

Differential Equations (92.236)

Listing of Matlab Lab Exercises

This page contains a summary list of the Matlab lab exercises available for this course. We will have roughly 10 – 12 lab sessions that highlight various aspects of the Matlab language, with a focus on its use in simulating and understanding physical systems. The exercises given here walk the student through the basic operation of Matlab, introduce the use of Matlab for evaluating and plotting functions, and highlight several techniques for solving and visualizing differential equations (both numerically and symbolically). These exercises, coupled with the several Matlab demos that are available, should give the student a good command of using Matlab for a variety of applications. The student should work through each of these exercises and really try to understand what is happening at each step. With this basic understanding and careful study of the full example applications, the Matlab component of the HW assignments and Exams for this course should be quite straightforward. Also the exercises and demos should give you a solid understanding of various problem solving strategies and techniques within Matlab.

Matlab Lab Exercises

<u>Lab 1</u>	Matlab overview
<u>Lab 2</u>	Slope fields and solution curves
<u>Lab 3</u>	Evaluate and plot multiple curves on single plot
<u>Lab 4+5</u>	More Matlab syntax
<u>Lab 6+7</u>	Numerical solution of ODEs
<u>Lab 8</u>	More on numerical solutions of ODEs
<u>Lab 9+10</u>	Symbolic operations in Matlab

Note: Of course, an additional source for learning the basics of the Matlab language is the actual code manual, “*MATLAB & Simulink Student Version*,” Release 14, The Mathworks, Inc., 2004. The many tutorial-type examples in the manual are probably the best source for the novice. It is highly recommended that you purchase the student edition and actually use the manual to learn Matlab. The exercises given here, in some cases, are just a little more focused towards the needs and goals for this course. They do not, however, serve as a replacement for the manual -- they should be used in addition to the Matlab manual!!! Note also that there is now a number of good books available that can help you learn the basics of Matlab. The Mathworks Home Page at www.mathworks.com is a good reference to these other sources.

Note: Just so you know, one of my favorite introductory books on the fundamentals of Matlab is the text by Amos Gilat, “*Matlab: An Introduction with Applications*,” 2nd Edition, John Wiley & Sons, 2005. This book is a great place to start and it is relatively inexpensive (under \$40)!!!

Overview of Matlab -- Lab Exercises

Getting Started with Matlab

Getting Started -- Try the following tasks:

1	Startup Matlab	double click Matlab icon or go to Start/Programs/Matlab
2	Change default directory	cd a:\ or cd c:\temp or browse for folder in menu
3	Check location	pwd
4	Get directory listing	ls or dir
5	Get general help	help or go to ? on top menu bar
6	Get help on 2-D plotting	help graph2d or select from menu
7	Get help on a command	help plot , help title , etc. or select from menu
8	Create simple plot	x = -2:0.5:2; f = x.*sin(x);
		plot(x,f), axis, title('f(x) = x*sin(x)')
		xlabel('x-values'), ylabel('y-values')
9	Explore other possibilities -- don't be afraid to try things...	

Creating a Matlab Script File -- Try the following tasks:

1. Under File, select New, M-file
2. Get into the Matlab editor and write a few lines of Matlab code to evaluate and plot a function. You might try the example given above as a start. Type in the commands and save the file, but don't exit the editor yet.
3. Return to the Matlab command window and simply type the name of the M-file to test it out (without the extension -- first name only).
4. Continue the above process of going between the Matlab workspace and the Editor to build a useful Matlab program.

Using a Pre-existing Matlab Script File -- Try the following tasks:

1. Download a sample M-file from the course website:
http://www.profjrwhite.com/diff_eqns/matlab_demos.htm
2. Now simply execute the M-file (as above) to see what it does. At this point, you might try **plot_demo1.m** and **plot_demo2.m**, as relatively simple examples of creating standard line plots in Matlab. Really try to understand the logic and command syntax in these files!!!

You are now well on your way to learning Matlab!!!

Slope Fields and Solution Curves -- Lab Exercises

Use of Functions in Matlab

Getting Started -- Try the following tasks:

1. Download a sequence of m-files from the website for this Differential Equations course:

http://www.profjrwhite.com/diff_eqns/matlab_demos.htm

Get the following files and be sure to store them in a *known* location:

[sf_demo1.m](#), [sf_demo2.m](#), [sf_demo_eqn.m](#), and [sfield.m](#)

2. Open Matlab and change the default directory to the desired location.
3. Run the Matlab script file **sf_demo1.m** by simply typing the name of the file at the Matlab prompt (just use the first part of the name, **sf_demo1**). That was pretty easy -- Matlab has performed a little magic to give you a plot of the slope field and several solution curves for one of the examples in Sec. 1.3 in your text by Edwards and Penny.

