Problem Set 3 Nonlinear Algebraic Systems
10.34 Fall 2005, KJ Beers

Ben Wang, Mark Styczynski
September 28, 2005
Revised: October 7, 2005

Problem 1: 2.A.1 (2 points)

You are asked to calculate the Jacobian matrix by hand for the following equations:

$$f_1(x_1, x_2) = x_1^3 - 3x_1x_2^2 - x_2 + 18 = 0 \quad (1)$$

$$f_2(x_1, x_2) = x_1^2 - 4x_1^2x_2 + x_2^3 - 2x_2^2 + 28 = 0 \quad (2)$$

The Jacobian Matrix is calculated by taking the derivative of each equation with respect to each variable such that each element of the Jacobian can be defined as:

$$J_{im}\Big|_{\underline{x}^{[k]}} = \frac{\partial f_i}{\partial x_m}\Bigg|_{\underline{x}^{[k]}} \quad (3)$$

Solving for our two equations we get the following matrix:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} = 3x_1^2 - 3x_2^2 & \frac{\partial f_1}{\partial x_2} = -6x_1x_2 - 1 \\ \frac{\partial f_2}{\partial x_1} = 2x_1 - 8x_1x_2 & \frac{\partial f_2}{\partial x_2} = -4x_1^2 + 3x_2^2 - 4x_2 \end{bmatrix} \quad (4)$$

Beginning with an initial guess:

$$\underline{x}^{[0]} = \begin{bmatrix} 2 & 1 \end{bmatrix}^T \quad (5)$$

We can derive an expression for $\underline{\mathbf{x}}^{[1]}$ by calculating $\underline{p}$ from the following update equation:

$$J^{[k]} \underline{p}^{[k]} = -\underline{f}\left(\underline{x}^{[k]}\right) \tag{6}$$

For k = 0, we can calculate the values of **J** and **f** by plugging in for $\mathbf{\underline{x}}^{[0]}$ into equations (1), (2), and (4):

$$\begin{bmatrix} 9 & -13 \\ -12 & -17 \end{bmatrix} \begin{bmatrix} p_1^{[0]} \\ p_2^{[0]} \end{bmatrix} = -\begin{bmatrix} 19 \\ 15 \end{bmatrix} \tag{7}$$

This is our classic A$\underline{x}$ = $\underline{b}$ and can be solved with Gaussian elimination. Now this can be done with Gaussian elimination or a little handiwork or if we are really lazy, which I am, use the "\" operator in Matlab. Our resulting first guess is:

$$\underline{p}^{[0]} = \begin{bmatrix} -0.4142 & 1.1748 \end{bmatrix}^T \tag{8}$$

Now that we have determined our update, we can calculate $\mathbf{\underline{x}}^{[1]}$ by plugging it into the update equation:

$$\underline{x}^{[1]} = \underline{x}^{[0]} + \underline{p}^{[0]} \tag{9}$$

and using $\mathbf{\underline{x}}^{[1]}$ as the new "initial guess" for the 2nd iterative step we evaluate $\underline{f}(\underline{x})$:

$$\underline{f}^{[1]} = \begin{bmatrix} -2.6870 & 9.4663 \end{bmatrix}^T \tag{10}$$

Now how do we check if reduced step Newton's method will accept this step? We look at the 2-norm of the values of the two equations and look if our values satisfy the following relationship

$$\left\| \underline{f}\left(\underline{x}^{[1]}\right) \right\|_2 < \left\| \underline{f}\left(\underline{x}^{[0]}\right) \right\|_2 \tag{11}$$

The values we calculate for this does indeed satisfy this equation:

$$\left\| \underline{f}\left(\underline{x}^{[1]}\right) \right\|_2 = 9.84 < \left\| \underline{f}\left(\underline{x}^{[0]}\right) \right\|_2 = 24.21 \tag{12}$$

So this move will not be required to implement a reduced step.

