MATLAB TRAINING SESSION III

NUMERICAL METHODS

MATLAB provides a variety of techniques and functions for solving problems numerically. On-line help for each of the commands shown below is available by typing `help ‘command_name’` or `doc ‘command_name’` at the MATLAB prompt.

SOLUTIONS To SYSTEMS Of LINEAR EQUATIONS:

There are a number of different possible situations that can occur in solving sets of equations with two and three variables. Solutions to sets of equations with more than three variables are discussed in terms of *hyperplanes*. Here we will present two different techniques for solving a system of simultaneous equations using matrix operations.

A linear equation with two variables, such as `y = mx + b`, defines a straight line. If we have two linear equations, they can represent two different lines that intersect at a point, or they can represent parallel lines that do not intersect. If the linear equation contains three variables then it represents a plane. If we have three equations with three variables the planes could intersect at a point, a line, or have no intersection common to all three. These ideas can be extended to more than three variables, although it is harder to visualize. We call the set of points defined by an equation of more than three variables a hyperplane.

In general, we can consider a set of M linear equations that contain N unknowns, where each equation defines a unique hyperplane in the system. If $M < N$, then the system is underspecified, and a unique solution does not exist. If $M = N$, then a unique solution will exist if none of the equations represent parallel hyperplanes. If $M > N$, then the system is overspecified and a unique solution does not exist. A system with a unique solution is called a *nonsingular* system of equations, otherwise it is called a *singular* set of equations. Consider the following system of three equations with three unknowns ($M = 3 = N$):

$$\begin{aligned} 3x + 2y - z &= 10 \\ -x + 3y + 2z &= 5 \\ x - y - z &= -1 \end{aligned}$$

We can rewrite this system of equations using the following matrices:

$$A = \begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix} \qquad X = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad B = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix} \quad \text{as} \quad AX = B$$

Matrix Division:

In MATLAB, a system of simultaneous equations can be solved using matrix division. The solution of the matrix equation AX = B can be computed using matrix left division, as in A\B; or the matrix equation XA = B can be computed using matrix right division, as in B/A.

MATLAB_window

```
>> A = [3,2,-1;-1,3,2;1,-1,-1]

A =

     3     2    -1
    -1     3     2
     1    -1    -1

>> B = [10;5;-1]

B =

    10
     5
    -1

>> X = A\B

X =

   -2.0000
    5.0000
   -6.0000

>> a = [3,-1,1;
        2,3,-1;
        -1,2,-1];
>> b=[10,5,-1];
>> x = b/a

x =

   -2.0000    5.0000   -6.0000

>> 
```

If a set of equations is singular, an error message is displayed; the solution may contain values of NaN meaning +/- infinity, depending on the values in A and B. It is also possible that a system of equations is very close to being singular. These systems are called *ill-conditioned* systems. MATLAB will compute a solution, but a warning message is printed indicating the results may be inaccurate.

Matrix Inverse:
The system of equations can also be solved using the inverse of a matrix.

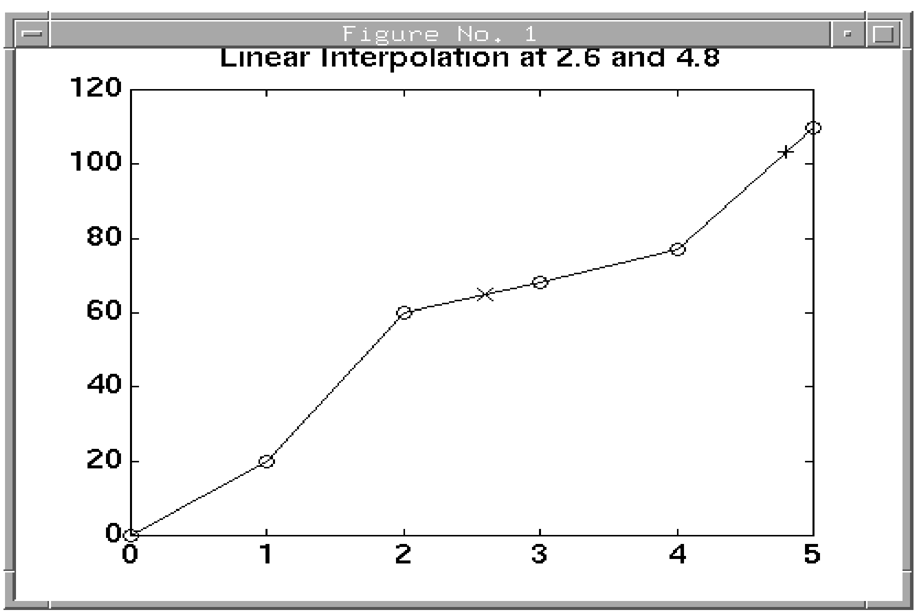
If $AX = B$ then $X = A^{-1}B$; or if $XA = B$ then $X = BA^{-1}$

```
MATLAB_window
>> X = inv(A)*B

X =

   -2.0000
    5.0000
   -6.0000

>> x = b*inv(a)

x =

   -2.0000    5.0000   -6.0000

>> 
```

