Name: Luyando Kwenda
Class: ENM 5020
Due Date: 16 March 2023
Assignment: 2

Application of Newtons Method, Analytical Continuation and Arc Length Continuation to a 2D Boundary Value Problem

Contents

1 Introduction

1.1 Overview

In a linear sense we are able to solve for an equation $Ax = b$ using methods like LU Decomposition as discussed in the previous assignment. However, when we are posed with a non-linear equation, we have to use other methods that focus on iterative schemes. Newtons Method is a **Fixed Point Iteration (FPI)** method that uses derivatives to solve for the solution to a curve. Given a curve, we know that the tangent line is a really good approximation to the curve at that particular point. Even though the root may be far from the actual point of intersection on the x-axis, we are able to repeat the derivative calculation many times until we get as close as possible to the real solution which is what we call convergence.

The things to consider when using this iteration method is the domain of convergence and the order of convergence. This convergence can be explained using the **Contraction Mapping Theorem** which implies that for any closed and bounded set of functions that map a closed and bounded region $R \in \mathbb{R}^n$ into itself and in this region R there is a unique solution to the equation. For the domain of convergence, **the Mean Value Theorem** states that for any continuously differentiable function over an interval, there is an intersection of this interval from the Contraction Mapping theorem that gives the domain of convergence. Its worth noting that we expect newtons method to converge quadratically.

The first step taken in newtons method is finding an initial guess for the solution which most likely end up converging and this might be a problem for certain cases. In a general sense Eq. (1) shows how newtons method works with the $k + 1$ term being the next guess.

$$x^{k+1} = x^k + R(x^k) \tag{1}$$

There many different subcategories of Newtons Method and in our case, we will focus on the Adaptive Newton Method. This method involves computing Simple Newton for a while and periodically updating the derivative matrix called the **Jacobian**.

In this assignment we are to use newtons method to solve the non-linear 2D BVP for different λ values Eq. (2) using Analytic and Arc Length Continuation. We will also discuss the timing analysis of the methods used

$$\nabla^2 u + \lambda u(1 + u) = 0 \tag{2}$$

2 Problem Setup and Formulation

2.1 Finite Difference Analysis

We have been tasked to analyse a 2D Boundary Value Problem (BVP) in the domain $D = (0 \leq x \leq 1) \cup (0 \leq y \leq 1)$ provided by Eq. (2) and the first step is to discretize the problem using a center difference which can be written as

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \lambda u(1 + u) = 0 \tag{3}$$

with $u = 0$ on all boundaries. The equation is not linear in the sense that we have a quadratic term u and we have to vary the value of lambda over a range of $0 \leq \lambda \leq 60$. Our problem will be defined by a 30 by 30 grid which gives a total of 900 by 900 points since this is a two dimensional problem. When discretized, we have

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h_x^2} + \frac{u_{l+n} - 2u_i + u_{l-n}}{h_y^2} + \lambda u_i(1 + u_i) = 0 \tag{4}$$

where $n = 30$, $h_x = \frac{1}{n-1} = h_y = h$ so we can say $h = \frac{1}{29}$. As for the index value l, we can define it by Eq. (5) with i being the ith row and j the jth column

$$l = (j - 1)(n + 1) + i \tag{5}$$

This form allows to write out a linear equation in the form $Ax = b$ which can be solved using linear methods. This however, does not provide a direect solution to the problem we use approximations as the initial inputs to the u_i values. The next step would be to use Newtons method that requires an inital guess. We also know that the actual equation created from discretization is $J\delta = -R$.

2.2 Newtons Method

In section 1.1, we gave a general overview of what happens in a newton iteration. Here we will provide a much more specific work flow of her dimensional cases. We call a function **R(x)** the residual function and when expanded using a taylor series approximation, we can say

$$R(x) = R(x_0) + \left.\frac{\partial R}{\partial x}\right|_{x_0} (x - x_0) + H.O.T = 0 \tag{6}$$

$$R(x^{k+1}) = R(x^k) + \left.\frac{\partial R}{\partial x^k}\right|_{x^k} (x^{k+1} - x^k) + H.O.T = 0 \tag{7}$$

Rearranging equation 9 we can rewrite this equation as

$$\left.\frac{\partial R}{\partial x^k}\right|_{x^k} (x^{k+1} - x^k) = -R(x^k) \tag{8}$$

This is how we obtained the equation stated in the previous section with the residual function given as R(x) and J is the jacobian matrix.

$$J\delta x^k = -R(x^k) \tag{9}$$

After deriving these matrices, our next goal will be to create an initial guess and start with an initial lambda value. The delta values we obtain are then added to the previous ones to get a newer values which are then used in the newer iteration until convergences occurs.

