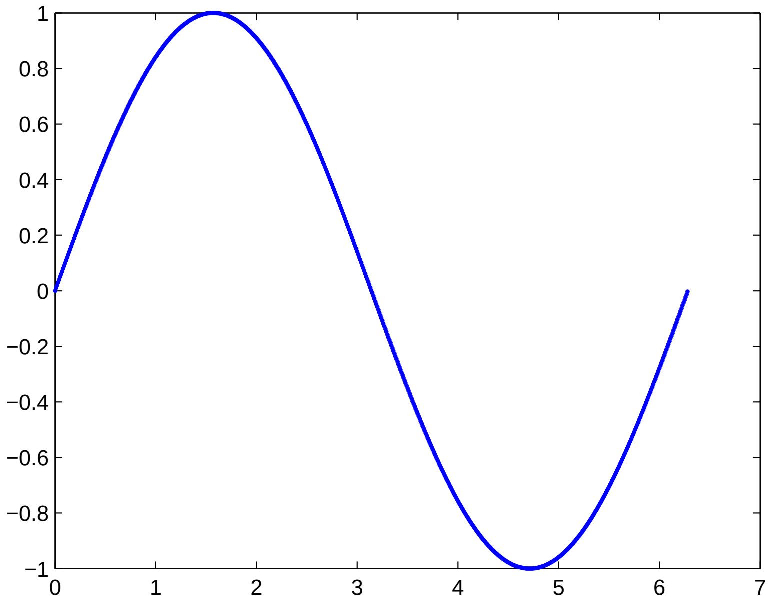
Numerically, we cannot represent a general continous function (x,f(x)) because it requires handling infinite data (for each point in the range, we need to keep f(x)). Instead, we represent a function by its values at a finite number of data points (x_i,f(x_i)), where the series of points {x_i} is typically referred to as the sampling points or the grid points. Accordingly, the "continous" functions in Matlab accepts a vector of point {x_i} and return a vector of values {f(x_i)}. We note that opposed to the numerical approach is the symbolic approach, which is the approach you know from all the basic math classes.

```
% define the grid {1,1.1,1.2...4.9,5} using the semicolon operator
x=1:0.1:5;
% f(x) = x^2/(4+x), notice the use of *term-by-term operators*
f1  =  x.^2./(4+x);
% sqrt(x) = x^(1/2)
f2 =  sqrt(x + x.^3);
% Note: MATLAB doesn't define the constant 'e'.  Use exp(1) to get e.
f3  =  exp(x);
% Note: in MATLAB log() means ln() (i.e., log in base e).
f4  =  log(x+4);
% 'pi' is a matlab constant.  Note: sin, cos , etc. are in radians (NOT in degrees!)
f5  =  cos(pi)*tan(x);
% abs(x) := |x|
f6  =  abs(f5);
% sign(x) gives -1 for x<0, 0 for x=0, and +1 for x>0
f7  =  sign(f5);
```