Of course, MATLAB will give similar warnings if A is singular or ill-conditioned.

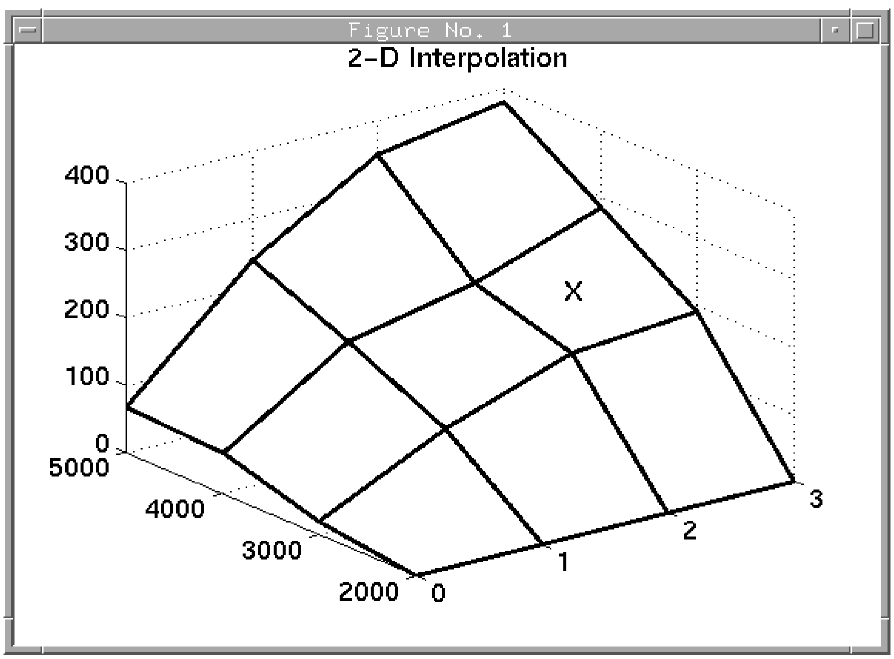
INTERPOLATION
One of the most common techniques for estimating data between two given data points is linear interpolation. The `table1` function performs a one dimensional linear interpolation using a table of data. The first argument of the function is the name of the table and the second argument is the value of x, for which we want to interpolate a corresponding y value. The data in the table must be given in ascending or descending order.

```
MATLAB_window
>> data1(:,1) = [0.0 1.0 2.0 3.0 4.0 5.0]';
>> data1(:,2) = [0.0 20.0 60.0 68.0 77.0 110.0]';
>> disp('Time(sec) Tempurature(degF)'), disp(data1)
Time(sec) Tempurature(degF)
     0     0
     1    20
     2    60
     3    68
     4    77
     5   110

>> y1 = table1(data1,2.6)

y1 =

   64.8000

>> y2 = table1(data1,4.8)

y2 =

  103.4000

>> plot(data1(:,1),data1(:,2),data1(:,1),data1(:,2),'o')
>> hold, plot(2.6,y1,'x',4.8,y2,'+'), hold
Current plot held
Current plot released
>> title(' Linear Interpolation ant 2.6 and 4.8 ')
>>
```



The `table2` function performs a two-dimensional interpolation using values from the first column and the corresponding row of the table. The data in the first column and row of the table must be increasing or decreasing and x and y must be within the limits of the table.