2.3 Initial Guess

In order to run the newton algorithm, we need to guess an initial input for u_i. The simplest way of obtaining this is buy linearising Eq. (2)

$$\nabla^2 u + \lambda u = 0 \tag{10}$$

This equation, from it structure, can be termed as an eigenvalue problem. We need to solve for the non-trivial solution to have relevant results. Eigenvalue problems have a known solution and in this case we have an equation that is a solution where x and y are locations in the grid. Here A is unknown and we guess a value of 0.1 as lambda comes from the left and m and n are positive integers

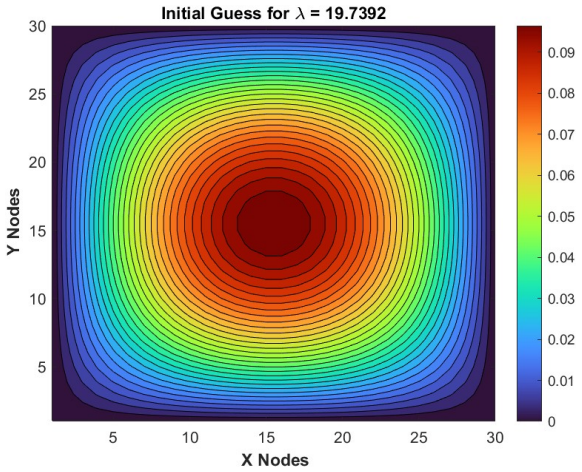
$$u(x, y) = A_{nm} sin(n\pi x) sin(n\pi y) \tag{11}$$

$$\lambda = \pi^2(m^2 + n^2) \tag{12}$$

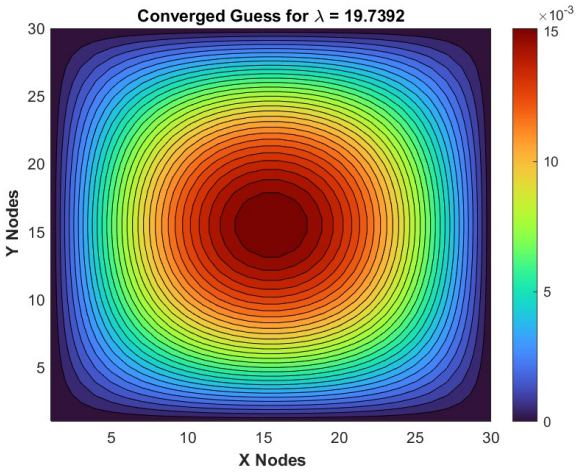
In this problem we have been given a range of lambda values $0 \leq \lambda \leq 60$ so our values of m and n are restricted i.e.

1. For $m = 1$ and $n = 1$ we have $\lambda = 2\pi^2 = 19.739$
2. For $m = 1$ and $n = 2$ we have $\lambda = 5\pi^2 = 49.348$.
3. For $m = 2$ and $n = 1$ we have $\lambda = 5\pi^2 = 49.348$

Even though the last two lambda values are identical, their solution branch is different and will be seen in the final results. After this initial guess is obtained, it is passed to the newton function to obtain values that converge given a tolerance. It will be seen from the figure below that for our first case, there is only one peak (could also be a valley depending on A but in the other two cases, there will be one peak and one valley and the only difference will be their orientation. The contour plots shown below plot initial guess and the converged case and as expected, the results are almost identical. The values of lambda chose that are close to the actual values are 19.5 and 49.0 for the other two cases.