Plotting graphs

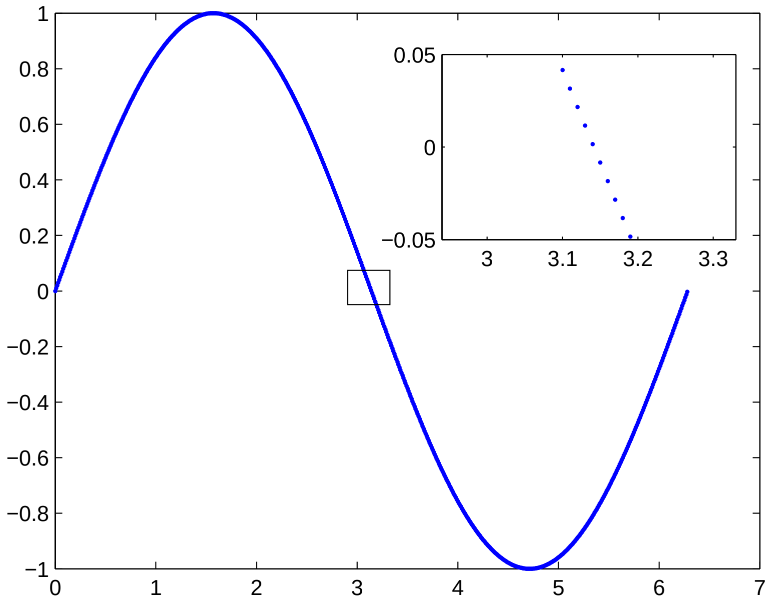
Matlab is well-known for its plotting capabilites and for their simplicity of use. We now go over the most basic plot command and its features - plot(x,y) which plots the data points {x_i,f(x_i)}

```
% Define the grid and the "continous" function sin
x=0:0.01:2*pi;
y=sin(x);
% Plot the points (x,sin(x))
plot(x,y,'.');
```



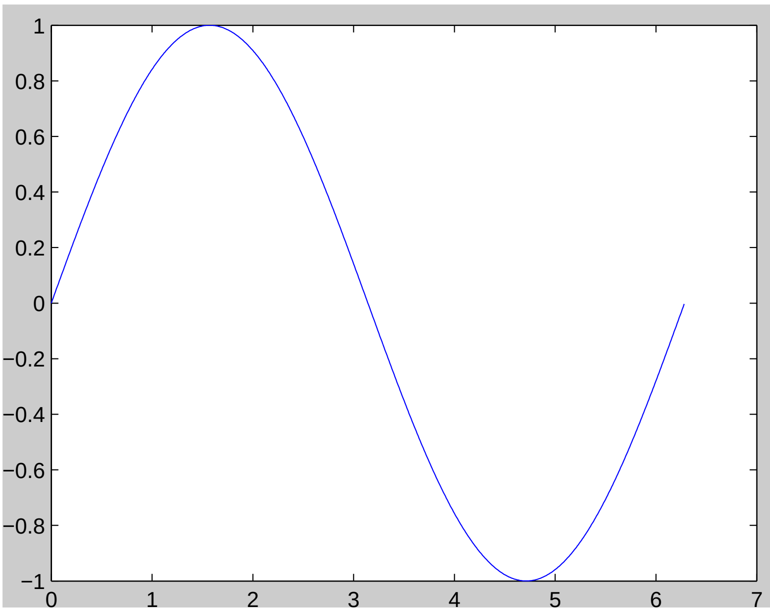
As you see, the plot command displays a graph of (x,sin(x)) which looks continous. However, a closer look at the data shows it is not really continous. *You can ignore the commands used to demonstrate it.*

```
plot(x,y,'.');
annotation('rectangle',[0.4518 0.4976 0.04643 0.05]);
axes('Position',[0.5554 0.5929 0.3232 0.2714]);
plot(x,y,'.');axis([2.94 3.33 -0.05 0.05])
```



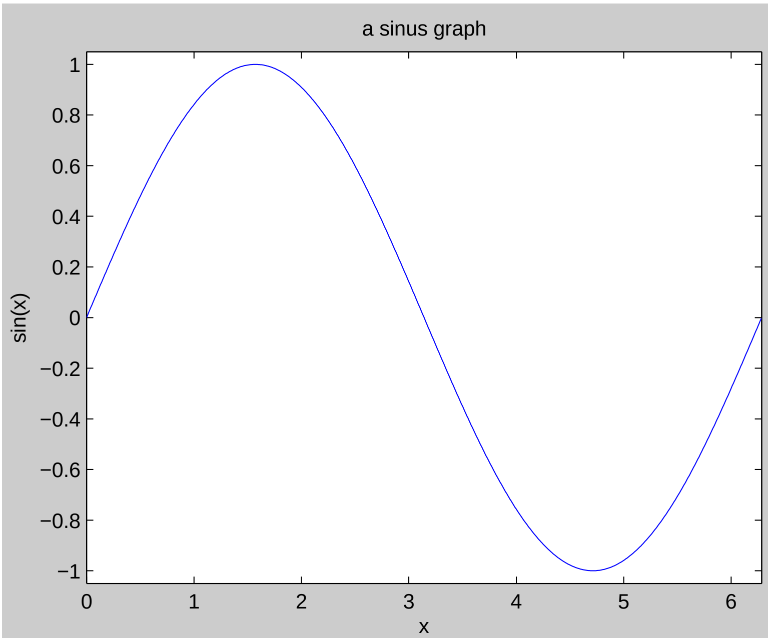
The default command connects every two points, making the graph look continous.

```
% Close the last graph to clean up the prior setting of double axes
close all;
% Plot using the default option
plot(x,y);
```



