Introduction to MATLAB®
for the Physics Lab

Table of Contents

We will go over some of the basic functions and the methods we looked during the class. If you have questions please e-mail me ahaje092@uottawa.ca.

Help/Docs

We can use **help** or **doc** command to explore an unknown fucntion or just to find out all the arguments and check for examples. **doc** command will open up a new window, where as **help** will show inline results.

```
help sin
```

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

   See also ASIN, SIND.

   Overloaded methods:
```

```
sym/sin
codistributed/sin
gpuArray/sin

Reference page in Help browser
   doc sin
```

Variables

```
%To create variables simply assign a value to a name.
```

```
var1 = 5.3
```

```
var1 =

    5.3000
```

A variable can be given a value explicitly

```
a = 10
```

```
a =

    10
```

Or as a function of explicit values and existing variables

```
c = 1.3*45-2*a
```

```
c =

   38.5000
```

To suppress output, end the line with a semicolon

```
varSuppressed = 13/3;
```

When solving a large problem try to use meaningful names for the variables. For example, instead of just using *a* and *b*, use *forceTotal* to describe the total force, and *unitRotOp* to represent an an unitary operator matrix. Note how I have using capitalization for better readability.

Vectors

The strength of MATLAB is in the matrix operation. Using matrix and vector allows us to do complicated calculations on large set of data usinga single line of command. We will see some examples later. Avoid using for..loops. These are terribly slow.

To create a row vector use

```
row = [1 2 5.4 -6.6] %or
row = [1, 2, 5.4, -6.6];
```

```
        row =

            1.0000    2.0000    5.4000   -6.6000
```

To create column vector use

```
column = [4;2;7;4]
```

```
        column =

             4
             2
             7
             4
```

You can tell the difference between a row and a column vector by:

Å Looking in the workspace

Å Displaying the variable in the command window

Å Using the **size** function

```
size(row)
```

```
        ans =

             1     4
```

```
size(column)
```

```
        ans =

             4     1
```

To get vectors length use the **length** function

```
length(row)
```

```
        ans =

             4
```

```
length(column)
```

```
ans =

     4
```

Matrices

Make matrices by merging the commands of row and column vectors

```
a = [1 2;3 4]

        a =

             1     2
             3     4
```

or by concatenating vectors or matrices (NOTE: the output differecnes below)

```
a = [1 2]

        a =

             1     2

b = [3 4]

        b =

             3     4

c = [5;6]

        c =

             5
             6

d = [a;b]

        d =

             1     2
             3     4

e = [d c]
```

```
e =

    1     2     5
    3     4     6
```

```
f = [[e e];[a b a]]
```

```
f =

    1     2     5     1     2     5
    3     4     6     3     4     6
    1     2     3     4     1     2
```

You can create a vector of strings as well. Strings are character vectors

```
str = ['Hello, I am ' 'John'];
```

save/clear/load

Use **save** to save variables to a file

```
save myFile a b
% saves variables a and b to the file myfile.mat
```

myfile.mat file is saved in the current directory Default working directory is \MATLAB Make sure youȓe in the desired folder when saving files.

Use **clear** to remove variables from environment

```
clear a b
% look at workspace, the variables a and b are gone
```

Use **load** to load variable bindings into the environment

```
load myFile
% look at workspace, the variables a and b are back
```

Can do the same for entire environment

```
save myenv; clear all; load myenv;
```

Scalar Operations

Arithmetic operations (+,-,*,/)

```
7/45
```

```
ans =

    0.1556
```

```
(1+i)*(2+i)

        ans =

           1.0000 + 3.0000i

1 / 0

        ans =

           Inf

0 / 0

        ans =

           NaN
```

Exponentiation (^)

```
4^2

        ans =

            16

(3+4*j)^2

        ans =

          -7.0000 +24.0000i

((2+3)*3)^0.1

        ans =

            1.3110
```

3(1+0.7) gives an error. multiplication has to explicitely stated

To clear command window

```
clc
```

To clear all variables

```
clear all
```

Built-in functions

MATLAB has an enourmous library functions. It is really really big and quite comprehensive. Covers functions from basic algebra to algebraic number theory, from numeric calculus to system dynamics. Ofcourse there are also a lot of free user defined functions people have created which can be downloaded.

```
sqrt(2)
```

```
ans =

    1.4142
```

```
log(2)
```

```
ans =

    0.6931
```

```
log10(0.23)
```

```
ans =

   -0.6383
```

```
cos(1.2)
```

```
ans =

    0.3624
```

```
atan(-.8)
```

```
ans =

   -0.6747
```

```
exp(2+4*i)
```

```
ans =

  -4.8298 - 5.5921i
```

```
round(1.4)
```

```
ans =

     1
```

```
floor(3.3)
```

```
ans =

     3
```

```
ceil(4.23)
```

```
ans =

     5
```

```
angle(i) % note that angles are in radian by default
abs(1+i)
```

```
ans =

    1.5708


ans =

    1.4142
```

```
besselj(1, 5)
```

```
ans =

   -0.3276
```

Transpose

The transpose operators turns a column vector into a row vector and vice versa

```
a = [1 2 3 4+i];
transpose(a)
```

```
ans =
```

```
    1.0000 + 0.0000i
    2.0000 + 0.0000i
    3.0000 + 0.0000i
    4.0000 + 1.0000i
```

```
a'
```

```
ans =

   1.0000 + 0.0000i
   2.0000 + 0.0000i
   3.0000 + 0.0000i
   4.0000 - 1.0000i
```

.' gives the Hermitian-transpose, i.e. transposes and conjugates all complex numbers

```
a.'
```

```
ans =

   1.0000 + 0.0000i
   2.0000 + 0.0000i
   3.0000 + 0.0000i
   4.0000 + 1.0000i
```

Addition and Subtraction

Addition and subtraction are element-wise; sizes must match (unless one is a scalar):

```
row = [1 2 5.4 -6.6];
column = [4;2;7;4];
% use the transpose to make size compatatible
c = row' + column
c = row + column'
%  Can sum up or multiply elements of vector
s = sum(row)
p = prod(row)
```

```
c =

    5.0000
    4.0000
   12.4000
   -2.6000


c =

    5.0000    4.0000   12.4000   -2.6000
```

```
s =

    1.8000


p =

  -71.2800
```

Element-wise Functions

All the functions that work on scalars also work on vectors. This is the most important characteristics of MATLAB. This allows us to think large matrix as a scalar and do not need loop through each element to apply a function

```
t = [1 2 3];
f = exp(t); %is the same as
f = [exp(1) exp(2) exp(3)];
```

Operators (* / ^) have two modes of operation: element-wise and standard To do element-wise operations, use the dot before the operation: .*, ./, .^ ***BOTH*** dimensions must match (unless one is scalar)!

```
a = [1 2 3];b = [4;2;1];
a.*b'
a./b'
a.^(b')
```

```
ans =

     4     4     3


ans =

    0.2500    1.0000    3.0000


ans =

     1     4     3
```

Multiplication can be done in a standard way or element-wise Standard multiplication **(*)** is either a dot - product or an outer-product. Remeber from the linear algebra that the inner dimensions must match !!! Left and right division (/ \) is same as multiplying by inverse

Functions for automatic initialization

```
o = ones(1,10) %row vector with 10 elements, all 1
```

```
o =

     1     1     1     1     1     1     1     1     1     1
```

```
z = zeros(23,1) % column vector with 23 elements, all 0
```

```
z =

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

```
r = rand(1,45) % row vector with 45 elements (uniform [0,1])
```

```
r =

  Columns 1 through 7

    0.5938    0.2827    0.1552    0.0007    0.2836    0.5508    0.8709

  Columns 8 through 14

    0.0423    0.9047    0.1310    0.8337    0.8005    0.9179    0.1373

  Columns 15 through 21

    0.5047    0.4050    0.1736    0.5752    0.6062    0.2144    0.5199

  Columns 22 through 28
```

```
        0.9892    0.4899    0.6949    0.4114    0.0348    0.2928    0.8014

      Columns 29 through 35

        0.3465    0.0833    0.5111    0.3668    0.7395    0.5247    0.8045

      Columns 36 through 42

        0.8169    0.1895    0.1237    0.8210    0.6379    0.0161    0.8960

      Columns 43 through 45

        0.5154    0.5445    0.6064
```

```
n = nan(1,69)
% row vector of NaNs (useful for representing uninitialized variables)
```

```
    n =

      Columns 1 through 13

       NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN

      Columns 14 through 26

       NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN

      Columns 27 through 39

       NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN

      Columns 40 through 52

       NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN

      Columns 53 through 65

       NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN

      Columns 66 through 69

       NaN   NaN   NaN   NaN
```

To initialize a linear vector of values use linspace

```
a = linspace(0,10,5)  % starts at 0, ends at 10 (inclusive), 5 values
```

```
    a =

             0    2.5000    5.0000    7.5000   10.0000
```

Can also use colon operator (:)

```
b = 0:2:10 % starts at 0, increments by 2, and ends at or before 10
% increment can be decimal or negative
c = 1:5 % if increment isn’t specified, default is 1
```

```
b =

     0     2     4     6     8    10


c =

     1     2     3     4     5
```

To initialize logarithmically spaced values use **logspace**, see help

Vector and Matrix Indexing

MATLAB indexing starts with 1, not 0 a(n) returns the nth element

```
a = [13 5 9 10];
a(1)
a(4)
```

```
ans =

    13


ans =

    10
```

The index argument can be a vector.

```
x = [12 13 5 8 9 10];
a = x(2:3) % is same as a = [13 5]
a = x(2:5:2) % is same as a = [13 8]
b = x(1:end-1)
```

```
a =

    13     5


a =
```

```
    13


b =

    12    13     5     8     9
```

for Matrix you will have to indiciate both row and column numeber

```
A = rand(5)
A(1:3,1:2) % specify contiguous submatrix
A([1 5 3], [1 4]) % specify rows and columns
```

```
A =

    0.7604    0.3320    0.1295    0.4372    0.1568
    0.8553    0.8397    0.8799    0.3798    0.3260
    0.3829    0.3717    0.0441    0.9797    0.3141
    0.0846    0.8282    0.6867    0.3990    0.8945
    0.7339    0.1765    0.7338    0.4402    0.2470


ans =

    0.7604    0.3320
    0.8553    0.8397
    0.3829    0.3717


ans =

    0.7604    0.4372
    0.7339    0.4402
    0.3829    0.9797
```

To select rows or columns of a matrix, use the :

```
c = [12 2; -2 13];
d = c(1,:)
e = c(:,2)
c(2,:) = [3 6]; %replaces second row of c
```

```
d =

    12     2


e =

     2
```

```
        13
```

To get the minimum value and its index:

```
vec = [5 3 1 9 7];
[minVal,minInd] = min(vec)

% *max* works the same way
% To find any the indices of specific values or ranges
ind = find(vec ==  9);
ind = find(vec > 2 & vec < 6);
```

```
        minVal =

             1


        minInd =

             3
```