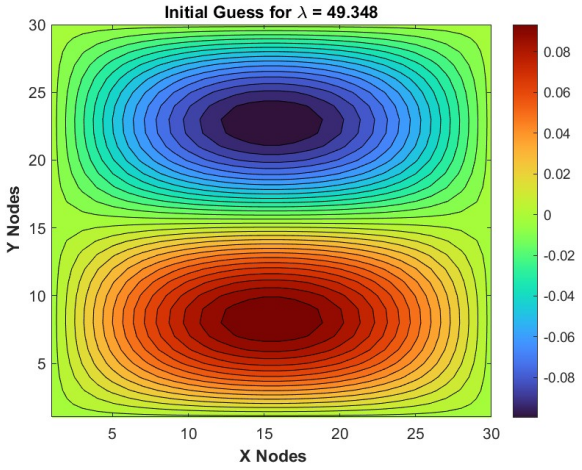


(a) Initial Guess

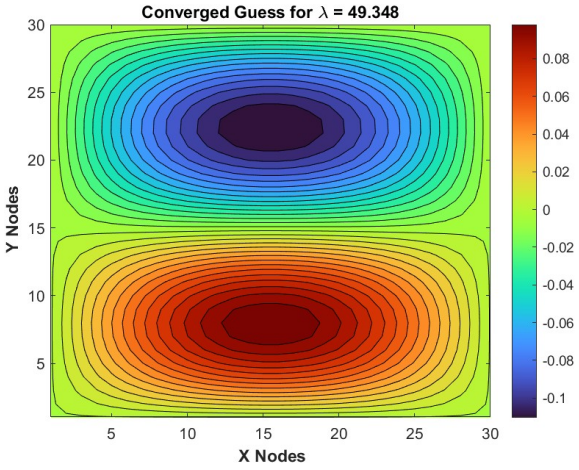


(b) Converged Guess

Figure 1: Contour Plots for $m = 1$ and $n = 1$

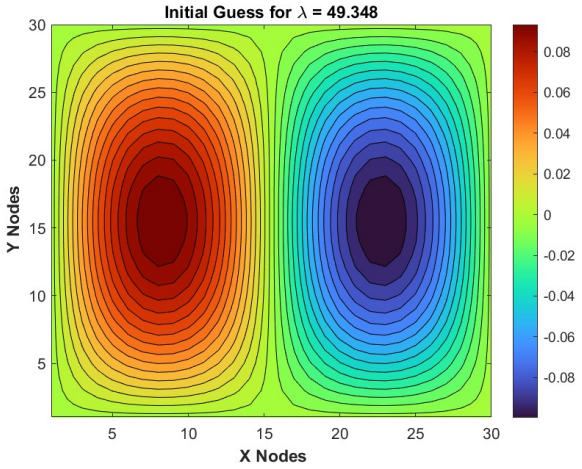


(a) Initial Guess

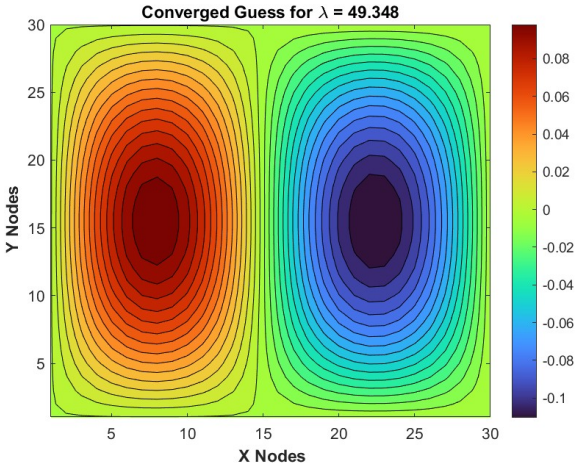


(b) Converged Guess

Figure 2: Contour Plots for $m = 2$ and $n = 1$



(a) Initial Guess



(b) Converged Guess

Figure 3: Contour Plots for $m = 1$ and $n = 2$

2.4 Analytical Continuation

Previously, we were able to solve for a particular value of lambda after getting a solution that converged with Newton's Algorithm. Now, we want to increase lambda which gives us another u vector.

$$u_{new} = u_{old} + \left.\frac{\partial u}{\partial \lambda}\right|_{\lambda_{old}} \partial\lambda \tag{13}$$

In order to get $\frac{\partial u}{\partial \lambda}$, we can use the residual and jacobian matrices to solve linearly as shown below. The R derivative is simply taken with respect to λ

$$J\left.\frac{\partial u}{\partial \lambda}\right|_{\lambda(old)} = -\left.\frac{\partial R}{\partial \lambda}\right|_{\lambda_{old}} \tag{14}$$

Since we choose a value of $\partial\lambda$ and we obtained a result from Eq. (14), are then able to solve Eq. (13) to get newer values.

