Problem Set 5 Initial Value Problems
10.34 Fall 2005, KJ Beers

Ben Wang, Mark Styczynski
October 21, 2005

Problem 1: 4.A.1

Note: there is a slight typo in the assignment. We define $x_1 = x$ and $x_2 = y$.

You are asked to derive the linear ODE system that describes the response of a system to small perturbations around the steady-state. The system is described by a system of ODEs

$$\underline{\dot{x}} = \underline{f}(\underline{x}) \tag{1}$$

$$\underline{f}(\underline{x_s}) = \underline{0} \tag{2}$$

A perturbation around the steady state can be written as

$$\underline{x}(t) = \underline{x_s} + \underline{\varepsilon}(t) \tag{3}$$

Since close to the steady state, the value of $\underline{f(x_s)} \approx 0$, we can write $\underline{f(x)}$ as a Taylor series expansion.

$$\underline{f}(\underline{x}) \approx \underline{f}(\underline{x_s}) + \left(\left. \frac{d\underline{f}}{d\underline{x}} \right|_{\underline{x_s}} \right) \left[\underline{x} - \underline{x_s} \right] + \ldots \tag{4}$$

Neglecting terms higher than second order, we are left with a linearized equation. Plugging equation (4) and (3) into equation (1):

$$\underline{\dot{x}_s} + \underline{\dot{\varepsilon}} = \underline{f}(\underline{x_s}) + \left(\left. \frac{d\underline{f}}{d\underline{x}} \right|_{\underline{x_s}} \right) \left[\underline{x} - \underline{x_s} \right] \tag{5}$$

Both $\underline{\dot{x}_s}$ and $\underline{f}(\underline{x_s})$ are equal to 0 and $\left[\underline{x} - \underline{x_s} \right]$ is equal to ε. So we are left with the linear ODE system that describes the response to a small perturbation:

$$\underline{\dot{\varepsilon}} = \left(\frac{d\underline{f}}{d\underline{x}}\bigg|_{\underline{x_s}} \right) \underline{\varepsilon} \tag{6}$$

or to be written as:

$$\underline{\dot{\varepsilon}} = J\big|_{x_s}\, \underline{\varepsilon} \tag{7}$$

where the J is the Jacobian. For our problem with the equations:

$$\dot{x} = f_1(x,y) = -2(x-1)(y-2)^2 + (x-1)(y-2) - (y-2) - 3(x-1) \tag{8}$$
$$\dot{y} = f_2(x,y) = -(x-1) + (x-1)^2(y-2) \tag{9}$$

The Jacobian for our system will be:

$$J = \begin{bmatrix} -2(y-2)^2 + (y-2) - 3 & -4(x-1)(y-2) + (x-1) - 1 \\ -1 + 2(x-1)(y-2) & (x-1)^2 \end{bmatrix}_{(1,2)} \tag{10}$$

plugging in these numbers the Jacobian will be

$$J = \begin{bmatrix} -3 & -1 \\ -1 & 0 \end{bmatrix} \tag{11}$$

Now is the steady state stable? To assess the stability we need to check the eigenvalues of this Jacobian by obtaining our characteristic polynomial:

$$|J| = \left| \begin{bmatrix} -3-\lambda & -1 \\ -1 & -\lambda \end{bmatrix} \right| = \lambda^2 + 3\lambda - 1 \tag{12}$$

Solving for lambda we get: $\lambda = -3 \pm \sqrt{13}$ ($\lambda_1 = -3.3028$ and $\lambda_2 = 0.3028$)

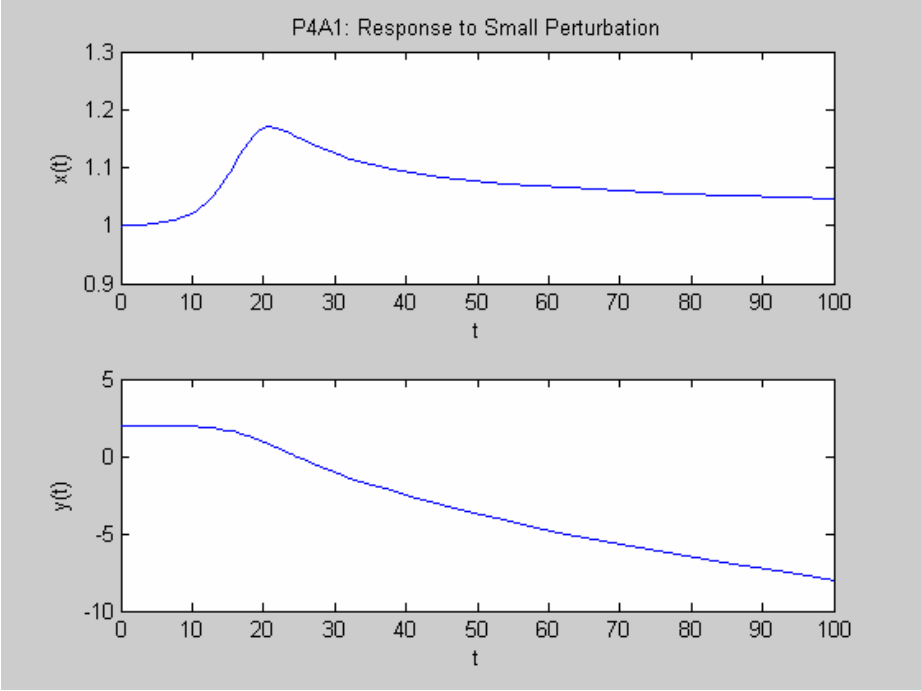
Because not all the real portions of the eigenvalues are less than zero, this system is not stable to small perturbations. Furthermore our eigenvalues are purely real, which means that our steady-state will not exhibit an oscillatory response to small perturbations.

