Applied Engineering Problem Solving (CHEN.3170)

Part I: Matlab Overview

Lesson 1: Getting Started With Matlab

The first part of this course focuses on Matlab fundamentals and on using Matlab to evaluate and plot 1-D and 2-D functions -- that is, f(x) versus x and f(x,y) versus x and y. In addition, many of the elements of a standard programming language (function files, program flow control, input and output processing, etc.) will be highlighted so that you will become comfortable with using Matlab for all your programming needs for addressing real engineering design and analysis problems.

There are a number of good books available for learning the fundamentals of Matlab as well as a host of websites devoted to this goal. The MathWorks home page at www.mathworks.com is a good starting point if you are looking for more information here. In addition, the ***help*** facility in Matlab is a great tool for detailed instructions on using specific functions and commands, and we will refer to the built-in Matlab ***help*** quite often throughout this course.

The two texts chosen for this course,

Amos Gilat, **Matlab -- An Introduction with Applications**, John Wiley & Sons

Steven C. Chapra, **Applied Numerical Methods with Matlab for Engineers and Scientists**, McGraw Hill

will be the primary sources for the technical content for this course. The reason for two texts is that they complement one another quite nicely, with each filling in the gaps of the other. Gilat's text is primarily about Matlab with a rather introductory treatment of the numerical methods needed for solving many engineering analysis problems. On the other hand, the book by Chapra gives a good overview of the numerical tools needed by design engineers, but only gives a brief introduction to many of Matlab's programming features. Together, however, they should provide a good foundation for enhancing your engineering problem-solving capabilities.

Note: The book by Gilat is an introductory text on how to use Matlab within a variety of engineering applications. It assumes no prior knowledge with Matlab or any other programming language. In fact, this text is often used within freshmen Introduction to Engineering courses. As such, it is easy to read and understand and it illustrates many of the key features within the Matlab programming environment quite nicely via a series of interesting and practical applications. I chose this introductory text for this junior-level course because I feel it is one of the best "getting started" Matlab texts and, once you are comfortable with the fundamentals, the extensive internal documentation that is available within Matlab negates any real need for a more advanced or comprehensive text on this subject. Indeed, you will be expected to go through Gilat's introductory text fairly rapidly, but there should be enough time for you to develop sufficient skills and confidence with Matlab to proceed to the numerical methods portion of this course. Overall, this is a great text that, if studied cover-to-cover, will give you all the foundation you need to become a competent and confident Matlab user -- and all this can be done in about five 2-hour sessions (or less).

It is expected that you will carefully read and study both texts. This is absolutely essential, since much of the course content is presented in the course textbooks rather than during a traditional lecture class. My role will be to highlight key aspects of the assigned reading and to clarify --

primarily via example -- how to use Matlab and the numerical methods discussed here to solve a variety of typical engineering problems. Thus, the brief lecture notes and the worked examples given here will assume that you have already completed the assigned readings. Since I am assuming a mature, intelligent audience for this course, there is no need to repeat everything from the assigned reading. Instead, my role will be to help you understand and apply this material in a variety of practical situations.

The goal of this first unit, "Getting Started with Matlab" is to introduce you to some of the subtle aspects of the Matlab language and to highlight some of its key features. At this point, you should read Chapters 1 and 2 in Chapra and Chapters 1-2 and 4-5 in Gilat, paying particular attention to the following:

- Interactive processing within the Matlab command window
- Defining arrays via direct assignment, by using the colon operator, and with the ***linspace*** command
- Scalar versus vector arithmetic in Matlab (this is a biggie!!!)
- Creating simple 2-D plots
- Writing and running simple Matlab programs (script files)
- Use of the ***help*** command
- Etc., etc., etc.

One of the most important applications of a computational tool like Matlab is for function evaluation and plotting. The ability to easily evaluate complicated expressions and to visualize their functional dependence is an essential component of most engineering analyses. We will focus much of our early study of Matlab towards this goal. In fact, with your first reading assignment now complete, you should now have the capability to do some simple function evaluation and plotting in Matlab.

To illustrate this, we will give a series of examples of increasing complexity to highlight and expand upon many of the skills you have attained from your first reading assignment. We will start by doing some simple interactive analyses within the command window and then quickly move to writing simple Matlab programs -- since this is clearly the only way to go for any analysis requiring more than a few lines of code.