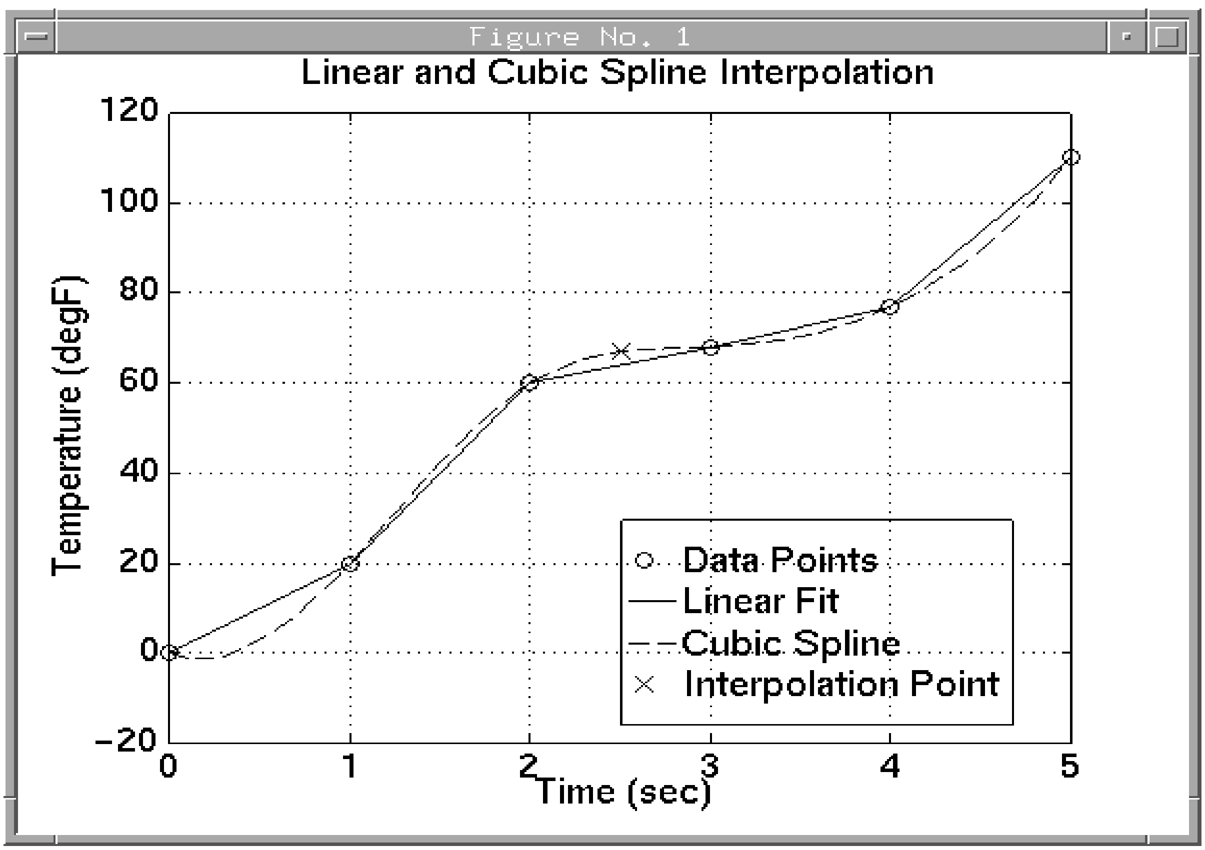
```
MATLAB_window
>> data2(1,:) = [0 2000 3000 4000 5000 6000];
>> data2(2,:) = [0 0 0 0 0 0];
>> data2(3,:) = [1 20 110 176 190 240];
>> data2(4,:) = [2 60 180 220 285 327];
>> data2(5,:) = [3 68 240 349 380 428];
>> disp('Temperature Data'),...
disp('  Time(s)         Engine Speed (rpm)'),...
disp(data2)
Temperature Data
  Time(s)         Engine Speed (rpm)
           0        2000        3000        4000        5000        6000
           0           0           0           0           0           0
           1          20         110         176         190         240
           2          60         180         220         285         327
           3          68         240         349         380         428

>> temp = table2(data2, 2.3, 3450)

temp =

  225.3150

>> mesh(data2(2:5,1),data2(1,2:5),data2(2:5,2:5))
>> grid, title('2-D Interpolation'), text(2.3,3450,temp,'X')
>> 
```



The `spline` function performs interpolation assuming that the data points are connected by a smooth third-degree polynomial. The first two arguments contain the x and y coordinates of the data and the third argument contains the x coordinate(s) for which we want to find the y value(s) on the spline.