We can test this out using MATLAB with code such as the following:

```
% benwang_P4A1.m
% Ben Wang
% HW#5 Problem #1
% due 10/21/05 9 am
```

```
% This routine will look at the effect of a small perturbation on the
% behavior of a system of coupled ODEs

% ======= main routine benwang_P4A1.m
function iflag_main = benwang_P4B1();

iflag_main = 0;

%PDL> clear graphs, screen etc. general initialization

clear all; close all; clc;

% specify inital state + random perturbation
x_0 = [1;2]+0.1*randn(2,1);

% solve ODE system
[t_traj, x_traj] = ode15s(@calc_f,[0 100],x_0);

% plot trajectories
figure;
subplot(2,1,1);
plot(t_traj,x_traj(:,1));
xlabel('t'); ylabel('x_1(t)');
title('P4A1: Response to Small Perturbation');
subplot(2,1,2); plot(t_traj,x_traj(:,2));
xlabel('t'); ylabel('x_2(t)');

return

%======= subroutine calc_f.m

function f = calc_f(t,p)

x = p(1);
y = p(2);

f = zeros(2,1);
f(1) = -2*(x-1)*(y-2)^2+(x-1)*(y-2)-(y-2)-3*(x-1);
f(2) = -(x-1)+(y-2)*(x-1)^2;

return;
```

Because of the random perturbations, we wouldn't have trajectories that are all the same, but you would have an example of a trajectory such as:



Upon examination of these graphs, we will note that as time increases, y(t) tends to diverge significantly, which corresponds to the positive eigenvalue. Thus our system is indeed unstable to a small perturbation. You will note also that x(t) tends to return to the stable solution, since it is associated with the negative eigenvalue. Since we are close to the stable manifold, it is a little more well-behaved.

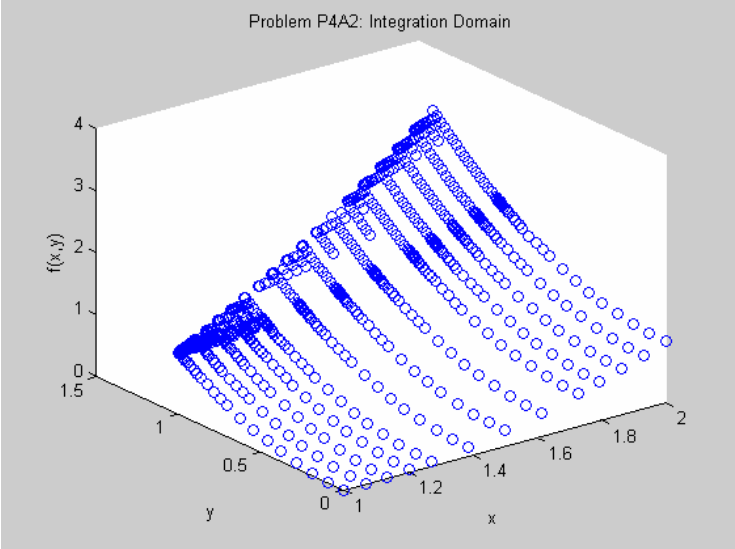
Problem 2: 4.A.2

You are asked to compute the value of the following definite integral using **dblquad**

$$\int_1^2 \int_0^{\sqrt{x}} \left[(x-1)^2 + y^2 \right] dy dx \qquad (13)$$

dblquad does not require much by way of input. As opposed to **trapz**, which requires you to enter in a specified vector of values, **dblquad** only asks for a function handle, and the limits of integration (and any changes in tolerance if required). The function that dblquad calls only needs the function that needs to be integrated. The only difficulty that arises is that some logic needs to be implemented to get the correct integration domain. This can be done in the function itself.

For example our integration domain will not be a simple box. Our function will look like this in 3 dimensions with the range of y varying as a function of x



A routine that would allow you to calculate the integral would be similar to something as follows:

```
% benwang_P4A2.m
% Ben Wang
% HW#5 Problem #2
% due 10/21/05 9 am

% We will write a routine that will implement a 2D numerical integration by
% use of the function dblquad

% ======= main routine benwang_P4A2.m
function iflag_main = benwang_P4A2();

iflag_main = 0;

%PDL> clear graphs, screen etc. general initialization
clear all; close all; clc;

%PDL> specify integration limits - need to come back to this
x_hi = 2;
x_lo = 1;
y_hi = sqrt(x_hi);     % max value
y_lo = 0;

% solve for integrand
int_2D = dblquad(@calc_integrand_f,x_lo,x_hi,y_lo,y_hi,1e-6)

xlabel('x'); ylabel('y'); zlabel('f(x,y)');
title('Problem P4A2: Integration Domain');
return

%======= subroutine calc_integrand_f.m
```

```
function f = calc_integrand_f(x,y)

f = zeros(size(x));

for i=1:length(f)
   if(y>sqrt(x(i)))
      f(i)=0;
      %plot(x(i),y); hold on;     % use this to see wherever dblquad sets
      %f(i) = 0;
   else
      f(i)=(x(i)-1)^2+y^2;
      plot3(x(i),y,f(i),'o'); hold on;     % use this to see wherever dblquad sets
      %f(i) = the function specified
   end
end

return;
```

The output should be something similar to:

```
int_2D =

   1.0612
```

Problem 3: 4.A.3

We are asked to plot the position of a point mass subject to harmonic forcing term and frictional dissipation. The equation of motion that describes this situation will be:

$$m\frac{d^2x}{dt^2} = -Kx - \zeta\frac{dx}{dt} + F_{ext}(t) \tag{14}$$

where

$$F_{ext}(t) = (10^{-2}N)\sin(\omega t) \tag{15}$$

We know that to solve a 2nd order ODE, we need to reduce the order by creating a system of two 1st order ODEs. We do this by rewriting equation (14) into:

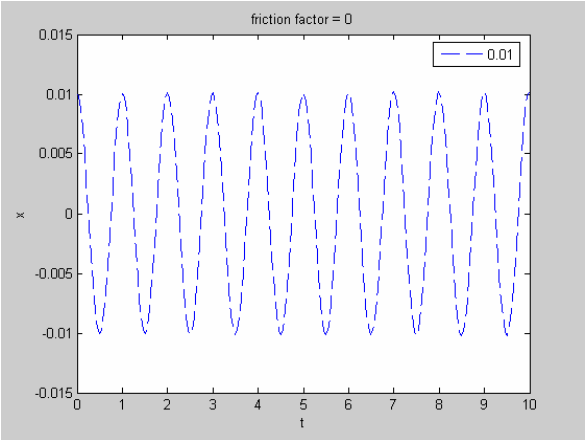
$$\begin{cases} \dot{v} = \dfrac{-Kx - \zeta v + F_{ext}(t)}{m} \\ \dot{x} = v \end{cases} \tag{16}$$

We now have two 1st order ODEs that can be solved with one of our methods. Some code that can be used to accomplish this is listed below, using ODE15s:

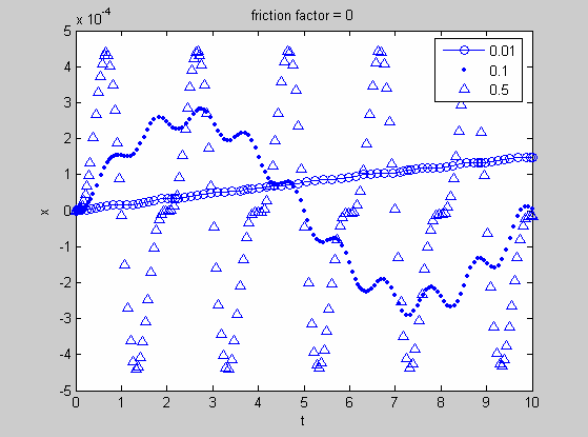
```
% benwang_P4A3.m
% Ben Wang
% HW#5 Problem #3
% due 10/21/05 9 am

% We will write a routine that will solve a 2nd order differential equation
% by reducing the order and creating a system of equations of 1st order
% ODEs.

% ======= main routine benwang_P4A3.m
function iflag_main = benwang_P4A3();

iflag_main = 0;

%PDL> clear graphs, screen etc. general initialization
clear all; close all; clc;

%PDL> specify some values specific to the problem

param.m = 1;                    % [kg]
param.w_c = 2*pi;               % [rad/s]
param.F = 1e-2;                 % [N]
param.K = param.w_c^2*param.m;  % [kg/s^2]

%PDL> specify vectors of different conditions

omega = param.w_c*[0.01 0.1 0.5 0.9 0.99 0.999];
friction = [0; 0.01; 0.1; 1];
for i = 1:length(friction)
   subplot(2,2,i); hold on;
   for j = 1:length(omega)
      x_0 = [0;0];          % initial position is at rest and without any velocity
      %PDL> solve the system of ODEs
      [t_traj, x_traj] = ode15s(@calc_f_P4A3,[0 10],x_0,[],param,friction(i),omega(j));
      %PDL> plot trajectories
      if j == 1;
         plot(t_traj,x_traj(:,2),'-o');
      elseif j == 2;
         plot(t_traj,x_traj(:,2),'.');
      elseif j == 3;
         plot(t_traj,x_traj(:,2),'^');
      elseif j == 4;
         plot(t_traj,x_traj(:,2),'+');
      elseif j == 5;
         plot(t_traj,x_traj(:,2),'*');
      else
         plot(t_traj,x_traj(:,2),'--');
      end
   end
   xlabel('t');
   ylabel('x');
   legend('0.01','0.1','0.5','0.9','0.99','0.999');
   a = num2str(friction(i));
```

```
    titlename = ['friction factor = ' a];
    title(titlename);
    hold off;

    %PDL> write matrix A that represents the Jacobian of our system:
    %A = [-param.K/param.m - friction(i)/param.m; 0 1];
    %b = eig(A);
end
return;
%======= subroutine calc_f_P4A3.m
function f = calc_f_P4A3(t,x_0,param,friction,freq)
f = zeros(size(x_0));
v = x_0(1);
x = x_0(2);
f(1) = 1/param.m*(-param.K*x - friction*v + param.F*sin(freq*t));
f(2) = v;
return;
```

Let's being by looking at the case when friction factor = 0, and there is no oscillatory forcing term:



Now if we look at some low frequency cases still with friction factor = 0 but add the forcing term, this code will give us this graph:

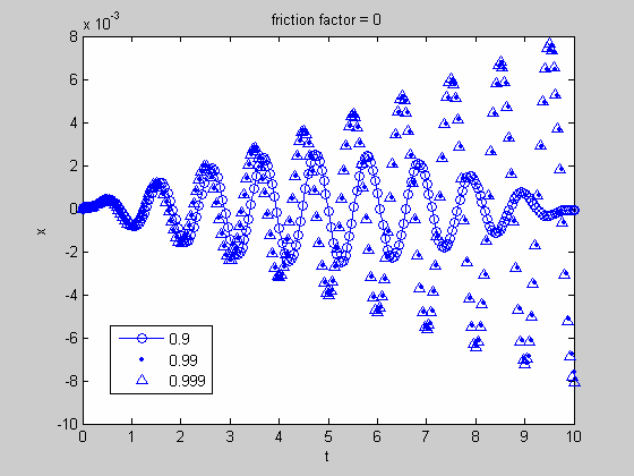


We are asked to describe what is going on here. Strange indeed! First let's take a look at the case when we do not account for the friction. Our ode has the form:

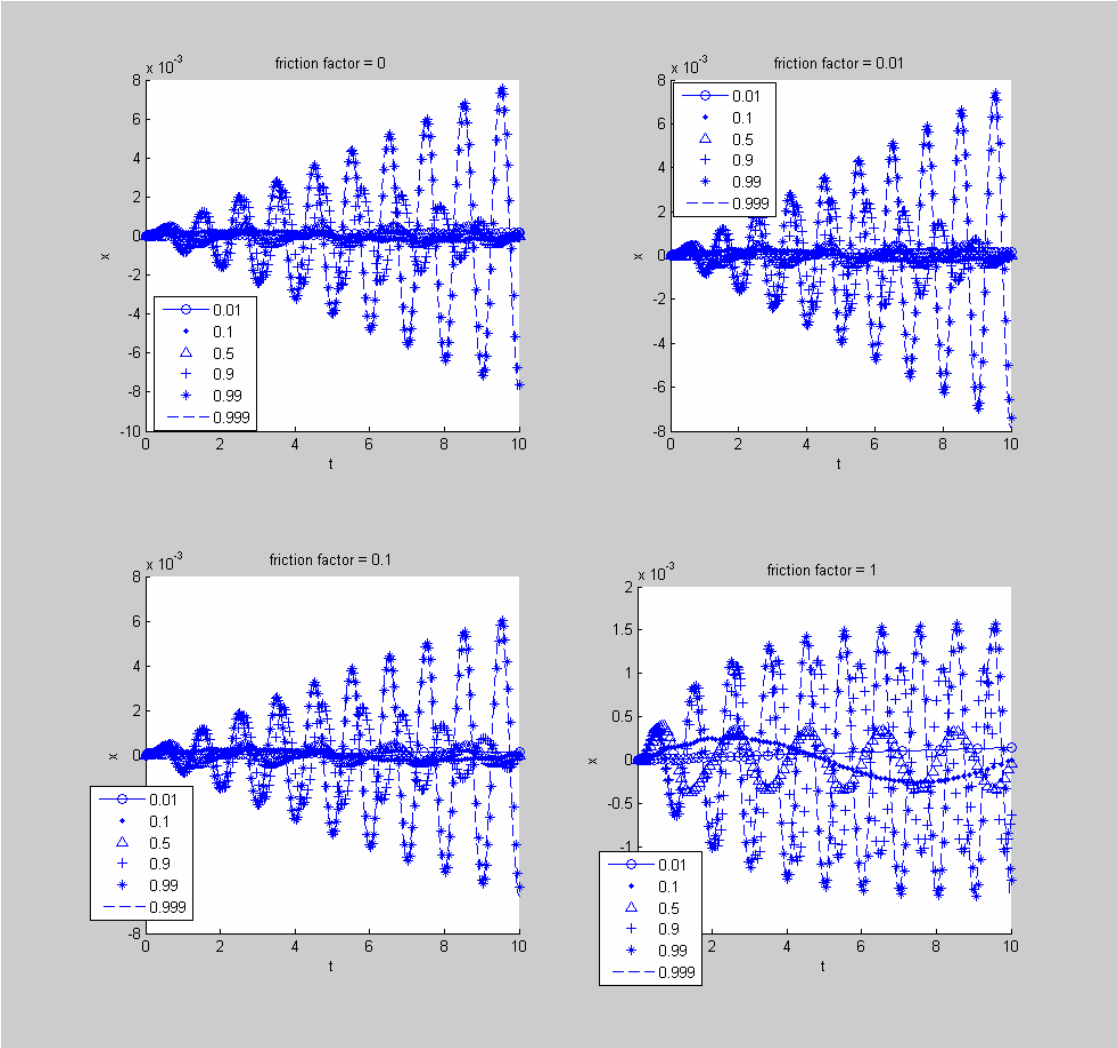
$$\ddot{x} + Cx + F(t) = 0 \qquad (17)$$

We remember from undergrad, that for C>0 and F(t) = 0, we have a solution that is typically made up of oscillatory terms (sin, cos). So that we have a solution that is oscillatory and with the forcing term, we have superposed on that some longer wavelength oscillations. So we would expect this result to be comparable to what we would expect analytically.

Now we look at the higher frequencies:



We see that as we increase our frequency such that it approaches the harmonic frequency, we start to get resonance. The amplitude of our solution increases with time, as expected with resonance. Now we move on to different friction factors:



Now we increase the friction factor. For low values of this factor, it appears that not a whole lot changes. Once the factor reaches a value of 0.1, it seems that amplitude of our solution is not as high as it was for lower values of the friction factor. By the time the friction factor = 1, we see significant amounts of damping and that our solution ceases to grow with time after t = 6 or so, reaching a "steady state" of sorts. Furthermore the amplitude of the solution is significantly reduced.