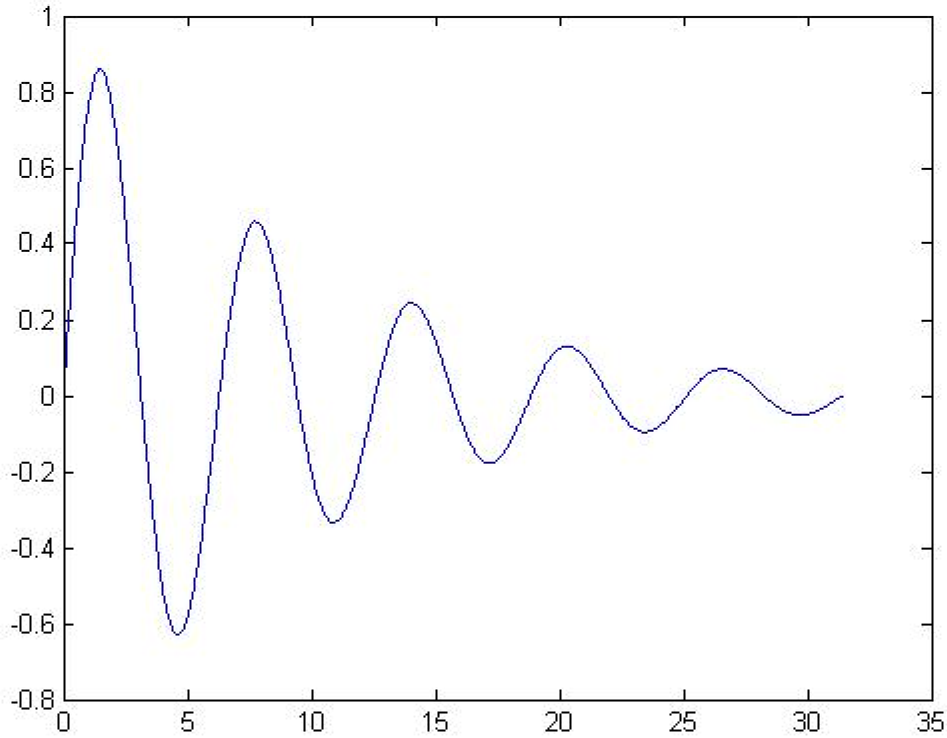
The Colon (:) operator

This is a versatile operator. See the following table

Format	Purpose
A(:,j)	is the jth column of A.
A(i,:)	is the ith row of A.
A(:,:)	is the equivalent two-dimensional array. For matrices this is the same as A.
A(j:k)	is A(j), A(j+1),...,A(k).
A(:,j:k)	is A(:,j), A(:,j+1),...,A(:,k).
A(:,:,k)	is the k^{th} page of three-dimensional array A.
A(i,j,k,:)	is a vector in four-dimensional array A. The vector includes A(i,j,k,1), A(i,j,k,2), A(i,j,k,3), and so on.
A(:)	is all the elements of A, regarded as a single column. On the left side of an assignment statement, A(:) fills A, preserving its shape from before. In this case, the right side must contain the same number of elements as A.

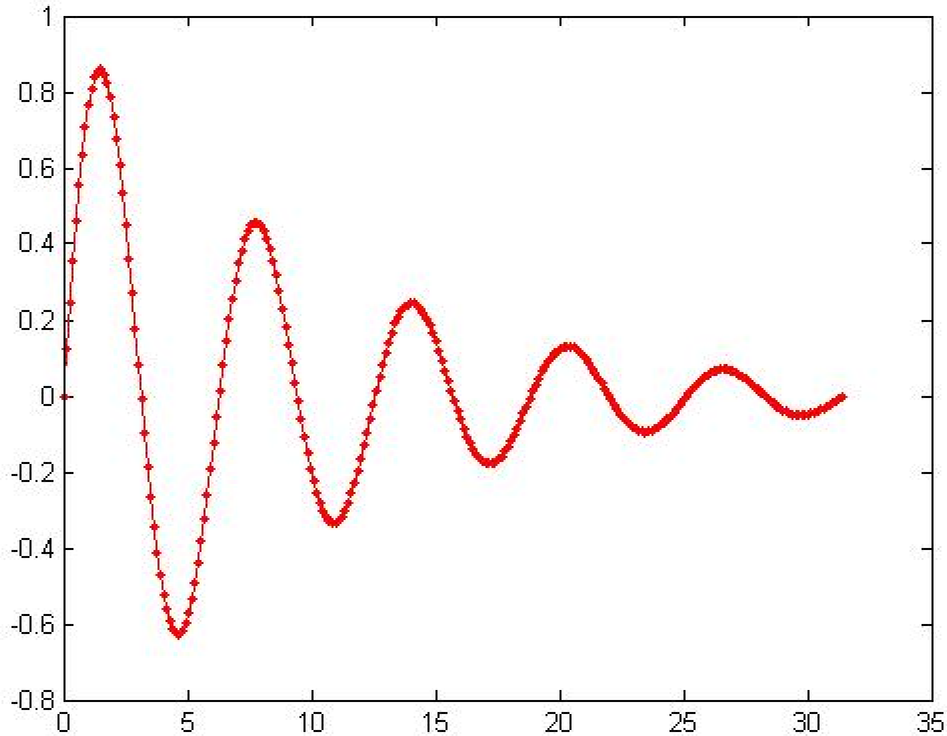
Basic Plotting

```
x = linspace(0,10*pi,1000);
y = exp(- 0.1*x).* sin(x);
plot(x, y)
```



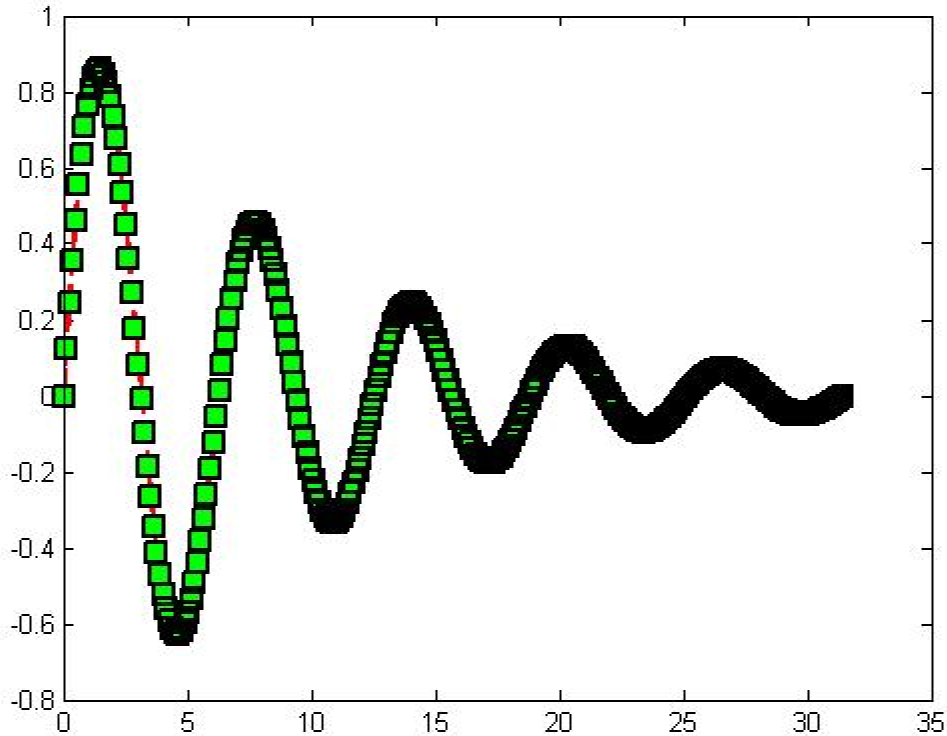
Can change the line color, marker style, and line style by adding a string argument

```
x = linspace(0,10*pi,250);
y = exp(- 0.1*x).* sin(x);
plot(x,y,'r.-')
```



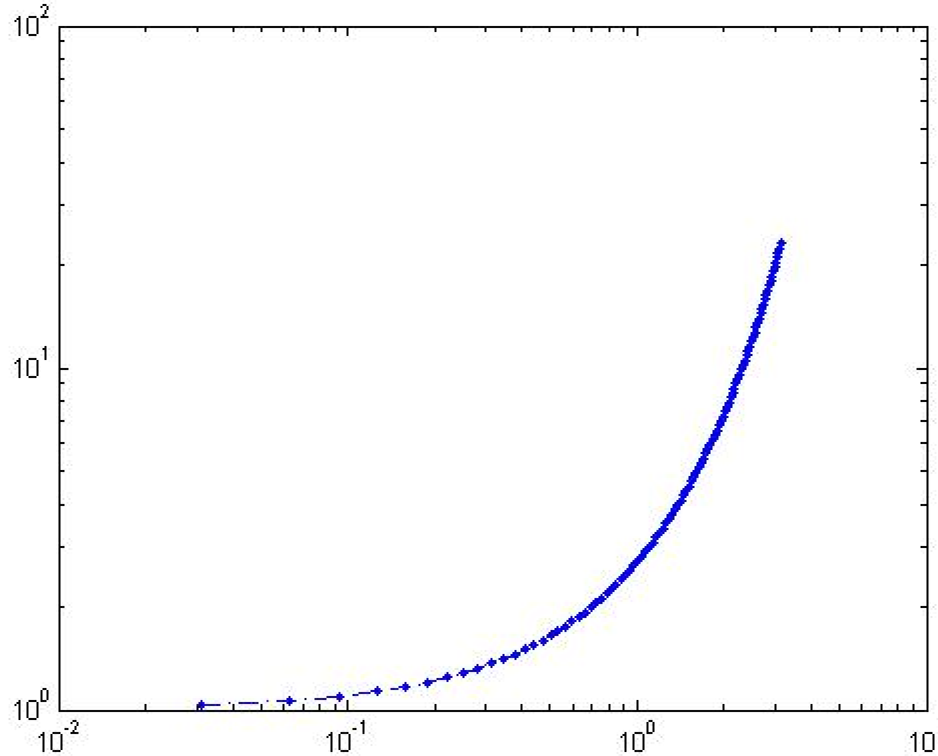
Everything on a line can be customized . Here we will use a vector of [R G B] values to define color

```
plot(x,y,'--s','LineWidth',2, 'Color', [1 0 0], 'MarkerEdgeColor','k', 'MarkerFace
```



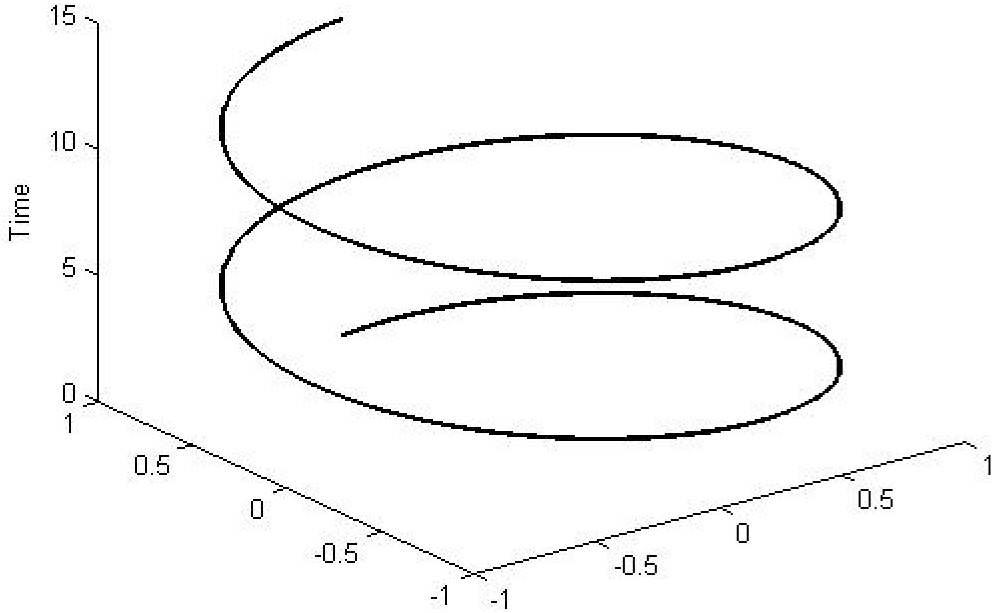
The same syntax applies for semilog and loglog plots

```
x = -pi:pi/100:pi;
y = cos(4*x).*sin(10*x).*exp(-abs(x));
semilogx(x,y,'k-')
semilogy(x, y,'r.-')
loglog(x, exp(x), 'b.-.')
```