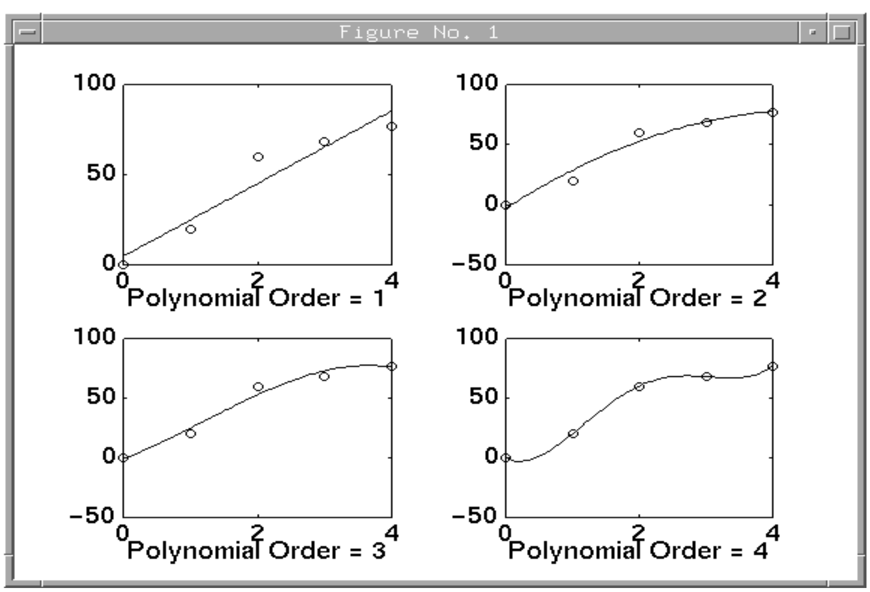
MATLAB_window

```
>> xdata = [0 1 2 3 4 5];
>> ydata = [0 20 60 68 77 110];
>> y3 = spline(xdata,ydata,2.5)

y3 =

   66.8750

>> new_x = 0:0.1:5;
>> new_y = spline(xdata,ydata,new_x);
>> plot(xdata,ydata,'o',xdata,ydata,'--',new_x,new_y,'-.',2.5,y3,'x')
>> title('Cubic Spline vs Linear Interpolation'), grid
>> legend('Data Points','Linear Fit','Cubic Spline','Interpolation Point')
>>
```



POLYNOMIAL CURVE FITTING (Least Squares):
We can use the `polyfit` function to compute an nth-order polynomial that best fits a set of data. It is up to the user to decide what order of polynomial to use for a particular set of data.

```
MATLAB_window
>> clg
>> x = [0 1 2 3 4];
>> y = [0 20 60 68 77];
>> newx = 0:0.05:4;
>> for i = 1:4
          poly_coef = polyfit(x,y,i)
          ncurve = polyval(poly_coef,newx);
          subplot(2,2,i), plot(x,y,'o',newx,ncurve);
          xlabel(['Polynomial Fit Order = ',num2str(i)])
   end

poly_coef =

   20.2000    4.6000


poly_coef =

   -3.8571   35.6286   -3.1143


poly_coef =

   -1.5833    5.6429   22.0119   -1.2143


poly_coef =

    3.5417  -29.9167   74.9583  -28.5833   -0.0000

>> 
```



POLYNOMIAL ANALYSIS:
A polynomial of the form $f(x) = a_1 x^n + a_2 x^{n-1} + \ldots + a_n x + a_{n+1}$ is represented in MATLAB as a row vector of the coefficients of the polynomial $fx = \left[a_1\ a_2\ \ldots\ a_n\ a_{n+1}\right]$.

Polynomial Analysis commands and functions are summarized below:

• `conv(p1,p2)`	Convolves vectors p1 and p2 (polynomial multiplication)
• `deconv(p1,p2)`	Deconvolves vector p2 from p1 (polynomial division)
• `polyval(p,s)`	Evaluates the polynomial p at x = s
• `roots(p)`	Determines the roots of polynomial p
• `poly(r)`	Determines the polynomial whose roots are r