Problem 4: 4.B.1

We are asked to find the height of the water in the tank and the volume of water in the tank as functions of time. The main differential equation that we can use to calculate h(t) is the equation that relates the velocity in the tank to the velocity in the exit pipe:

$$\frac{\pi}{4} D_p{}^2 v_p = \frac{\pi}{4} [D(h)]^2 \left| \frac{dh}{dt} \right| \qquad (18)$$

we can do some rearrangement to get it into the form of our typical IVP problem:

$$\frac{dh}{dt} = f(t) = -\left(\frac{D_p}{D(h)}\right)^2 v_p \tag{19}$$

To get an ODE for volume:

$$\frac{dV}{dt} = \frac{d(\dot{h}A)}{dt} = \frac{\pi}{4}\frac{d(\dot{h}D(h)^2)}{dt} \tag{20}$$

We insert the minus sign in equation 19 to account for the fact that we expect the height to decrease with time, not increase. We can solve this equation with our classic iterative, time-stepping methods, but the we do not have a value for v_p. We can get v_p from Bernoulli's equation:

$$\frac{1}{2}\rho\left[\left(\frac{dh}{dt}\right)^2 - {v_p}^2\right] + \rho g\left[h + L_p\right] = \frac{1}{2}\rho v_p^2\left[K_L + f_D\left(\frac{L_p}{D_p}\right)\right] \tag{21}$$

This is essentially an energy balance posed as an algebraic problem. Basically we need a value for v_p to plug into equation 19 and we can get it from equation 20. Together with equations 19 and 20 , they form a Differential-Algebraic system. **ODE15s** is capable of handling this problem. We just have to use the 'Mass Matrix' to inform **ODE15s** that we are solving simultaneously differential equations and algebraic equations.

Of course our world is not so simple and we have the nasty f_D term that further complicates our situation, shown in equation (21). To top it off we don't even know what the deal is in the region where Re>2100 but Re<4000. Especially since f_D depends on Re (equation 22) which then depends back upon v_p.

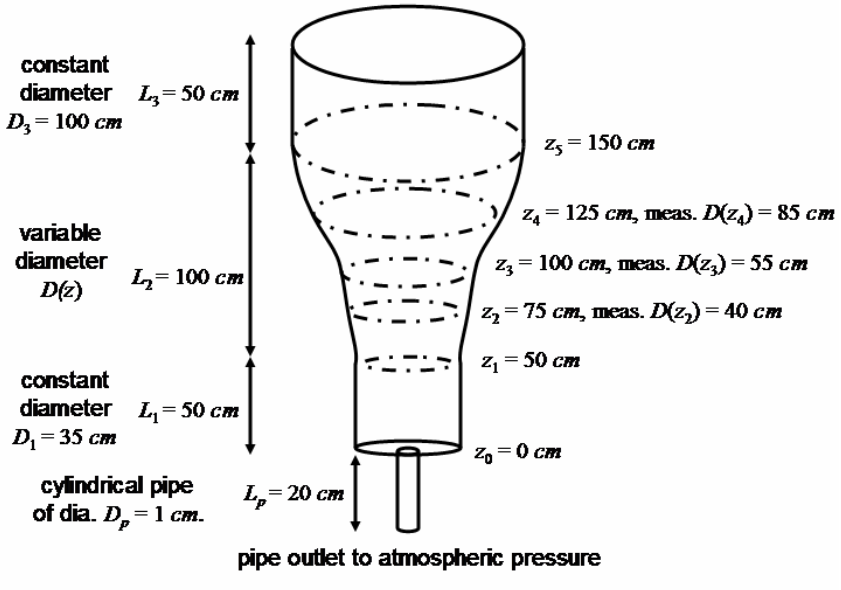
$$f_D = \begin{cases} 64/\mathrm{Re} & \mathrm{Re} < 2100 \\ ??? & 2100 < \mathrm{Re} < 4000 \\ \dfrac{1}{\sqrt{f_D}} = -2\log_{10}\left[\dfrac{(e/D_p)}{3.7} + \dfrac{2.51}{\mathrm{Re}\sqrt{f_D}}\right] & \mathrm{Re} > 4000 \end{cases} \tag{22}$$

$$\mathrm{Re} = \frac{\rho v_p D_p}{\mu} \tag{23}$$

This seemingly has become an iterative nightmare! No problem though! As long as we can phrase it correctly as differential and algebraic equations in ODE15s,

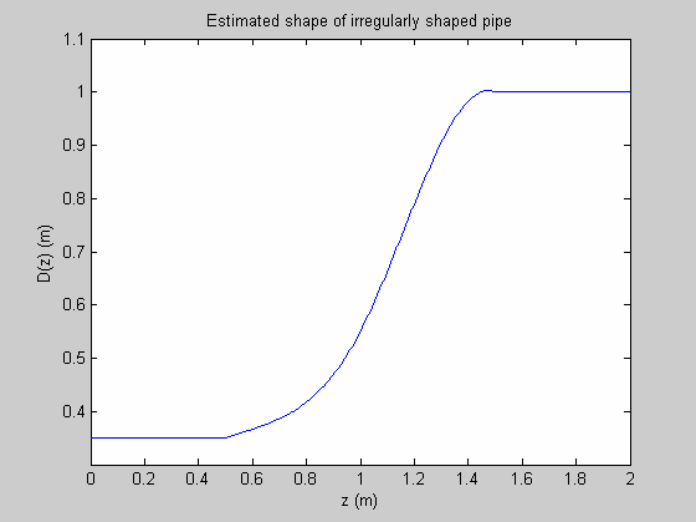
the algorithm will take care of everything! We will come back to these points in a little bit.

We face some difficulties when trying to figure out the geometry of the tank because the diameter changes with respect to h. We are given a few data points with regard to the geometry and we can utilize the matlab function, **interp1**, to approximate the shape of the tank. In our code we will use a 'spline' interpolation. You can choose to determine the tank shape with whatever interpolation you want: linear, sinusoidal, whatever. There are many more open-ended aspects to this problem and we leave it up to you to make your own decisions on what assumptions you choose to make about the shape of the tank As long as you state your assumptions clearly, your answer will be accepted.