As our first example, consider the simple pendulum model described in the available pdf file **pendulum_dynamics.pdf**. As developed in detail, the linearized model for the specific case described here gives the pendulum angular position versus time as

$$\theta(t) = \frac{\pi}{6} e^{-t} \left(\cos 3t + \frac{1}{3} \sin 3t \right)$$

and the angular velocity as

$$\omega(t) = \theta'(t) = -\frac{5}{9} \pi e^{-t} \sin 3t$$

Through the following interactive sessions, we can identify several things about this physical system (the Matlab commands and associated responses are given here using an 8-pt Courier font):

1. Evaluate the initial position and initial angular velocity of the pendulum.

```
>> format compact
>> to = 0
to =
     0
>> poso = (pi/6)*exp(-to)*(cos(3*to)+(1/3)*sin(3*to))
poso =
    0.5236
>> poso*180/pi
ans =
   30.0000
>> velo = (-5/9)*pi*exp(-to)*sin(3*to)
velo =
     0
```

Notice that these values are consistent with the initial conditions given in the problem statement in **pendulum_dynamics.pdf**. The first line using the ***format*** command simply eliminates an extra blank line that occurs after each command that is executed within the command window (to save space in this listing).

2. Evaluate the total energy in the system at t = 0, where

$$E_{tot} = mgL\left(1-\cos\theta\right)+\frac{1}{2}mL^2\left(\theta'\right)^2$$

```
>> m = 1;   g = 10;   L = 1;
>> Etoto = m*g*L*(1-cos(poso)) + 0.5*m*L^2*velo^2
Etoto =
    1.3397
```

Notice the semicolon that is used to eliminate echoing the result to the screen, the use of multiple Matlab commands on a single line, and the use of the previously defined initial position and velocity variables, poso and velo. Also note that every variable on the right hand side (RHS) of the assignment operator (the = sign) must be already defined. We can always see what variables are currently defined, along with their size and class, with the ***whos*** command. This is a very useful command that is used a lot, especially when debugging programs. For example, at this stage in our interactive session, we have the following variables in memory:

```
>> whos
  Name        Size                    Bytes  Class
  Etoto       1x1                         8  double array
  L           1x1                         8  double array
  ans         1x1                         8  double array
  g           1x1                         8  double array
  m           1x1                         8  double array
  poso        1x1                         8  double array
  to          1x1                         8  double array
  velo        1x1                         8  double array

Grand total is 50 elements using 148 bytes
```

These are all scalar variables (1x1 arrays) and we can print any variable to the screen by simply typing its name. For example,

```
>> g
g =
    10
```

3. We have now evaluated $\theta(t)$, $\theta'(t)$, and $E_{tot}(t)$ at the specific time t = 0. However, what I really want to do is to evaluate these at a whole sequence of different times, t_i, and then plot $\theta(t_i)$ versus t_i. For example, let t take on values of 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, and 5 seconds. We can store these values of time in a vector variable t. In Matlab, we can generate this vector in a number of ways, as follows:

```
>> t = [0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5]
t =
   0   0.5000   1.0000   1.5000   2.0000   2.5000   3.0000   3.5000   4.0000   4.5000   5.0000
>> t = 0:0.5:5
t =
   0   0.5000   1.0000   1.5000   2.0000   2.5000   3.0000   3.5000   4.0000   4.5000   5.0000
>> t = linspace(0,5,11)
t =
   0   0.5000   1.0000   1.5000   2.0000   2.5000   3.0000   3.5000   4.0000   4.5000   5.0000
```

Note that all these vectors are identical. Explicit entry within brackets is appropriate for a small number of values or when the increment is not uniform. The colon operator (which I refer to as the repeat operator) is useful when you want to specify the desired increment between entries in the vector. Finally, the ***linspace*** command is best when you want to explicitly specify the start and end values and the number of values in the vector, but don't really care to set the increment. Often, this is the best choice -- for example, what happens in the second case with $\Delta t = 0.4$ seconds?

Note that if you want to find out how to use a specific Matlab command, you simply type ***help command_name***. For example, we can find more information about ***linspace***, as follows:

```
>> help linspace

 LINSPACE Linearly spaced vector.
    LINSPACE(X1, X2) generates a row vector of 100 linearly
    equally spaced points between X1 and X2.

    LINSPACE(X1, X2, N) generates N points between X1 and X2.
    For N < 2, LINSPACE returns X2.

    See also LOGSPACE, :.
```

Now, getting back to the task at hand, we have a vector of 11 different times, t_i for i = 1, 2, … 11, stored in vector variable t. We can access any discrete time of interest by specifying the index i. For example, $t_5 = 2$ seconds or, in Matlab, we have

```
>> t(5)
ans =
     2
```

Thus, to access the information stored in the i^{th} element of the vector, we simply specify the index or element number of interest. This suggests that if we want to evaluate $\theta(t)$ for several different values of t, then the position versus time also becomes a discrete vector quantity, or

$$\theta(t) \rightarrow \theta(t_i) \rightarrow \theta_i$$

Thus, the discrete subscript notation, θ_i, represents the i^{th} value of the vector that stores the position information. To evaluate $\theta(t)$ for several discrete values of t in Matlab, we can use either scalar arithmetic or vector arithmetic, as follows:

Scalar Form (requires a loop over the number of time points)

```
>> pos1 = zeros(size(t));
```

```
>> for i = 1:length(t)
     pos1(i) = (pi/6)*exp(-t(i))*(cos(3*t(i)) + (1/3)*sin(3*t(i)));
   end
>> pos1
pos1 =
  0.5236  0.1281  -0.1816  -0.0627  0.0614  0.0283  -0.0202  -0.0122  0.0064  0.0050  -0.0019
```

Vector Form (evaluates the position for all time points in a single step)

```
>> pos2 = (pi/6)*exp(-t).*(cos(3*t) + (1/3)*sin(3*t));
>> pos2
pos2 =
  0.5236  0.1281  -0.1816  -0.0627  0.0614  0.0283  -0.0202  -0.0122  0.0064  0.0050  -0.0019
```

It is important as a new user of Matlab to fully understand exactly what is happening in these two examples. In the first case, we compute one position at a time, θ(i) = f{t(i)}, and in the vector form, we can think of the arithmetic as **θ** = f{**t**}, where the bold emphasis represents a vector quantity. Clearly the scalar form, with one computation per step, requires that we repeat the computation for each value of i = 1, 2, … length(t) -- thus, the need for the standard ***for … end*** structure in Matlab. The first time through the loop i = 1, the second time i = 2, and so on up to i = length(t), where the default increment for the looping variable i is unity, since it represents an integer index that goes from 1, 2, … 11 (for the present case). Note that the initial ***zeros*** command for the scalar form simply initializes the vector of interest (allocates memory space and sets all the entries to zero).

Now, for the vector case, because t is a vector variable, then exp(t), cos(3*t), and sin(3*t) are all vectors. When evaluating functions containing these quantities, we need to be careful to do element-by-element operations (as needed). Since, by definition, matrix addition and subtraction are done element-by-element (more on this later), the only explicit dot arithmetic or element-by-element operation needed here is for the multiplication of exp(t) with (cos(3*t) + (1/3)*sin(3*t)). Thus, the **.*** operation between these two terms is absolutely essential. Note that, in this case, without the dot you would get a Matlab error, "inner matrix dimensions must agree". However, don't always count on Matlab to tell you when you have done something wrong -- because sometimes it will do valid calculations that are not at all what you wanted. So it is up to you to specify the correct form of arithmetic in Matlab!!!

Note: We will spend lots of time talking about matrix arithmetic (Matlab's default form) a little later in the course. For now, for most function evaluation and plotting exercises (which is our current focus), you will want to perform element-by-element vector operations wherever possible -- they are much more efficient and easier to code than the corresponding scalar code, as shown above in the comparison of the scalar and vector forms (see further discussion of this point in your texts).

Before getting too far off the track, let's complete the goal of Task #3 -- that is the plotting of θ(t) versus t. We can do this with a series of Matlab plotting commands, as follows (see ***help plot***):

```
>> plot(t,pos2*180/pi),grid
>> title('Pendulum Dynamics')
>> xlabel('Time (sec)'),ylabel('Angular Position (degrees)')
```

with the results as shown in Fig. 1 (note the units conversion directly within the ***plot*** command).