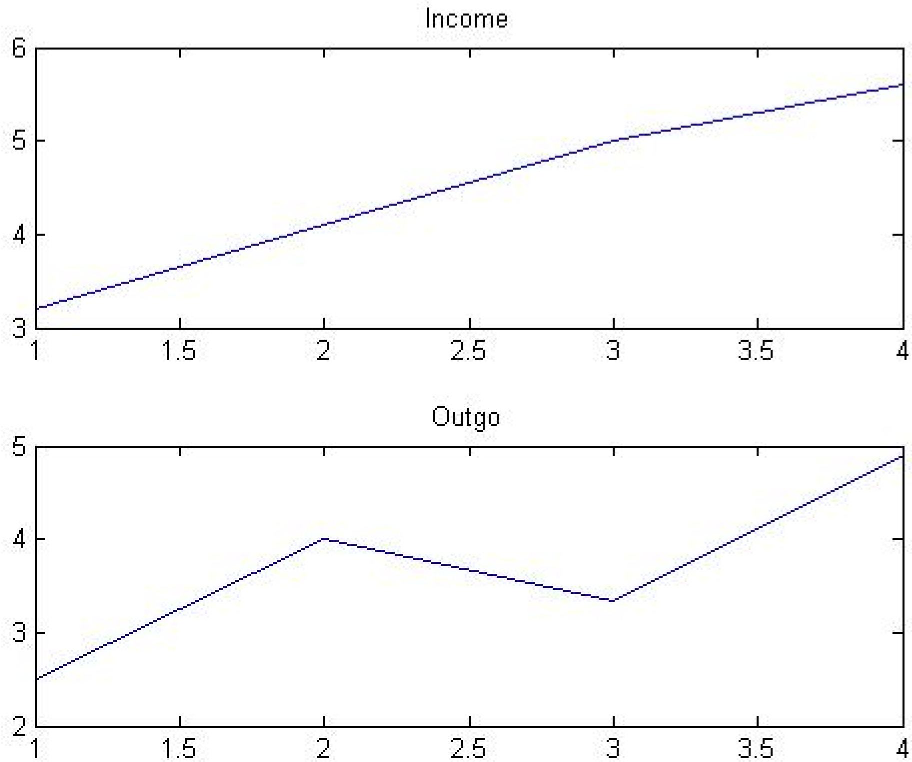
We can plot in 3 dimensions just as easily as in 2

```
time = 0:0.001:4*pi;
x = sin(time);
y = cos(time);
z = time;
plot3(x,y,z,'k','LineWidth',2);
zlabel('Time');

% Can set limits on all 3 axes
% xlim, ylim, zlim
```



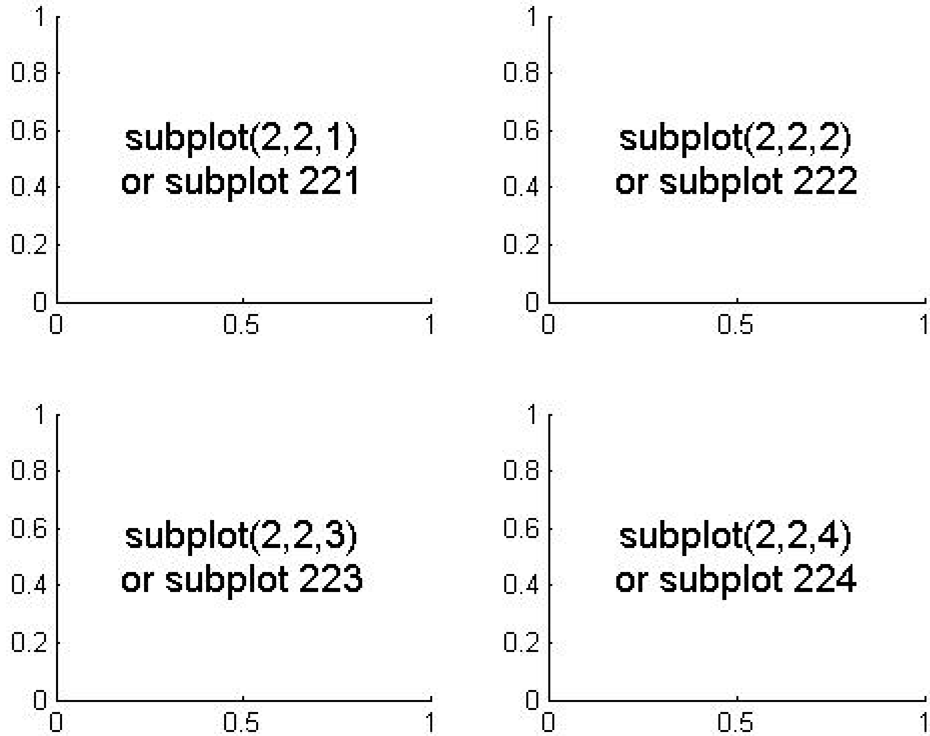
Multiple Plots in one Figure

```
income = [3.2,4.1,5.0,5.6];
outgo = [2.5,4.0,3.35,4.9];
subplot(2,1,1); plot(income)
title('Income')
subplot(2,1,2); plot(outgo)
title('Outgo')
```



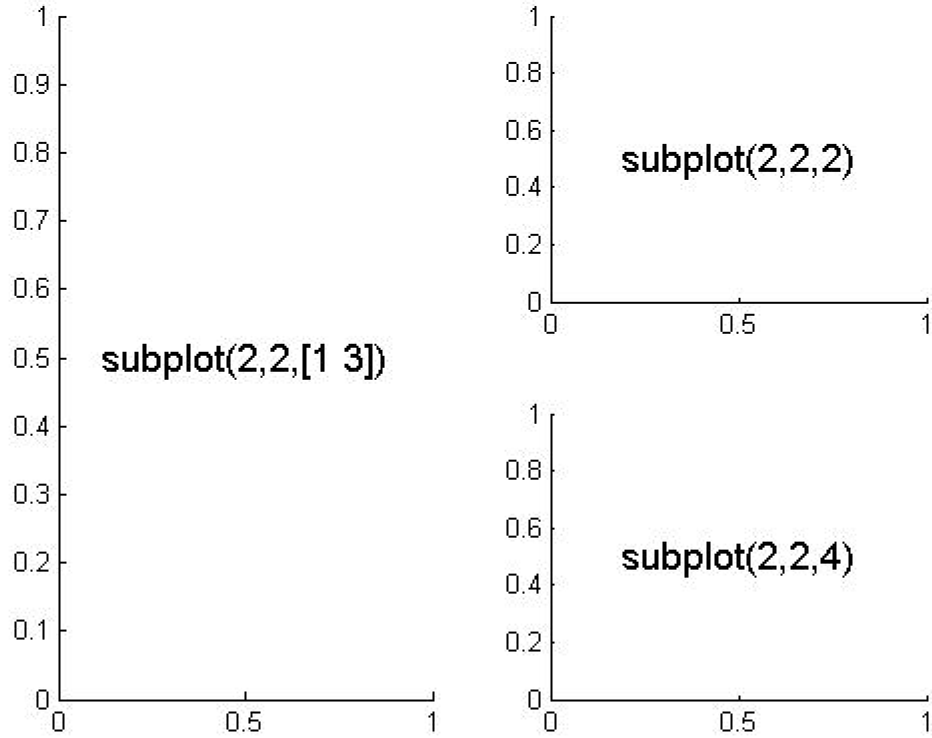
Subplots in Quadrants The following illustration shows four subplot regions and indicates the command used to create each.

```
figure
subplot(2,2,1)
text(.5,.5,{'subplot(2,2,1)';'or subplot 221'},...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,2)
text(.5,.5,{'subplot(2,2,2)';'or subplot 222'},...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,3)
text(.5,.5,{'subplot(2,2,3)';'or subplot 223'},...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,4)
text(.5,.5,{'subplot(2,2,4)';'or subplot 224'},...
    'FontSize',14,'HorizontalAlignment','center')
```



Asymmetrical Subplots The following combinations produce asymmetrical arrangements of subplots.

```
figure
subplot(2,2,[1 3])
text(.5,.5,'subplot(2,2,[1 3])',...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,2)
text(.5,.5,'subplot(2,2,2)',...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,4)
text(.5,.5,'subplot(2,2,4)',...
    'FontSize',14,'HorizontalAlignment','center')
```



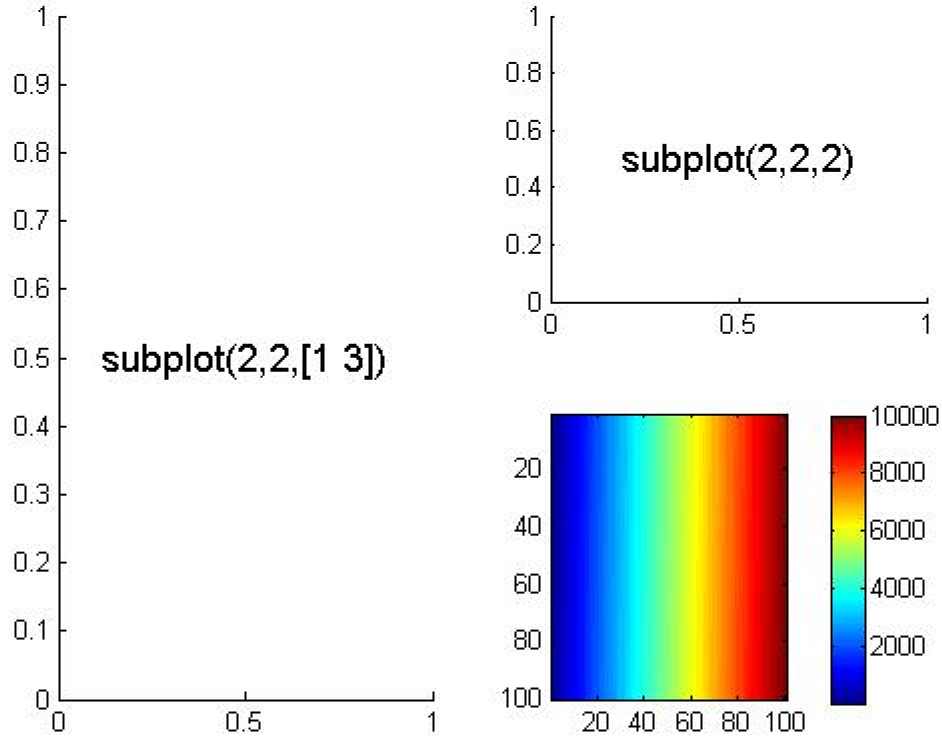
Saving Figures, inserting Legends and titles

Figures can be saved in many formats. The common ones are: * **.fig** preserves all information and is matlab * **.JPEG** compressed image. use this if you want to insert to a MS Office document * **.eps** encapsulated post script: high-quality scaleable format. * **.pdf** pdf format can be used in latex

Check the MATLAB plot documentation on how to insert legend and axis and plot titles. You can get more details about legend by typing ***doc legend***