This problem was graded in the following manner:

- 0.5 points for correct Jacobian

- 0.5 points for getting correct x[1]
- if x[1] is incorrect then 0.5 points are awarded if del_x or p is correct
- 0.5 points for discussion of reduced step method via the norm

Problem 2: 2.A.2 (2 points)

This problem is the numerical implementation of problem 2.A.1. Professor Beers has OK'd the use of FSOLVE as a tool to accomplish this problem and surely an appropriate implementation using FSOLVE will be accepted. The following code can be used to accomplish this:

```
% benwang_P2A2.m
% Ben Wang
% HW#3 Problem #1
% due 9/28/05 9 am

% We solve a system of nonlinear algebraic equations with FSOLVE

% ======= main routine benwang_P2A2.m
function iflag_main = benwang_P2A2();

iflag_main = 0;

%PDL> clear graphs, screen etc. general initialization

clear all; close all; clc;

%PDL> Determine relevant initial guess

x0 = [2;1];

Options = optimset('LargeScale','off','Tolfun', 1e-8);
Options = optimset(Options,'Jacobian','on');

[x,f,Jac] = fsolve(@calc_NAE,x0,Options);

iflag_main = 1;

x
f

return;

%====== subroutine calc_NAE ===========

% NAE stands for nonlinear algebraic equations
function [f,Jac] = calc_NAE(x);

%PDL> Determine relevant equations for system
f = zeros(2,1);
```

```
f(1) = x(1)^3 - 3*x(1)*x(2)^2 - x(2) + 18;
f(2) = x(1)^2 - 4*x(1)^2*x(2) + x(2)^3 - 2*x(2)^2 + 28;

%PDL> Write out the Jacobian for our system
Jac = zeros(2,2);
Jac(1,1) = 3*x(1)^2 - 3*x(2)^2;
Jac(1,2) = -6*x(1)*x(2) - 1;
Jac(2,1) = 2*x(1) - 8*x(1)*x(2);
Jac(2,2) = -4*x(1)^2 + 3*x(2)^2 - 4*x(2);

return;
```

Your output should look something like:

```
Optimization terminated: first-order optimality is less than options.TolFun.

x =

    2
    2


f =

    0
    0


ans =

    1
```

Given an initial guess of [2 ; 1], you should arrive at a solution of [2 ; 2]. If you start with other initial guesses, fsolve will potentially find other viable solutions. Remember that Newton's method can find one of multiple solutions. (Thanks to David Liu for experimenting with this.)

