10.34: Numerical Methods Applied to Chemical Engineering
Prof. K. Beers
Solutions to Problem Set 2: Linear Algebra

Mark Styczynski, Ben Wang

1. (2 points) *Solve problem 1.A.3, with your routine taking the name username_calc_poly_coeff.m.*

The problem can be formulated as

$$p(x_k) = a_0 + a_1 x_k + a_2 x_k^2 + \cdots + a_N x_k^N$$

for all x_k. We also note that this approximation is for

$$x_0 < x_1 < x_2 < \cdots < x_N$$

Where the N in both denotes the same value. So, we can make an N^{th} degree polynomial approximation whenever we have N+1 points. This overall problem can be expressed in matrix notation as

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ 1 & x_2 & x_2^2 & \cdots & x_2^N \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^N \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} p(x_0) \\ p(x_1) \\ p(x_2) \\ \vdots \\ p(x_N) \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_N) \end{bmatrix}$$

Or much more simply as

$$X\underset{\sim}{a} = \underset{\sim}{p} = \underset{\sim}{f}$$

So we just multiply by the inverse of X, which is the same as Gaussian elimination or the backslash operator.

Code to calculate the polynomial coefficients can be found below.

```
function a = marksty_calc_poly_coeff(x,f)
% function a = marksty_calc_poly_coeff(x,f)
% Input:
%  x: N-element column vector of x-coordinates to be fit
%  y: N-element column vector of y-coordinates, which are the
%    actual values of the function to be fit for all x
% Output:
%  a: N-element column vector of polynomial coefficients
%    a = [a_0, a_1, ... a_(N-1)]

% PDL> First check to make sure that input is valid
if (length(x) ~= length(f)),
    error('myHomework:badInput', ...
        'Length of x and f vectors not the same!  Exiting.')
end

% PDL> Find out how many data points need to be fit; the best
%   polynomial fit is one less than the number of points
numPoints = length(x);
```

```
approxDegree = numPoints - 1;

% PDL> Make a matrix containing the manipulations of each
%  X value... [1, x, x^2, x^3, ...]
%  So x is being raised to one less than the column number
X = [];
for i=1:approxDegree + 1,
    X(:,i) = x.^(i-1);
end

% PDL> Calculate the polynomial coefficients
a=X\f;
```

Test and show the results of your program by fitting f(x) = exp(x) at the points x = 1, 1.5, and 2. What happens to the accuracy of your approximation as you move outside of the range [1, 2]?

We can now use a script to call this function and then create a plot to show the fit both in the region approximated and outside of it.

```
% Mark Styczynski
% 10.34
% HW 2, P1A3

% PDL> Get all of the necessary data calculated
clear all; close all;
x = [1 1.5 2]';
f = exp(x);

% PDL> Get polynomial coefficients
a = marksty_calc_poly_coeff(x,f);

% PDL> Make a new vector to plot the functions in a more
%  detailed manner
xnew = linspace(-1,4)';
p = zeros(size(xnew));
for i=1:length(x),
    p = p + a(i)*xnew.^(i-1);
end

% PDL> Plot and make pretty
figure(1)
plot(xnew,exp(xnew),'-')
hold on
plot(xnew,p,'--')
plot(x,f,'o')
xlabel('x')
ylabel('e^x')
legend('Exact function','Polynomial fit','Location','Best')
title(['A polynomial fit of degree ' int2str(length(x))-1 ...
    ' of e^x: Problem 1.A.3'])
```