2.5 Arc Length Continuation

In some cases, Analytic Continuation fails when we encounter bifurcation points where the slope is zero. In order to correct this, we use arc length continuation which is more like solving for a straightened curve or tracing out the curve. The first stage is creating an augmented jacobian matrix that is padded with specific vectors and also creating an augmented residual matrix that is differentiated with respect to a new variable, s. The full process is describe in the assignment document so only the main equations will be stated in this report.

$$\hat{J} = \begin{pmatrix} \frac{\partial R}{\partial u} & \frac{\partial R}{\partial \lambda} \\ \frac{\partial \nu}{\partial u} & \frac{\partial \nu}{\partial \lambda} \end{pmatrix} \tag{15}$$

$$\hat{R} = \begin{pmatrix} \frac{\partial R}{\partial s} \\ \frac{\partial \nu}{\partial s} \end{pmatrix} \tag{16}$$

Eq. (15) and Eq. (16) can then be solved linearly as shown

$$\hat{J}\begin{pmatrix} \frac{\partial u}{\partial s} \\ \frac{\partial \lambda}{\partial s} \end{pmatrix} = \hat{R} \tag{17}$$

Its worth noting that the step size of s ,ds, is a chose variable and in this case we chose 0.1 and is constant in all our iterations. The results are then run through a new newton loop that takes into account this augmentation. In order to calculate for convergence, we used the l2-nor which was discussed in the previous assignment. After every loop, we multiply the result by ds and add the knew delta values to the previous ones to provide a new step. Further information can be see in the code section of the appendix

3 Results

After a series of trial and error to see which initial lambda guesses would produce a continuous curve, $\lambda = 19.1$, $\lambda = 49.6$ and $\lambda = 49.6$ were the values chosen. The table below summarizes the lambda values.

m =1 n=1	m = 1 n=2	m=2 n=1
$\lambda_{guess} = 19.1$	$\lambda_{guess} = 49.6$	$\lambda_{guess} = 49.6$
$\lambda_{actual} = 19.7392$	$\lambda_{actual} = 49.348$	$\lambda_{actual} = 49.348$

In the first branch, since the guessed value is less that the actual known value of lambda, we are able to perform arc length continuation to the right of the curve produced which will give positive norm values. When we move to the left, the norm values are expected to be negative as seen from the norm plot below. In the other two cases, we chose an initial lambda guess that was on the right so we would expect the norm values to be negative as we move towards the actual guess.

At first glance, it may seem like the our last two branches would produce the same result but we are able to differentiate them using contour plots. In the l2-norm plot below, their norm values lie over each other. Despite changing the A value in the initial guess, this will always be the result.

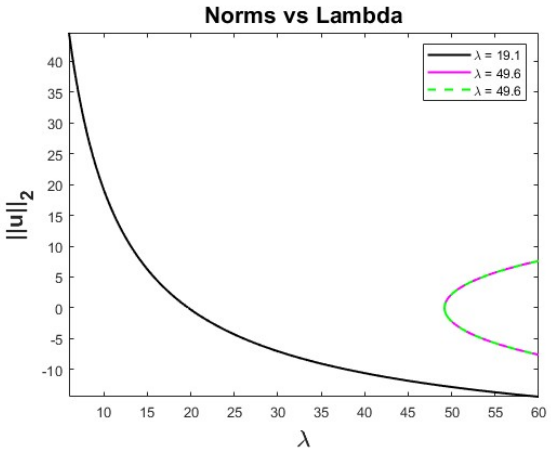
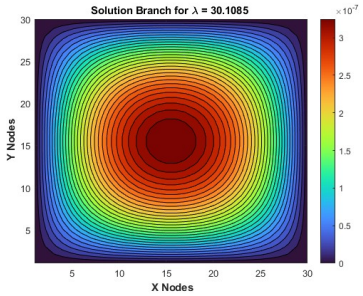


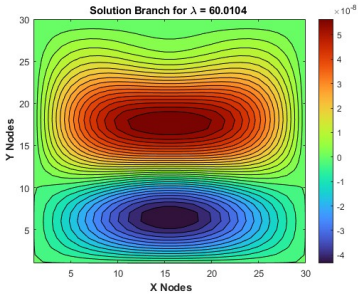
Figure 4: l2 Norm versus Lambda

In the figure below, we show contour plots for three different branches at chosen lambda values i.e. $\lambda = 30.1085$ and $\lambda = 60.0104$. In to demonstrate why we obtained the same norm for the last two branches, we use the same lambda value but the only difference we will observe is the orientation of the valleys and peaks. As mention before, figure 5a shows a hill because value of lambda is greater that $2\pi^2$. For the last two branches, when compared to figures 2b and 3b, the plots show that they have greater peaks than valleys. If more plots are shown for an increasing lambda value closer to the known eigenvalue, we will notice the trend from the hills and valley being symmetric in the graph to what is seen below.