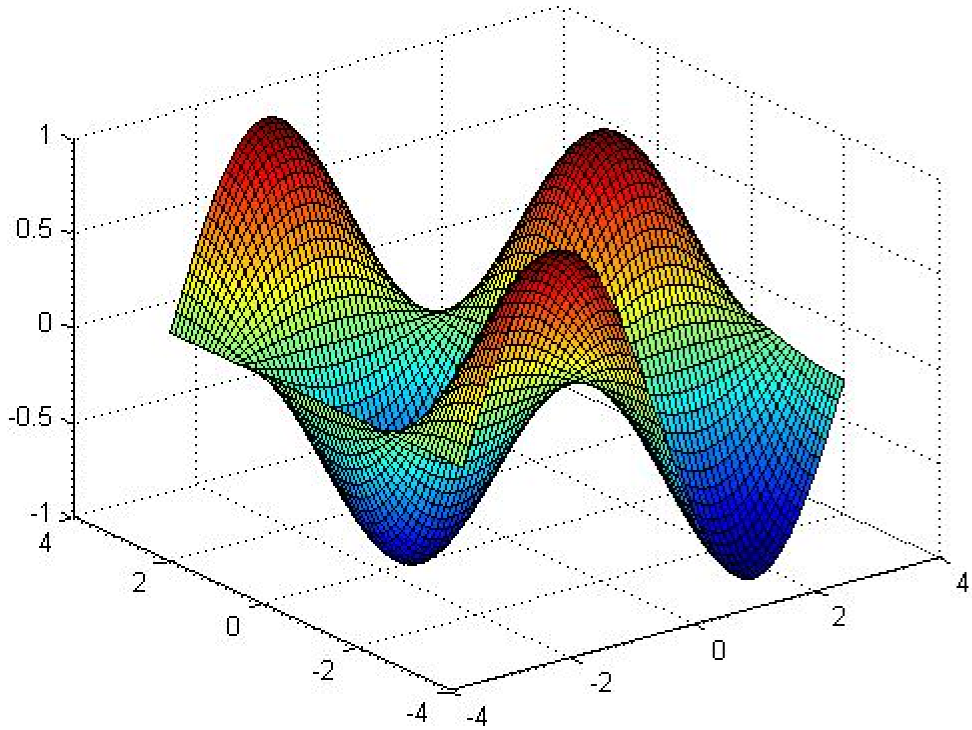
Visulaizing matrices

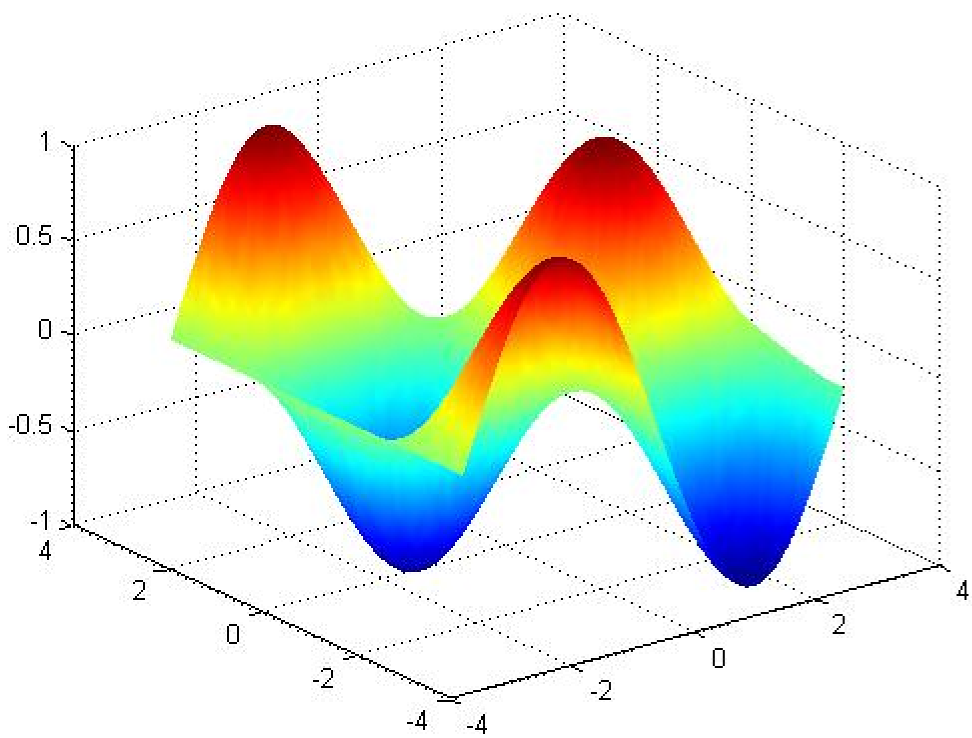
```
mat = reshape(1:10000,100,100);
imagesc(mat); % automatically scales the values to span the entire colormap
colorbar % adds the colorbar legend.
% note how the plot is made as a subplot and in the subplot(2,2,4)
% for a new plot you will use the fiugre command to open an empty plot
% space
```



Surface Plot

```
figure()
x = -pi:0.1:pi; % make x and y vectors
y = -pi:0.1:pi;
[X,Y] = meshgrid(x,y); %(meshgrid takes in two vectors and return two matrix
% with x and y points;
Z  = sin(X).*cos(Y); % calculate the value of the funciton
surf(X,Y,Z) % the surface plot
figure() % new figure window
surf(X,Y,Z)
shading interp % using this command makes a smoother plot by interpolating
% between points.
```

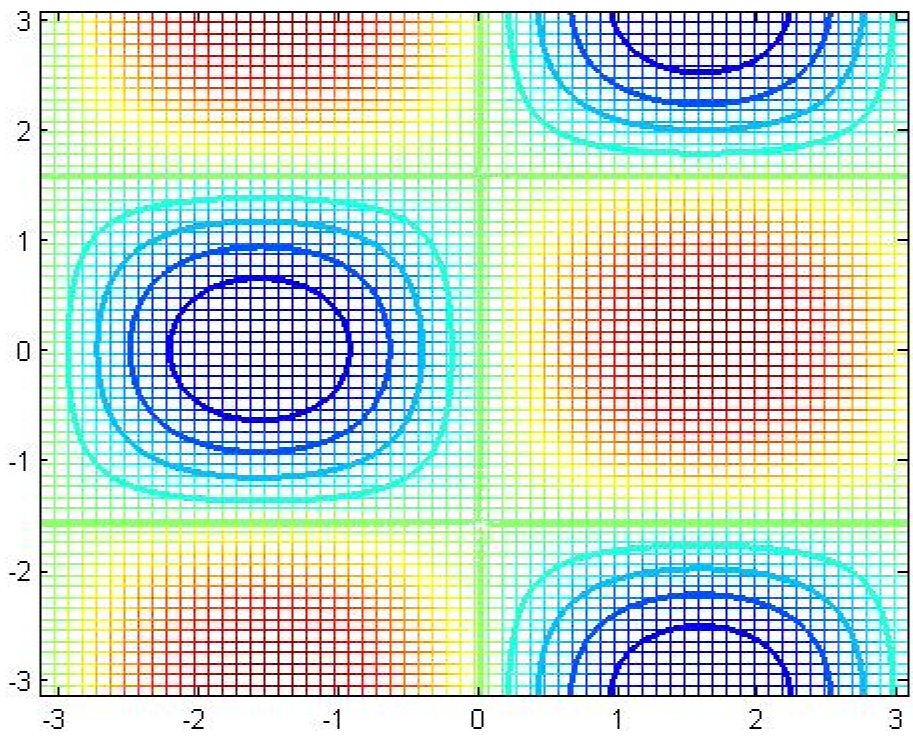

1
0.5
0
-0.5
-1
4
2
0
-2
-4
-4
-2
0
2
4

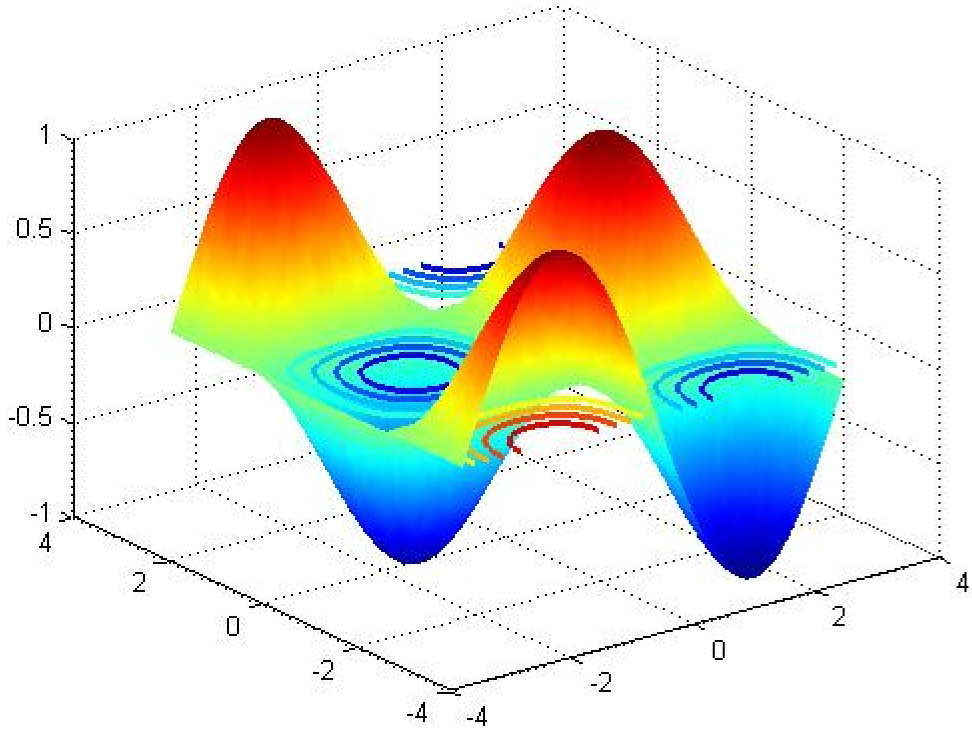

1
0.5
0
-0.5
-1
4
2
0
-2
-4
-4
-2
0
2
4

Contour plot

You can make surfaces two-dimensional by using contour

```
x = -pi:0.1:pi; % make x and y vectors
y = -pi:0.1:pi;
[X,Y] = meshgrid(x,y); %(meshgrid takes in two vectors and return two matrix
% with x and y points;
Z  = sin(X).*cos(Y); % calculate the value of the funciton
contour(X,Y,Z,'LineWidth',2)
hold on %holds on the plot for next plot to be overlayed on top of the
% existing one
mesh(X, Y, Z) % shows the mesh points of the calulated values
% next few lines will create a new plot show the surface plot and overlap
% the contour plot on it.
figure()
surf(X,Y,Z)
shading interp
hold on
contour(X,Y,Z,'LineWidth',2)
hold off % to take the hold off
```