We write a subroutine `get_tank_shape.m` which will generate a matrix with values of *z* and their corresponding diameter *D(z)*. It will also return the total volume of the tank, neglecting the volume in the small little pipe at the bottom. The result of such an interpolation is that we get the following vectors plotted against each other in the following graph and a tank volume of:

```
tank_volume =

    0.7859 m^3
```



We can use this tank_volume value as our initial condition for the ODE in V. From the vectors, we know that whatever the water height (or z) is, we have an corresponding diameter for our tank.

Alright. So we've gotten that out of the way. Believe it or not, we have the essentials of what we need to solve this problem. Let's begin by looking at a top down picture:

We need to solve for h(t). We have an ODE describing the behavior of dh/dt (and correspondingly dV/dt, since it is A*dh/dt):

$$\frac{dh}{dt} = f(t) = -\left(\frac{D_p}{D(h)}\right)^2 v_p \qquad (19)$$

With our tank fitting, for any reasonable *h*, we know the *D(h)*. D_p is just a parameter so all we need now is v_p. **ODE15s** can do this, and all we need is an initial condition for *h*. So let's set *h(t=0)*= 2, since our tank is full.

How do we get v_p?

Well we go back to equation (20) and re-arrange to get:

$$0 = \frac{1}{2}\rho {v_p}^2\left[\left(\frac{D_p}{D(h)}\right)^4 - 1\right] + \rho g\left[h + L_p\right] - \frac{1}{2}\rho v_p^2\left[K_L + f_D\left(\frac{L_p}{D_p}\right)\right] \qquad (24)$$

We will solve for v_p within **ODE15s**. We do this by setting up a **Mass Matrix**. How do we do this? Let's look at our situation right now. We have an ODE for h and V. And an algebraic equation for v_p. So we can define our system:

$$\dot{h} = f(h, v_p) \tag{25}$$

$$\dot{V} = p\left(\dot{h}, D\right) \tag{26}$$

$$0 = g(h, v_p, \ldots) \tag{27}$$

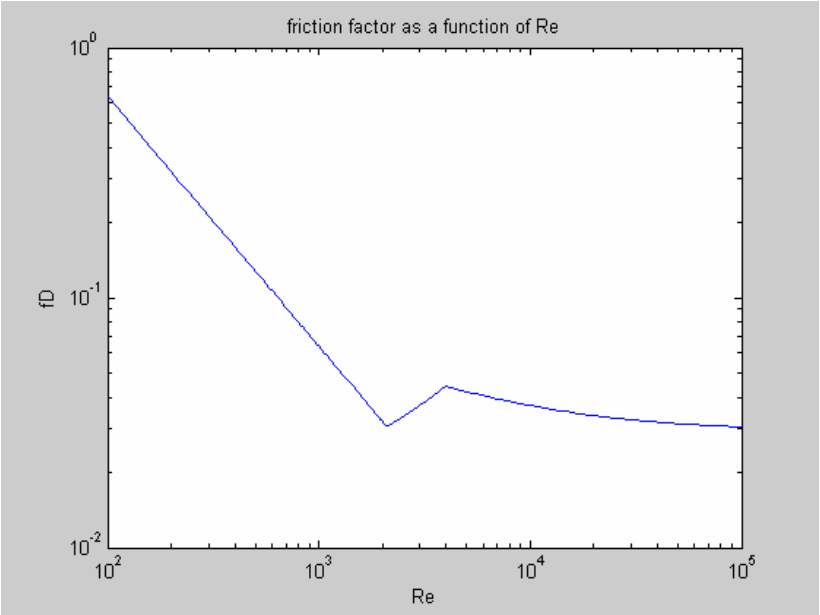
This can be written into the form:

$$M\,\underline{x} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{h} \\ \dot{V} \\ v_p \end{bmatrix} = \begin{bmatrix} f \\ p \\ g \end{bmatrix} \tag{27}$$

We can pass *M* to **ODE15s**, and it will know that the first and second equations will be a differential and the third equation will be an algebraic one. Here *M* = [1 0 0 ; 0 1 0; 0 0 0], as written above. So in this way we can do a simultaneous solution for both h, V and v_p. We recognize that we need some guess for v_p, in order to use **ODE15s**. That is fine, we can take an arbitrary guess or make a more informed guess. We can make an informed guess by plugging in for a few assumed values for the Reynolds number, the f_D and solve for v_p.

Everything in equation (24) is a parameter except for v_p, *D(h)*, *h*, and f_D. Now we have a guess for *h* in our **ODE15s**, and it will be solved for simultaneously, so the algorith will always have a value for it. Likewise we will have *D(h)* just by looking it up from `get_tank_shape.m`.

Ok, so we've taken a reasonable step in the right direction. Now let's move on to see how the friction factor f_D behaves which is the last variable unknown in equation (24). We write a function `get_fD.m` which gives a functional form for how f_D changes with Re. This might help us to get a better guess of v_p to use in **ODE15s**. When we run this function we get:



Of course we see this disconnect between Re = 2100 and Re = 4000. Here we need to make some assumption as to how f_D behaves in this region. You can assume pretty much any functional form including large oscillations or a step function. Here we assume that it transitions linearly, which seems reasonable.

Seeing this graph we can now say, if we needed to guess an f_D and we thought that we might be in a high Re regime, we can say 0.03 might be a good guess.