MATLAB_window

```
>> % Work with p1 = x^2-1 and p2 = x-2
>> p1 = [1 0 -1]; p2 = [1 -2];
>> % Multiply p1 & p2 = x^3 - 2x^2 - x + 2
>> p = conv(p1,p2)

p =

     1    -2    -1     2

>> r = roots(p)

r =

    2.0000
    1.0000
   -1.0000

>> % Factor out (x-1) from p
>> p3 = deconv(p,[1 -1])

p3 =

     1    -1    -2

>> % Evalute P(x) at x = 1
>> polyval(p,1)

ans =

     0

>> % Build p(x) from its roots
>> p4 = poly(r)

p4 =

    1.0000   -2.0000   -1.0000    2.0000
```

NONLINEAR EQUATIONS AND OPTIMIZATION

MATLAB has functions for solving nonlinear equations and unconstrained nonlinear minimization. The functions are included in the class of function functions because they depend on user written m-file functions to define the nonlinear equations.

- `fmin('fun',xl,xh)` Finds the minimum of a function **fun.m** of one variable in the range **[xl,xh]**
- `fmins('fun',x0)` Finds the minimum of a multivariable function **fun.m** near the initial guess vector **x0**
- `fzero('fun',x0)` Finds the zero point of the function **fun.m** of one variable near the initial guess **x0**
- `fsolve('fun',x0)` Finds the solution of a multivariable set of equations in function **fun.m** near the initial guess vector **x0**.

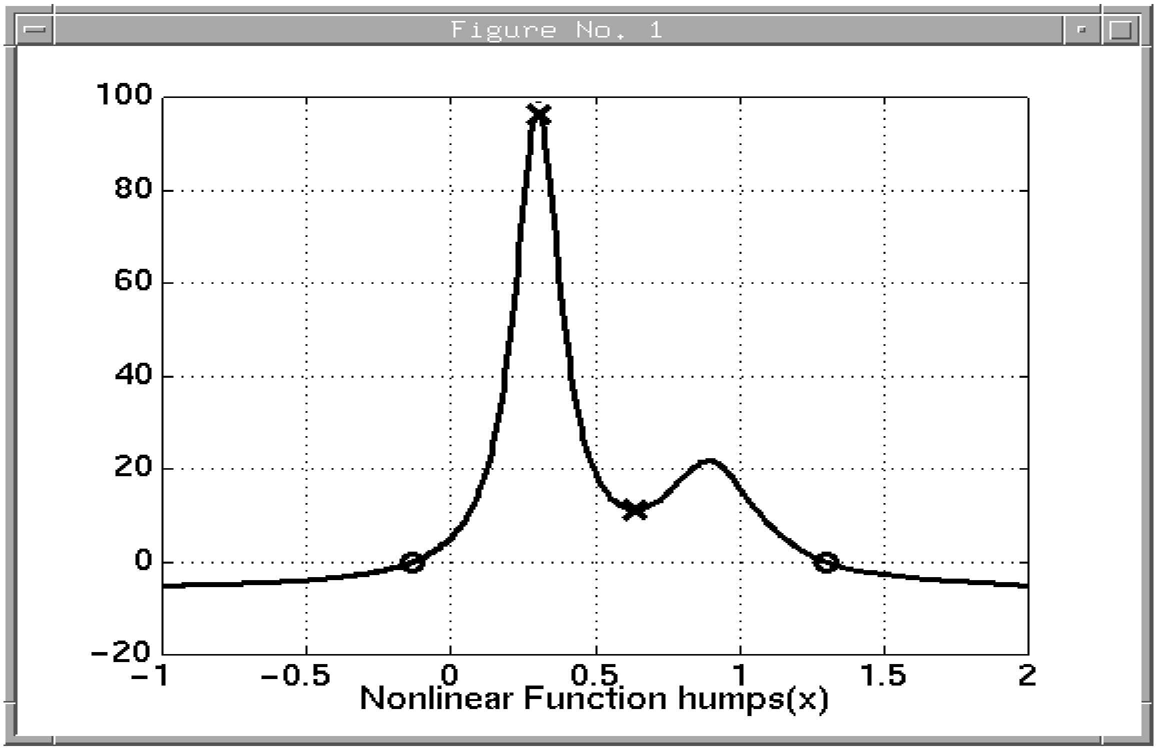
(Write the following M-file function and call it mhumps.m)

editor_window

```
function y = mhumps(x)
% Negative of the built-in humps(x)

y = -1*humps(x);

"mhumps.m" 5 lines, 78 characters
```