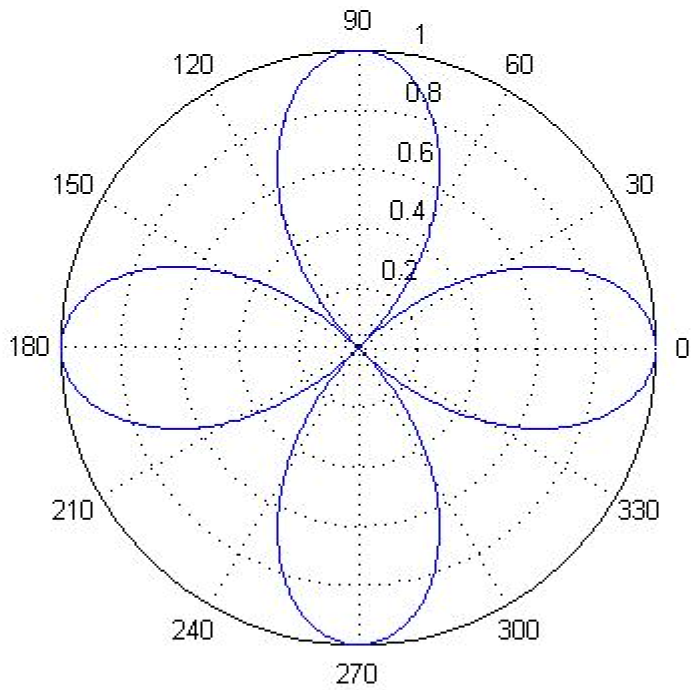




Other specialized plotting funcitons

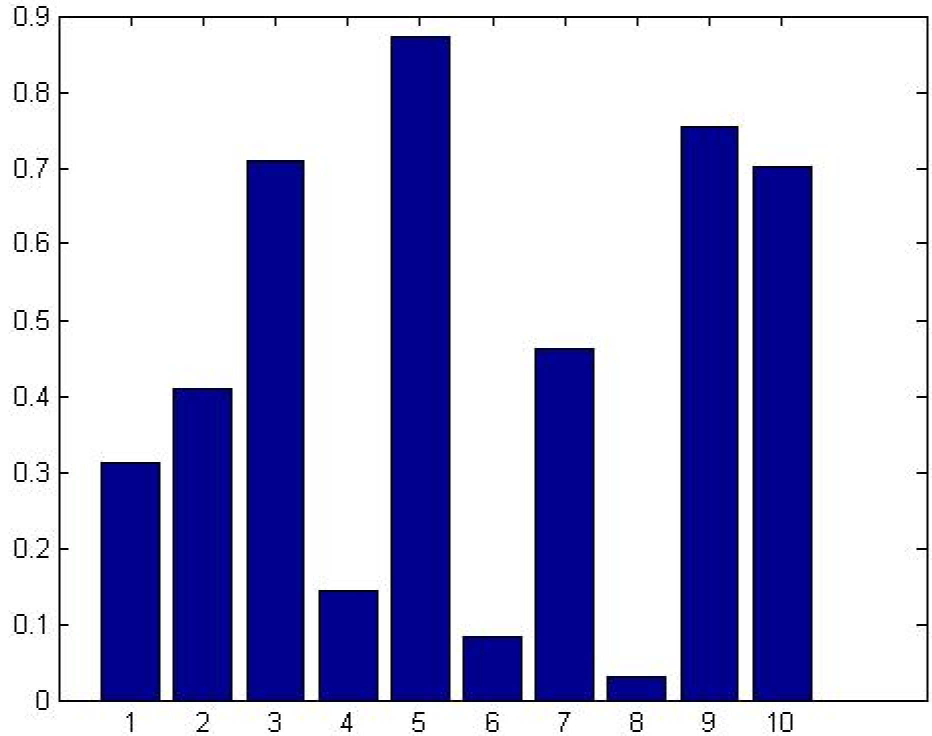
MATLAB has a lot of specialized plotting functions (check documentations for more details to make polar plots

```
figure
polar(0:0.01:2*pi,cos((0:0.01:2*pi)*2))
```



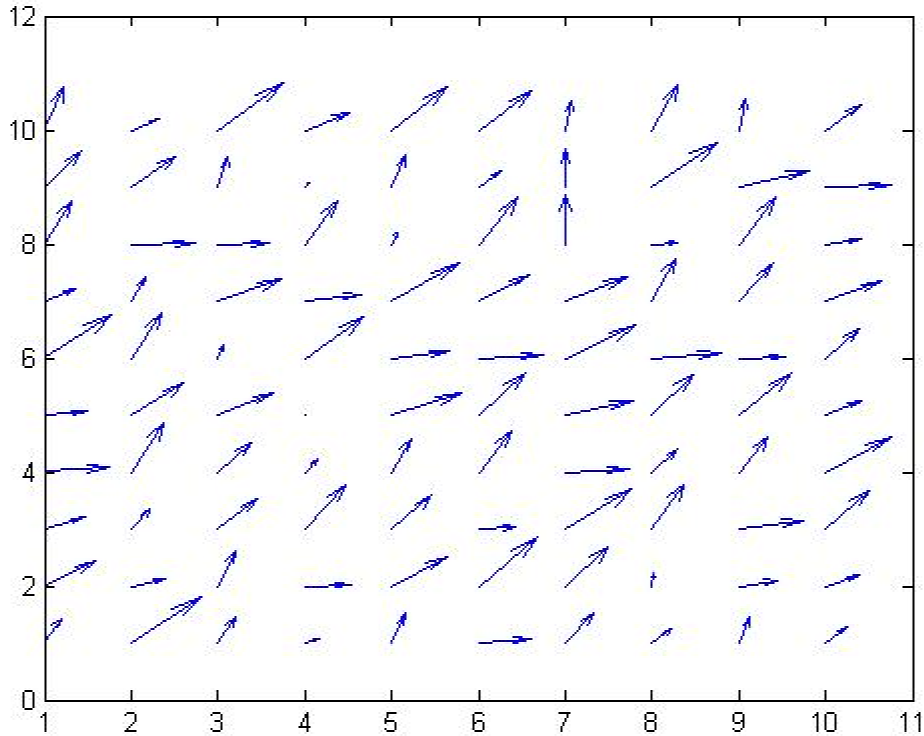
to make bar graphs

```
figure
bar(1:10,rand(1,10));
```



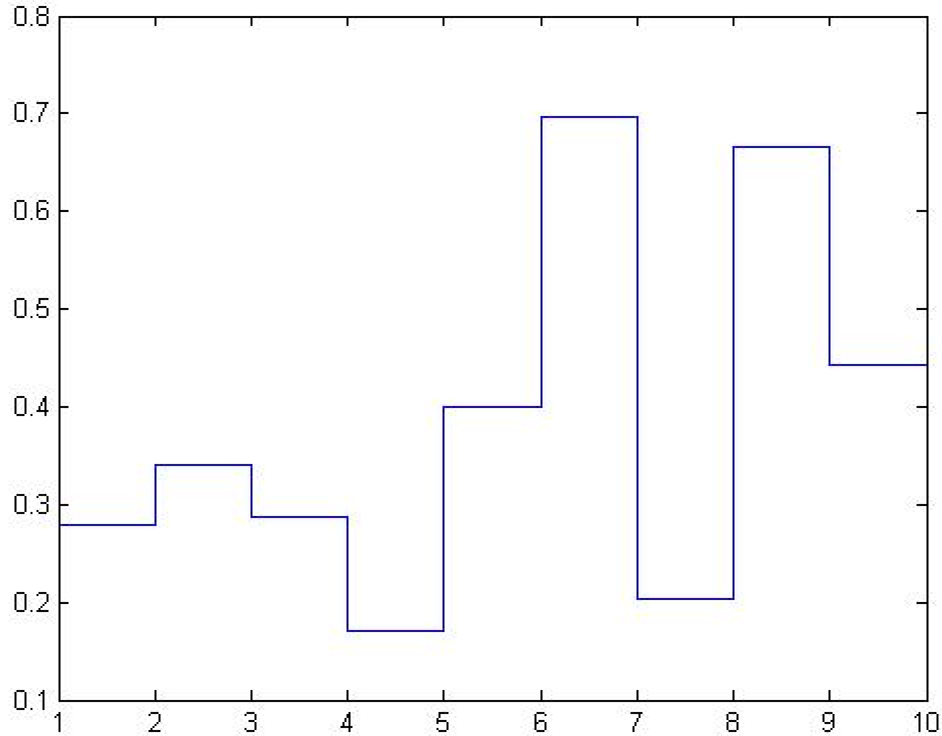
to add velocity vectors to a plot

```
figure
[X,Y] = meshgrid(1:10,1:10);
quiver(X,Y,rand(10),rand(10));
```

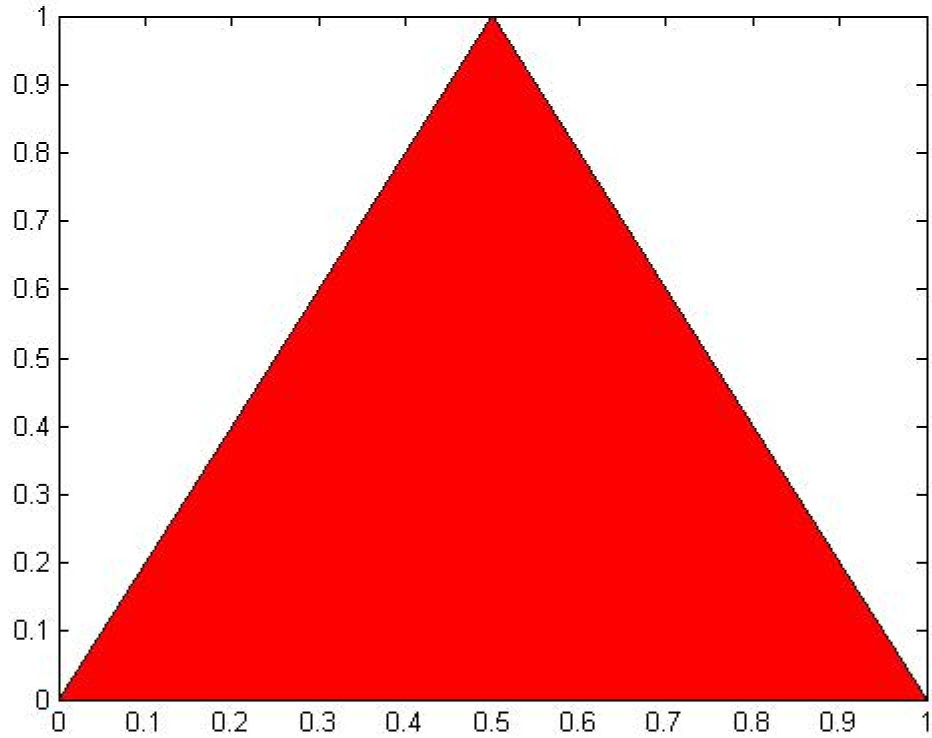


stairs-plot piecewise constant functions

```
figure
stairs(1:10,rand(1,10));
```




```
%*fill* draws and fills a polygon with specified vertices
fill([0 1 0.5],[0 0 1],'r');
```



Systems of Linear Equations

Let us solve the following system of linear equations

$$x + 2y - 3z = 5$$

$$3x - y + z = -8$$

$$x - y + z = 0$$

```
% to solve this we will have to create a coefficient matrix A and b so that
% the systems of equation can be written as $Ax = b$. To solve it we will use
% the *\* (left division)
A = [1 2 -3;-3 -1 1;1 -1 1];
b = [5;-8;0];
x = A\b
```

```
x =

    2.0000
    3.0000
    1.0000
```

Linear Algebra

```
mat = [1 2 -3;-3 -1 1;1 -1 1];
r = rank(mat) % calculates the rank of the above matrix
d = det(mat) % calcualtes the determinant of the matrix
E = inv(mat) %calculates the  inverse of the matrix
[V,D] = eig(mat) % eigen value decomposition
```

```
r =

     3


d =

   -4.0000


E =

         0   -0.2500    0.2500
   -1.0000   -1.0000   -2.0000
   -1.0000   -0.7500   -1.2500


V =

  -0.6641 + 0.0000i  -0.6641 + 0.0000i   0.0274 + 0.0000i
   0.3952 - 0.5029i   0.3952 + 0.5029i   0.8257 + 0.0000i
   0.1989 + 0.3321i   0.1989 - 0.3321i   0.5634 + 0.0000i


D =

   0.7085 + 3.0148i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.7085 - 3.0148i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i  -0.4171 + 0.0000i
```