However, if you read the problem statement, it asks you to implement a single step of a Newton's method step and to check if a reduced-step Newton's method would accept the given first step. Effectively you might interpret this problem assignment as implementing Newton's method on your own, or at least a single step of Newton's method. This will give you insight into how you take a nonlinear algebraic system of equations and 'convert' it into a linear algebraic system of equations, by way of the Jacobian, as illustrated in equations 6 – 9 in this document. The following code can be used to solve this:

```
% benwang_HW3_P2.m
% Ben Wang
% HW#2 Problem #1
% due 9/28/05 9 am
```

```matlab
% For this problem we write a routine that will allows us to perform a
% series of  iterative step using Newton's method without a reduced step
% to solve a simple system of nonlinear algebraic equations.

% ======= main routine benwang_HW3_P2.m
function iflag_main = benwang_HW3_P2();

iflag_main = 0;

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

%PDL> Set a tolerance for convergence
tolerance = 1e-6;

%PDL> Input relevant initial guess
x0 = [2;1];

%PDL> We call a function that evaluates the value of the nonlinear
%algebraic equations to be solved
[f] = calc_func(x0);

%PDL> Store in a vectors: magnitude, z, y: the norm, x-coordinate, and
%y-coordinate for each Newton step.
magnitude(1) = norm(f);
z(1) = x0(1);
y(1) = x0(2);

i = 1;      % initializes number of steps required for convergences
%PDL> Call a while loop that while the norm(f) is greater than some specified
% tolerance, it will continue to make Newton steps, and record


while norm(f) > tolerance
   i = i+1;                        % update number of Newton steps
   [f,x] = calc_func_iter(x0);     % calculate a Newton update step
   x0 = x;                         % re-evaluate x for a new "initial" guess
   magnitude(i) = norm(f);         % calculate and store the norm
   z(i) = x(1);                    % calculates the x-axis of our current Newton location
   y(i) = x(2);                    % calculates the y-axis of our current Newton location
end

numsteps = i-1;                    % total steps moved
plot(z,y,'-o')
xlabel('x(1)');
ylabel('x(2)');
iflag_main = 1;
f
x
numsteps
return;

%====== subroutine calc_func ===========
```

```
% This is a function that returns the value of f given an input of x
function [f] = calc_func(x)

%PDL> Determine relevant equations for system
f = zeros(2,1);
f(1) = x(1)^3 - 3*x(1)*x(2)^2 - x(2) + 18;
f(2) = x(1)^2 - 4*x(1)^2*x(2) + x(2)^3 - 2*x(2)^2 + 28;

return;

%====== subroutine calc_func_iter ===========

function [f,x] = calc_func_iter(x)

% This is a function that performs a single Newton step

%PDL> Determine relevant equations for system
f = zeros(2,1);
f(1) = x(1)^3 - 3*x(1)*x(2)^2 - x(2) + 18;
f(2) = x(1)^2 - 4*x(1)^2*x(2) + x(2)^3 - 2*x(2)^2 + 28;

%PDL> Write out the Jacobian for our system
J = zeros(2,2);
J(1,1) = 3*x(1)^2 - 3*x(2)^2;
J(1,2) = -6*x(1)*x(2) - 1;
J(2,1) = 2*x(1) - 8*x(1)*x(2);
J(2,2) = -4*x(1)^2 + 3*x(2)^2 - 4*x(2);

%PDL> To calculate our update to x, we solve the linear equation J*p = -f

p = J\-f;
x = x+p;
f = calc_func(x);
return;
```

The output should look something like this:

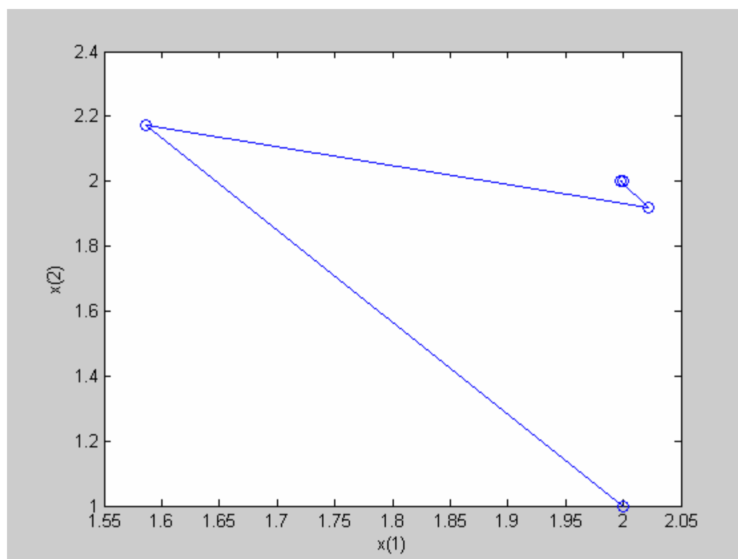
```
f =

  1.0e-010 *

    0.1351
    0.3487


x =

    2.0000
    2.0000


numsteps =
```

```
     5


ans =

     1
```

and if we access the [f, x] values after one step we get :

```
x =

    1.5858
    2.1748


f =

   -2.6870
    9.4663
```

These numbers are similar to the ones we calculate by hand. We can plot all the consecutive steps in an iterative form of this step and arrive at:



Point distribution:

- 1 point were awarded for workable code
- 1 point was awarded if you arrived at either x[1] or x[solution]

Problem 3: 2.B.2 (3 points)

This problem is largely about setting up the correct mass balance equations and then utilizing FSOLVE to solve a set of nonlinear algebraic equations that describe a CSTR, very much like the example that Professor Beers went over in class. We solve for the steady state concentrations where changes in species

concentration with time approach zero, allowing us to utilize Newton's method. The main difference in this case is that we have 5 species to account for, 3 reactions, different feed ratios, and different temperatures.