The last graph is ugly and not very informative. Let us now improve its look

```
plot(x,y);
% Set the axis boundaries. Note: The data should not touch the axis,
% therefore the y axis is set to be -1.05 to 1.05.
axis([0 2*pi -1.05 1.05])
% Add a label for the x-axis
xlabel('x');
% Add a label for the y-axis
ylabel('sin(x)');
% Add a title for the y-axis
title('a sinus graph')
```



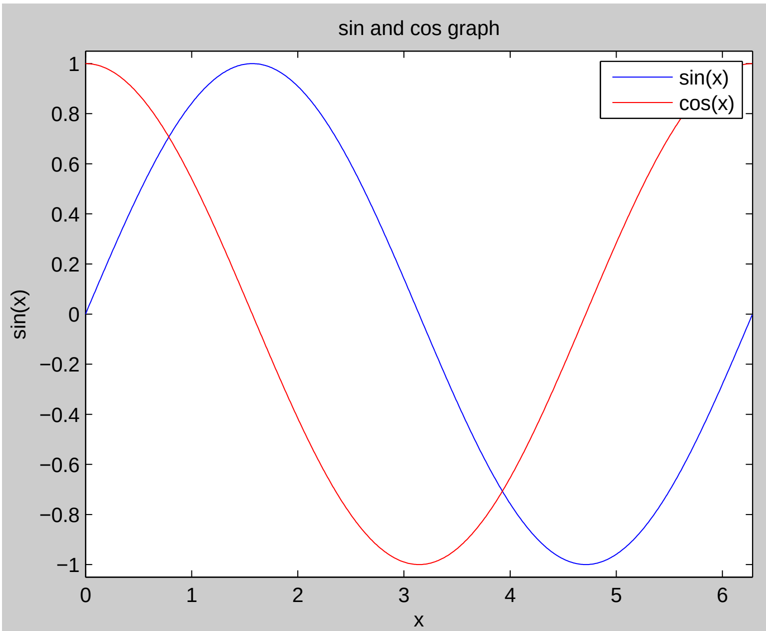
Plotting multiple graphs together

Often we need to plot more than one function, for example - to compare the output of two processes or get an easy look at various measures at once. There are two ways to plot multiple functions

- Adding graphs to an existing axes, so that in the same axes you get multiple plots. This is done by the *'hold'* command.
- Adding a new axes are plotting the graph there. This is done by the *'subplot'* command.

```
% add the cos graph to the exisiting plot
% tell Matlab to hold the graph for the next plot
hold
%  plot the additional graph, the additional parameter 'r' is for 'red'
plot(x,cos(x),'r');
% add a legend for the two plots
legend('sin(x)','cos(x)')
% correct the title
title('sin and cos graph')
```