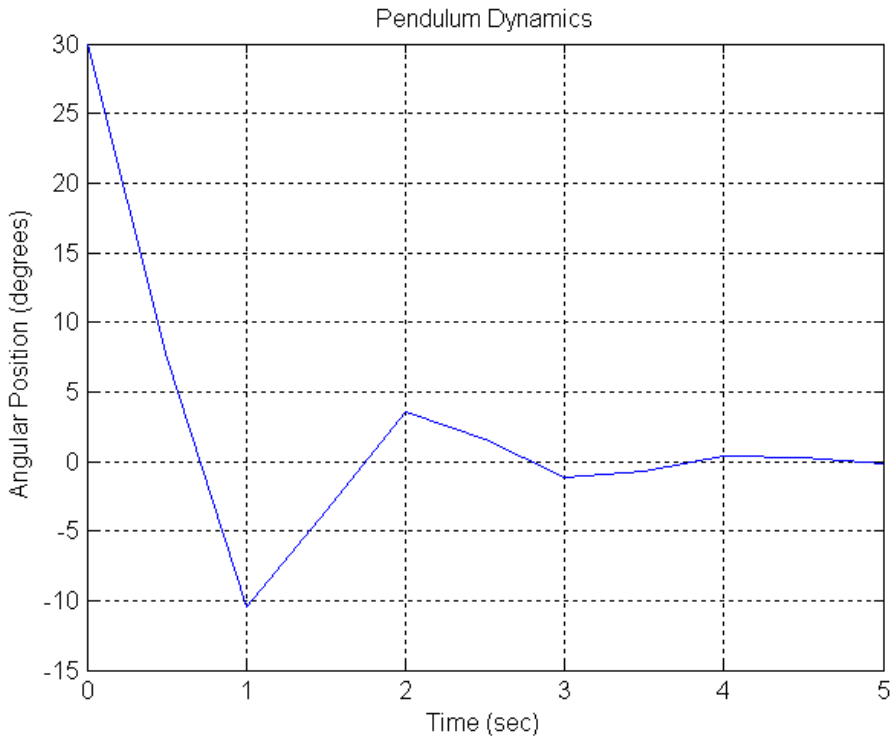

Fig. 1 First try at plotting θ(t) for a simple linear pendulum model.

Well, this plot is sort of what was expected. Since the linearized pendulum model has some friction, we expect the pendulum to eventually return to θ = 0 degrees after oscillating about this equilibrium point, with continually decreasing amplitude. This, of course, can also be seen directly from the equations being plotted -- the e^{-t} term gives the decay and the cos(3t) and sin(3t) terms give the oscillatory behavior. This is basically what is observed, but clearly the plot is too coarse to see any fine resolution (note that Matlab draws straight lines between individual data points). This is an example of a curve that has a bad case of what I call the "jaggies". One solution to this problem is to simply use a finer time discretization -- maybe 101 points instead of 11 points, for example. To do this in Matlab's interactive mode is somewhat tedious, since we would have to redefine the t vector and then re-execute several commands (the Command History Window helps here). However, since we also want to do some other analyses with this model anyway, it is now time to write our first Matlab program.

4. I recommend strongly that any time you have a problem that requires more than a few lines of code, you should put that code, along with appropriate internal documentation, into a Matlab script file. Thus, the goal of Task #4 is to take the above interactive session and put the desired commands into an m-file using the Matlab built-in editor.

This was done by opening a new m-file from the Matlab command window, typing the desired commands within the editor, and then saving the file with the name **pendulum_1.m** in the desired directory. Once the file is available, we can run it by simply typing the name of the script file at the Matlab prompt, or

```
>> pendulum_1
>>
```

No output is echoed to the screen because all the assignment commands in **pendulum_1.m** have semicolons at the end of the line (this is the usual case). However, a number of plots are produced, and these will be described shortly.

Note: Matlab has a preset default directory that you should change at the beginning of each Matlab session to have the "current directory" point to the location of your m-files (e.g. D:\user_directory for a flash drive, etc.). If Matlab returns with an "undefined function or variable" error when you enter your m-file name (without the .m extension), be sure to check that the spelling and the current directory are correct. The ***dir*** and ***pwd*** commands are useful for fixing these types of problems. See further discussions of these types of commands in your texts or within the Matlab ***help*** facility.

Our **pendulum_1.m** file is listed in Table 1. Note that it does essentially the same tasks as above, with a number of embellishments, as follows:

a. Several lines of comments are added to the top of the file to identify the purpose of this m-file. The comments in Matlab follow the **%** sign, and they can be on separate lines or they can be on the same line following a Matlab command. In general, all the code you write should be well documented with internal comments!!!

b. There are at least three primary sections to each code -- an input section, a computational section, and an output section. The input section identifies the key parameter values to be used in the calculations. These variables and their associated units should be appropriately defined via comments directly after their assignment. The computational section, which immediately follows the input section, does the real numerical work. Note that here we only included the vector form since it is more efficient and easier to code -- as long as we are careful to use the appropriate dot or element-by-element arithmetic. In this section, we evaluate the position and velocity of the pendulum point mass as well as the kinetic, potential, and total energy in the system versus time. Note that the equations implemented here are more general in that they are direct functions of the mass, m, length, L, friction coefficient, c, gravitational constant, g, and the initial angular position and velocity, θ_o and ω_o. In general, when performing engineering analysis, you should maintain as much generality in your computations as appropriate for the system under study. With the current equations, we could easily study the effects of varying any one of these design and operational variables.