We assume that there is negligible volume change (V_{CSTR} is constant) associated with species generation resulting from reactions and assume isothermal operation. This can be expressed with the following:

$$[\text{accumulation}] = [\text{input}] - [\text{output}] + [\text{generation by reaction}] \quad [=] \ \frac{\text{mol}}{\text{s}}$$

Our equations can be expressed as:

$$\frac{dV_{CSTR}C_A}{dt} = 0 = v_0 C_{A,0} - v_0 C_A - V_{CSTR} k_1 C_A C_B - V_{CSTR} k_3 C_A \qquad (13)$$

$$\frac{dV_{CSTR}C_B}{dt} = 0 = v_0 C_{B,0} - v_0 C_B - V_{CSTR} k_1 C_A C_B - V_{CSTR} k_2 C_B C_C \qquad (14)$$

$$\frac{dV_{CSTR}C_C}{dt} = 0 = -v_0 C_C + V_{CSTR} k_1 C_A C_B - V_{CSTR} k_2 C_B C_C \qquad (15)$$

$$\frac{dV_{CSTR}C_D}{dt} = 0 = -v_0 C_D + V_{CSTR} k_2 C_B C_C \qquad (16)$$

$$\frac{dV_{CSTR}C_E}{dt} = 0 = -v_0 C_E + V_{CSTR} k_3 C_A \qquad (17)$$

These equations are part of our function [f] = `calc_concentrations()`, which fsolve calls and solves.

The problem asks to vary temperature (we are given rate data for two temperatures) and a parameter gamma (which is the ratio of input feed of B to input feed of A). The easiest way to implement this is to nest a few for loops and make repeated calls to FSOLVE, while passing on different values for rate data and gamma.

The last thing we need is an initial guess: we can begin with a simple guess of initial feed concentration for A and B, with C, D and E = 0. So thus our initial guess:

$$\begin{bmatrix} C_{A,0} & C_{B,0} & 0 & 0 & 0 \end{bmatrix} \qquad (18)$$

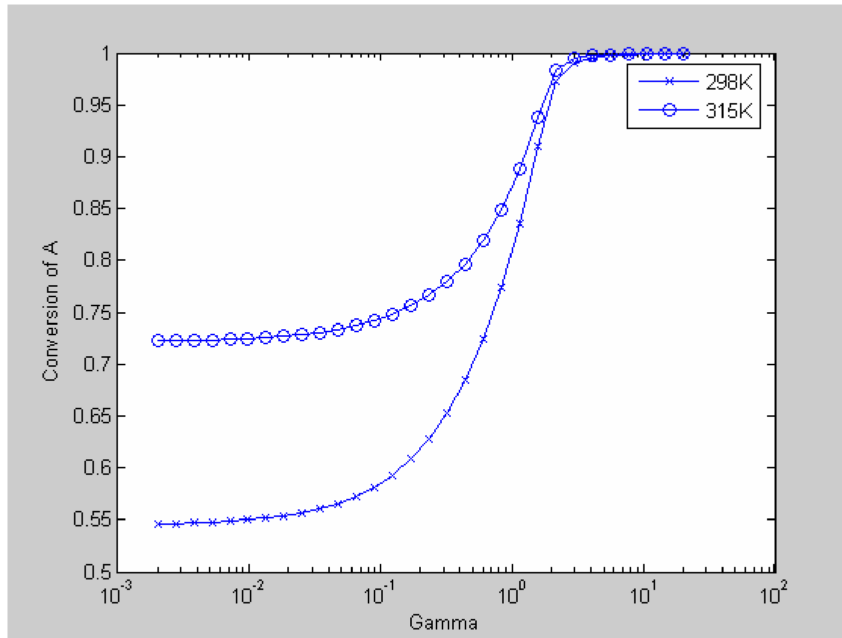
Armed with this information we can use the following code to create a plot of conversion of A vs. gamma and T:

```
% benwang_P2B2.m
% Ben Wang
% HW#3 Problem #3
% due 9/28/05 9 am

% For this problem we write a routine that will allow us to solve the
% steady state CSTR problem with 5 species and 3 reactions at (least) 2
% different temperatures.  We will make basic assumptions of negligible
% volume change and isothermal operation

% ======= main routine benwang_P2B2.m
function iflag_main = benwang_P2B2();
iflag_main = 0;

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

%PDL> Set data values and store in data structure Data with corresponding
%units

Data.T = [298; 315];
Data.V = 1000;          % [Liters]
Data.v = 0.1;           % [Liters/s]
Data.c_A0 = 0.5;        % [M]
Data.k1(1) = 2.1e-2;    % [L/mol-s] 298K
Data.k2(1) = 1.5e-2;    % [L/mol-s] 298K
Data.k3(1) = 1.2e-4;    % [1/s]     298K
Data.k1(2) = 3.6e-2;    % [L/mol-s] 298K
Data.k2(2) = 4.5e-2;    % [L/mol-s] 298K
Data.k3(2) = 2.6e-4;    % [L/s]     298K

c_B0 = logspace(-3,1,30);                    % a variety of c_B0 inputs
gamma = c_B0/Data.c_A0;                        % convert conc into rel conc.

% Initialize matrices that store the information for each call of FSOLVE
Full_data = zeros(2, length(gamma), 5);
conversion_A = zeros(2, length(gamma));

% Fsolve options
Options = optimset('LargeScale','off','Display','off');

%PDL> Set up nested for loops that can pass different experimental
%parameters

for i = 1:length(Data.T)
   for j = 1:length(gamma)

      % Selects the appropriate rate constant
      Data.k1_T = Data.k1(i);
```