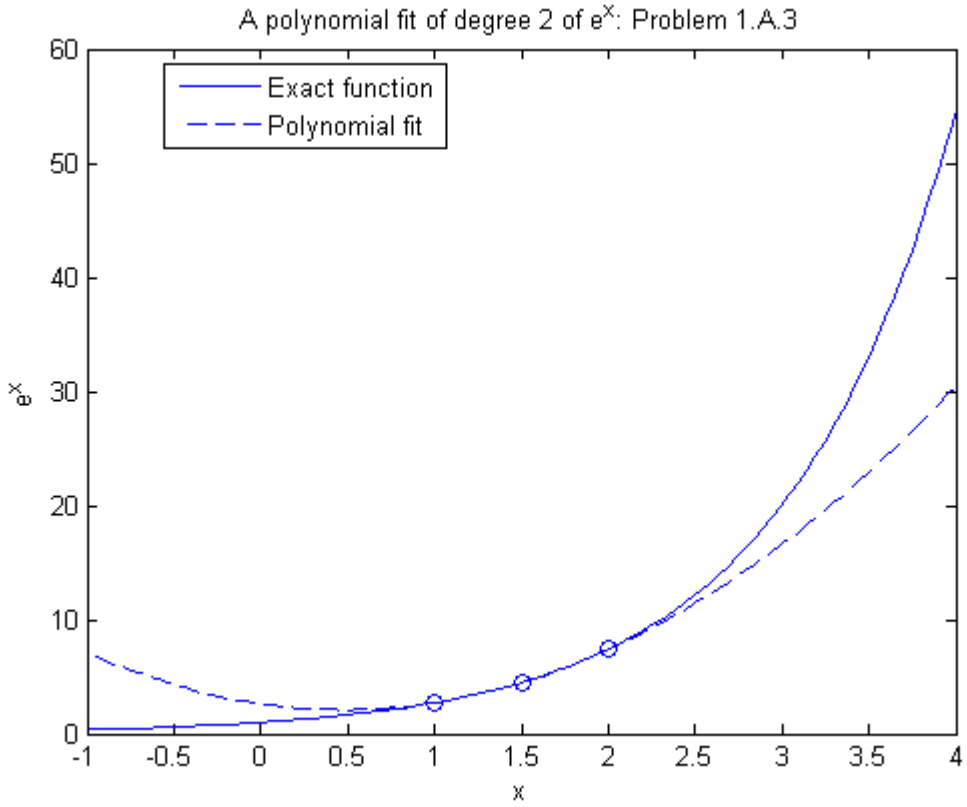
The resulting values of a are:

>> marksty_P1A3

a =

2.6233
-2.1930
2.2879

We can plot this up and see that in the region between 1 and 2, the approximation is pretty good (with the three data points given being fit exactly), but as we move a bit outside of this region, the approximation is not at all acceptable. The plot looks like the one below:



Grading:
(no points off)
Input should support column vectors
x and f should be inputted as function arguments

(1 point off)
Polynomial degree needs to be less than the number of data points to be meaningful here
Plot and explanation insufficient to prove how good polynomial approximation is outside of range
Hard-coding data or significant errors in code

2. (3 points) Solve problem 1.A.4 and name your routine *username_Gram_Schmidt.m.*

We need to take as input a vector $\underline{v}$ of unknown dimension N and return an orthonormal basis set containing $\underline{v}$. Note that this implies that the input vector $\underline{v}$ has a magnitude of 1.

In order to perform Gram-Schmidt orthogonalization, we need not one vector but a set of linearly independent vectors. So, we need to take $\underline{v}$ and from it determine what other vectors we can use that are linearly independent of $\underline{v}$ and thus span space of dimension N. There are many ways to do this, and any method you choose that gets this done will suffice. I'll choose what I believe to be an easy method: base the linearly independent vectors on the traditional unit magnitude orthonormal vectors $\underline{e}^{[i]}$ for all i between 1 and N. This is equivalent to an identity matrix of dimension N. Using this as the starting point, but realizing that $\underline{v}$ needs to be in our basis set, we see that the next step is to choose which $\underline{e}^{[i]}$ should be replaced by $\underline{v}$ such that the entire new set is still linearly independent. Since each $\underline{e}^{[i]}$ has only one non-zero value, and that non-zero value occurs nowhere else in the matrix of all $\underline{e}^{[i]}$, we must make sure that $\underline{e}^{[i]}$ is replaced by a vector without a zero in row i; otherwise, the set of vectors will no longer be linearly independent, nor will it be a basis set. Thus, $\underline{v}$ can be put in any column j for which $\underline{v}[j]$ is not zero. I implement this by just finding the index of the first non-zero element in $\underline{v}$.