Back to the structure of our **ODE15s**, it seems reasonable that we might want to keep an algebraic equation for Re, since we need it to calculate f_D. Since to solve for Re, we need a guess of v_p, it seems logical that we might want to make sure that they were simultaneously solved. Again this will go into our function which **ODE15s** calls. We will also modify our mass matrix:

$$\begin{aligned} \dot{h} &= f(h, v_p) \\ \dot{V} &= p(\dot{h}, D(h)) \\ 0 &= g(h, v_p, \ldots) \\ 0 &= k(v_p, \ldots) \end{aligned} \tag{28}$$

giving rise to:

$$M\underline{x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{h} \\ \dot{V} \\ v_p \\ \mathrm{Re} \end{bmatrix} = \begin{bmatrix} f \\ p \\ g \\ k \end{bmatrix} \tag{29}$$

We will need an initial condition for Re, but that can be a rough estimate. We know it will be roughly large because of the high density and low viscosity, approximately. You can do a rough order of magnitude analysis. Now we solve for all four variables, h, V, v_p, and Re, we can calculate f_D. To calculate f_D we need to implement some logic to assess in what regime we are in. If we are in either the laminar or transition regime, to calculate f_D is fairly straightforward. However if we are in the turbulent regime (which we happen to be in all of the process of draining), we need to invoke **fsolve** to solve the implicit equation for f_D. But we remember this from our previous problem sets and it's not too much of a problem.

This is pretty much it. Simultaneously solve for h, V, Re, and v_p, using **ODE15s**, and make occasional function calls to fsolve, looking for the value of f_D.

We write code that can accomplish this:

```
% benwang_P4B1.m
% Ben Wang
% HW#5 Problem #4
% due 10/21/05 9 am

% Tank draining problem
% ======= main routine benwang_P4B1.m
function iflag_main = benwang_P4B1();
iflag_main = 0;

%PDL> clear graphs, screen etc. general initialization
clear all; close all; clc;

%PDL> specify some values specific to the problem
param.viscosity = 1e-3;        % [Pa s]
param.density = 1000;          % [kg/m^3]
param.KL = 0.2;                %
param.Dp = 0.01;               % [m]
param.e = 0.045e-3;            % [m]
param.Lp = 0.2;                % [m]
param.g = 9.8;                 % [m/s^2]
param.h_0 = 2;                 % [m]
param.vp_0 = 5;                % [m/s]
param.Re = 1e4;                % guess

%PDL> get tank shape.  Call a function that returns a matrix that creates a
%correspondence between height and diameter. This function also returns the
```

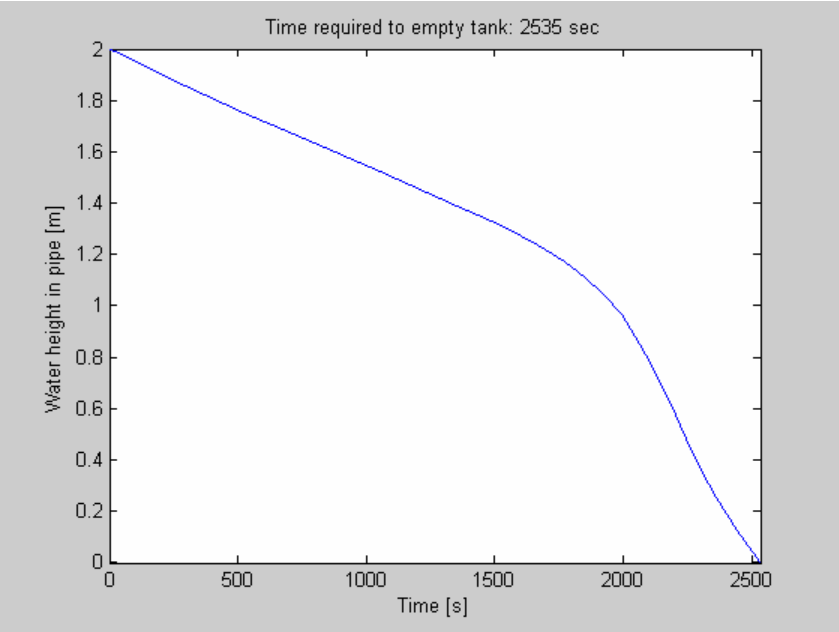
```
%tank volume
[param.diam, param.volume] = get_tank_shape();

%PDL> Determine functional form of fD = fD(Re).  Call a function that returns
%a matrix that corresponds to a given value of fD at a given value of Re
[param.re param.fD] = get_fD(param);

%PDL> specify initial guess
guess = [param.h_0 param.volume param.vp_0 param.Re];

%PDL> solve DAE system.  Include some parameters and options to communicate
%to ODE15s
M = zeros(4,4);     % mass matrix
M(1,1) = 1;         % equation is differential
M(2,2) = 1;         % equation is differential

% we want to automate the termination of this integration when the height
% of the water reaches zero.  We can do this with the Events option
% available to ODE15s

options_ode = odeset('Events',@f_events,'Mass',M,'MassSingular','yes',...
   'MStateDependence','none');
t_end = 3000;

[t_traj, x_traj, TE, YE, IE] = ode15s(@calc_f_P4B1, [0 t_end], guess, options_ode,
param);

% graphs
figure;
plot(t_traj, x_traj(:,1));
xlabel('Time [s]');
ylabel('Water height in pipe [m]');
axis tight;

empty = num2str(round(TE));
title_label = ['Time required to empty tank: ' empty ' sec'];
title(title_label);
figure;
plot(t_traj, x_traj(:,2));
xlabel('Time [s]');
ylabel('Volume of water in pipe [m^3]');
return

%======= subroutine calc_f_P4B1.m
% the meat of the problem.  Where all the DAE equations, appropriate
% function calls to solve for fD are located.
function [f] = calc_f_P4B1(t,x,param)

f = zeros(size(x));
h = x(1);
V = x(2);
vp = x(3);
Re = x(4);
```

