MATLAB® for *ODEs*

SUPPLEMENTAL NOTES FOR "NUMERICAL METHODS" ME 330

PREPARED BY:

Dr. Kevin R. Anderson
Associate Professor
Mechanical Engineering
California State Polytechnic University at Pomona

kranderson1@csupomona.edu

www.csupomona.edu/~kranderson1/

MATLAB Environment

The MATLAB programming environment provides the user with an interactive work space, in which the user enters commands and view results. After launching MATLAB from the Desktop Icon (or via Start -> Programs -> MATLAB), commands are entered at the >> prompt. A number of Operating System (OS) level control-type commands are available at the MATLAB >> prompt, including:

>> `cd`	Changes the current working directory
>> `delete`	Delete a file
>> `diary`	Saves your MATLAB session as a text file
>> `dir`	List the contents of the current directory
>> `!`	Execute an OS command
>> `quit`	Exit MATLAB

MATLAB Variables and Statements

Scalar, vector and matrix quantities can be easily defined in MATLAB using assignment statements. Some examples follow:

`a= 14.7`	defines a scalar
`r = [1 2 3 4]`	defines a row vector
`c = [1.2 3.2 7.5 5.6]'`	defines a column vector (prime means transpose)
`b = [1 2 3; 4 5 6;7 8 9]`	defines a 3 x 3 matrix, rows are delineated by a ; (Is this example singular ?)

Array elements may be referenced directly, *e.g.* `c(2) = 3.2`, or `b(3,2) = 8`. Built into MATLAB are the usual mathematical functions, such as `sin, cos, tan, atan`, *etc.* as well as everybody's favorite constant `pi`. Additionally, a large number of array functions are provided to aid you in performing operations on matrices some examples of which follow:

>> `A = [0.5 0.5 0.5; 0.5 1.5 1.5; 0.5 1.5 2.5]`

>> `detA = det(A)`	returns the determinant of A into the scalar named detA
>> `Ainv = inv(A)`	returns the inverse of A into the array named invA
>> `y = eig(A)`	computes the eigenvalues of A into column vector y
>> `[x1,x2] = eig(A)`	returns the the matrices U and D, D is a diagonal matrix with the eigenvalues along the diagonal, U is a full matrix with the eigenvectors as column vectors

MATLAB Logic

The MATLAB programming language provides loops, branches and functions, *akin* to any other high level modular programming language, *i.e.* `for ,if else.`

MATLAB Script Files and Functions

Script files and functions are *batch* files which contain a series of >> input command lines appended together. These files are given the extension .m and are referred to in MATLAB slang as "**M-files**". Scripts are created in the user's text editor of choice and executed by typing the name of the script at the >> prompt.

The generic structure of a `function` declaration in MATLAB is given below:

```
function[output variables] = function_name(input variables)
```

MATLAB Plotting Routine

Plotting your results is achieved with the command

```
>> plot(t, v), xlabel(' x axis name'), ylabel(' y axis
name'), title(' Welcome to MATLAB for Dummies')
```

MATLAB® ODE Solvers

In addition to the many variants of the predictor-corrector and higher order Runge-Kutta (R-K) algorithms there are many algorithms in MATLAB which employ adaptive step-size control. The variable step-size algorithms use larger step sizes when the solution is changing more slowly. MATLAB provides two functions, called *solvers*, that implement R-K methods with adaptive step-size. These are the `ode23` and `ode45` functions. The `ode23` function uses a combination of 2nd order and 3rd order R-K, while `ode45` uses a combination of 4th and 5th order R-K methods and is an adaptive step-size algorithm based on the Runge-Kutta-Fehlberg integration scheme. In general, `ode45` is more robust and accurate than `ode23`, but it employs larger step sizes which may produce a solution plot that is not as smooth as the same plot produced with `ode23`. These two solvers are characterized as low-order and medium-order, respectively. Another solver in MATLAB is `ode113`, which is based on a variable-order algorithm. MATLAB contains four additional solvers: these are `ode23t`, `ode15s`, `ode23s` and `ode23tb`. These and the other solvers outlined above are listed below in Table 1.