Beyond this, I wanted to make my Gram-Schmidt routine somewhat automatic in that the first column is $\underline{v}$… that way, I know that it has to be included and can iterate over the other columns normally. To do this, I performed a column swap that you can see in the code.

After that, I just followed the Gram-Schmidt routine as described on pages 40-41. Implementing either of the methods (Gram-Schmidt followed by normalization of each, or Gram-Schmidt with a normalization step for each vector along the way) is acceptable.

The code to perform these operations is below. Note that it takes a lot longer to describe and explain it than to actually do it.

```
function V = marksty_Gram_Schmidt(v)
% function V = marksty_Gram_Schmidt(v)
% Input:
%  v: N-element, unit-magnitude column vector
%     (Note the assumption of unit-magnitude, otherwise v
%      itself could not be included in the orthonormal basis set)
% Output:
%  V: NxN matrix of orthonormal basis vectors, with v as the
%      first vector and all vectors as columns.

% PDL> Find out dimension of the space
dimension = length(v);

% PDL> Make a linearly independent basis set based on identity
%  matrix, with v substituted in for one column.  But make sure
%  vectors are still linearly independent by finding the first
```

```
%  nonzero element of v, and making that row number be the column
%  that is changed in the identity matrix.
%  Then put v at the front of the matrix.

nonZeroIndex = find(v,1);
linIndVecs = eye(dimension);
linIndVecs(:,nonZeroIndex) = v;
linIndVecs(:,[1,nonZeroIndex]) = linIndVecs(:,[nonZeroIndex,1]);

% PDL> Set up intermediate (w) and output(V) matrices.
%  w are not necessarily normalized yet, V are.
V = [];
w = [];
V(:,1) = v;
w(:,1) = v;

% PDL> Perform Gram-Schmidt, with normalization.
for i=2:dimension,
    w(:,i) = linIndVecs(:,i);
    for j=1:i-1,
        w(:,i) = w(:,i) - (V(:,j)' * linIndVecs(:,i)) * V(:,j);
    end
    V(:,i) = w(:,i)/norm(w(:,i));
end
```

Demonstrate its use for the vector v = [1.3; 2.1; -0.9] and show that your generated set of basis vectors is indeed orthogonal.

As Dr. Beers pointed out in his email, you were supposed to just normalize this vector before giving it to your routine. I do that in the sample code below, and then I also demonstrate that the initial column vector is indeed in the final matrix and that the vectors are orthonormal. In this case, this means that dotting each vector with itself should yield 1, and dotting each vector with other vectors should yield 0. This can be done quickly for a matrix of column vectors V by performing V'*V, which should return the identity matrix. Due to roundoff error, it is extremely difficult to ever get these operations to equal exactly 0 or 1, so we allow for some tolerance.

Grading of your homework assignment was facilitated by a script using operations strikingly similar to those in this script.

```
% Mark Styczynski
% 10.34
% HW2, P1A4

clear all; close all;

% Let's demonstrate that our Gram_Schmidt routine works.
% Note that my routine expects that the input vector is of
%  unit magnitude, so as to truly have "v" be in the vector.
% So we take the vector given in the PDF:
vec = [1.3; 2.1; -0.9];
```

```
% Normalize it.
vec = vec/norm(vec);

% And see what our function gives us.
orthonormalBasis = marksty_Gram_Schmidt(vec)

% Is this right?  Let's check it.
% First make sure that the normalized vector is somewhere in
%  the matrix:

orthoSize = size(orthonormalBasis);
numRows = orthoSize(1);
numCols = orthoSize(2);
foundFlag = 0;
for i=1:numCols,
    if mean(orthonormalBasis(:,i) == vec) == 1
        foundFlag = 1;
    end
end
foundFlag

% Now let's make sure that the matrix is orthonormal... if it is
%  then multiplying its transpose by itself should give something
%  very close to the identity matrix.
orthonormalBasis' * orthonormalBasis
% Looks OK.
% We can make sure we're really really close, too.  If we are,
%  this will be all ones.
(orthonormalBasis'*orthonormalBasis) - eye(numCols) < 1e-10
% Looks good!
```

Grading:
Some of you slipped through on grading when you really shouldn't have. In the future, when Dr. Beers tells you to be able to take a vector in R^d and doesn't limit the value of d, you shouldn't use an assumption that d = 3.