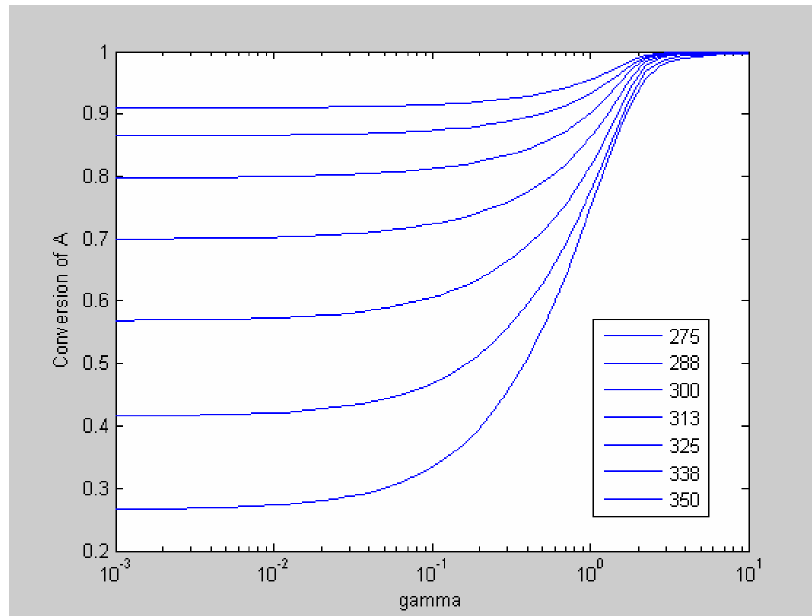
```
        Data.k2_T = Data.k2(i);
        Data.k3_T = Data.k3(i);

        % Selects the correct feed of B
        Data.c_B0 = gamma(j)*Data.c_A0;

        % Determine relevant initial guess
        c0 = [Data.c_A0, Data.c_B0, 0, 0, 0];

        % Call Fsolve to solve nonlinear algebraic equations
        [x,f] = fsolve(@calc_concentrations, c0, Options, Data);

        % Save all of the returned data in matrix Full_Data
        Full_Data(i,j,1) = x(1);
        Full_Data(i,j,2) = x(2);
        Full_Data(i,j,3) = x(3);
        Full_Data(i,j,4) = x(4);
        Full_Data(i,j,5) = x(5);

        % Calculates conversion of A :(Initial - Final)/Initial
        conversion_A(i,j) = (Data.c_A0 - x(1))/Data.c_A0;
    end
end

figure;
semilogx(gamma,conversion_A(1,:),'-x');
hold on;
semilogx(gamma,conversion_A(2,:),'-o');
xlabel('Gamma');
ylabel('Conversion of A');
legend('298K', '315K');

iflag_main = 1;

return;

%====== subroutine calc_concentrations ===========
% This is a function that returns the value of f given an input of x
function [f] = calc_concentrations(x, Data)

% change names to more physical handles
cA = x(1); cB = x(2); cC = x(3); cD = x(4); cE = x(5);

f = zeros(5,1);

% balance of A including input, output, reactions
f(1) = Data.c_A0*Data.v - Data.v*cA - Data.V*Data.k1_T*cA*cB - ...
      Data.V*Data.k3_T*cA;                                          % [mol/s]

% balance of B including input, output, reactions
f(2) = Data.c_B0*Data.v - Data.v*cB - Data.V*Data.k1_T*cA*cB - ...
      Data.V*Data.k2_T*cC*cB;
```

```
% balance of C including output, reactions
f(3) = - cC*Data.v + Data.V*Data.k1_T*cA*cB - Data.V*Data.k2_T*cC*cB;

% balance of D including output, reactions
f(4) = - Data.v*cD + Data.V*Data.k2_T*cC*cB;

% balance of E including output, reactions
f(5) = - Data.v*cE + Data.V*Data.k3_T*cA;

return;
```