Table 1 MATLAB ODE Solvers

Solver Name	Description
`ode23`	Non-stiff, low-order solver
`ode45`	Non-stiff, medium-order solver
`ode23s`	Stiff, low-order solver
`ode23t`	Moderately stiff, trapezoidal rule solver
`ode23tb`	Stiff, low-order solver
`ode15s`	Stiff, variable-order solver

Some of the solvers are especially useful for the solution of "stiff" systems. Recall, a stiff ODE is one whose response changes rapidly over a time scale that is short compared to the time scale over which the solution is desired. Stiff equations present additional strains to the numerical algorithm. A small step size is required to resolve the rapid gradient behaviors, while many steps are needed to obtain the solution over the longer time interval, and thus a large truncation error may accumulate. The `ode23` and `ode45` solvers handle moderately stiff ODEs nicely, but if you run into trouble, consult Table 1 and try using `ode15s`, a variable-order method; `ode23s`, a low-order method; `ode23tb`, a low-order method,; or `ode23t`, a trapezoidal method.

Solver Syntax

When used to solve the 1st order ODE $dy/dt = f(t,y)$ the basic MATLAB syntax is given by

```
[t,y] = ode23(' ydot', tspan, y0)
```

where `ydot` is the name of the function file whose inputs must be t and y, and whose output must be a column vector representing $f(t,y)$. The number of rows in this column vector must equal the order of the equation. The vector `tspan` contains the starting and ending values of the independent variable t, and optionally any intermediate value of t where the solution is desired. For example, if no intermediate values are desired `tspan = [t0, tf]`, where `t0` and `tf` are the desired starting and ending values of the independent parameter t. Using `tspan = [0, 5, 10]` informs MATLAB to find the solution at $t = 5$ and $t = 10$. You can solve equations backward by specifying `t0 > tf`. The parameter `y0` is the initial value $y(0)$. The function file must have two input arguments, t and y, even for equations where $f(t,y)$ is not a function of t. You need not use array operations in the function file since the ODE solvers call the file with scalar values for the arguments.

Example 1) "A simple first order ODE"

Let's solve an equation whose solution is known in closed form. Consider the model

$$0.1\dot{y} + y = 0$$

Use MATLAB solver `ode45` to find the free response with y(0) = 2. Compare with the exact analytic result.

First, recast the equation as

$$\dot{y} = -10y$$

Next define the following MATLAB function file, noting that the order of the input arguments must be t and y.

```
function ydot = eqn1(t,y)
% comments in MATLAB use the percent sign
% free response of a  first order model
ydot = -10*y;
```

The initial time is $t = 0$, so set `t0` to be 0. The time constant is 0.1, so the response will be 2% of its initial value at $t= 4(0.1)$ seconds, and 1% at $t = 5(0.1) = 0.5$ seconds, so we choose `tf` to be between 0.4 and 0.5 seconds, depending on how much of the response we wish to see. The analytic solution is given by

$$y(t) = e^{-10t}$$

The function is called as shown in the following MATLAB script, and the solution is plotted along with the analytic solution `y_true`.

```
% file eqnplot1.m
% note the .m extension of all MATLAB script files
[t,y] = ode45(' eqn1', [0,0.4], 2);
y_true = 2*exp(-10*t);
plot(t,y,'o',t,y_true), xlabel('t'), ylabel('y')
```

To run the above script we'd type

```
>> eqnplot1
```

Note, we need not generate the array `t` to evaluate `y_true` since `t` is generated by the `ode45` function. The solution is shown below in Figure 1, the R-K solution is denoted by the circles, while the analytic solution is shown by the solid line. Note that the step size automatically selected by `ode45` varies from 0.02 to 0.025.

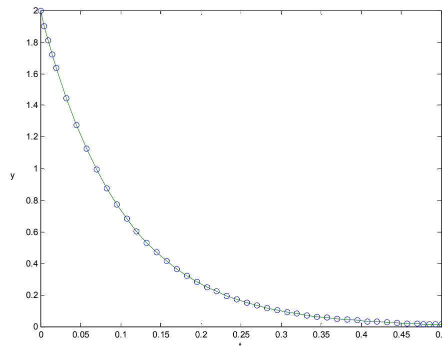


Figure 1 – Output for Example 1 Free Response of a 1st Order Model

Example 2) Moving on to Higher Order Equations

To use the ODE solvers to solve and equation higher than order first order, we must first write the higher order ODE as a system of 1st order ODEs. Consider the following 2nd order ODE

$$5\ddot{y} + 7\dot{y} + 4y = f(t)$$

Solving for the highest order derivative affords,

$$\ddot{y} = \frac{1}{5}f(t) - \frac{4}{5}y - \frac{7}{5}\dot{y}$$

Define two new variables x_1 and x_2 to be y and its derivative dy/dt as follows:

$$x_1 = y$$
$$x_2 = \dot{y}$$

Consequently,

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = \frac{1}{5}f(t) - \frac{4}{5}x_1 - \frac{7}{5}x_2$$