(-1 point)
Input vector not being in final basis set
(up to -1 point)
Lack of orthonormality of matrix

3. (2 points) Solve problem 1.B.2 to use linear regression to estimate the Michaelis-Menten rate constants of the enzymatic reaction from the data of Table 1.2. The tables are found in the back of the text. Write your routine as username_P1B2.m. *Make a plot comparing the data of Table 1.2 with the predictions of your fitted model.*

We follow the exact derivation in the book on pages 82-84. You may want to particularly note the parts of this derivation on pages 83-84 for future reference, since if you know the model you want to fit and have all of the appropriate data, this is exactly

the way you would do it to minimize your error for a non-linear model function. In order to solve equation 1.274, we assign X'X to be A, X'y to be q, and perform A\q. This gives us values for b_0 and b_1 that can be manipulated to obtain the physically meaningful constants' values.

This is the function to actually perform the least squares calculations:

```
function [k2 Km] = marksty_Mich_Ment(data)
% function [k2 Km] = marksty_Mich_Ment(data)
% Calculates the rate constants for data fit to a Michaelis-Menten
%  kinetic model.  Note: we assume that all units for time, volume,
%  and concentration are the same
% Input:
%  data: Data structure containing:
%   data.S: N-element column vector of substrate concentrations
%   data.r: N-element column vector of reaction rates
%   data.E0: N-element column vector of initial enzyme concentrations
% Output:
%  k2: Forward reaction rate constant
%  KM: Affinity, or concentration for half-saturation

% PDL> Form the x and y matrices
y = data.E0 ./ data.r;
x = 1./data.S;
X = [ones(size(x)) x];

% PDL> Perform the least squares fit
b = (X' * X) \ (X' * y);

% PDL> Extract the parameter values
k2 = 1/b(1);
Km = b(2)*k2;
```