You should be able to get a graph that looks like:



Your sexier TA, Mark, fit the given rate constants to Arrhenius relationships and can now plot out operation of the CSTR at many different temperatures than only those specified. It is only necessary to plot conversion as a function of gamma and T, but plotting more curves can give you a better understanding of your program and of the behavior of this CSTR.



Problem 4: 2.B.3 (3 points)

This problem combines what we learned from the Chapter 1 (Finite Differences/Linear Systems) with what we learned in Chapter 2 (Nonlinear Algebraic Equations) and is very similar to a problem in a previous assignment. We are asked to nondimensionalize the equation:

$$0 = D\frac{d^2C_A}{dx^2} - kC_A^2 \tag{19}$$

We can use the following dimensionless quantities:

$$\frac{C_A}{H_AP_A} = \phi \qquad\qquad \frac{2x}{B} = \eta \qquad\qquad (20), (21)$$

and rewrite the equation:

$$0 = \frac{4H_AP_AD}{B^2}\frac{d^2\phi}{d\eta^2} - k(H_AP_A)^2\phi^2 \tag{22}$$

collecting similar terms yields:

$$0 = \frac{4D}{k(H_A P_A)B^2} \frac{d^2\phi}{d\eta^2} - \phi^2 \qquad (23)$$

We can collect all the terms in front of the Laplacian and call it the Damkohler number (which here represents the ratio of Diffusion to Reaction). This may be different than what Deen's book says, but it would be related to it by a power of 2 or the inverse.

$$0 = Da \frac{d^2\phi}{d\eta^2} - \phi^2 \qquad (24)$$

Now we have only one adjustable parameter which we will use!

Next we are asked to use finite differences to convert this nonlinear differential equation into a system of nonlinear algebraic equations, recalling:

$$\left. \frac{d^2\phi}{d\eta^2} \right|_i = \frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{(\Delta\eta)^2} \qquad (25)$$

The set of equations for i = 2:N-1 will be:

$$\frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{(\Delta\eta)^2} - \frac{\phi_i^{\,2}}{Da} = 0 \qquad (26)$$

With the boundary conditions accounted for by:

left BC: $$\frac{-2\phi_1 + \phi_2}{(\Delta\eta)^2} - \frac{\phi_1^{\,2}}{Da} - \frac{1}{(\Delta\eta)^2} = 0 \qquad (27)$$

right BC: $$\frac{-2\phi_{N-1} + \phi_N}{(\Delta\eta)^2} - \frac{\phi_{N-1}^{\,2}}{Da} - \frac{1}{(\Delta\eta)^2} = 0$$

(28)

We now have N nonlinear equations that cannot be solved with the linear solver "\". It can be solved with FSOLVE quite easily.