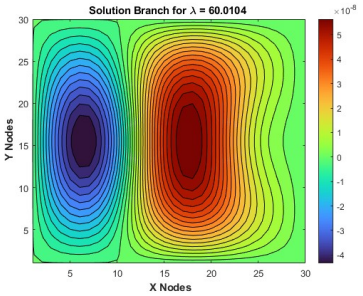
In figures 5a and 6a we note that the difference is that as we choose a lambda value less than 2π, we obtain a valley whereas if greater than, we have a mountain



(a) Branch 1 $m = 1$ **and** $n = 1$

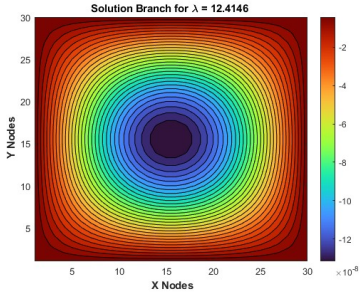


(b) Branch 2 $m = 2$ **and** $n = 1$

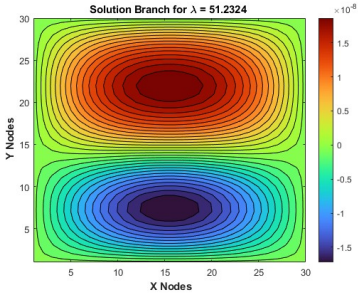


(c) Branch 2 $m = 1$ **and** $n = 2$

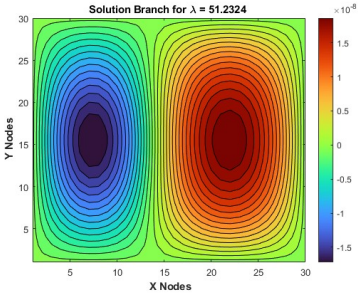
Figure 5: Contour Plots for the 3 different branches



(a) Branch 1 $m = 1$ **and** $n = 1$



(b) Branch 2 $m = 2$ **and** $n = 1$



(c) Branch 2 $m = 1$ **and** $n = 2$

Figure 6: Contour Plots for the 3 different branches

4 Conclusion

In summary, we had been given a non-linear 2D Boundary Value Problem that we could not solve using linear methods. We implemented an iterative scheme using Newtons method to solve for convergence. The first step we took was finding an initial guess for the solution given a lambda range. When the equation is linearized, we are able to obtain a solution that has eigenvalues,lambda. From this, we are only able to obtain 3 lambda values that fall within the range with two of them being equal.

The initial guesses are passed through newton's loop for convergence and then analytic continuation which gives the next point. But because we are likely to get a point where the derivative is zero, the linear equations solved using the residual and the jacobian are unsolvable. Hence we rely on arc length continuation which helps us when we encounter these bifurcation points. From this point onward, we used an augmented newtons loop to solve for different lambda values to obtain new solutions.

From the 3 branches we obtained, we noted that in the last two branches where $m = 2$, $n = 1$, $m = 1$, $n = 2$ we observed that increasing the lambda values from the known eigenvalue produced a greater peak intensity. The only notable difference is the orientation rotation of 90 degrees. The first branch with equal m and n values produced either a peak of a value depending on what lambda value is chosen.