Table 1 Listing of the pendulum_1.m program.

```
%
%   PENDULUM_1.M  Evaluate and plot the dynamics of a simple linear pendulum
%
%   This example evaluates and plots the angular position and velocity of a pendulum
%   with a point mass at the end.  The mathematical model was linearized about the
%   equilibrium position (pendulum at rest).  The model was developed by performing
%   an appropriate force balance and the resulting linear constant coefficient 2nd
%   order IVP was solved analytically.  The exact expressions for the angular position
%   and velocity versus time are evaluated and plotted in this file (see the lecture
%   notes for the formal development and analytical expressions).
%
%   The primary purpose of this file is to serve as a typical example for function
```

```
%   evaluation and plotting in Matlab.  This is a common need for many engineering
%   design and analysis problems.  Thus, the student should master this capability
%   ASAP to become an effective "problem-solver" in school and on the job.  In this
%   particular example, as a bonus, you may also gain a little insight into the
%   actual dynamics of pendulum motion.  This is my intention anyway...
%
%   File prepared by J. R. White, UMass-Lowell (last update: August 2017)
%

     clear all;   close all;   nfig = 0;
%%
%   define problem parameters
     m = 1;                    % pendulum mass (kg)
     c = 2;                    % friction coeff (kg/s)
     L = 1;                    % length of pendulum arm (m)
     g = 10;                   % gravitational acceleration (m/s^2)
     poso = 30*pi/180;         % initial position (degrees converted to radians)
     velo = 0;                 % initial velocity (radians/s)
%%
%   evaluate roots of characteristic equation
     a = -c/(2*m);   b = sqrt(g/L - a^2);
%
%   compute equation coefficients
     c1 = poso;          c2 = (velo - a*c1)/b;
     d1 = b*c2 + a*c1;   d2 = a*c2 - b*c1;

%   define discrete time domain variable (with time in seconds)
     Nt = 101;   to = 0;   tf = 5;   t = linspace(to,tf,Nt);
%
%   evaluate analytical expressions for angular postion and velocity
%   (be careful with the "dot" operations -- only need one dot in each equation)
     pos = exp(a*t).*(c1*cos(b*t) + c2*sin(b*t));
     vel = exp(a*t).*(d1*cos(b*t) + d2*sin(b*t));
%
%   let's also compute the energy components versus time
     Ep = m*g*L*(1 - cos(pos));    % potential energy
     Ek = 0.5*m*L^2*vel.^2;        % kinetic energy
     Etot = Ep + Ek;               % total energy
%%
%   now let's plot the results in a variety of ways
%   position (in degrees) vs time
     nfig = nfig+1;   figure(nfig)
     plot(t,pos*180/pi,'r-','LineWidth',2),grid
     title('PENDULUM\_1: Pendulum Dynamics  --  \theta(t) vs t')
     xlabel('Time (sec)'),ylabel('Angular Position (degrees)')
%   position and velocity vs time  (use subplot command)
     nfig = nfig+1;   figure(nfig)
     subplot(2,1,1),plot(t,pos*180/pi,'r-','LineWidth',2),grid
     title('PENDULUM\_1: Pendulum Dynamics  --  \theta(t) and \omega(t) vs t')
     ylabel('Angular Position (degrees)')
     subplot(2,1,2),plot(t,vel*180/pi,'g--','LineWidth',2),grid
     xlabel('Time (sec)'),ylabel('Angular Velocity (deg/sec)')
%   just for fun, let's look at the phase plane diagram (velocity vs position)
     nfig = nfig+1;   figure(nfig)
     plot(pos*180/pi,vel*180/pi,'r-','LineWidth',2),grid
     title('PENDULUM\_1: Phase Plane Plot  --  \omega(t) vs \theta(t)')
     xlabel('Angular Position (degrees)'),ylabel('Angular Velocity (deg/sec)')
%   finally, let's plot the energy components (units -> J = N-m)
     nfig = nfig+1;   figure(nfig)
     plot(t,Ep,'r-',t,Ek,'g--',t,Etot,'b-.','LineWidth',2),grid
     rr = axis;  rr(2) = 3;  axis(rr);
     title('PENDULUM\_1: Energy Components  --  E_{tot} = E_p + E_k')
     xlabel('Time (sec)'),ylabel('Energy (J)')
     legend('Potential Energy -> E_p','Kinetic Energy -> E_k','Total Energy -> E_{tot}')
%
%   end of problem
```

The third part of almost every program is the output section. Here we create a variety of plots to help us visualize the full dynamics of the linearized pendulum model. The first plot of position versus time, as shown in Fig. 2, should be compared directly to the one generated during our interactive session (see Fig. 1). They are, of course, quite similar, except the one generated from the script file has a much finer time resolution (101 points), which leads to a much better representation of θ(t) versus t.