Polynomials

Matlab represnts a polynomials by a vector of coefficients

$ax^3 + bx^2 + cx + d$ is represented by a vector [a b c d]

```
P = [1 0 -2]; % represents $x^2-2$
P = [2 0 0 0]; % represents $2x^3$
```

To get roots use the function **roots**

```
P = [1 0 -2];
```

```
r = roots (P)
P = poly(r) % this creats the polynomial from the roots*)
```

```
r =

    1.4142
   -1.4142


P =

    1.0000   -0.0000   -2.0000
```

We can evaluate polynomials at one or many points

```
P = [1 0 -2];
x0 = 4;
y0 = polyval(P,x0)
```

```
y0 =

    14
```

or at many points

```
P = [1 0 -2];
x = [4 3 2];
y = polyval(P,x)
```

```
y =

    14     7     2
```

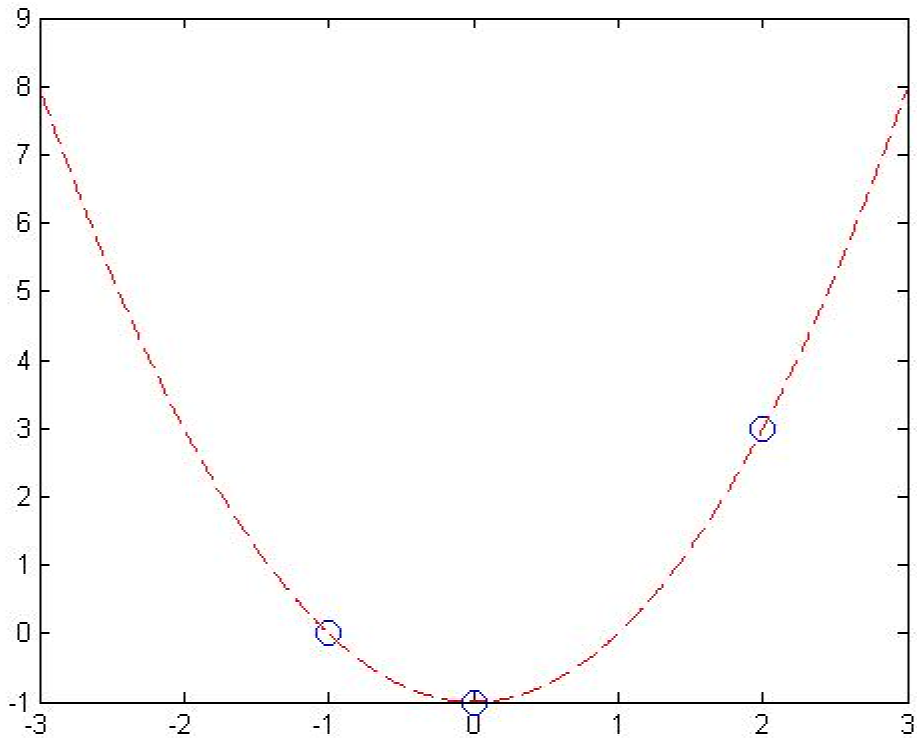
Polynomial Fitting

To find the best *second order* polynomial that fits the points (-1, 0), (0, -1) and (2, 3) we do the following:

```
X = [-1 0 2];
Y = [0 -1 3]
p2 = polyfit (X, Y, 2)

% Now check the fitness by plotting the fucntion

plot(X,Y,'o', 'MarkerSize', 10)
hold on;
x = -3:.01:3;
plot(x, polyval(p2,x), 'r--')
```

```
hold off;
```

```
Y =

     0    -1     3


p2 =

    1.0000   -0.0000   -1.0000
```

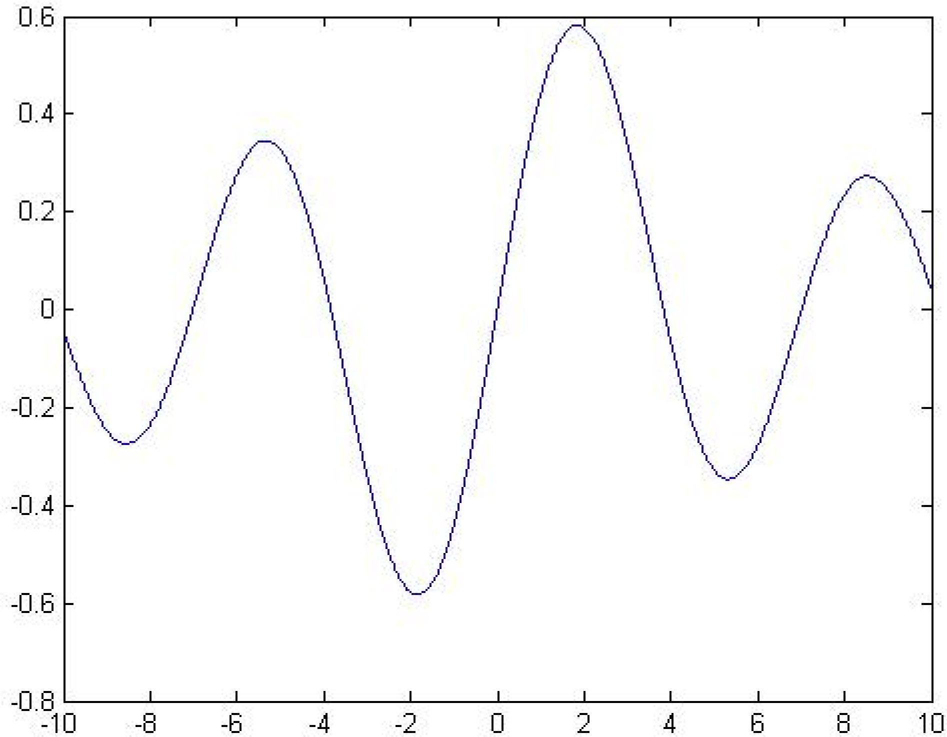


Nonlinear Root Finding

```
% *fzero* function calculate the roots of _any _ arbitrary function.
% You need to pass the function ans give na initial guess of the root
% It by default uses Newton's method to find the root. To find multiple
% root you will have to pass multiple initial guesses
x = -10:0.001:10;
plot(x, besselj(1, x))

y = inline('besselj(1, x)', 'x'); % creats a funcntion y(x) = cos(exp(x))
                                    % + x.^2 -1

% note the use of the .^ instead of ^
```

```
x = fzero(y, 1)
```

```
x =

   3.6401e-26
```



Creating functions

There are three basic ways to create functions * inline function (example in the root finding) * anonymous function * using a .m file and saving it to the workspace

See MATLAB documentation for more details A function that might be helpful for a fucntion with optional inputs is **nargin**. We did an example of it in during the tutorial session. Please type *doc nargin * in the command line to get more details

A function is a group of statements that together perform a task. In MATLAB, functions are defined in separate files. The name of the file and of the function should be the same.

Functions operate on variables within their own workspace, which is also called the local workspace, separate from the workspace you access at the MATLAB command prompt which is called the base workspace.

Functions can accept more than one input arguments and may return more than one output arguments

Syntax of a function statement is: function [out1,out2, ..., outN] = myfun(in1,in2,in3, ..., inN)

The following function named mymax should be written in a file named mymax.m. It takes five numbers as argument and returns the maximum of the numbers. Create a function file, named mymax.m and type the following code in it :

```
%     function max = mymax(n1, n2, n3, n4, n5)
%     %This function calculates the maximum of the
%     % five numbers given as input
%         max =  n1;
%         if(n2 > max)
%             max = n2;
%         end
%         if(n3 > max)
%             max = n3;
%         end
%         if(n4 > max)
%             max = n4;
%         end
%         if(n5 > max)
%             max = n5;
%         end
```

The first line of a function starts with the keyword function. It gives the name of the function and order of arguments. In our example, the mymax function has five input arguments and one output argument.

The comment lines that come right after the function statement provide the help text. These lines are printed when you type:

help mymax MATLAB will execute the above statement and return the following result:

```
_This function calculates the maximum of the five numbers given as
input_
```

You can call the function as:

mymax(34, 78, 89, 23, 11) MATLAB will execute the above statement and return the following result:

ans = 89

Anonymous Functions

An anonymous function is like an inline function in traditional programming languages, defined within a single MATLAB statement. It consists of a single MATLAB expression and any number of input and output arguments.

You can define an anonymous function right at the MATLAB command line or within a function or script.

This way you can create simple functions without having to create a file for them.

The syntax for creating an anonymous function from an expression is

```
f = @(arglist)expression
```

Example In this example, we will write an anonymous function named power, which will take two numbers as input and return first number raised to the power of the second number.

```
power = @(x, n) x.^n; % creats a function of x and n. note the use of .^ ;
% this way we can pass on vectors
result1 = power(7, 3)
result2 = power(49, 0.5)
result3 = power(10, -10)
result4 = power (4.5, 1.5)
%
```

```
        result1 =

           343


        result2 =

             7


        result3 =

           1.0000e-10


        result4 =

            9.5459
```

Primary and Sub-Functions

Any function other than an anonymous function must be defined within a file. Each function file contains a required primary function that appears first and any number of optional sub-functions that comes after the primary function and used by it. Primary functions can be called from outside of the file that defines them, either from command line or from other functions, but sub-functions cannot be called from command line or other functions, outside the function file.

Sub-functions are visible only to the primary function and other sub-functions within the function file that defines them.

Example

Let us write a function named quadratic that would calculate the roots of a quadratic equation. The function would take three inputs, the quadratic co-efficient, the linear co-efficient and the constant term. It would return the roots.

The function file quadratic.m will contain the primary function quadratic and the sub-function disc, which calculates the discriminant.

Create a function file quadratic.m and type the following code in it :

```
function [x1,x2] = quadratic(a,b,c)
    %this function returns the roots of
    % a quadratic equation.
    % It takes 3 input arguments
```

```
    % which are the co-efficients of x2, x and the constant term
    % It returns the roots
    d = disc(a,b,c);
    x1 = (-b + d) / (2*a);
    x2 = (-b - d) / (2*a);
end % end of quadratic

function dis = disc(a,b,c)
    %function calculates the discriminant
    dis = sqrt(b^2 - 4*a*c);
end % end of sub-function
```

You can call the above function from command prompt as:

```
quadratic(2,4,-4)

MATLAB will execute the above statement and return the following
result:
```

ans = 0.7321

Nested Functions

You can define functions within the body of another function. These are called nested functions. A nested function contains any or all of the components of any other function.

Nested functions are defined within the scope of another function and they share access to the containing function's workspace.

A nested function follows the following syntax:

```
function x = A(p1, p2)
    ...
    B(p2)
    function y = B(p3)
        ...
    end
    ...
end
```

Example

Let us rewrite the function quadratic, from previous example, however, this time the disc function will be a nested function.

Create a function file quadratic2.m and type the following code in it:

```
function [x1,x2] = quadratic2(a,b,c)
    function disc  % nested function
       d = sqrt(b^2 - 4*a*c);
    end % end of function disc
    disc; % caled the neested function to calculate d
    x1 = (-b + d) / (2*a);
    x2 = (-b - d) / (2*a);
```

```
end % end of function quadratic2
```

You can call the above function from command prompt as:

```
quadratic2(2,4,-4)
```

MATLAB will execute the above statement and return the following result:

ans = 0.7321

Private Functions

A private function is a primary function that is visible only to a limited group of other functions. If you do not want to expose the implementation of a function(s), you can create them as private functions.