Because N can be a very large number depending on how fine you want the grid mesh to be, we are asked to calculate the Jacobian and pass it to FSOLVE to minimize computation time. Normally, calculating the Jacobian by hand, for even a 10x10 matrix (containing 10 equations, 10 variables) can be time consuming.

However if we look at our equations we see that determining the Jacobian is easy and automatable.

It is easy because we note that the Jacobian of a finite difference system is typically sparse and diagonal. The only nonzero terms are located:

$$J_{ij} = \frac{df_i}{d\eta_j} \qquad \begin{cases} j = i-1 \\ j = i \\ j = i+1 \end{cases} \tag{29}$$

In fact most of the Jacobian, from i = 2:N-1 will be described by:

$$\begin{cases} J_{i,i-1} = \dfrac{1}{(\Delta\eta)^2} \\ J_{i,i} = -\dfrac{2}{(\Delta\eta)^2} - \dfrac{2\phi_i}{(\Delta\eta)^2} \\ J_{i,i+1} = \dfrac{1}{(\Delta\eta)^2} \end{cases} \tag{30}$$

With the differences at the boundaries being:

$$\text{left BC:} \qquad J_{1,1} = \frac{-2}{(\Delta\eta)^2} - \frac{2\phi_1}{Da} \qquad J_{1,2} = \frac{1}{(\Delta\eta)^2} \tag{31}$$

$$\text{right BC:} \qquad J_{N,N-1} = \frac{1}{(\Delta\eta)^2} \qquad J_{N,N} = \frac{-2}{(\Delta\eta)^2} - \frac{2\phi_N}{Da} \tag{32}$$