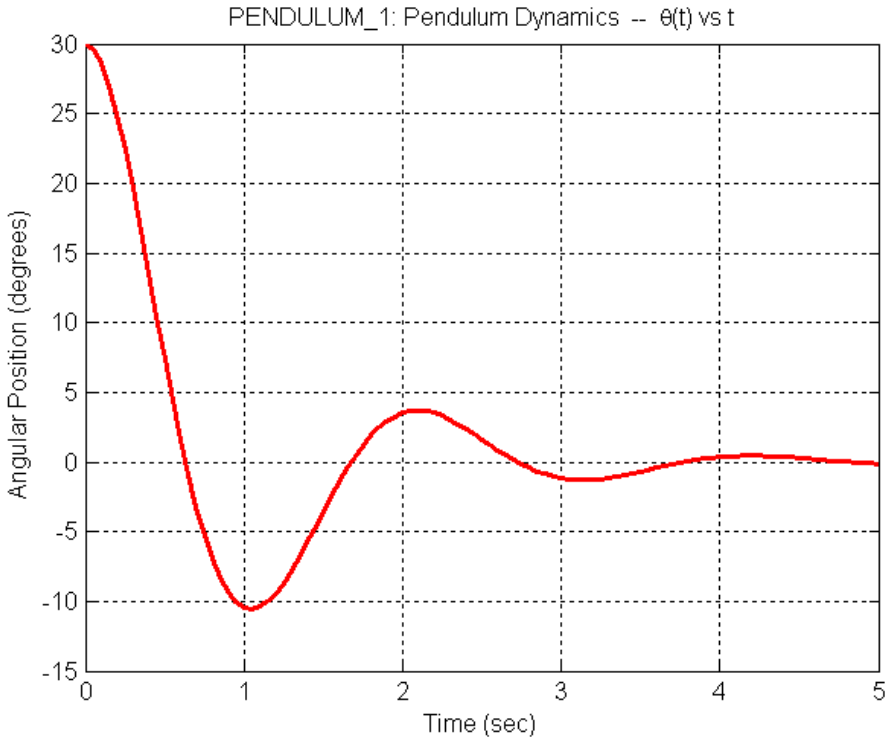

Fig. 2 Angular position versus time for a simple linear pendulum model.

Continuing the discussion of the figures generated in **pendulum_1.m**, we see that Fig. 3 shows both the position and velocity versus time on the same page. This was accomplished using Matlab's ***subplot*** command. Note here, that θ(t) and ω(t) on the same axis is probably not appropriate since they have different units. However, we often like to see these on the same plot for comparison purposes -- and the ***subplot*** command allowed us to do this in a straightforward way. Both θ(t) and ω(t) behave as expected, and careful consideration of Fig. 3 shows that the maxima and minima in the position plot occur at the points where the angular velocity is zero (as expected).

Figure 4 presents a slightly different view of the same information as contained in Fig. 3. Here we plot ω(t) versus θ(t). This type of plot is often referred to as a phase plane plot, where two dependent variables are plotted in the x-y plane. This eliminates the independent variable from the visualization and shows how, in this case, the pendulum angular velocity and position are related, independent of time. The spiraling in motion towards the final equilibrium point ($\theta_{eq} = 0$ and $\omega_{eq} = 0$, for this case) is a common pattern that is observed for many stable unforced linear systems.

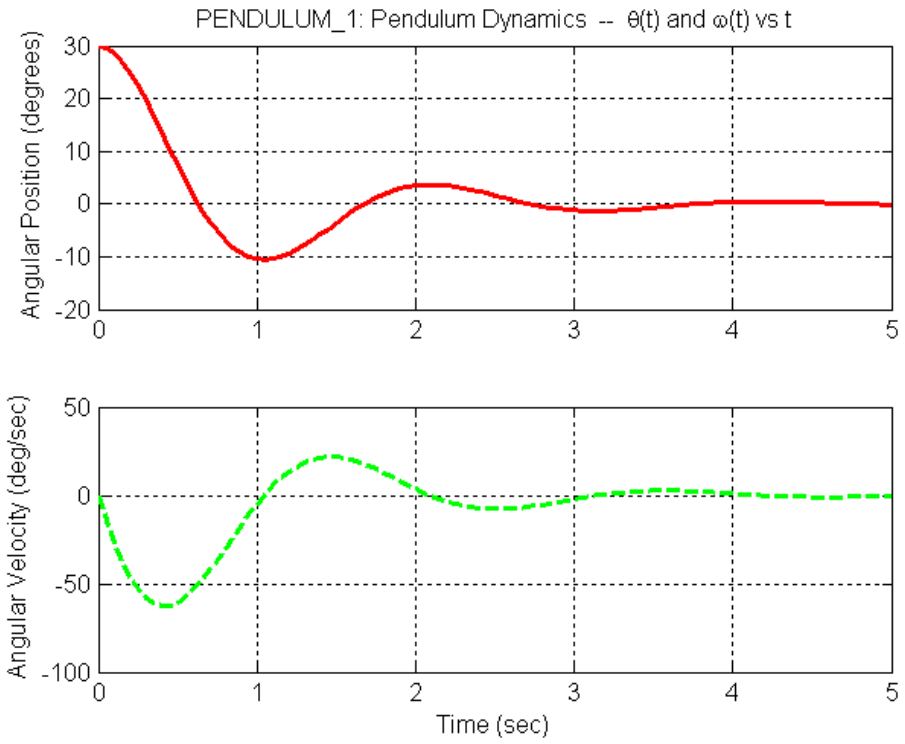

Fig. 3 Pendulum angular position and velocity versus time.