Now What Did You Just Do? -- Try the following tasks:

Part I -- The function file for a given ODE

1. Open the function file **sf_demo_eqn.m** in the Matlab editor, and notice the form:

function [output_arguments] = function_name(input_arguments)

→ do something of interest (being sure to define the output arguments)

Note: In this case there is only a single output argument, yp , and two input variables, x and y . These can be named anything you desire, but it makes sense to give the variables meaningful names. For this example we have the IVP, $y' = f(x, y) = \sin(x - y)$ with $y(-5) = -1.2$. Therefore, the interpretation of the variables is obvious, and the goal of the function is to simply return the value of $y' = f(x, y)$ for any x, y combination that is input to the function.

2. Call the function with the specific input $x = 1$ and $y = 0$, as follows:

sf_demo_eqn(1,0)

What happened? Use your calculator to make sure the returned value makes sense.

3. Now let's pass in a sequence of values:

x = [-3 -1.5 0 1.5 3.0], y = [-5 -2.5 0 2.5 5.0], sf_demo_eqn(x,y)

For fun also try:

[xx,yy] = meshgrid(x,y), sf_demo_eqn(xx,yy)

Do you understand what has happened here?

Also try:

```
yp = sf_demo_eqn(xx,yy)
```

Here the output of the function is simply stored in variable `yp` and it can be used in additional calculations as needed.

Part II -- Slope fields using SFIELD.M

4. Open the function file `sfield.m` in the editor.

Note: This file has the same form as any function,

```
function [output_arguments] = function_name(input_arguments)
```

→ do something of interest (being sure to define the output arguments)

but it is a little more complicated than the previous function file. As discussed in class, *this function's role in life is to plot slope fields*. In particular,

- It generates a meshgrid (a rectangular array of x,y pairs based on an input x,y range, `rm` = `[minx maxx miny maxy]`, and the number of points, `Np`, to use in each direction).
 - It then evaluates the function file denoted by string variable `fn` at each of the x,y points in the grid. This represents the slope of $y(x)$ at each x,y point [i.e. $y' = f(x,y)$].
 - These slopes can be interpreted as unit vectors that show the direction or slope of a solution curve, $y(x)$, at each x,y point. For visualization, the unit vectors at each point are converted into their x and y components and plotted with a built-in Matlab routine called `quiver.m` (this is a standard file used for plotting vector fields in lots of applications).
5. Let's plot the slope field for the ODE represented by function file `sf_demo_eqn.m` over the domain $-5 \leq x \leq 5$ and $-5 \leq y \leq 5$ using a 20×20 grid, as follows:

```
sfield('sf_demo_eqn',[-5 5 -5 5],20);
```

Pretty neat, don't you think? This should look similar to the plots generated in class and also given in your textbook.

Part III -- Solution curves using ODE23.M

6. Try the following:

```
hold on (this "holds" the current figure open so that new curves appear in this figure)
[x,y] = ode23('sf_demo_eqn',[-5 5],-1.2);
plot(x,y, 'r-')
hold off
```

You have just solved your first ODE in Matlab! Notice how the solution curve is parallel to the slope or direction field vectors at each point -- the solution, $y(x)$, essentially *threads* its way through the slope field!

Note: Although we have not talked about numerical solution schemes yet (we will in Chapter 2), we note here that **ode23** is one of Matlab's built-in ODE solvers that uses a particular implementation of the so-called Runge-Kutta scheme for numerically integrating ODEs. The input arguments to **ode23** include:

- the function name for the ODE you want to solve (here, the file **sf_demo_eqn.m** must be in the default directory),
- the domain limits for the independent variable (here we used $[-5 \ 5]$), and
- the initial value of the dependent variable [$y(-5) = -1.2$ in this example].

The output arguments from the **ode23** routine contain the desired solution, $y(x)$ versus x .

Part IV -- Putting it all together into a Matlab program

- Now open **sf_demo1.m** in the Matlab editor and notice that this file simply includes the steps that we performed interactively:
 - It generates the slope field for the desired ODE.
 - It identifies several different initial points and uses **ode23** within a Matlab *for loop* to solve the ODE and plot its solution curve for each initial condition.
- Now let's solve a new problem, this time focusing on only the generation of the slope fields (we will put off further work with the **ode23** routine for a little while):
 - First, you should close any open files. Now open only **sf_demo2.m** and run it. This file is identical to the first part of **sf_demo1.m** -- it only generates the slope field (i.e. no calls to the **ode23** routine).
 - Now open and edit the existing ODE function file, **sf_demo_eqn.m**, to include the equation of interest [for example, try generating a slope field for $y' = 2x^2y^2$].
 - Modify the main script file, **sf_demo2.m**, to solve this particular problem:
 - Choose an appropriate grid and number of points for the slope field.
 - Now simply execute the new main program and see what happens -- always study and interpret the solutions to make sure they make sense.
 - Repeat this process as needed to solve a variety of problems.