Employing matrix notation we can write the above system of two 1st order ODEs as follows:

$$\begin{Bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{4}{5} & -\frac{7}{5} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} + \begin{Bmatrix} 0 \\ \frac{f(t)}{5} \end{Bmatrix}$$

The above set of 1st order ODES is known as the so-called *Cauchy* form or more commonly is regarded as the *state-variable* form which in general reads

$$\{\dot{x}\} = [A]\{x\} + \{f\}$$

Now we code up a MATLAB function file that computes the values of the state variable derivatives and stores them in a column vector. To do this, we must have a function specified for *f(t)*. Suppose here for demonstration purposes that *f(t) = sin t.*

Then the required MATLAB function file is:

```
function xdot = example2(t,x)
% compute the derivative of two equations
xdot(1) = x(2);
xdot(2) = (1/5)*(sin(t)-4*x(1)-7*x(2));
xdot = [xdot(1); xdot(2)];
```

In fact, the above code could be written more compactly as:

```
function xdot = example2(t,x)
% computes the derivative of two equations
xdot = [x(2);(1/5)*(sin(t)-4*x(1)-7*x(2))];
```

Suppose we wanted to solve the system given above for

$$0 \le t \le 6$$
$$x(0) = 3$$
$$\dot{x}(0) = 9$$

Then the initial condition vector is [3, 9], to use `ode45` we key in,

```
% calling script for example2 function file
[t,x] = ode45('example2', [0,6],[3,9]);
%plot(t,x), xlabel('t'), ylabel('x')
gtext('x1'), gtext('x2')
plot(t,x(:,1),'+',t,x(:,2),'o'),xlabel('t'), ylabel('x')
gtext('x1 = +'), gtext('x2 = o')
```

and save the file as `example2.m`

Each row in the vector x corresponds to a time returned in the column vector t. If you type plot(t,x) you will obtain a plot of both x1 and x2 versus t. Note x is a matrix with two columns; the first column contains the values of x1 at the various times generated by the solver. The second column contains the values of x2. Thus, to plot only x1, type plot(t,x(:,1)). The colon represents all rows.

To execute the above script file we'd type `>> example2`

The resulting output of the above MATLAB script is shown below in Figure 2.

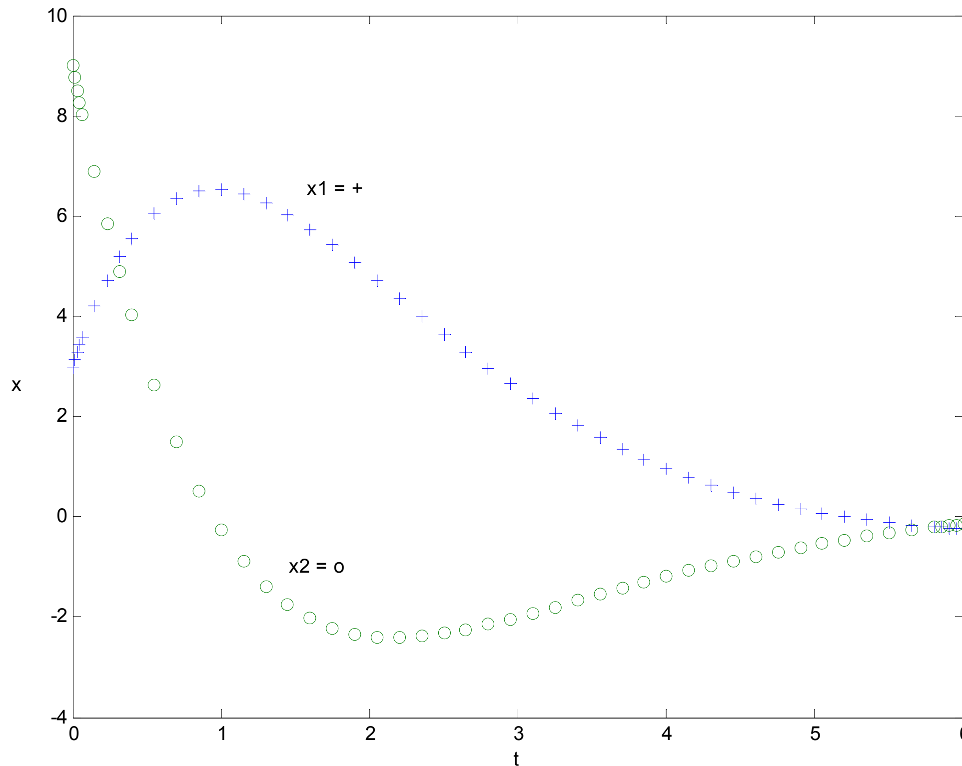


Figure 2 – State-Space Representation of a System of Two 1st Order ODEs

Example 3) "Use of Matrix Notation"