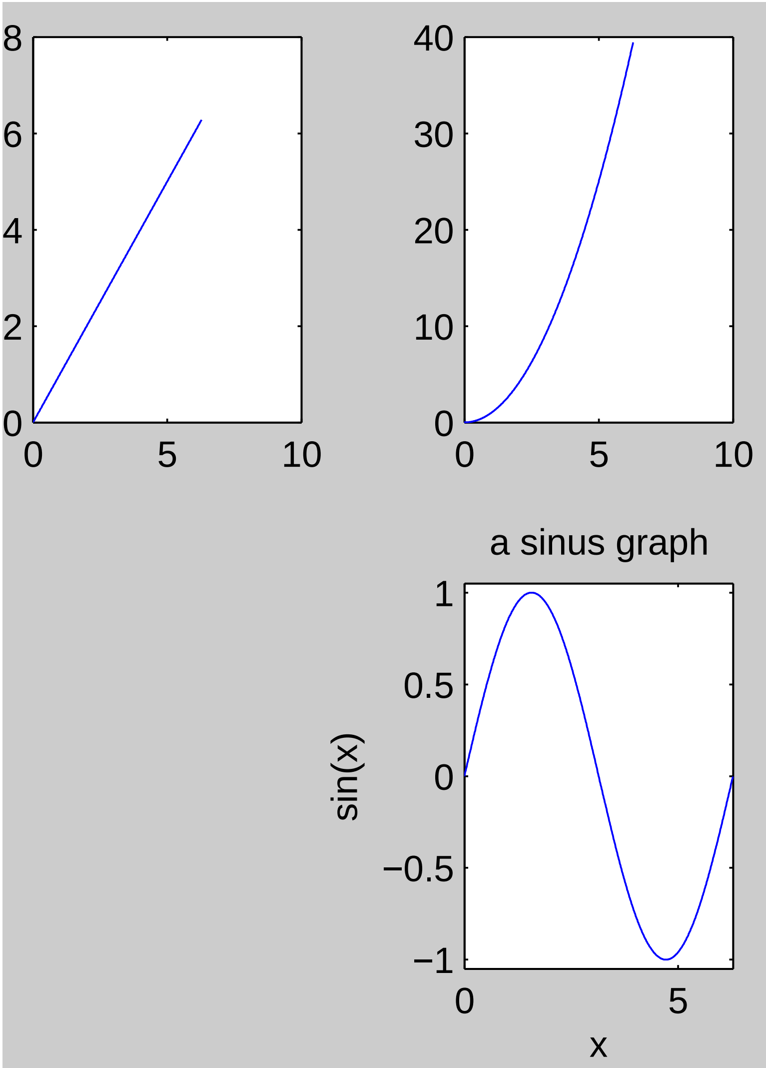
```
Current plot held
```



We now split the page into several axes by the subplot command Its syntax is subplot(row_num,col_num,curr_plot)

```
% split the screen into 2 rows and 3 columns of axes, set the next plot to be at
% the first axes
subplot(2,3,1);
% plot a graph of x
plot(x,x);
% Now go to the second axes
subplot(2,3,2);
% plot a graph of x^2
plot(x,x.^2);
% Next, plot a graph of sin in the fifth axes.  Notice, all editing of the
```