**You are now an expert with using Matlab functions and script files
and in generating slope fields and actual solutions to first-order ODEs!!!**

NOTE. There is a really neat Java program called, **dfield**, by Prof. Polking (Rice University) that plots slope fields and solution curves for 1st order ODEs. It is a nice program for helping you visualize direction fields -- use the URL <http://math.rice.edu/%7Edfield/dfpp.html> to try it out. Have fun...

Evaluating and Plotting Functions in Matlab -- Lab Exercises

Element-by-Element Operations and Matlab *for loops*

Getting Started -- Try the following tasks:

1. Download m-file **hw5_demo.m** from the website for the Differential Equations course:
http://www.profjrwhite.com/diff_eqns/matlab_demos.htm
 and store it in a *known* location (**a:** or **c:\temp**):
2. Open Matlab and change the default directory to the desired location (**a:** or **c:\temp**).
3. Run the Matlab script file **hw5_demo.m** by simply typing the name of the file at the Matlab prompt (just use the first part of the name, **hw5_demo**). This file simply generates a few plots and our goal for this lab period is to dissect this file to fully understand how this was done. Along the way, we will highlight two key features in Matlab:
 - **element-by-element** operations with vectors and arrays using the “**dot**” operator
 - using *for loops* in Matlab to perform repetitive tasks

Now What Did You Just Do?

Open the script file **hw5_demo.m** in the editor, and notice the basic structure and operations performed. We will discuss each line in detail. However, the key sections of code are:

- Define some variables (some constants and some vector quantities).
- Evaluate two functions over a range of flow rates (q) and time points (t):

$$V_1(t, q) = V_{10} e^{-3qt/V_T}$$

$$V_2(t, q) = \frac{0.5qV_{10}}{V_T} t e^{-3qt/V_T}$$

- Plot $V_1(t)$ and $V_2(t)$ versus time for three different flow rates.

The first and third sections are reasonably straightforward, but the middle section, which does the actual function evaluation, has a number of subtle points that need further discussion. The remainder of the period will focus on this section, highlighting the storage of information in 2-D arrays, using Matlab’s **element-by-element** operations to efficiently do the function evaluations, and using Matlab’s *for loop* construction to loop over a given set of code. When complete, you should have a much better understanding of these elements of the Matlab language. These features are essential for many applications.

You are now an expert with using Matlab *for loops* and element-by-element operations to evaluate and plot functions of two variables!!!

More on Matlab Syntax -- Lab Exercises

Some more exercises to highlight several common operations in Matlab

Scalar Arithmetic:

With $x = 3$ and $y = 4$, compute the following quantities:

$$\text{a. } \frac{3}{2}xy \quad \text{b. } \left(1 - \frac{1}{x^5}\right)^{-1} \quad \text{c. } \frac{4(y-5)}{3x-6}$$

Vector (element-by-element) Arithmetic:

With $\underline{x}^T = [3 \ 1 \ 0]$ and $\underline{y}^T = [0 \ 1 \ 1]$, compute the same quantities as above.

2-D Array (element-by-element) Arithmetic:

With $\underline{\underline{X}} = \begin{bmatrix} -3 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ and $\underline{\underline{Y}} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & -2 \end{bmatrix}$, compute the same quantities as above.

Array Operations:

With $\underline{\underline{A}} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$, perform the following operations:

- Extract the 3rd column of matrix A and store it in vector B.
- Extract the 1st and 3rd columns of matrix A and store them in matrix C.
- Add the 1st and 3rd rows of matrix A together and store the result in vector D.
- Change the value in the 2nd row and 3rd column of A to -7 (instead of $+7$) and call the result AA (do not destroy/change the original A matrix).
- Create a matrix that contains rows 1 and 3 from A, the second row of AA, and the result of Step c. The resultant 4x4 matrix should be

$$BB = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \\ 5 & 6 & -7 & 8 \\ 10 & 12 & 14 & 16 \end{bmatrix}$$

- First clear all variables from the Matlab workspace, and then create a **4x4 magic matrix** (using Matlab's **magic** command) -- call this matrix A. Now do the following:
 - Sum each column of the A matrix.
 - Sum the rows of the A matrix.
 - Extract the main diagonal elements of A (top left to lower right) and sum them.
 - Extract the opposite diagonal elements of A (top right to lower left) and sum them.