5 Appendix

Below is the code used in the analysis of the problem

5.1 Main Function

```
% Thisis the main script that runs all the code and generates the norm vs
% lambda plots

close all;clear;clc

n = 30;

%call function that solves for all the initial guesses. Here we have thress
%cases so we have to loop through all the cases (3)
u_mat = initial_guess(n);

lambda1 = 19.1;
u1 = u_mat(:,1);
%for the forward direction
ds = 0.1;
dLambda = 0.1;
[u_normsf1, myLambsf1] = getNorms(n,u1,lambda1,ds,dLambda);
ds = -0.1;
dLambda = -0.1;
%for the backward direction
[u_normsb1, myLambsb1] = getNorms(n,u1,lambda1,ds,dLambda);

%Change the vectors by concatination and plot
myLambs1 = [fliplr(myLambsb1) myLambsf1 ];
u_norms1 = [fliplr(u_normsb1) u_normsf1 ];
minInd = find(u_norms1==min(u_norms1));
next = minInd+1;
u_norms1(next:end) = -u_norms1(next:end);
plot(myLambs1,u_norms1,'k-','LineWidth',1.5)
hold on


lambda2 = 49.6;
u2 = u_mat(:,2);
%for the forward direction
ds = 0.1;
dLambda = -0.1;
[u_normsf2, myLambsf2] = getNorms(n,u2,lambda2,ds,dLambda);
%for the backward direction
ds = -0.1;
dLambda = -0.1;
[u_normsb2, myLambsb2] = getNorms(n,u2,lambda2,ds,dLambda);

myLambs2 = [ fliplr(myLambsb2) myLambsf2];
u_norms2 = [fliplr(u_normsb2) u_normsf2 ];
minInd = find(u_norms2==min(u_norms2)); %twice equal values
next = minInd(1)+1;
u_norms2(next:end) = -u_norms2(next:end);
plot(myLambs2,u_norms2,'m-','LineWidth',1.5)
```

```

hold on



lambda3 = 49.6;
u3 = u_mat(:,3);
%for the forward direction
ds = 0.1;
dLambda = -0.1;
[u_normsf3, myLambsf3] = getNorms(n,u3,lambda3,ds,dLambda);
%for the backward direction
ds = -0.1;
dLambda = -0.1;
[u_normsb3, myLambsb3] = getNorms(n,u3,lambda3,ds,dLambda);

myLambs3 = [fliplr(myLambsb3) myLambsf3];
u_norms3 = [fliplr(u_normsb3) u_normsf3];

minInd = find(u_norms3==min(u_norms3)); %twice equal values
next = minInd(1)+1;
u_norms3(next:end) = -u_norms3(next:end);
plot(myLambs3,u_norms3,'g--','LineWidth',1.5)

xlabel('')
legend ('\lambda = 19.1','\lambda = 49.6','\lambda = 49.6')

title('Norms vs Lambda','fontweight','bold','fontsize',16);
xlabel('\lambda','fontweight','bold','fontsize',16);
ylabel('||u||_{2}','fontweight','bold','fontsize',16)
exportgraphics(gcf,'normplot.jpg')
```

5.2 Initial Guess Script

```
%This script generates the inital guesses for the lambda values
%given our range of values, we have three possible solutions to get a
%lambda value less than 60 exculding zero

function u_mat = initial_guess(N)

%we have u = Asin(n*pi*x)sin(m*pi*y)
m_vec = [1 1 2];
n_vec = [1 2 1];

%create vector to store possible lamda values
lambda_vec = zeros(1,length(m_vec));

%preallocate initial values vector
A = 0.1;
u0 = zeros(N^2,1);
u_mat =zeros(N^2,3);

for ii = 1:length(m_vec)
    m = m_vec(ii);
```

```
    n = n_vec(ii);
    lambda = pi^2 *(m^2 + n^2); %calculate current value of lambda

    %store lambda values in vector
    lambda_vec(ii) = lambda;

    for jj = 1:N
        for kk = 1:N
            l = (kk-1)*N + jj;

            x = (jj-1)/(N-1);
            y = (kk-1)/(N-1);

            u0(l) = A * sin(n*pi*x) * sin(m*pi*y);

        end
    end

 u_mat(:,ii) = u0;
 u0 = [];

end



end
```

5.3 Initial Guess Plotting Script

```
%This script plots the initial and converged u values
close all
ind =3;
n = 30;
[u_mat,lambda_vec] = initial_guess(n);
guess = u_mat(:,ind);
guess = (reshape(guess,[n,n]));
lambda = lambda_vec(ind);

figure
contourf(guess,n)
colorbar
colormap("turbo")
xlabel('\bf X Nodes')
ylabel('\bf Y Nodes')
title(['Initial Guess for \lambda = ', num2str(lambda)])
exportgraphics(gcf,"guess m1n2 .jpg")

% Use Newton function to calculate the new J and R values
lambda1 = 49.6;
[J,R,u00,counter] = NewtonsMeth(n,u_mat(:,ind),lambda1);
u1 = (reshape(u00,[n,n]));

figure
contourf(u1,n)
```

```
colorbar
colormap("turbo")
xlabel('\bf X Nodes')
ylabel('\bf Y Nodes')
title(['Converged Guess for \lambda = ', num2str(lambda)])
% name = num2str(n);
% name = convertCharsToStrings(name);
 exportgraphics(gcf,"Converged m1n2 .jpg")
```