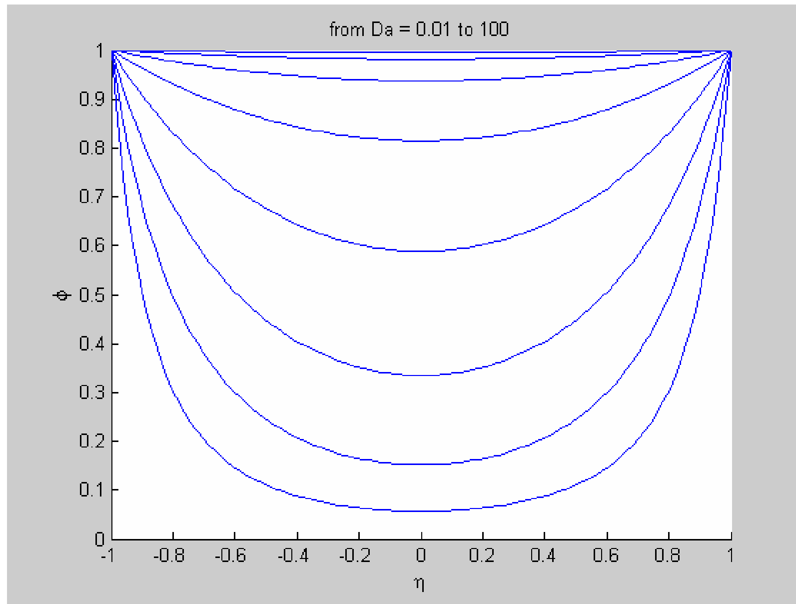
These are all the equations you need and you can write a routine loop that calls FSOLVE multiple times to solve for an array of Damkohler numbers. The following code should be able to accomplish this:

```
% benwang_P2B3.m
% Ben Wang
% HW#3 Problem #4
% due 9/28/05 9 am
```

```
% This problem is similar to problem from Assignment #2, only this time
% we have a system of nonlinear algebraic equations rather than a linear
% system.  We use finite differences to transform the differential equations
% into algebraic equations and then include the nonlinear term to form the
% set of equations to solve.  We then use FSOLVE.

% ======= main routine benwang_P2B3.m
function iflag_main = benwang_P2B3();
iflag_main = 0;

%PDL> clear graphs, screen etc. general initialization

clear all; close all; clc;

%PDL> Let's store the parameters about the spatial grid in a data structure
%Grid.

Grid.N = 100;
Grid.B = 2;
Grid.dy = Grid.B/(Grid.N+1);

%PDL> Let's store the bulk of our parameters in a data structure
%Parameters

Parameters.diff = 1;
Parameters.k = 1;
Parameters.H_A = 1;
Parameters.P_A = 1;

%PDL> The problem is easily nondimensionalized so that there is only one
%relevant parameter that uses all these values in the Parameters data
%structure.  It comes out to be a Damkohler number, which is D/kcb^2.
%This is a difference compared to assignment HW#2, P4. We then
%nondimsionalize concentration with H_A*P_A.

%PDL> The remaining differential equation is just (Da)d^2(phi)/dy^2 = phi^2.
%We can convert this to finite differences relatively easy.

% Specify a range of Damkohler numbers
Parameters.DaVector = logspace(-2,2,8);

% Let's create the grid.
y = linspace(-Grid.B/2, Grid.B/2, Grid.N+2);

figure;
hold on;

% Give an an initial guess for the concentration(y) as = 0.
zero_guess = zeros(Grid.N,1);

Options = optimset('Jacobian','on','Display','off');
```

```
%Initialize matrix to store all values of concentration as function of Da
conc = zeros(Grid.N,length(Parameters.DaVector));

for j = 1: length(Parameters.DaVector)
   [x1(:,j),f1] = fsolve(@sys_eqn, zero_guess, Options, Grid, Parameters,j);
   plot(y,[1;x1(:,j);1],'-');
end

xlabel('\eta');
ylabel('\phi');

iflag_main = 1;

return;

% ===== function sys_eqn =====
function [f, Jac] = sys_eqn(x, Grid, Parameters,k)

f = zeros(Grid.N,1);

% now we define the Jacobian, which is a tri-diagonal matrix with the only
% non-zero elemetns located along the main diagonal

Jac = spalloc(Grid.N, Grid.N, 3*Grid.N);

% Left Boundary Conditions
f(1) = (-2*x(1) + x(2))/Grid.dy^2 - x(1)^2/Parameters.DaVector(k) + (Parameters.H_A *
Parameters.P_A)/Grid.dy^2;
Jac(1,1) = -2/Grid.dy^2 - 2*x(1)/Parameters.DaVector(k);
Jac(1,2) = 1/Grid.dy^2;

% These equations will account for the finite differences in the middle of
% the domain
for i = 2: Grid.N-1
   f(i) = (x(i-1) - 2*x(i) + x(i+1))/Grid.dy^2 - x(i)^2/Parameters.DaVector(k);
   Jac(i, i-1) = 1/Grid.dy^2;
   Jac(i, i) = -2/Grid.dy^2 - 2*x(i)/Parameters.DaVector(k);
   Jac(i, i+1) = 1/Grid.dy^2;
end

% Right boundary conditions
f(Grid.N) = (x(Grid.N-1) - 2*x(Grid.N))/Grid.dy^2 - x(Grid.N)^2/Parameters.DaVector(k) +
...
   (Parameters.H_A * Parameters.P_A)/Grid.dy^2;

Jac(Grid.N, Grid.N-1) = 1/Grid.dy^2;
Jac(Grid.N, Grid.N) = -2/Grid.dy^2 - 2*x(Grid.N)/Parameters.DaVector(k);

return
```

Your output should look something like:



where you have a family of graphs that correspond to different Damkohler numbers.

Comments:

- In the future, use optimset to turn off the command line output while using FSOLVE
- Remember to write routines or functions that can be run in closed form. That is, if you want to (and you should) write your functions in modular form, you should also write an overarching main() function that calls all these separate functions for your task. All parameters and other important problem-specific data should be housed here.
- Make sure you have all the values at the edges go to the correct boundary value. If you use insufficient grid points, or don't include that point, you will have values that are wrong at the edges.