Logical Arrays:

- Find the locations in array **A** whose values are greater than 10. You can do this by typing **C=A>10** at the Matlab prompt (check the Matlab workspace at this point and notice the **class** of the **C** matrix).
- Now type **D = A(C)**. What has happened?
- We can also do a similar task with Matlab's **find** command. Try typing **[I,J] = find(A>10)**. Now type **for k = 1:length(I), DD(k) = A(I(k),J(k)); end**. Are **D** and **DD** the same? Try to understand what has happened here.

A Searching Problem:

Suppose that you need to find all integers between 1 and 1000 that are divisible by 7, and have a remainder of 1 when divided by 2, 3, 4, 5, or 6. Can you write a short Matlab code to do this? Note that setting elements of an array equal to the empty array, [], eliminates them from the original array. With this fact, try the following (and make sure you understand each step):

```
n = 7:7:1000; Nn = length(n)
lookfor remainder
help rem
for i = 2:6
n(rem(n,i)~=1)=[]; Nn = length(n)
end
n
```

Decision-Making Structures:

Problem Description: Let x_1 be an integer. Suppose that the following rule is used to define a sequence of numbers based on x_1 , as follows:

$$x_{k+1} = \begin{cases} x_k / 2 & \text{if } x_k \text{ is even} \\ 3x_k + 1 & \text{if } x_k \text{ is odd} \end{cases}$$

For an arbitrary input positive integer, evaluate and plot the sequence x_k , stopping when $x_k = 1$. Put the Matlab code in a script file.

Problem Solution: What is needed here is a **while** loop that stops when $x_k = 1$, and an **if else end** structure to implement the above rule for incrementing x_k . A Matlab solution to this problem is given below. Put this code into a Matlab script file (comments not essential), run it, and make sure you understand the syntax for the decision making structures and why they are used in this context.

<pre>% % Demo illustrating several control structures % clear all, close all x = zeros(500,1); x(1) = round(abs(input('Enter a number: '))); k = 1; while (x(k)>1) & (k<500) if rem(x(k),2) == 0, x(k+1) = x(k)/2; else, x(k+1) = 3*x(k)+1; end k = k+1; end plot(x),xlabel('k value'),ylabel('x value') title('Sequence of x values')</pre>	<pre>% allocate storage for x % input integer % initialize counter % x is even % x is odd % increment counter % plot function</pre>
--	--

You should now have a better understanding of some of Matlab's syntax!!!

Numerical Solution of ODEs in Matlab -- Lab Exercises
The Euler Method

Getting Started -- Do the following:

1. Download a sequence of m-files from the website for this Differential Equations course:
http://www.profjrwhite.com/diff_eqns/matlab_demos.htm
 Get the following files and be sure to store them in a *known* location (**a:** or **c:\temp**):
ndemo1.m, **eqn1.m**, **euler1.m** and **euler.m**
2. Open Matlab and change the default directory to the desired location (**a:** or **c:\temp**).
3. Run the Matlab script file **ndemo1.m** by simply typing the name of the file at the Matlab prompt. This program is interactive in that it asks the user to input the number of integration steps to use over a given domain, and it does this repeatedly until the user types in a value of zero for the number of steps. Try various values, **N = 5**, **N = 10**, and **N = 100**, and you will see that the numerical solution of the ODE gets closer and closer to the exact solution as **N** increases or as the **step size**, Δx , decreases.

Well that was pretty easy -- you have just solved the IVP defined by

$$\frac{dy}{dx} = e^{-2x} - y \quad \text{with} \quad y(0) = 1$$

using the **Euler Method** for discretizing the ODE. You have also seen that “**as Δx decreases, the numerical solution becomes closer to the exact solution.**” Let’s now dissect these programs a little just to make sure you know what has happened.

Now What Did You Just Do? -- Try the following tasks:

Part I -- The function file to evaluate $y' = f(x,y)$

1. Open the function file **eqn1.m** in the Matlab editor. The input arguments to this file are specific values of x and y . Knowing x and y , we can evaluate the function $f(x,y)$ which is the derivative of $y(x)$ at the point $[x,y]$. Thus, the goal of this routine is to return $y' = f(x,y)$ for the input x,y pair. In this case we give the result, y' , a variable name called **yp** and pass this variable back to the calling program as part of the function’s output argument list.
2. Following this discussion, let’s create a function file to handle the first problem for your HW assignment. To do this we can simply edit the current file by making the changes as indicated in the following table:

	Example Problem	HW Problem #1
Differential equation	$\frac{dy}{dx} = e^{-2x} - y$	$\frac{dy}{dx} = -y$
Matlab syntax in function	yp = exp(-2*x)-y;	yp = -y;

Now save this new file as **p1_eqns.m** to refer to the Prob. #1 equations (be sure to hit the **Save As** option in the **File** menu in the editor and be sure you know where the file is being saved).