MATLAB_window

```
>> clg, fplot('humps',[-1,2])
>> xlabel('Nonlinear Function humps(x)'), grid, hold
Current plot held
>> xmin = fmin('humps',0,1), xmax = fmin('mhumps',0,1)

xmin =

    0.6370


xmax =

    0.3004

>> zs(1) = fzero('humps',0); zs(2) = fzero('humps',1); zs

zs =

   -0.1316    1.2995

>> plot([xmin xmax],[humps(xmin) humps(xmax)],'rx',zs,[0 0],'o')
```



NUMERICAL INTEGRATION

MATLAB has two quadrature functions for performing numerical function integration. The `quad` function uses an adaptive form of Simpson's rule, while `quad8` uses an adaptive Newton-Cotes 8-panel rule. The `quad8` function is better at handling certain types of singularities at the end points of the function to be integrated. The quad functions have 4 main arguments. The first argument is the name of the function to be integrated which can be a MATLAB function or a user written M-file function. The second and third arguments are the lower and upper integral limits a and b. The fourth argument is optional and represents the tolerance or the desired accuracy of the result.

EXAMPLE

Suppose we are interested in $K_Q = \int_a^b \sqrt{x}\,dx$ we know that $K = \frac{2}{3}(b^{3/2} - a^{3/2})$

Write the following M-file and call it integ.m

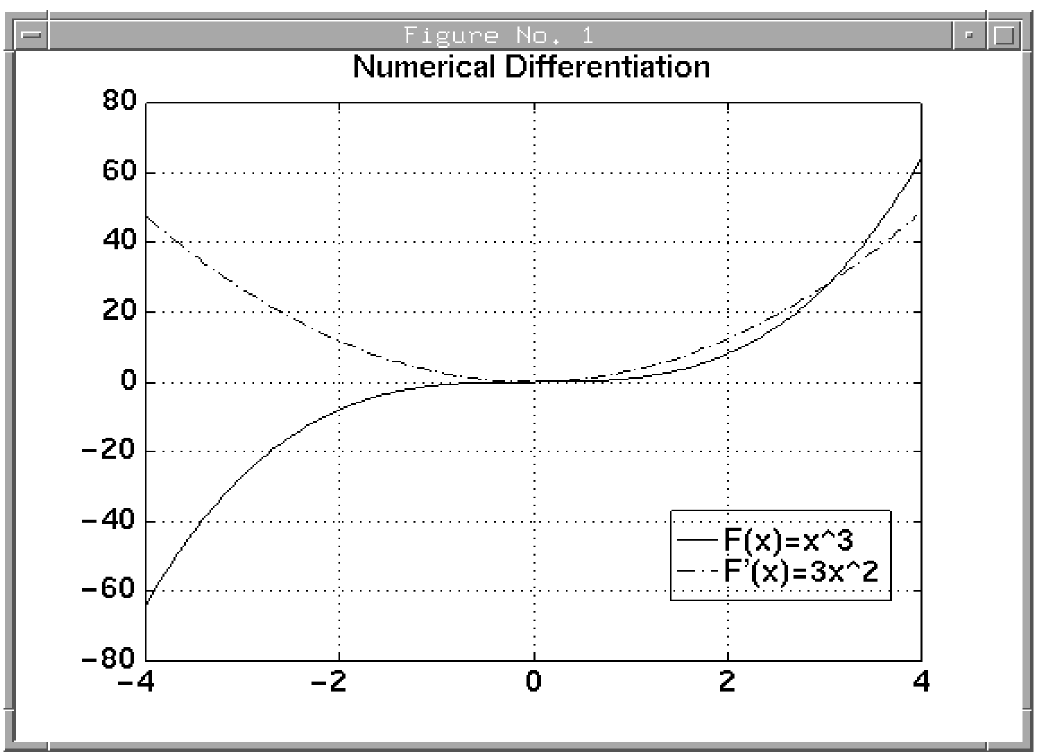
editor_window
```
% A program to integrate srqt(x)

a = input('Enter left endpoint > 0 :');
b = input('Enter right endpoint >= 0 :');
K = 2/3*(b^(3/2) - a^(3/2));
Kq = quad('sqrt',a,b);
Kq8 = quad8('sqrt',a,b);
disp(sprintf('\nInterval [%4.2f,%4.2f]',a,b))
disp(sprintf('Analytical: %f \n Numerical: %f %f ',K,Kq,Kq8))

~
"matlab/files/integ.m" 10 lines, 302 characters
```

```
MATLAB_window
>> integ

Enter left endpoint > 0 :.1

Enter right endpoint >= 0 :1

Interval [0.10,1.00]
Analytical: 0.645585
 Numerical: 0.645583 0.645585
>>
```