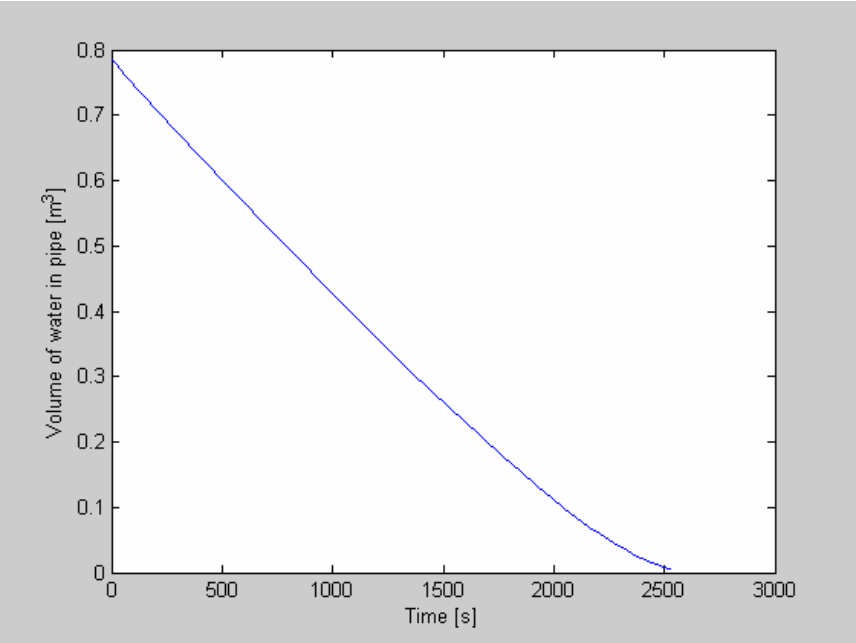
```
if h<0.005
   diam_index = 1;            % check to make sure we don't have negative indices
else
   diam_index = round(h*200);
end
D = param.diam(diam_index,2);

fD = calc_fD(Re, param);

% now we have our DAE system to solve
f(1) = -(param.Dp/D)^2*vp;                    % use physical intuition, tank must drain
f(2) = f(1)*pi/4*D^2;                         % vout*A
f(3) = param.density/2*vp^2*((param.Dp/D)^4-1)+...
   param.density*param.g*(h+param.Lp) - ...
   param.density*vp^2/2*(param.KL + fD*param.Lp/param.Dp);
f(4) = Re - param.density*vp*param.Dp/param.viscosity;
return

%======= subroutine events_f.m
% this function allows the termination of integratoin when the height of
% the water passes through 0
function [value, isterminal, direction] = f_events(t,y, param)
value = y(1);
isterminal = 1;
direction = -1;
return
%====== subroutine calc_fD.m
function fD = calc_fD(Re, param)
options_fsolve = optimset('Display','off');
if Re<2100
   fD = 64/Re;
elseif Re>4000
   fD = fsolve(@calc_fD_function, 64/Re, options_fsolve, param, Re);
else
   % manually implement an interpolation between turbulent and laminar
   % regime
   a = 64/2100;                               % fD at high end of laminar Re
   b = fsolve(@calc_fD_function, 64/Re, options_fsolve, param, 4e3);
   fD = (b-a)/(4000-2100)*(Re-2100) + a;
end
return;
%====== subroutine calc_fD_P4B1.m
% returns a value of fD in the turbulent regime
function f = calc_fD_function(x, param, Re)
fD = x(1);
f(1) = 1/fD^0.5 + 2*log10((param.e/param.Dp/3.7)+2.51/(Re*fD^0.5));
return
%====== subroutine get_fD.m
% this function prints out a diagram showing the anticipated
function [re fD] = get_fD(param)
N = 100;
re_laminar = linspace(100,2100,N);
re_turbulent = linspace(4000,100000,N);
re_transition = linspace(2100,4000,N);
re = [re_laminar re_transition re_turbulent]';
```

```
vp = re*param.viscosity/param.density/param.Dp;
fD_laminar = zeros(length(re_laminar),1);
fD_turbulent = zeros(length(re_turbulent),1);

options_fsolve = optimset('Display','off');
for i=1:N
   fD_laminar(i) = 64/re_laminar(i);
   fD_turbulent(i) = fsolve(@calc_fD_function, fD_laminar(i), options_fsolve, param, ...
      re_turbulent(i));
end

a = [re_laminar(N) re_turbulent(1)];
b = [fD_laminar(N) fD_turbulent(1)];
fD_transition = interp1(a,b,re_transition,'linear');
fD = [fD_laminar; fD_transition'; fD_turbulent];
figure;
loglog(re,fD);
xlabel('Re');
ylabel('fD');
title('friction factor as a function of Re');
return

%======= subroutine get_tank_shape.m
% returns a matrix containing values of z and the corresponding diameter of
% the tank for each value of z
function [A, volume] = get_tank_shape()

z_lo = linspace(0,0.5,100);
z_hi = linspace(1.5,2,100);
z_mid = linspace(0.5,1.5,200);
d_lo = 0.35*ones(size(z_lo));
d_hi = ones(size(z_hi));

% now to get d_mid we need to interpolate from the data that we were given
% in the problem specification

a = [0.5; 0.75; 1.0; 1.25; 1.5];
b = [0.35; 0.4; 0.55; 0.85; 1];
d_mid = interp1(a,b,z_mid,'spline');
z = [z_lo z_mid z_hi]';
d = [d_lo d_mid d_hi]';
A = [z d];

figure;
plot(A(:,1),A(:,2));
xlabel('z (m)');
ylabel('D(z) (m)');
title('Estimated shape of irregularly shaped pipe');
% we need to get the inital volume in the tank.  we can get this by using
% trapz on our new interpolation vectors
d_volume = pi/4*(d.^2);
volume = trapz(z,d_volume);

tank_volume = volume
```

```
return
```

The output from this program will yield:





and we find that it takes: 2535 seconds for the tank to drain!