Part II -- The Euler Method (euler.m and euler1.m)

- Open the function file **euler.m** in the editor. This file implements a general form of the **Euler Method**. However, it is more complicated than necessary at this time. The added complexity gives this routine greater flexibility for later use in this course (i.e. it will handle a system of first-order ODEs). At this point we don't need all this detail, so a single-equation version, called **euler1.m**, has been generated for discussion here. Open **euler1.m** and briefly compare it to **euler.m** (the lines in **bold** represent the key changes). You should see that the two files are similar, but that **euler1.m** is slightly less complicated.
- Now let's focus on the **euler1.m** file. A listing of **euler1.m** is shown below:

```
%
% EULER1.M Use Euler Method for Numerical Solution of IVPs
%
% This function solves a single first-order ODE via Euler's Method
%
% The inputs to the function are:
%   fxy = string variable with the name of the file containing f(x,y)
%   xo,xf = initial and final values of the independent variable (scalars)
%   yo = initial value of dependent variable at xo
%   N = number of intervals to use between xo and xf
% The outputs to the function are:
%   X = vector containing values of the independent variable
%   Y = the dependent variable at each value of the independent variable
%
%
%   function [X,Y] = euler1(fxy,xo,xf,yo,N)
%
% compute step size and size of output variables
%   if N < 2   N = 2;   end % set minimum number for N
%   h = (xf-xo)/N;      % step size
%   X = zeros(N+1,1);  % initialize independent variable
%   Y = zeros(N+1,1);  % initialize dependent variables
%
% set initial conditions
%   X(1) = xo;   Y(1) = yo;
%
% begin computational loop
%   for i = 1:N
%       k1 = h*feval(fxy,X(i),Y(i)); % evaluate function
%       Y(i+1) = Y(i) + k1; % increment dependent variable
%       X(i+1) = X(i) + h; % increment independent variable
%   end
%
% end of function
```

The first part of this file computes the step size, h , and does some initialization. The final few lines of code contain the **basic Euler algorithm**:

Loop over number of time steps	Loop over all i
Evaluate function	$k_1 = hf(x_i, y_i)$
Increment dependent variable	$y_{i+1} = y_i + k_1$
Increment independent variable	$x_{i+1} = x_i + h$
End loop	End loop

Note that Matlab's **feval** function is used to implement the actual function evaluation. This is needed since we are using the name of a generic function, **fx**, in the current routine. The command **feval(fx,x,y)** is essentially equivalent to **eqn1(x,y)** or **p1_eqns(x,y)**, where the local variable, **fx**, has been equivalenced to the name of the ODE function file, **eqn1** or **p1_eqns**, via the input argument list in these examples.

5. Now let's test drive **euler1.m**. For example, let's solve the base problem from above with the following commands:

```
xo = 0; xf = 0.5; yo = 1; N = 10;
[x,y] = euler1('eqn1',xo,xf,yo,N); plot(x,y)
```

Pretty easy, don't you think? This should look similar to the plots generated above (except we have not included the exact solution). Using the same basic scheme, you can solve the differential equation associated with your HW assignment. For example, Prob. #1 becomes

$$\frac{dy}{dx} = -y \quad \text{with} \quad y(0) = 2$$

and we already have the ODE function built into function file **p1_eqns.m**. Thus, solution of this problem can be accomplished by

```
xo = 0; xf = 0.5; yo = 2; N = 10;
[x,y] = euler1('p1_eqns',xo,xf,yo,N); plot(x,y)
```

This is pretty crude but it contains the basics. Note that **euler.m** could be used if desired.

Part III -- Putting it all together into a Matlab program -- **ndemo1.m**

6. The program **ndemo1.m** simply elaborates on the above basic solution scheme. It includes the evaluation of the exact solution for comparison purposes and for studying the sensitivity of the solution to the choice of step size. It also includes some intermediate results that are tabulated to help compare with the hand calculations you were asked to do. Finally, it enhances the plots a little with a title, x-axis label, etc.. Putting everything into a program let's you do more things -- these all could be done interactively but it could become pretty tedious. The program also can be used as formal documentation of exactly what computations were performed. Following this guide, you should modify **ndemo1.m**, as needed, to solve the requested HW problems.

You are now an expert with using the Euler Method for solving simple IVPs!!!