NUMERICAL DIFFERENTIATION

The `diff` function computes the difference between adjacent values in a vector, generating a vector with one less element. To find the derivative of a function we find `dy/dx` by `diff(y)/diff(x)` where `y` is generated by taking samples of `f(x)` at `x`.

```
MATLAB_window
>> % Differentiate x^3 numerically over x=[-4,4]
>> x = -4:0.05:4;   F = x.^3;
>> dFdx = diff(F)./diff(x);
>> clg, plot(x,F,x(1:length(x)-1),dFdx,'-.')
>> title('Numerical Differentiation'), grid
>> legend('F(x)=x^3','F''(x)=3x^2')
>>
```



ORDINARY DIFFERENTIAL EQUATIONS

MATLAB contains two functions for computing numerical solutions to first-order ordinary differential equations; `ode23` and `ode45`. The ode23 function uses second-order and third-order Runga-Kutta integration equations; The `ode45` function uses fourth-order and fifth-order Runga-Kutta integration equations. The simplest form of the ode functions requires four arguments. The first argument is the name (in quotation marks) of a MATLAB function or M-file that returns `y' = g(x,y)` when it receives values for x and y. The second and third arguments represent the end points of the interval over which we want to find `y = f(x)`. The fourth argument contains the initial conditions or left boundary points that are needed to determine a unique solution to the ODE. The ode functions produces two outputs; a set of x coordinates and the corresponding set of y coordinates, which represents points of the function `y = f(x)`.

When `x` and `y` are vectors, then the equation represents n-coupled first-order ODE's. A higher order differential equation can be written as a system of coupled first-order differential equations using a change of variables. The M-file function used to evaluate the differential equation must compute the values of the derivative in a vector. The initial conditions or boundary points also must be a vector containing value for all the lower order terms. The discussions above will be made clearer by example

Lets solve $\frac{d^2x}{dt} + 2\frac{dx}{dt} + 2x = 4u(t)$

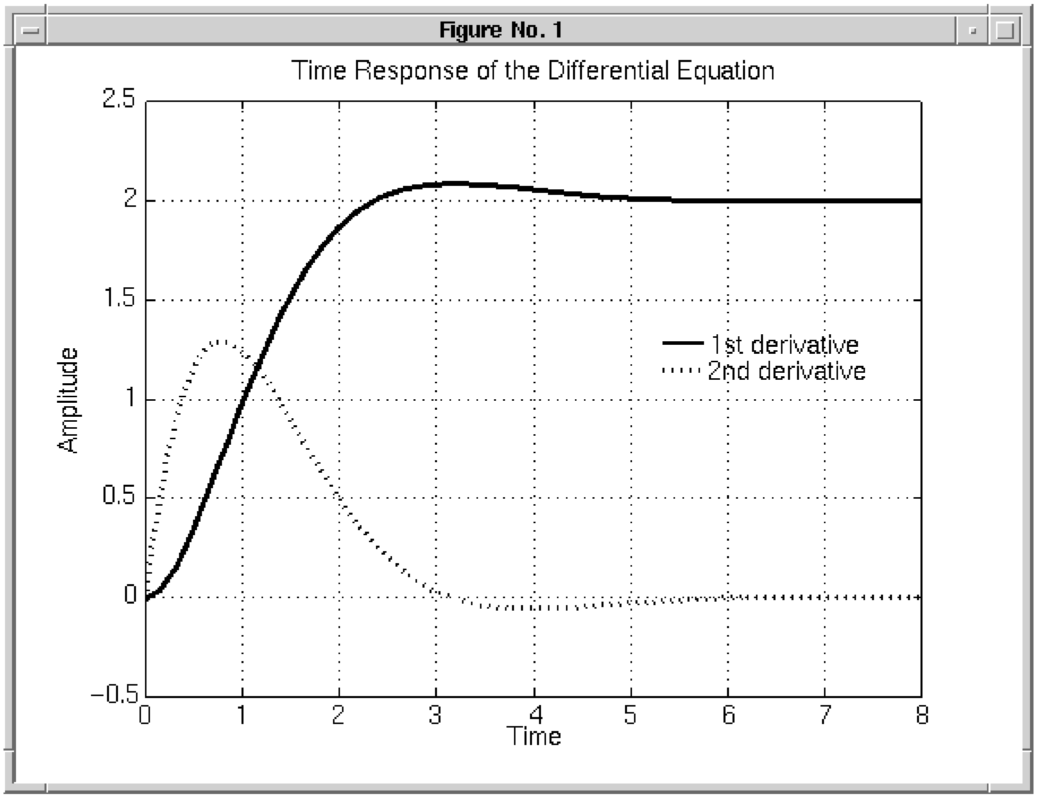
$$\dot{z}_1 = z_2$$
$$\dot{z}_2 = -2z_1 - 2z_2 + 4$$