We can use matrix operations to reduce the number of lines to be typed into the MATLAB derivative function file. Consider, the following 2nd order ODE which is the classic spring-mass-damper model describing the motion of a mass connected to a spring, and viscous damping.

$$m\ddot{x} + c\dot{x} + kx = f(t)$$

where *x* is the displacement of the mass, *c* is the linear damping coefficient, *k* is the linear spring constant and *f(t)* is the forcing function. The above system can be placed into state-space (Cauchy) form by defining the following quantities:

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = \frac{1}{m} f(t) - \frac{k}{m} - \frac{c}{m} x_2$$

In state-space form the above system reads

$$\begin{Bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} = \begin{bmatrix} 0 & 1 \\ -\dfrac{k}{m} & -\dfrac{c}{m} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} + \begin{Bmatrix} 0 \\ \dfrac{1}{m} \end{Bmatrix} f(t)$$

Which in compact notation is given by

$$\{\dot{x}\} = [A]\{x\} + \{b\} f(t)$$

The following MATLAB function file illustrates how easily it is to use matrix operations within MATLAB. Here m = 1 kg, c = 2 N-s/m , k = 5 N/m and f = 10 N (a constant input step function).

```
function xdot = msd(t,x)
% function file for mass with spring and damping
% position is first variable, velocity is second variable
global c f k m
A = [0, 1; -k/m, -c/m];
B = [0; 1/m];
xdot = A*x+B*f;
```

The corresponding driver file is given by

```
% driver for k-m-c example
global c f k m
m = 1; c = 2; k = 5; f = 10;
[t,x]= ode23('msd', [0,5],[0,0]);
plot(t,x), xlabel('Time (sec)'), ylabel('Displacement (m)
and Velocity (m/s)')
gtext('Displacement')
gtext('Velocity')
```

Suppose we save the above script as `kmcdriver.m`, then we 'd execute is typing

```
>> kmcdriver
```

Figure 3 shows the results for initial conditions of $x_1(0) = x_2(0) = 0$.

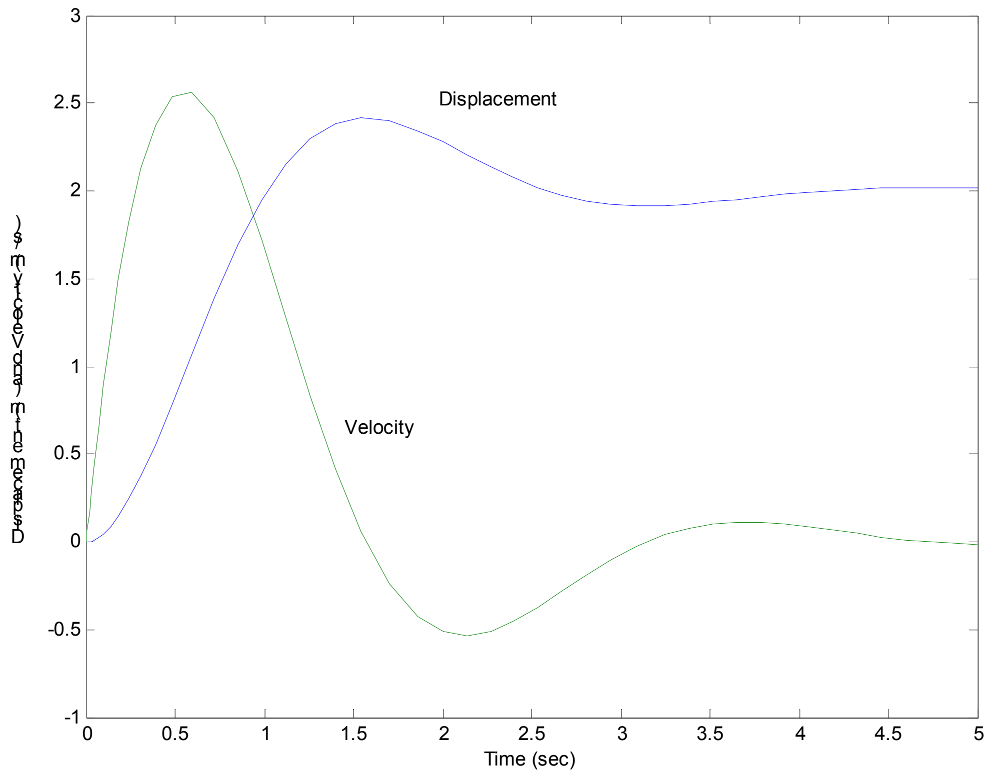
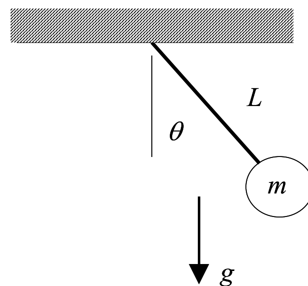