More on the Numerical Solution of ODEs in Matlab -- Lab Exercises
INODE -- Interactive Solution of Simple IVPs (using a variety of methods)
ODE23 -- Matlab's Built-In ODE Solver (with adaptive step control)

Getting Started -- Do the following:

1. Download a sequence of m-files from the website for this Differential Equations course:
http://www.profjrwhite.com/diff_eqns/matlab_demos.htm
 Get the following files and be sure to store them in a *known* location (**a:** or **c:\temp**):
inode.m & **sfode.m** and **euler.m**, **impeuler.m**, **rk4.m** & **sfield.m** (if needed)
2. Open Matlab and change the default directory to the desired location (**a:** or **c:\temp**).

The INODE.M Code -- Try the following tasks:

1. Create an ODE function file that contains the following ODE:

$$\frac{d}{dt}P = 4P - P^2 - 3$$

and save it to disk (make sure you know the name and location of all your files – I called my file **eqnpop1.m**). If you don't remember the format for this function file, use **eqn1.m** from last week's lab exercises as a guide. Note also that the P^2 term should be written as $P.*P$ in Matlab to allow matrix element-by-element evaluation (needed later for the **sfield.m** function).

2. Run the Matlab script file **inode.m** with the ODE function file you just created with the following parameters: $t_0 = 0$, $P_0 = 5$, and $t_f = 3$.

Now run **inode.m** again with the same parameters, but this time manually type in the equation of interest [i.e. $f(t,P) = 4P - P^2 - 3$]. Note that this option does not need a pre-defined ODE function file -- the ODE of interest is defined interactively. However, this **must** use x and y notation -- here you simply type **4*y - y.*y - 3**. Pretty easy, don't you think???

Run **inode.m** again trying a different method this time (the choices use the EULER, IMPEULER, RK4, or ODE23 routines). Do you see any differences?

3. Now run **inode.m** again (any option you desire) with a different initial condition. This time try $P_0 = 0.5$. What happened? Try another method to see if this helps.
4. To figure out what is going on, let's generate the *slope field* for this problem. To do this type the following Matlab commands:

```
clear all, close all  
nfig = 1; figure(nfig); v = [0 3 -5 5];
```

sfield('eqnpop1', v, 25); axis(v), hold on

Does this plot of the direction field help? What do you expect the steady state solution to be for $P_0 > 3$? How about $1 < P_0 < 3$? What if we let P_0 be less than 1 (as we did above in Step 3)?

Based on this analysis, the behavior in Step 3, where the solution went to $-\infty$, is exactly as it should be. The solution is unbounded if the initial condition is less than unity. For P_0 greater than unity, the solutions are bounded and approach an equilibrium state of $P(t) = 3$.

But how do we get our ODE solver to show these results in a well-behaved manner, without going off to $-\infty$? One way, of course, is to simply limit the integration interval to cover a range where the function does not blow up. For example, for $P_0 = 0.5$, a $t_f < 0.7$ would give a nice plot. But the important question is "How do you know this ahead of time???". The answer lies in the use of the built-in **ode23.m** routine.

ODE23 to the Rescue!!! -- Try the following task:

Let's plot a series of solution curves using Matlab's **ode23.m** routine for several different initial conditions, as follows (make sure you typed **hold on** in the previous step):

```
ode23('eqnpop1',[0 3], 5)
ode23('eqnpop1',[0 3], 3.5)
ode23('eqnpop1',[0 3], 2)
ode23('eqnpop1',[0 3], .5)
ode23('eqnpop1',[0 3], 0)
```

Everything seems fine here. The solutions follow the direction field, just as expected!

One nice thing about well-written commercial software is that it is usually more flexible and robust than the simple algorithms that we often use to get across the basic ideas associated with a particular technique. Matlab's built-in ODE solvers are no exception. They have been tested and modified over the years to handle as many situations (and to be as forgiving) as possible. In this case, calling the **ode23.m** routine with *no output arguments automatically plots the solutions and the routine itself is smart enough to know when it is approaching a singularity* - so it stops in a graceful manner. When we called **ode23.m** from the **inode.m** code (see above), the code also ended gracefully -- but the results looked quite odd. Without output arguments, the integration process is animated so we can actually 'see' the singularity approaching. This is very useful when trying to understand what is happening physically. In the case of the other methods, catastrophic failure is possible, making the **ode23.m** routine the clear choice when singularities are present (as well as in most other situations).

Final Note: The combination of the numerical tools that we have introduced thus far in the semester (**sfield.m**, **ode23.m**, **inode.m**, etc.) represent a powerful and relatively easy-to-use inventory of software tools for *solving* and *analyzing* a variety of initial value problems (IVPs) involving a single ODE. Many of these same tools can also be used to work with a *system of ODEs*, as demonstrated in class (we will elaborate on this capability later in the course).

With this exercise, you are now an expert with using these tools for solving and understandings simple IVPs!!!