5.4 Jacobian and Residual Function

```
%This function creates the Jacobian and R matrices
%INPUT: n is the grid dimension
%       u is the initial guess and lambda
function [J,R] = BVP_2DSol(n,u,lambda)

%Equal in the x and y spaces
nx = n-1;
ny = nx;
hx = 1/nx;
hy = 1/ny;

%Preallocate J and R matrix (Ax = b)
J = zeros(n^2,n^2);
R = zeros(n^2,1);

%Populate the matrix
for ii = 1:n  %loop through ith row
    for jj = 1:n  %loop through the columns of ith row

        %define the indexing
        l = (jj-1)*n + ii;

        if (ii == 1 || jj == 1 || ii == n || jj == n)
            J(l,l) = 1;
            R(l) = u(l);
        else
            %filling the matrix by first choosing a column and going up
            %through the rows
            %in the x direction
            J(l,l) = - 4/(hx^2) + lambda*(1+2*u(l));

            J(l,l+1) = 1/(hx^2);
            J(l,l-1) = 1/(hx^2);

            %in the y direction
            J(l,l-n) = 1/(hy^2);
            J(l,l+n) = 1/(hy^2);

            R(l)= ((u(l+1) - 2*u(l) + u(l-1))/(hx^2)) + ((u(l+n) - 2*u(l) + u(l
                ↪ -n))/(hy^2)) + (lambda * u(l) *(1 + u(l)));
        end

    end
```

```
end

end
```

5.5 Analytic Continuation Function

```
function u = analytic_continuation(u0,d_lambda,n,lambda)

[J,~] = BVP_2DSol(n,u0,lambda);

%First we have to find dR_dL
dR_dL = zeros(n^2,1);

%Populate the matrix
for ii = 1:n  %loop through ith row
    for jj = 1:n  %loop through the columns of ith row

        %define the indexing
        l = (jj-1)*n + ii;

        if (ii == 1 || jj == 1 || ii == n || jj == n)
            dR_dL(l) = 0;
        else
           dR_dL(l) = (u0(l))^2 + u0(l);
        end

    end
end

%next we solve the linear equation
du_dL = J\-dR_dL;

%GET NEW U VALUE
u = u0 + du_dL * d_lambda;

end
```

5.6 Arc Length Continuation Function

```
% This function performs arc length continuation

function [u2,lambda2] = arc_length_continuation(u1,u0,lambda1,lambda0,n,ds)

[J,~] = BVP_2DSol(n,u1,lambda1);
% Preallocate a n^2+1 by n^2+1 matrix
J_aug = zeros(1+n^2,1+n^2);

%Fill the matrix accordingly
J_aug(1:n^2,1:n^2) = J;

%First we have to find dR_dL
dR_dL = zeros(n^2,1);
```

```

%Populate the matrix
for ii = 1:n  %loop through ith row
    for jj = 1:n  %loop through the columns of ith row

        %define the indexing
        l = (jj-1)*n + ii;

        if (ii == 1 || jj == 1 || ii == n || jj == n)
            dR_dL(l) = 0;
        else
           dR_dL(l) = (u1(l))^2 + u1(l);
        end

    end
end

J_aug(1:end-1,end) = dR_dL;

%calculate dn_dL
dn_dL = -2 * (lambda1 - lambda0);
J_aug(end,end) = dn_dL;

%calculate dn_du
dn_du = zeros(1,n^2);
for ii = 1:length(dn_du)
    dn_du(ii) = -2 * (u1(ii) - u0(ii));
end

J_aug(end,1:end-1) = dn_du;

%calculate dn_ds
dn_ds = 2 * ds;

dR_ds = zeros(n^2,1);
dRhat_ds = [dR_ds;dn_ds];

%we can now solve Ax = b
vals = J_aug\-dRhat_ds;
du_ds = vals(1:end-1);
dL_ds = vals(end);

du = du_ds *ds;
dL = dL_ds *ds;
disp(dL)
%calculate new lambda and new u values
u2 = u1 + du;
lambda2 = lambda1 + dL;

end
```