Private functions reside in subfolders with the special name private. They are visible only to functions in the parent folder.

Example Let us rewrite the quadratic function. This time, however, the disc function calculating the discriminant, will be a private function.

Create a subfolder named private in working directory. Store the following function file disc.m in it:

```
function dis = disc(a,b,c)
    %function calculates the discriminant
    dis = sqrt(b^2 - 4*a*c);
end % end of sub-function
```

Create a function quadratic3.m in your working directory and type the following code in it:

```
function [x1,x2] = quadratic3(a,b,c)
    %this function returns the roots of
    % a quadratic equation.
    % It takes 3 input arguments
    % which are the co-efficients of x2, x and the
    %constant term
    % It returns the roots
    d = disc(a,b,c);
    x1 = (-b + d) / (2*a);
    x2 = (-b - d) / (2*a);
 end % end of quadratic3
```

You can call the above function from command prompt as:

```
quadratic3(2,4,-4)
```

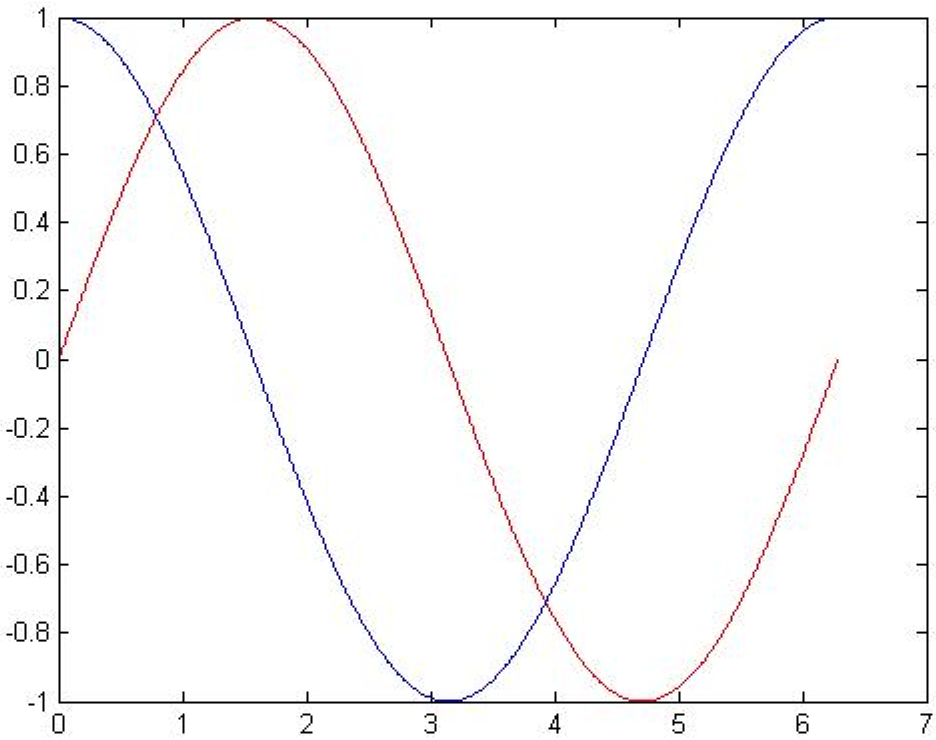
MATLAB will execute the above statement and return the following result:

ans = 0.7321

Numerical Differentiation and Integration

Matlab can differentiate numerically. Let's look at an example. basically for this you will supply a vector and it will find the derivatives

```
x = 0:0.01:2*pi;
y = sin(x);
dydx = diff(y)./diff(x);
plot(x, y, 'r-', x(2:end), dydx, 'b-') % note that the length of dydx is
% one less than the length of x
```



You can also operate on matrices

```
mat = [1 3 5; 4 8 6];
dm = diff(mat, 1, 2) %first difference of mat along the 2nd dimension
[dx, dy] = gradient(mat) %returns the gradient. the function gradiaent
% returns to vectors
```

```
dm =

     2     2
     4    -2


dx =

     2     2     2
     4     1    -2


dy =
```

```
3      5      1
3      5      1
```

For integration you can use either Adaptive Simpson's quadrature or trapezoidal rule. Adaptive meaning the sampling of the function depends on the oscillatory nature of the funciton. Hence this should be your first preference for any oscillatory function

For adaptive input must be a function

Trapezoidal rule only works on a vector

```
q2 = quad(@(x) sin(x).*x, 0, pi)% this does the integration with respect to
% x from 0 to pi
```

```
q2 =

    3.1416
```

```
x = 0:0.01:pi;
z = trapz(x,sin(x)) % z is the integral of sin(x) from 0 to pi
```

```
z =

    2.0000
```

Another example:

```
x = 0:0.01:pi;
z2 = trapz(x, sqrt(exp(x))./x )
```

```
z2 =

   Inf
```

Solving Differential equation

There are a number of tools available to solve differential equaiton. The most popular and generalized is **ode45** . There is another function **dsolve** which is simpler to implement and can solve simple differential equaltion. For a system of equation use **ode45**

Let us look in to an example

Consider the nonlinear system

$$x' = -x + 3z$$

$$y' = -y + 2z$$

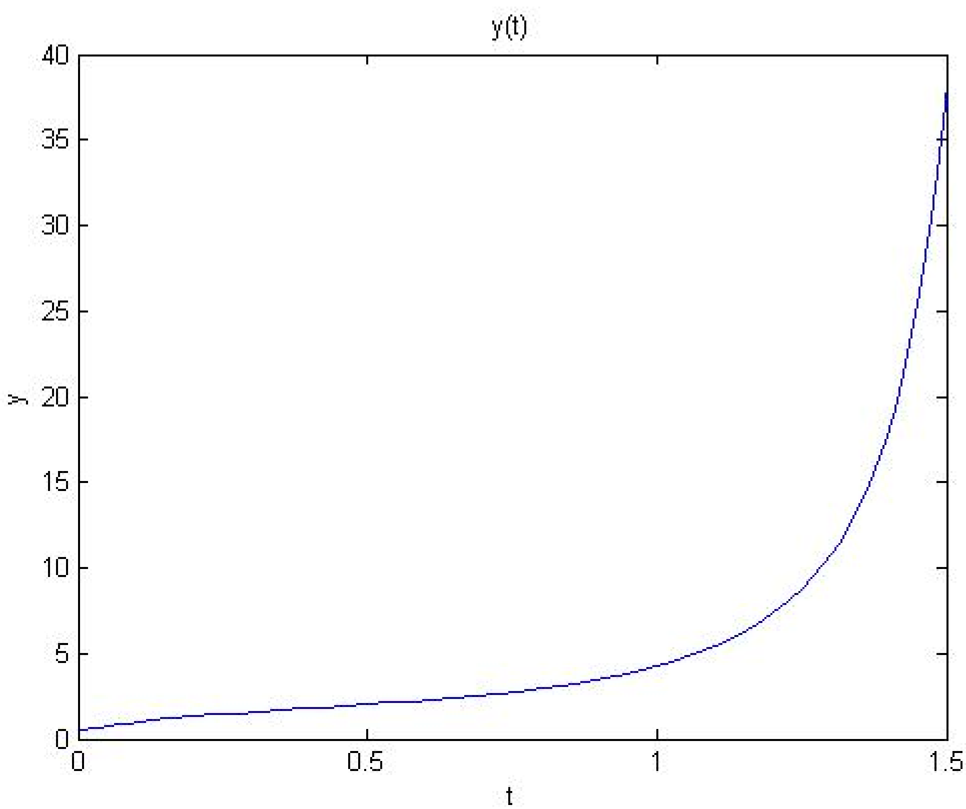
$$z' = x^2 - 2z$$

Think of x, y, z as the coordinates of a vector x. In MATLAB its coordinates are x(1),x(2),x(3) so I can write the right side of the system as a MATLAB function

```
f = @(t,x) [-x(1)+3*x(3);-x(2)+2*x(3);x(1)^2-2*x(3)];

%The numerical solution on the interval $[0,1.5]$ with $x(0) = 1,y(0) = 1/2,
%z(0) = 3$ is

[t,xa] = ode45(f,[0 1.5],[0 1/2 3]);
```

We can plot the components using plot. For example, to plot the graph of y I give the command:

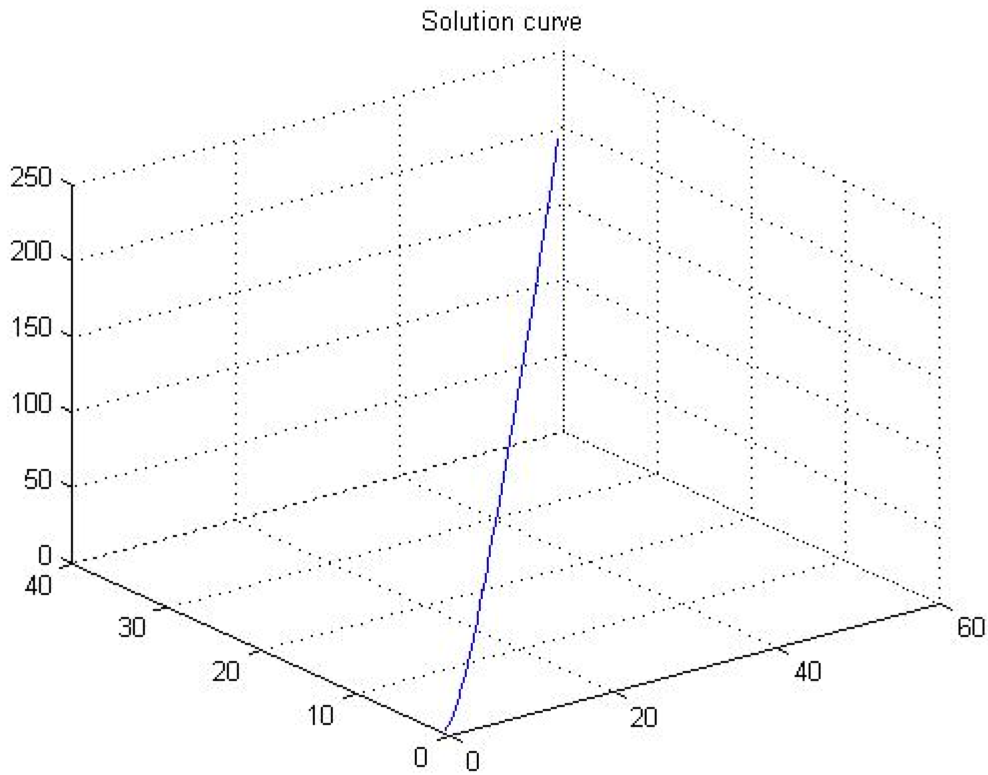
```
plot(t,xa(:,2))
title('y(t)')
xlabel('t'), ylabel('y')
%
```



We can plot the solution curve $(x(t), y(t), z(t))$ in phase space using plot3.

```
plot3(xa(:,1),xa(:,2),xa(:,3))
```

```
grid on
title('Solution curve')
```



Suppose I just want to plot the part which corresponds approximately to the time interval $[1, 1.5]$. Remember that the t produced by ode45 is a vector with a lot of components. We want to know which component corresponds approximately to $t = 1$. One way is to look at the values of t, but with a very long list of values this wouldn't be easy. So first I'll find how many components t has, using the command size.

```
size(t)
```

```
ans =

    69     1
```

This tells us that t has 69 rows and 1 column. Now We do some guessing: t(46) is two-thirds down the list of components of t so We can look at it.

```
t(46)
```

```
ans =

    0.8747
```

We look at components with slightly larger index:

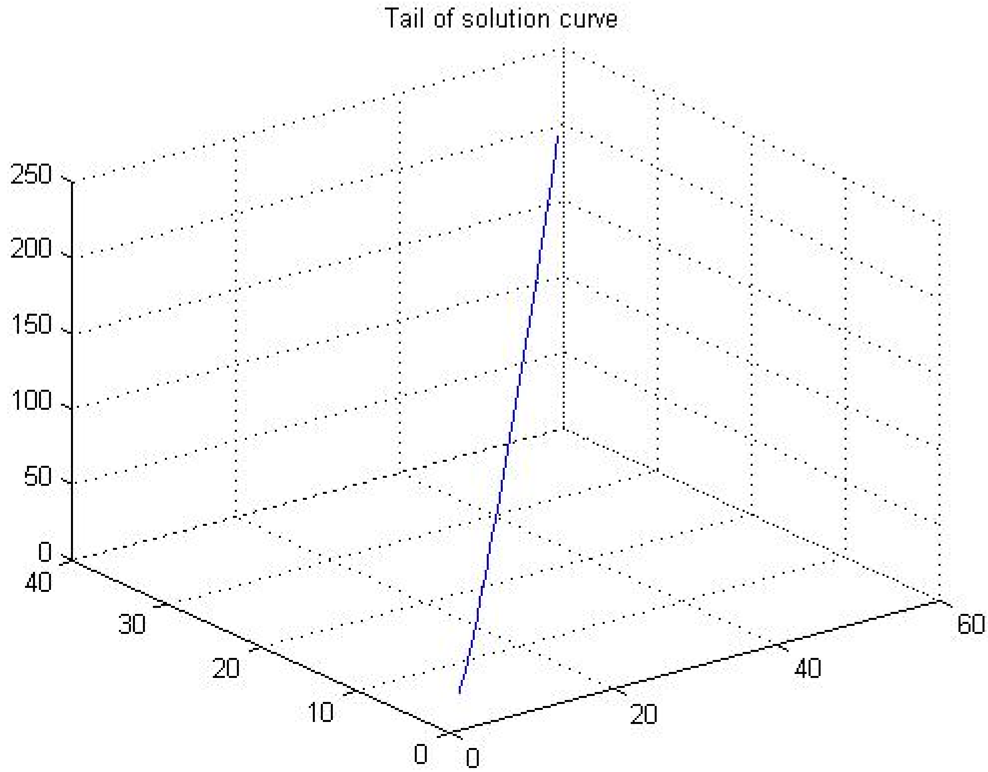
```
t(47:50)
```

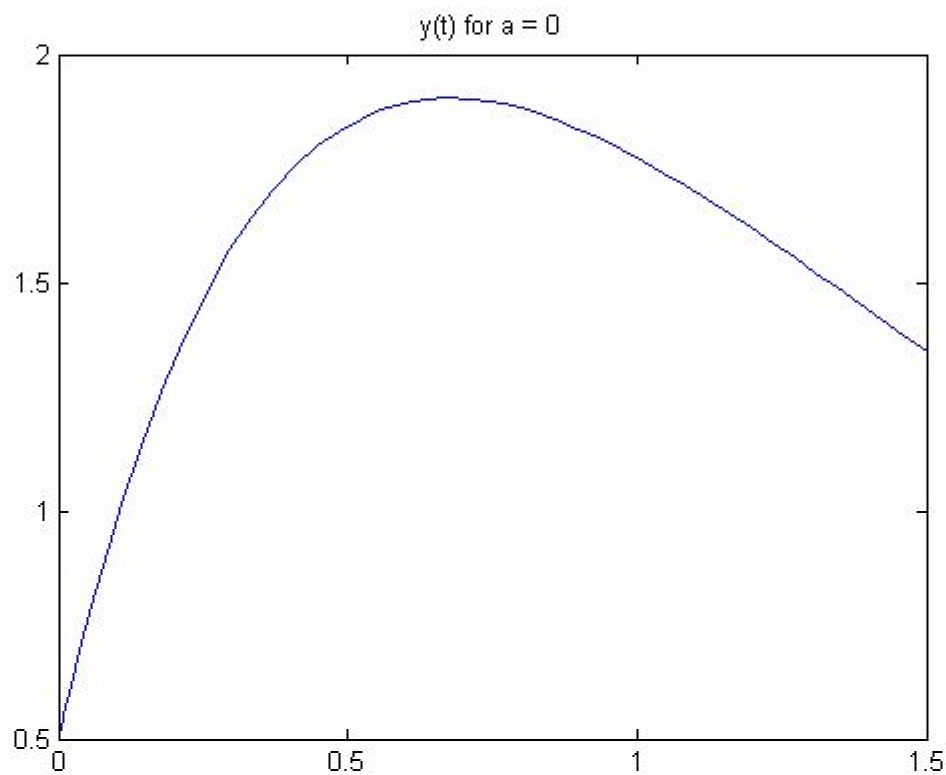
```
ans =

    0.9122
    0.9497
    0.9872
    1.0247
```

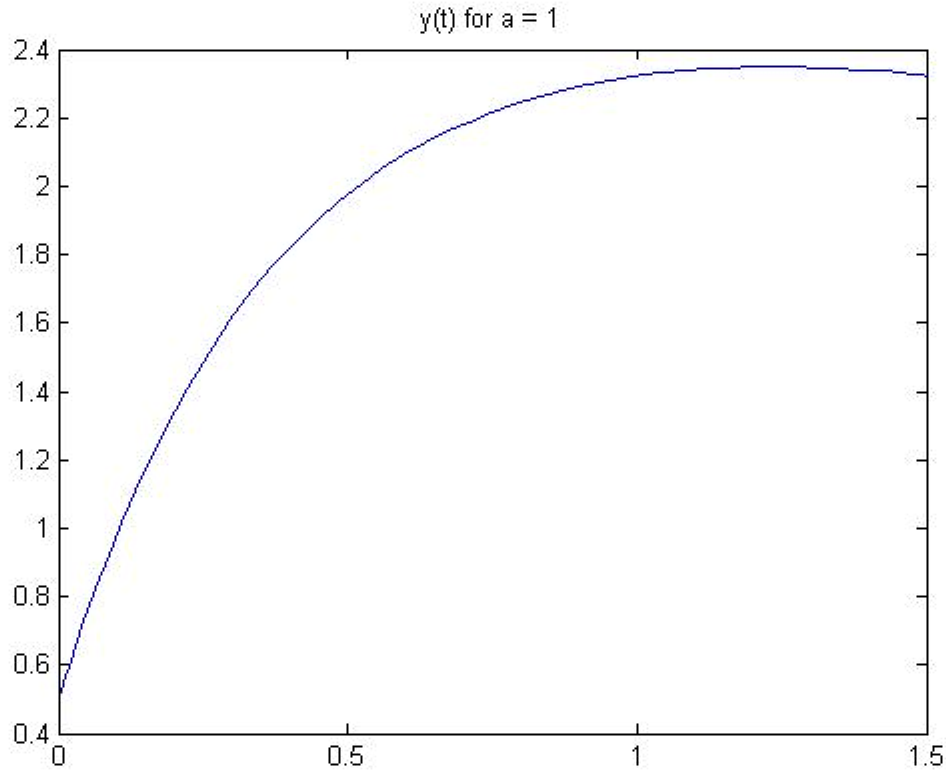
We see that t(49) and t(50) are the closest, one a little larger, the other a little smaller than 1. We'll use 49 as our index. (You can probably do this more elegantly using the Events option.) So we can plot the tail of the solution curve with the following command.

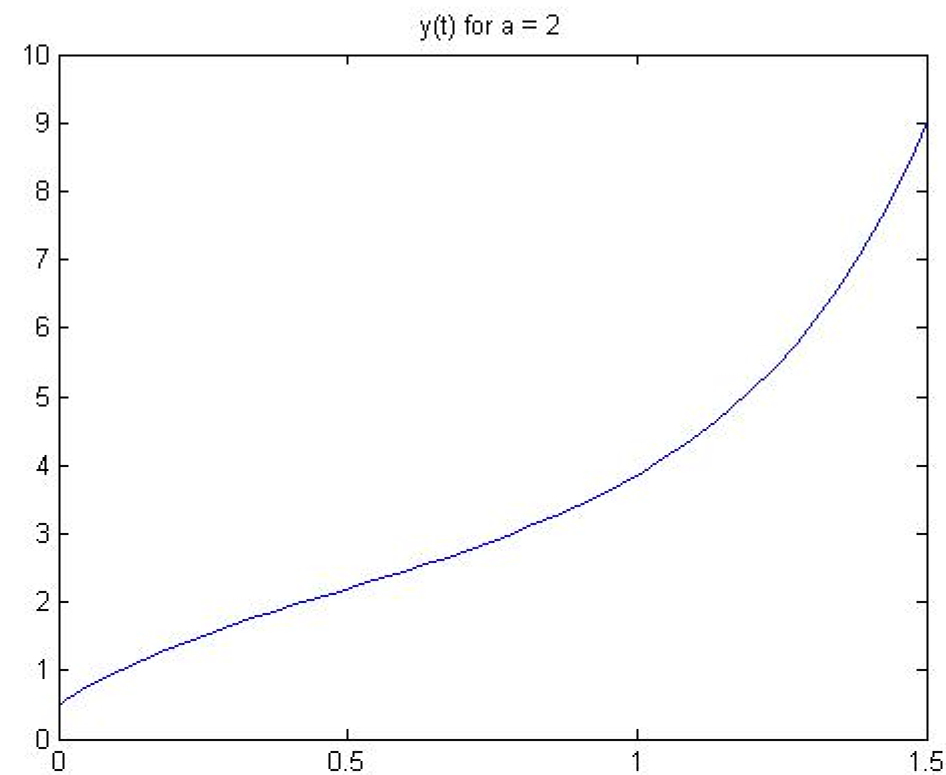
```
plot3(xa(49:69,1),xa(49:69,2),xa(49:69,3))
grid on
title('Tail of solution curve')

% *Using ode45 on a system with a parameter*
%
% Suppose we want to change the system to
%
% $x' = -x+az$
%
% $y' = -y+2z$
%
% $z' = x^2-2z.$
%
% and we would like to use a loop to solve and plot the solution for
% $a = 0,1,2$. We will use the following MATLAB code

syms t x a % we are using MATLABs symbolic toolbox. this command tells
% matlab that these are my variables without any value

g = @(t,x,a)[-x(1)+a*x(3);-x(2)+2*x(3);x(1)^2-2*x(3)] % Create the fucntion

for a = 0:2
    [t,xa] = ode45(@(t,x) g(t,x,a),[0 1.5],[1 1/2 3]);
    figure
    plot(t,xa(:,2))
    title(['y(t) for a = ',num2str(a)'])
end
```

```
g =

    @(t,x,a)[-x(1)+a*x(3);-x(2)+2*x(3);x(1)^2-2*x(3)]
```


Tail of solution curve
250
200
150
100
50
0
40
30
20
10
0
0
20
40
60


y(t) for a = 0
2
1.5
1
0.5
0
0.5
1
1.5


y(t) for a = 1
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0
0.5
1
1.5


y(t) for a = 2
10
9
8
7
6
5
4
3
2
1
0
0
0.5
1
1.5

References

For more details you can check the following places

1. MATLAB Documentation
2. Matlab: A Practical Introduction to Programming and Problem Solving
3. MATLAB: An Introduction with Applications
4. Essential MATLAB for Engineers and Scientists
5. A First Course in Computational Physics
6. Scientific Computing with MATLAB and Octave
7. Ofcourse the internet is full of examples including some I have reproduced here.

Published with MATLAB® R2013a