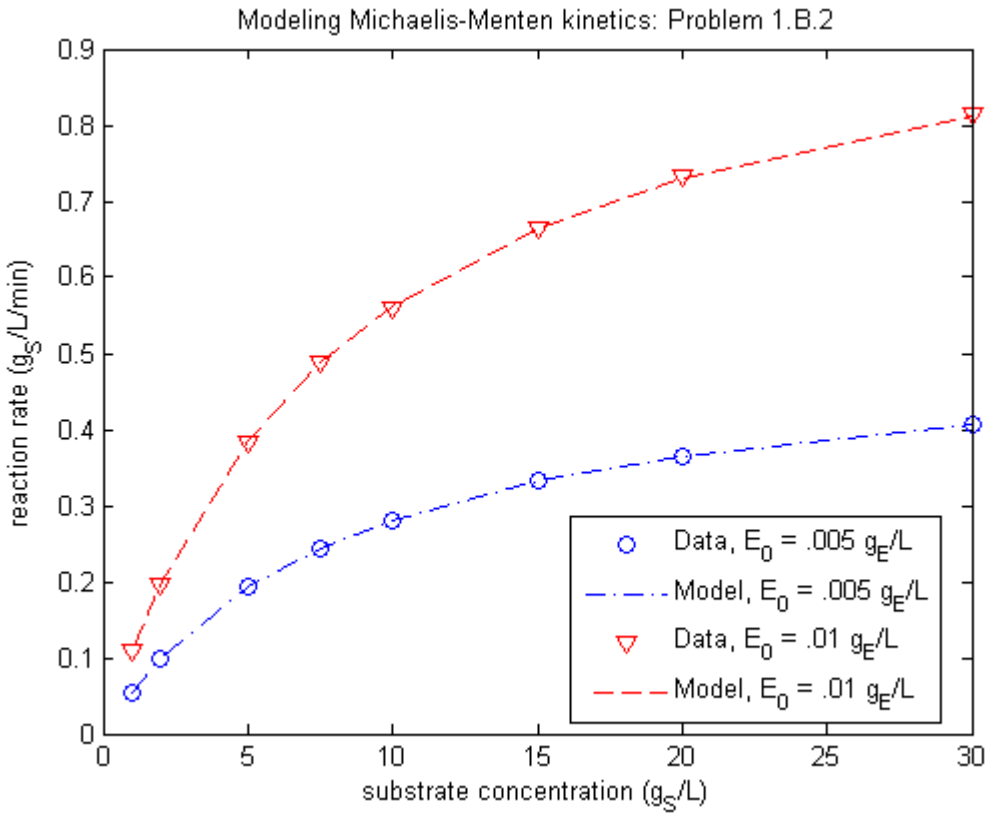
This is the master function to call that function and perform all of the plotting and formatting.

```
% Mark Styczynski
% 10.34
% HW2, P1B2

clear all; %close all;

% PDL> Set input data
eachS = [1 2 5 7.5 10 15 20 30]';
rLowE0 = [.055 .099 .193 .244 .280 .333 .365 .407]';
lowE0 = .005*ones(size(rLowE0));
rHighE0 = [.108 .196 .383 .488 .560 .665 .733 .815]';
highE0 = .01*ones(size(rHighE0));

% PDL> Make data structures to carry input data
data.S = [eachS; eachS];
```

```
data.r = [rLowE0; rHighE0];
data.E0 = [lowE0; highE0];

% PDL> Fit data to Michaelis-Menten kinetics
[k2 Km] = marksty_Mich_Ment(data)

% PDL> Get model predictions, and plot them against data
rCalcLowE0 = k2*lowE0.*eachS./(Km + eachS);
rCalcHighE0 = k2*highE0.*eachS./(Km + eachS);
figure(2)
plot(eachS,rLowE0,'bo')
hold on
plot(eachS,rCalcLowE0,'b-.')
plot(eachS, rHighE0,'rV')
plot(eachS,rCalcHighE0,'r--')
title('Modeling Michaelis-Menten kinetics: Problem 1.B.2')
xlabel('substrate concentration (g_S/L)')
ylabel('reaction rate (g_S/L/min)')
legend('Data, E_0 = .005 g_E/L', 'Model, E_0 = .005 g_E/L', ...
    'Data, E_0 = .01 g_E/L', 'Model, E_0 = .01 g_E/L', ...
    'Location', 'Best')
hold off
```

And here are our results and our wonderful plot.

```
>> marksty_P1B2
k2 =
  104.5041


Km =
   8.5924
```



Grading:
(.5 points each)
Proper plots
Proper values
(1 point)
Solving correct system (not straight-line fit)

4. (3 points) Solve problem 1.B.3 using finite differences to obtain a set of sparse linear algebraic equations. Write your routine as username_P1B3.m. *While I ask for you to put the problem in dimensionless form, this is for now not strictly necessary. I am sure that by the end of 10.50, putting problems in dimensionless form will seem like second nature.*

We will do this problem in dimensionless form… the solution without dimensionless form is quite similar. Not surprisingly, the numbers given to you for the problem and the way you should define your dimensionless number correspond quite well.

We start with our basic equation,

$$D_A \frac{dC_A}{dx} - kC_A = 0 \text{ where } C_A(B/2) = C_A(-B/2) = H_A p_A$$

Note that you could have solved the equation using a symmetry boundary condition such that there was no change in concentration at $x = 0$, but there really wasn't a need to do this since the equations were given to you explicitly in the problem.