Create the M-file called diffeq1.m

```
editor_window
function zdot = diffeq1(t,z)
% Linear State Equation
zdot(1) = z(2);
zdot(2) = -2*z(1) - 2*z(2) + 4;

~
"diffeq1.m" 5 lines, 102 characters
```

Now with MATLAB:

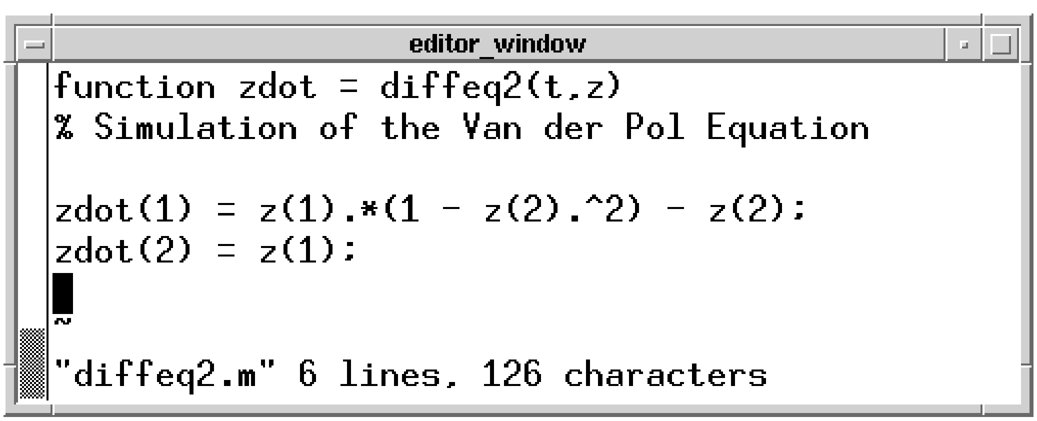
```
MATLAB_window
>> to = 0; tf = 8;          % Time Interval
>> z0 = [0 0];              % Initial Conditions
>> [t,z] = ode23('diffeq1',to,tf,z0);
>> clg, plot(t,z(:,1),t,z(:,2),':'), grid
>> title('Time Response of the Differential Equation')
>> xlabel('Time'), ylabel(' Amplitude ')
>> legend('1st derivative','2nd derivative')
>>
```



Lets solve $\ddot{x} + (x^2 - 1)\dot{x} + x = 0$

$$\dot{z}_1 = z_1(1 - z_2^2) - z_2$$
$$\dot{z}_2 = z_1$$

Create the M-file called diffeq2.m



editor_window

```
function zdot = diffeq2(t,z)
% Simulation of the Van der Pol Equation

zdot(1) = z(1).*(1 - z(2).^2) - z(2);
zdot(2) = z(1);

~
"diffeq2.m" 6 lines, 126 characters
```

Now with MATLAB:

MATLAB_window

```
>> to = 0; tf = 20;          % Time Interval
>> z0 = [0 0.25];            % Initial Conditions
>> [t,z] = ode23('diffeq2',to,tf,z0);
>> clg, plot(t,z(:,1),t,z(:,2),'--'), grid
>> title('Nonlinear State Equation Simulation')
>> xlabel('Time'), ylabel(' Amplitude ')
>> set(gca,'xtick',[0 2.5 5 7.5 10 12.5 15 17.5 20])
>> line([1 2],[-1.5 -1.5]), text(2,-1.5,' 2nd Derivative')
>> line([1 2],[-2 -2],'linestyle','--'), text(2,-2,' 1st Derivative')
>>
```