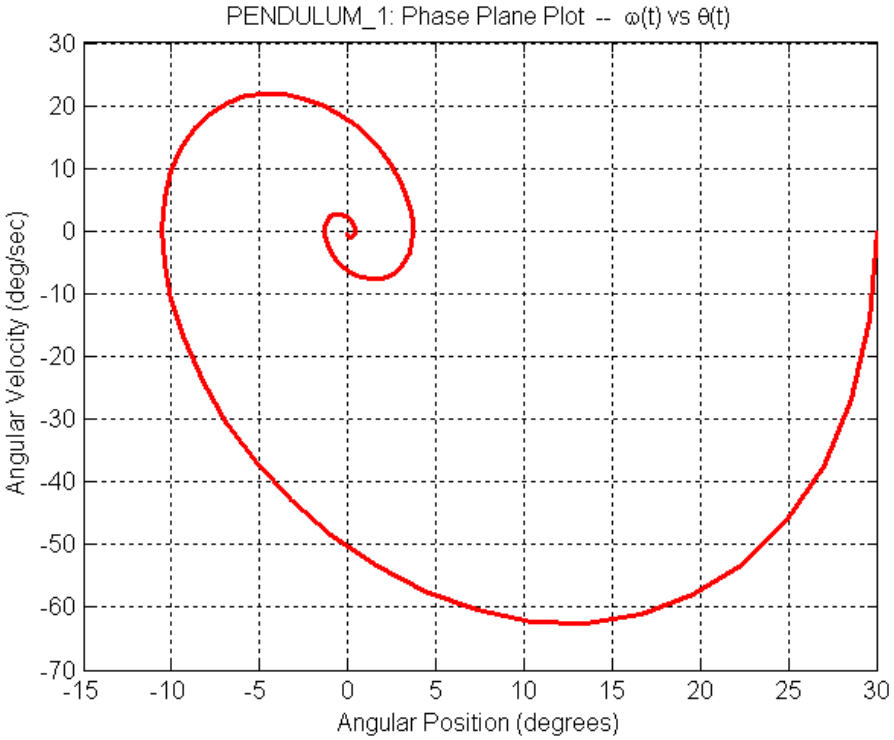

Fig. 4 Phase plane plot of pendulum dynamics.

The last plot, Fig. 5, shows the three time-dependent energy terms computed in **pendulum_1.m**. As expected, the total energy at each point is the sum of the kinetic and potential energies and, because of friction in the system, the total energy eventually decays to zero. As apparent, part of the initial potential energy in the system gets converted into kinetic energy, with the remainder being lost to friction. After the first minimum in $K_p(t)$, some of the available kinetic energy gets transformed back into potential energy as the pendulum swings towards is maximum negative position (about –10 degrees). These two energy components continue to oscillate with an ever-decreasing magnitude due to the friction loss in the system. Note also that, since friction is proportional to the velocity, there is less friction loss when the kinetic energy term is small, and this explains why the total energy curve has some relatively flat plateau regions. Of course, the overall behavior is exactly what we expected for this simple under-damped system.

c. Several common plotting examples are given in **pendulum_1.m**, including the use of the ***figure*** command to open multiple figure windows, the ***subplot*** command to place multiple axes on the same figure, the ***legend*** command to help label plots with multiple curves, the ***axis*** command to set user-defined axis limits, and the setting of various line styles and other plot properties and enhancements -- such as the use of Greek letters in the titles and plot labels and the use of the underscore character to produce subscripts. You will want to become familiar with a variety of ways to enhance the information content of your plots. Remember that a "picture is worth a thousand words" -- but only if the picture or plot is properly labeled!!!