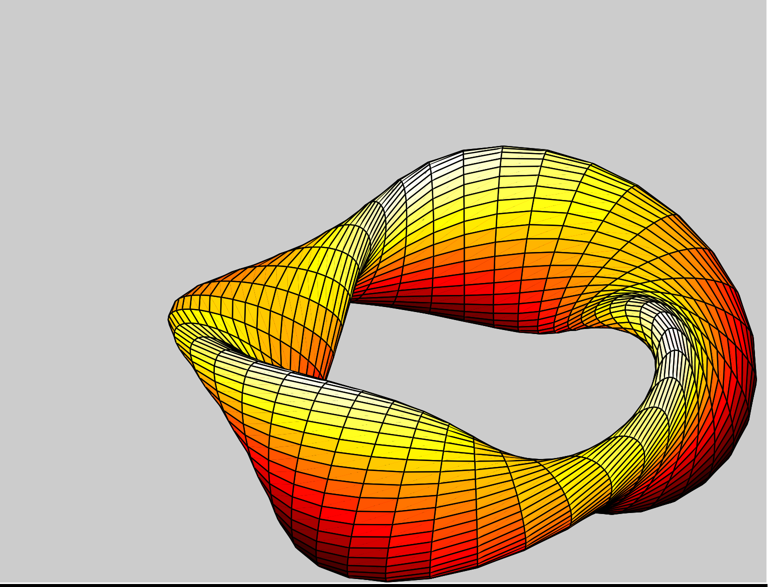
```
% graph applies only to the current axes
subplot(2,3,5);
% plot a graph of sin(x)
plot(x,y);
% set axis
axis([0 2*pi -1.05 1.05])
% Add a label for the x-axis
xlabel('x');
% Add a label for the y-axis
ylabel('sin(x)');
% Add a title
title('a sinus graph')
```