Figure 3 – Linear Spring Mass Damper System

<u>Example 4)</u> "The Pit and the Pendulum" starring Vincent Price

As an example of how MATLAB can be used to solve a non-linear ODE, consider the pendulum shown below:



The pendulum shown consists of a ball with concentrated mass, m attached to a rod of length L whose mass is assumed to be negligible with respect to that of the ball. The governing ODE for this dynamical system is given by the following 2nd order, non-liner ODE for the displacement angle θ:

$$\ddot{\theta} + \frac{g}{L}\sin\theta = 0$$

Note the above ODE is non-linear since

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} + \cdots$$

Now, to solve the above 2nd order ODE, recast as a system of two first order ODES via defining

$$x_1 = \theta$$
$$x_2 = \dot{\theta}$$

Hence,

$$\dot{x}_1 = \dot{\theta} = x_2$$
$$\dot{x}_2 = \ddot{\theta} = -\frac{g}{L}\sin x_1$$

Then the following MATLAB function file would be appropriate (remembering that the output `xdot` must by a column vector)

```
function xdot = pendul(t,x)
global g L
xdot = [x(2); -(g/L)*sin(x(1))];
```

Which is called in the following driver script using a pendulum length of $L = 1$ m and acceleration due to gravity $g = 9.81$ m/s^2 as `global` MATLAB parameters.

```
%driver for non-linear pendulum
global g L
g = 9.81; L = 1;
[ta, xa] = ode45('pendul', [0,5], [.5,0]);
[tb, xb] = ode45('pendul', [0,5], [.8*pi, 0]);
plot(ta,xa(:,1),tb,xb(:,1)),xlabel('Time(sec)')
ylabel('Angle (rad)')
```

```
gtext('Case 1'), gtext('Case 2')
```

Where the vectors `ta` and `xa` contain the results of the case where $\theta(0) = 0.5$ and the vectors `tb` and `xb` contain the results of the run with $\theta(0) = 0.8\pi$. The results are shown below in Figure 4 assuming we named the file `pendulumstudy.m`

```
>> pendulumstudy
```