Symbolic Operations in Matlab -- Lab Exercises

Some exercises to highlight the use of Matlab's SYMBOLIC capability

Background Note:

Up to this point we have done purely numerical operations in Matlab. This is the traditional approach for solving problems on the computer, and much of the current computational work done in industry follows this approach. However, a relatively new capability (last 5-10 years or so) that is gaining popularity involves the use of *symbolic operations* directly on the computer. Computer programs that use these symbolic techniques are often referred to as *computer algebra systems*, and they are becoming quite powerful. In particular, Matlab's *Symbolic Math Toolbox* offers this general capability within the Matlab environment by providing an interface to the **Maple** code (Maple is a commercial software package that emphasizes symbolic manipulations or computer algebra techniques). This set of labs will introduce some of the basic symbolic capability in Matlab, eventually leading up to the analytical solution of Ordinary Differential Equations directly on the computer. Many of the exercises given here are derived from the Matlab's User's Guide and you are encouraged to refer to that manual for further examples and guidance.

Define Some Symbolic Variables:

First define a bunch of symbolic variables: `syms a1 b1 c1 a2 b2 c2 d2 A B C1 C2 x y t`

Now let's form some symbolic functions:

Mathematical Function	Matlab Syntax for Function
$f_1(x) = a_1 + b_1x + c_1x^2$	<code>f1 = a1 + b1*x + c1*x^2</code>
$f_2(x) = a_2 + b_2x + c_2x^2 + d_2x^3$	<code>f2 = a2 + b2*x + c2*x^2 + d2*x^3</code>
$g(t) = e^{at}(C_1 \cos(\beta t) + C_2 \sin(\beta t))$	<code>g = exp(A*t)*(C1*cos(B*t)+C2*sin(B*t))</code>
$u(x, y) = 2xy^2 + \sin(x + y)$	<code>u = 2*x*y^2 + sin(x+y)</code>

Differentiation of Symbolic Functions:

Differentiate each of the above functions using Matlab's **diff** command. For example, try **diff(f1)**. What happened? Is the result correct? Do the same thing for the other functions. What happens when you type **diff(u)**? How do you get Matlab to take the partial derivative with respect to the variable y ? Type **help sym/diff** to see the various options associated with taking symbolic derivatives.

We can also take higher order derivatives. For example **diff(f1,2)** takes the second derivative of $f_1(x)$. Try this for all the functions!

Can you show that the mixed partial derivatives of $u(x,y)$ are equal? Take du/dx and du/dy as above and save the results as string variables. For example, try typing

dudx = diff(u,x); dudy = diff(u,y);

Now take the first derivative of these symbolic variables, as follows:

uyx = diff(dudx,y); uxy = diff(dudy,x);

Are these latter two variables the same? They should be. Pretty neat stuff, don't you think!!!

Integration of Symbolic Functions:

Using Matlab's **int** function, perform the following integrals using $f_1(x)$:

Mathematical Operation	Matlab Syntax for Operation
$I_1 = \int f_1(x)dx$	I1 = int(f1)
$I_2 = \int_0^5 f_1(x)dx$	I2 = int(f1, 0, 5)

Do these make sense? Check out the **subs** command. Now let $a_1 = 1$, $b_1 = -2$, and $c_1 = 1$, and substitute these numerical constants into the symbolic expressions for I_1 and I_2 , as follows:

I1 = subs(I1,{a1 b1 c1},{1 -2 1}), I2 = subs(I2,{a1 b1 c1},{1 -2 1})

What are the values for the integrals I_1 and I_2 ? Why is I_1 a function of x and I_2 is simply a scalar number? Note also that the curly brackets in Matlab refer to **cell arrays**. The sequence given by the set, **{a1 b1 c1},{1 -2 1}**, replaces the numerical variables into the symbolic variables in sequence. The cell array simply let's us make all three substitutions in one call to the **subs** command.

Try integrating and substituting values for the constants in the other functions. Note that with $u(x,y)$, you can only integrate with respect to one variable at a time. For example,

Mathematical Operation	Matlab Syntax for Operation
$I_u = \int_0^{2\pi} \left[\int_0^{\pi} u(x,y)dx \right] dy$	Iu = int(int(u,x,0,pi),y,0,2*pi)

Pretty impressive!!! Could you do this integral by hand this fast? I certainly can't...