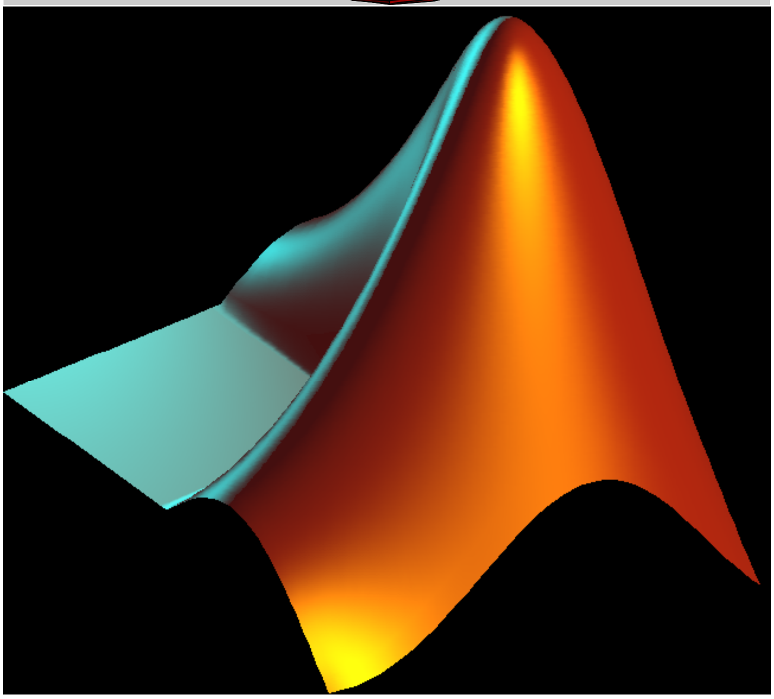


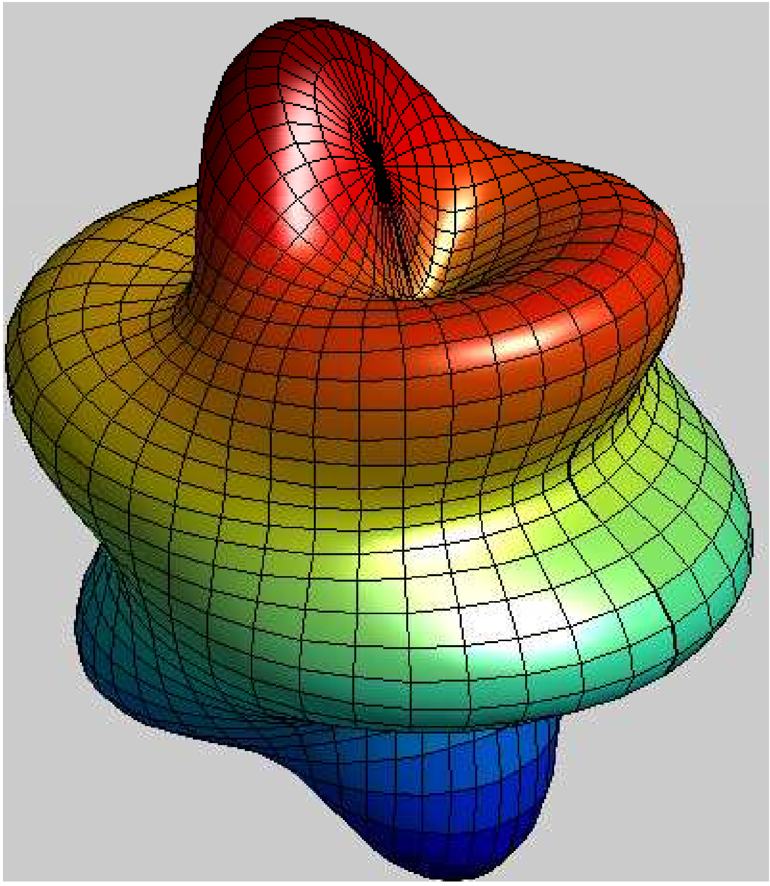
Examples of more sophisticated graphics

The number of line codes needed to produces these graphs is no more than 10-20 lines.

```
culler;
logo;figure;
spharm2;
```







Flow control

Here I just give examples for the most basic flow control commands. For more info see

`http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/`

Conditional control

```
a=4;
if a==5
    a=a+1
else
    a=3
end
```

```
a =

     3
```

Loop Control

```
for ix=1:3
    a=a+ix
end
```

```
a =

     4


a =

     6


a =

     9
```

Notice that I name the enumerator index 'ix' and not 'i' or 'j'. This is because 'i' and 'j' are the complex imaginary numbers, e.g.,

```
i_square=i^2
```

```
i_square =

    -1
```

Saving results

We can save all our results for future reference. Here we discuss three different objects:

- *Command window output:*

In this case, we can only save future output to the command window. The command

```
diary 'MyCommandWindow'
```

saves all output to command window into the .txt file 'MyCommandWindow' until this option is turned off by the command

```
diary off
```

- *Variables:*

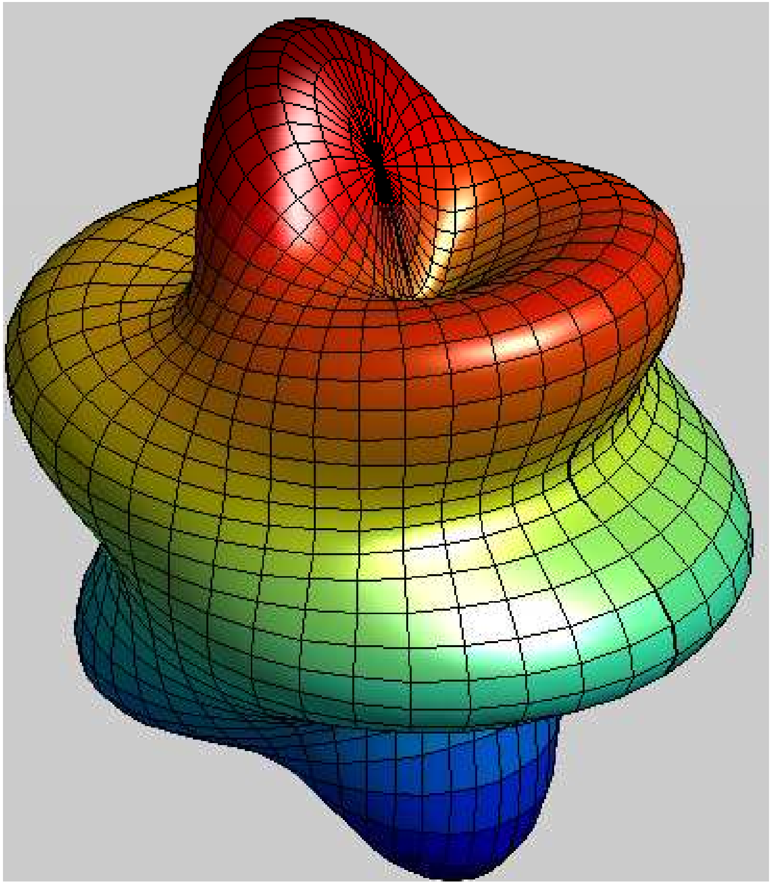
The following commands save & load the entire workspace into the .mat file 'MyMatFile'

```
save 'MyMatFile'
load 'MyMatFile'
```