We first want to scale all of our variables by characteristic values. We know the concentration will vary from some number possibly as low as zero in the middle (for instance, if diffusion is extremely slow) to a set value of $H_A p_A$ on the solid-gas interface. Thus, we choose our characteristic concentration to be $H_A p_A$ and define

$$\chi = \frac{C_A}{H_A p_A}$$

We next need a characteristic length… since the problem should be symmetric, the geometry of the problem is best described by the half-width of the slab, $B/2$. So, we can then define

$$\xi = \frac{x}{B/2}$$

Note that my choice of Greek letters here is purely arbitrary, and that xi is only chosen because it looks like a scribble. From these two equations, we know that

$$C_A = H_A p_A \chi \text{ and } x = \xi \frac{B}{2}$$

Inserting these two equations into the initial problem we get

$$\frac{D_A H_A p_A}{(B/2)^2} \frac{d^2\chi}{d\xi^2} - k H_A p_A \chi = 0$$

Which can be simplified by canceling out the $H_A p_A$ and moving all of the remaining constants to be in front of the linear term:

$$\frac{d^2\chi}{d\xi^2} - \frac{B^2 k}{4 D_A} \chi = 0$$

The constant coefficient in front of the linear term is a form of the Damkohler number, which you may remember from undergraduate reactions/kinetics courses or the like. This dimensionless number indicates the ratio of reaction to diffusion. Though this would normally be written as Da, that would be too annoyingly close to D_A for my handwriting, so I defined it as phi, leaving us with

$$\frac{d^2\chi}{d\xi^2} - \phi\chi = 0$$

What a nice dimensionless equation. Let's not forget about our boundary conditions, though… noting that the boundary conditions are now for dimensionless locations -1 and 1, and by dividing both sides of the boundary condition by the concentration normalization factor, we get:

$$\frac{C_A}{H_A p_A}(\xi = \pm 1) = \frac{H_A p_A}{H_A p_A}, \text{ or } \chi(\xi = \pm 1) = 1$$

OK, so that's our dimensionless equation. Now we want to make it a finite difference approximation. We assume a uniform grid spacing of N intermediate points, with points on the ends as well. Start with the first order differential approximation,

$$\frac{d\chi}{d\xi} \approx \frac{\chi_{i+1/2} - \chi_{i-1/2}}{\xi_{i+1/2} - \xi_{i-1/2}}$$

Now we need to take the derivative of that, so

$$\frac{d\left(\frac{\chi_{i+1/2} - \chi_{i-1/2}}{\xi_{i+1/2} - \xi_{i-1/2}}\right)}{d\xi} \approx \frac{d\left(\frac{\chi_{i+1/2}}{\xi_{i+1/2} - \xi_{i-1/2}}\right)}{d\xi} - \frac{d\left(\frac{\chi_{i-1/2}}{\xi_{i+1/2} - \xi_{i-1/2}}\right)}{d\xi}$$

$$\approx \frac{1}{\xi_{i+1/2} - \xi_{i-1/2}}\left(\frac{d\chi_{i+1/2}}{d\xi} - \frac{d\chi_{i+1/2}}{d\xi}\right)$$

$$\approx \frac{1}{\xi_{i+1/2} - \xi_{i-1/2}}\left(\frac{\chi_{i+1} - \chi_i}{\xi_{i+1} - \xi_i} - \frac{\chi_i - \chi_{i-1}}{\xi_i - \xi_{i-1}}\right)$$

$$\approx \frac{1}{\Delta\xi}\left(\frac{\chi_{i+1} - \chi_i}{\Delta\xi} - \frac{\chi_i - \chi_{i-1}}{\Delta\xi}\right)$$

Where we take the last step by noting that the distance between two half grid points is equal to the distance between two gridpoints and is equal to the uniform grid spacing. From this, we see that

$$\frac{d^2\chi}{d\xi^2} \approx \frac{1}{(\Delta\xi)^2}(\chi_{i+1} - 2\chi_i + \chi_{i-1}), \text{ where } \Delta\xi = \frac{2}{N+1}$$