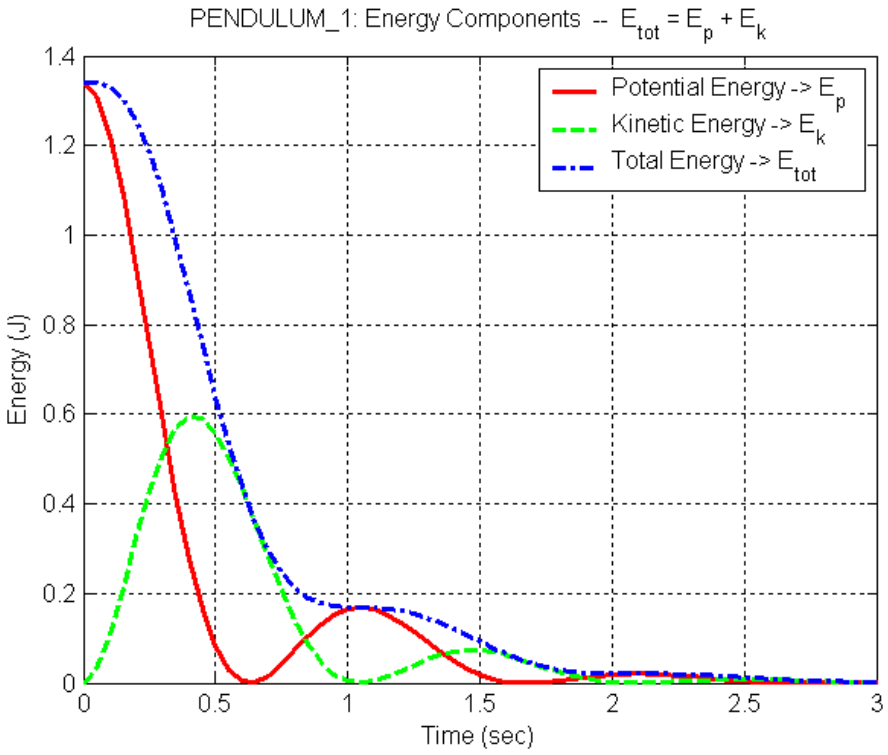

Fig. 5 Energy components for pendulum dynamics problem.

Note: I commonly set the LineWidth property to 2 within my Matlab plots. This is done primarily for presentation purposes, so the audience can see the curves better when viewing in a classroom environment at some distance. However, this attribute is also a pretty good choice for normal written documentation as well. This, of course, is a personal preference, and you can adjust this property, as desired, for your particular needs.

Note: Another aspect of writing Matlab programs that you should consider is to develop the codes gradually. For example, I did not write all of **pendulum_1.m** and then try to run it. Instead, I usually write the beginning comments first to present an overview of what is to follow. Then I compose the section that defines all the problem parameters and corresponding units. Once I have a good handle of how to proceed and the base data for the calculations, I then start the computational section. Once the base code for evaluation of the equations are typed, I run the code, look at the intermediate results (from the command prompt), do any necessary hand calculations, etc., etc. -- all to validate the actual computations that are performed. When this seems okay, I then make the first few plots that use the key parameters in the problem -- $\theta(t)$ versus t and $\omega(t)$ versus t, in this case. Upon careful study and rationalization that these are okay, I then proceed to complete any other desired analyses (the phase plane plot and the energy components versus time plot, for example). Finally, when everything is basically working, I go back and make the code easier to read with proper alignment of the commands and comments, and easier to understand with the addition of more explanations, as needed.

Breaking each program into a number of small explicit tasks makes the overall challenge more manageable, and it also leads to better code, fewer mistakes, and less overall time and frustration in your programming assignments. Eventually, you will develop your own programming style -- one that works well for you -- but just remember that the phrase "divide and conquer" is a nice philosophy to follow for many design and analysis problems, including those that involve programming in Matlab!!!

Well, this completes our first illustrative example that uses Matlab as a visualization tool to assist in understanding the behavior of physical systems. I hope that your understanding of simple pendulum dynamics is improved somewhat after studying the figures generated here. In this case, the system is simple enough that we had a pretty good idea of what to expect -- so no real surprises were observed. However, for many systems, our prior understanding may be somewhat limited, and the evaluation and plotting of the functional relationships within the system will then contribute significantly to our overall knowledge and understanding of the physical behavior of the system. With Matlab, doing this is rather straightforward, and the payback in terms of the new insight gained is usually well worth the small effort needed to produce the actual visualizations…

To complete this "Getting Started…" lesson, we provide three more examples of function evaluation and plotting in Matlab. These additional illustrative applications are available in separate pdf files, as follows:

maxwell_1.pdf -- Maxwell Boltzmann Distribution

pipe_flow_1.pdf -- Laminar vs. Turbulent Flow in a Pipe

pipe_insulation_1.pdf -- Pipe Insulation Considerations

After you have completed your reading assignment from the course texts and after reviewing the above examples, you should now be ready to attempt HW #1 on your own (see **hw1xxx.pdf**, where the xxx refers to the current semester and year). This homework asks you to solve a few problems with Matlab that are roughly similar to the applications given above and to the examples from your textbooks. You should solve each problem as requested in the assignment sheet in a separate Matlab script file. You should include the m-file, the resultant plots, any hand calculations and manipulations, and a brief description of the results as components of the complete solution for each problem -- and these should be all organized into a professional HW package to be turned in by the announced HW deadline.

Well, GOOD LUCK with HW #1!!! When complete, you should have a much better understanding of some of Matlab's syntax and fundamental capabilities. The intent is for you to get fairly well acquainted with simple 1-D function evaluation and plotting in Matlab. Hopefully we have been successful in accomplishing this goal!!!