Plotting Symbolic Functions:

Certainly we can also plot symbolic relationships. Since we have already defined the constants for use with $f_1(x)$, let's use this function to do some simple plots. In particular, let's plot $f_1(x)$, df_1/dx , and $\int f_1(x)dx$ over the range $0 \leq x \leq 5$. We can do this as follows:

```
F1 = subs(f1,{a1 b1 c1},{1 -2 1}); D1 = diff(F1); I1 = int(F1);
Nx = 51; xp = linspace(0,5,Nx);
F1p = double(subs(F1,x,xp));
D1p = double(subs(D1,x,xp));
I1p = double(subs(I1,x,xp));
plot(xp,F1p,'r-',xp,D1p,'g--',xp,I1p,'b-.'),grid
```

```

title('Evaluating and Plotting Symbolic Functions')
xlabel('X Values'),ylabel('Various Function Values')
legend('Function','First Derivative','Integral')

```

Solving Algebraic Equations:

We can also solve algebraic equations with Matlab. For example, what if we wanted to know the value of x where $f_1(x) = 3$? With the above constants this problem becomes:

$$\text{find } x \text{ such that } f_1(x) - 3 = 1 - 2x + x^2 - 3 = x^2 - 2x - 2 = 0$$

In Matlab, we can use the **solve** command, as follows:

```
solve(x^2-2*x-2)
```

or

```
solve(F1 - 3)
```

We can also solve systems of algebraic equations. For example, the analytical solution to the following 2x2 system:

$$3x + 2y = 3 \quad \text{and} \quad -2x + y = -7$$

is given by:

```

S = solve(3*x + 2*y - 3, -2*x + y + 7); % this solves a simple set of 2x2 equations
S.x, S.y % this prints x and y to the screen

```

Solving Ordinary Differential Equations:

The real goal of the above discussion and examples with symbolic variables is to give enough background so that we can solve ODEs analytically on the computer. This will be pretty powerful capability if we can actually do this with a set of relatively simple commands. As a test, let's give Matlab's **dsolve** command a workout for a few different types of ODEs that we have treated thus far in the semester (be sure to type **help dsolve** to get a good idea of the various forms that can be used).

a. Solve the first order IVP:

$$xy' + y = 2e^{2x} \quad \text{with } y(1) = 0$$

```
dsolve('Dy + y/x = (2/x)*exp(2*x)', 'x') % this gives the general solution
```

```
dsolve('Dy + y/x = (2/x)*exp(2*x)', 'y(1) = 0', 'x') % this gives the unique solution
```

b. Solve the second order IVP:

$$y'' - 4y = 4x^2 \quad \text{with } y(0) = -\frac{1}{2} \quad \text{and} \quad y'(0) = 4$$

```
Sh = dsolve('D2y - 4*y = 0', 'x'), pretty(simple(Sh)) % homogeneous soln
```

```
Sg = dsolve('D2y - 4*y = 4*x^2', 'x'), pretty(simple(Sg)) % general soln
```

```
Su = dsolve('D2y - 4*y = 4*x^2', 'y(0) = -1/2', 'Dy(0) = 4', 'x') % unique soln
pretty(simple(Su))
```

c. Solve the second order IVP:

$$y'' + 6y' + 13y = 10\sin 5t \quad \text{with} \quad y(0) = 0 \quad \text{and} \quad y'(0) = 0$$

Q1 = dsolve('D2y + 6*Dy + 13*y = 10*sin(5*t)', 'y(0) = 0', 'Dy(0) = 0', 't')

pretty(simple(Q1))

d. Re-solve the problem in Part c as a system of two 1st order ODEs, where $z_1 = y$ and $z_2 = dy/dt$:

$$\frac{d}{dt} z_1 = z_2 \quad \text{and} \quad \frac{d}{dt} z_2 = -13z_1 - 6z_2 + 10\sin 5t \quad \text{with} \quad z_1(0) = 0 \quad \& \quad z_2(0) = 0$$

Q2 = dsolve('Dz1 =z2', 'Dz2 = -13*z1 -6*z2 + 10*sin(5*t)', 'z1(0) = 0', 'z2(0) = 0', 't')

pretty(simple(Q2.z1)), pretty(simple(Q2.z2))

Final Note:

There is a lot of good stuff here. In particular, there are several examples that should make your life much easier in this course and in several of your other technical classes. You should do your best to understand the basic capability that is illustrated here -- it represents some pretty powerful mathematical analysis capability. You should also note that some of the examples from earlier in the semester (the *Two Salty Tanks* problem for example) give other illustrations of the use of computer algebra to solve a coupled set of first order differential equations for a real problem of interest. Also, a final Matlab demo will be given at the end of the semester illustrating how to take *Laplace transforms* and *inverse Laplace transforms* analytically using Matlab's symbolic processing capability.

You should now be an expert with Matlab's SYMBOLIC capability!!!

**Happy Problem Solving with Matlab's
NUMERICAL and SYMBOLIC processing tools!!!**