Because the total dimensionless distance is 2, from -1 to 1. Thus, the original dimensionless equation for each interior (non-boundary) point can be written as

$$\frac{1}{(\Delta\xi)^2}(\chi_{i+1} - 2\chi_i + \chi_{i-1}) - \phi\chi_i = 0$$

By enforcing the boundary conditions of unity dimensionless concentration, this gives altered equations for $i = 1$ and $i = N$, respectively giving

$$\frac{1}{(\Delta\xi)^2}(\chi_{i+1} - 2\chi_i) - \phi\chi_i = -\frac{1}{(\Delta\xi)^2} \text{ and } \frac{1}{(\Delta\xi)^2}(-2\chi_i + \chi_{i-1}) - \phi\chi_i = -\frac{1}{(\Delta\xi)^2}$$

After all of this work, we can now implement this sparse matrix in Matlab. I chose not to use spalloc, but rather just to use a regular matrix; you could easily have used sparse matrix representation with the method described in class or presented on page 80. Some code to perform just the nondimensional calculations is below.

```
function [ksi chi] = nondim_diff_rxn(phi,N)
% function [ksi chi] = nondim_diff_rxn(phi,N)
% Performs nondimensional analysis of concentration profile for
%  an infinite reactive slab with constant gas concentration on
%  either side and a first-order reaction inside as a consumption
%  term.
% Input:
%  Da: dimensionless number reflecting the relative rate of
%   first-order reaction and diffusion, also known as the
%   Damkohler number
%  N: the number of grid points for the FD approximation
% Output:
```

```
%  ksi: N-element column vector of the dimensionless distance
%   from the center of the slab
%  chi: N-element column vector of dimensionless concentrations
%   for each corresponding element in ksi

% PDL> Calculate distance between gridpoints
deltaKsi = 2/(N+1);

% PDL> Make FD matrix.
% Main diagonal and bordering diagonals.
A = eye(N)*((-2)/(deltaKsi^2) - phi);
for i=2:N-1,
    A(i,i+1) = 1/(deltaKsi^2);
    A(i,i-1) = 1/(deltaKsi^2);
end
A(1,2) = 1/(deltaKsi^2);
A(N,N-1) = 1/(deltaKsi^2);

% PDL> Enforce boundary conditions
b = zeros(N,1);
b(1) = -1/(deltaKsi)^2;
b(N) = -1/(deltaKsi)^2;

% PDL> Make a vector for outputting abscissa.
ksi = linspace(-1,1,N+2)';

% PDL> Calculate and output ordinate.
chi = A\b;
chi = [1; chi; 1];
```

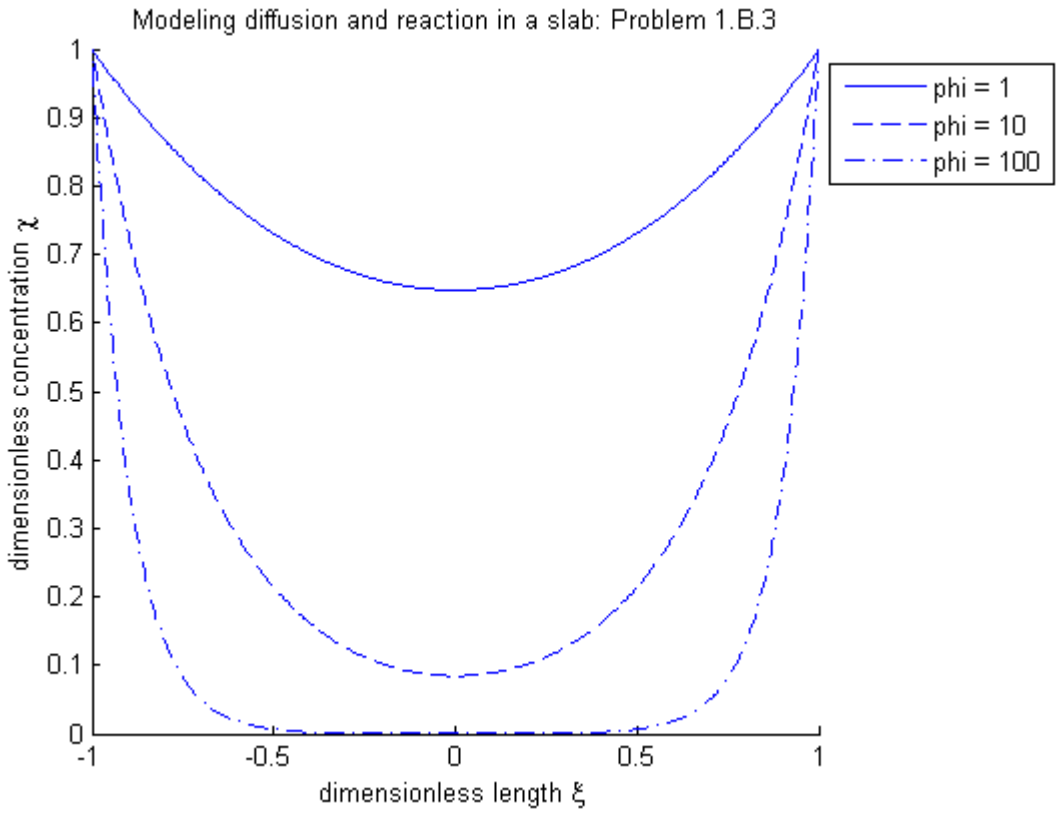
Show your numerical solution for $D_A = 1$, $k = 1$, $B = 2$, $H_A p_A = 1$ and then redo the calculation for $D_A = 0.1$ and 0.01... You should for each calculation increase the number of grid points until you obtain sufficient accuracy (for example, no longer see any significant change in the solution when you vary the number of grid points).