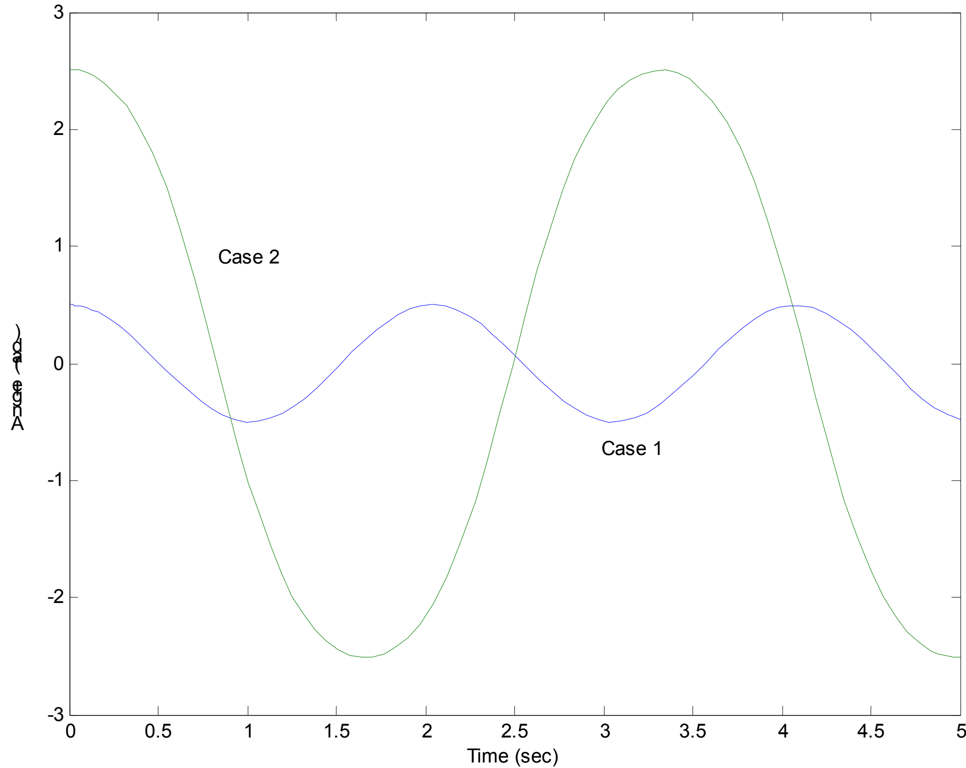


Figure 4 – Output for Non-linear Pendulum Problem

Note, from Figure 4, for Case 1, the period is a little greater than 2 seconds. This is in agreement with the value predicted by a linearization of the governing non-linear ODE. Linearizing, for small angles, $\sin\theta \sim \theta$,

Thus,

$$\ddot{\theta} + \frac{g}{L}\theta = 0$$

Which is a linear ODE and afford the following free response for $\theta(0) = \theta_o$:

$$\theta = \theta_o \cos\left(\sqrt{\frac{g}{L}}\ t\right)$$

Thus the amplitude of oscillation is θ_o while the period is given by

$$\mathrm{T} = \frac{2\pi}{\sqrt{\frac{g}{L}}}$$

Which for L = 1 m and g = 9.81 m/s^2 affords T = 2 seconds, offering confidence in the solution given for Case 1 by `ode45`. For Case 2 where $\theta(0) = 0.8\pi$, the period of the MATLAB R-K solution is about 3.3 seconds in duration.

This illustrates an important property inherent to non-linear differential equations, the free response of a linear equation has the same period for any initial conditions, while in contrast, the form of the free response of a non-linear ODE often depends on the particular values of the initial conditions.

Example 5) A non-linear dynamics problem

As another illustration of the behavior of non-linear systems, consider the following ODE:

$$\frac{d^2x}{d\tau^2} + 0.4\frac{dx}{d\tau} + x + 0.5x^3 = 0.5\cos(0.5\tau)$$

which describes a spring-mass-damper system with m = 1 kg, c = 0.4 N-s/m and the spring force is given by the cubic polynomial comprising the last tow terms on the left hand side of the non-liner ODE:

$$f(x) = x + 0.5x^3$$

This is a typical model of materials whose spring deflections do not obey Hooke's linear relationship of $f(x) = kx$. Many real world materials behave non-linear, including composites and polymers. To solve the above ODE is MATLAB, recast as a system of 1st order ODEs using the following substitution

$$y = \frac{dx}{d\tau}$$

Consequently,

$$\frac{dx}{d\tau} = 0.5\cos(0.5\tau) - x - 0.5x^3 - 0.4y = F(\tau, x, y)$$

The MATLAB function file and driver script are as follows:

```
%function file for non-linear spring mas damper ODE
function xdot = nonlin(t,x)
xdot = [0.5*cos(0.5*t)-x(2)-0.5*x(2)^3-0.4*x(1);x(1)];

%script file for non-linear spring problem
t0 = 0;
tf = 40;
x0 = [0,0.05]';
[t,x] = ode45('nonlin',t0,tf,x0)
x1 = x(:,1);
x2 = x(:,2);
plot (x2,x1)
```

Suppose we named the above script `nonlinspring.m,` then keying in

```
>>nonlinspring
```

Would yield the results of the shown on the following page in Figure 5. Figure 5 shows the Phase Plane ($dx/d\tau$ vs. x) representation of the non-linear spring system, where the iso-cline curve indicates the trajectory of the solution starting at the initial condition location and spiraling inward until the limit cycle (stable state) of the system. Here, the limit cycle is reached in less than two cycles.

Figure 5 – Phase Plane Plot of Non-liner Spring Problem