- Graphs

The following commands save the current figure

```
% save as jpeg (not optimal for graphs, good compression)
print -djpeg 'myPic.jpeg'
% save as tiff (much better for graphs, more space)
print -dtiff 'myPic.tif'
```



Cleaning up

Since memory is not erased at the begging and end of a script, it is a good habit to clean up before and after the script run.

```
% Close all plot windows
close all;
% Erase the command window (You can still see the last command in the 'command history' w
clc;
% Clear all variables from the workspace.
clear all;
```

More references on the web

Tutorials on Matlab's site:

http://www.mathworks.com/academia/student_center/tutorials

Movie tutorial on Matlab's site

`http://www.mathworks.com/support/product/demos_index_by_product.html?product=ML`

Cleve's Moler (free online) book

`http://www.mathworks.com/moler/chapters.html`

Exercise 1

- Create a vector of the even whole numbers between 31 and 75.
- Create a vector x with the elements,

$$x_n = \frac{(-1)^{n+1}}{2n-1}, \quad n = 1 : 100$$

- Approximate the value of pi using the identity

$$\frac{\pi^2 - 8}{16} = \sum_{n=1}^{\infty} \frac{1}{(2n-1)^2(2n+1)^2}$$

Do this by taking the sum of only 100 terms using Matlab's *'sum'* command. Use the Matlab constant *'pi'* to calcuate the accuracy of your calculation.

- Let A=[8 1 6;3 5 7;4 9 2]. Calcuate the sum over the rows of A by multiplication of A by a appropriate vector.
- Let A=[8 1 6;3 5 7;4 9 2]. Calcuate the sum over the rows of A by using the command *'sum'*.

Exercise 2

In this exercise you will write a program that calculates the factorials 1!, 2!, 3! ... 15! in three different ways

- Look for help on the command *'factorial'* and write a program that calculates 1!, 2!, 3! ... 15!
- Write a program that calculates 1!, 2!, 3! ... 15! without using *'factorial'*. Make the program efficient by calculating

$$(n+1)! = (n+1)n!$$

- Vectorize the code by using the command *'cumprod'* (look it up...).

Exercise 3

In this exercise you produce a graphical example of the accuary of a Taylor series

- Plot sin(x) in the domain [0,pi]
- Add a plot of the first term of its Taylor series (i.e, sinx=x+...). To distinguish between the graphs, use the line specification 'r–' for the Taylor series.
- Add a plot of the Taylor series with two terms (i.e, sin=x-xˆ3/6+...) and three terms. Use the line specification 'g:' and 'k.-', respectively.
- Make the graph readable by adding axis labels, adding a legend and setting the axis.

Exercise 4

The Legendre polynomials (*Pn(x)*) are defined by the following recurrance relation

$$(n+1)P_{n+1}(x) - (2n+1)xP_n(x) + nP_{n-1}(x) = 0$$

with

$$P_1(x) = x, \quad P_0(x) = 1$$

- Compute the next three Legendre polynomials by implementing the recursive relation in Matlab and using vectorial operations only for x=-1:0.01:1.
- Plot all 5 over the interval [-1,1] (use x=-1:0.01:1).
- Make sure the graph is well presented

```
% That's it, let's clean up
close all;clc;clear all;
```