This is very straightforward… you'll find that for these values, solution of my dimensionless equation corresponds exactly to the solution of the regular dimensional equations. Code to call the original function and perform appropriate plotting and formatting is below.

```
% Mark Styczynski
% 10.34
% HW2, P1B3

clear all; %close all;

% PDL> Collect input values.
D = [1 .1 .01];
k = 1;
B = 2;
Hapa = 1;
```

```
% PDL> Calculate dimensionless number.
phi = B^2*k/4./D;

% PDL> Decide on number of grid points.  This was done
%  after everything, by making sure the curves were smooth and
%  that increasing the number of points didn't change anything.
N = 499;

% PDL> Perform calculations for each value of D, store values.
for i=1:length(phi),
    [ksiMat(:,i) chiMat(:,i)] = marksty_nondim_diff_rxn(phi(i),N);
end

% PDL> Plot and make pretty.
figure(2)
hold on
plot(ksiMat(:,1),chiMat(:,1),'-')
plot(ksiMat(:,2),chiMat(:,2),'--')
plot(ksiMat(:,3),chiMat(:,3),'-.')
string1 = ['phi = ' int2str(phi(1))];
string2 = ['phi = ' int2str(phi(2))];
string3 = ['phi = ' int2str(phi(3))];
title('Modeling diffusion and reaction in a slab: Problem 1.B.3')
xlabel('dimensionless length \xi')
ylabel('dimensionless concentration \chi')
legend(string1,string2,string3,'Location','BestOutside')
hold off
```

And here is the plot.



What happens to the concentration profile as the diffusivity is reduced?
You can clearly see that the concentration profile tends to "flatten out" as the diffusivity increases and "hollow out" as the diffusivity decreases. This makes intuitive sense: if the reaction consumes all of species A faster than it can diffuse into the slab, then the very inside portion of the slab will have very little A in it. If diffusion and reaction rates are of the same order, though, then some A will be able to reach the middle of the slab. In the most extreme cases, a negligible reaction (making diffusion dominant) would cause the profile to be completely flat, with dimensionless concentration of 1 across the entire slab. Alternatively, if diffusion were so slow that the reaction consumed all of the species as soon as it entered the slab, there would be essentially a bolus of A at either interface of the slab… this would correspond to a negative step function at dimensionless length = -1 and a step function at dimensionless length = 1.

Grading:
1 pt - quick derivation
1.5 pt - implementation/code/plots
0.5 pt - analysis