5.7 Newton's Method Function

```
% This function performs newtons method
```

```
function u = NewtonsMeth(n,u,lambda)

[J,R] = BVP_2DSol(n,u,lambda);

%solve the linear equation J*delta = -R
delta = J\-R;

d1 = delta.^2;
d2 = sum(d1(:));
check = sqrt(d2);

%assign a tolerance value
conv = 10^-5;

while check > conv
    %add the difference to the new initial guess
    u = u + delta;
    [J,R] = BVP_2DSol(n,u,lambda);

    %solve the linear equation J*delta = -R
    delta = J\-R;
    d11 = delta.^2;
    d22 = sum(d11(:));
    check = sqrt(d22);
end

end
```

5.8 Newton's Augmented Method Function

```
% This function performs newtons method
function [u,lambda] = NewtonsMethAug(u2Guess,u1,lambda2,lambda1,n,ds)

%creates the regular jacobian matrix and residial vector
[J,R] = BVP_2DSol(n,u2Guess,lambda2);

% ext we will create the augmented J matrix, J_aug
% Preallocate a n^2+1 by n^2+1 matrix
J_aug = zeros(1+n^2,1+n^2);

%next we create the augmented forms of J and R
J_aug(1:n^2,1:n^2) = J;

%create dR_dL
%First we have to find dR_dL
dR_dL = zeros(n^2,1);

%Populate the matrix
for ii = 1:n  %loop through ith row
    for jj = 1:n  %loop through the columns of ith row

        %define the indexing
        l = (jj-1)*n + ii;

```

```
        if (ii == 1 || jj == 1 || ii == n || jj == n)
            dR_dL(l) = 0;
        else
            dR_dL(l) = (u2Guess(l))^2 + u2Guess(l);
        end

    end
end

J_aug(1:end-1,end) = dR_dL;

%calculate dn_du
dn_du = zeros(1,n^2);
for ii = 1:length(dn_du)
    dn_du(ii) = -2 * (u2Guess(ii) - u1(ii));
end
J_aug(end,1:n^2) = dn_du;

%calculate dn_dL
dn_dL = -2 * (lambda2 - lambda1);
J_aug(end,end) = dn_dL;

% create the augmented R vector
diff_u = sqrt(sum((u2Guess - u1).^2));
diff_lambda = lambda2 - lambda1;
nu = ds^2 - diff_u^2 - diff_lambda^2;
R_hat = [R;nu];

%solve the linear equation J*delta = -R
delta = J_aug\-R_hat;
du = delta(1:end-1);
dlambda = delta(end);

d1 = delta.^2;
d2 = sum(d1(:));
check = sqrt(d2);

%assign a tolerance value
conv = 10^-4;

u = u2Guess;
lambda = lambda2;

while check > conv
    %add the difference to the new initial guess
    u = u + du;
    lambda = lambda + dlambda;

    [J,R] = BVP_2DSol(n,u,lambda);

    %next we create the augmented forms of J and R
    J_aug(1:n^2,1:n^2) = J;

    %create dR_dL
    %create dR_dL
    %First we have to find dR_dL
```

```
    dR_dL = zeros(n^2,1);

    %Populate the matrix
    for ii = 1:n  %loop through ith row
        for jj = 1:n  %loop through the columns of ith row

            %define the indexing
            l = (jj-1)*n + ii;

            if (ii == 1 || jj == 1 || ii == n || jj == n)
                dR_dL(l) = 0;
            else
                dR_dL(l) = (u(l))^2 + u(l);
            end

        end
    end

    J_aug(1:end-1,end) = dR_dL;

    %calculate dn_du
    dn_du = zeros(1,n^2);
    for ii = 1:length(dn_du)
        dn_du(ii) = -2 * (u(ii) - u1(ii));
    end
    J_aug(end,1:end-1) = dn_du;

    %calculate dn_dL
    dn_dL = -2 * (lambda - lambda1);
    J_aug(end,end) = dn_dL;

    % create the augmented R vector
    diff_u = sqrt(sum((u - u1).^2));
    diff_lambda = lambda - lambda1;
    nu = ds^2 - diff_u^2 - diff_lambda^2;
    R_hat = [R;nu];


    %solve the linear equation J*delta = -R
    delta = J_aug\-R_hat;

    du = delta(1:end-1);
    dlambda = delta(end);
    %lambda1 = lambda;


    d1 = delta.^2;
    d2 = sum(d1(:));
    check = sqrt(d2);
    %disp(check)

   % [J,~] = BVP_2DSol(n,u,lambda);

end

end
```

5.9 L2-Norm and Lambda Function

```
%This function solves and loops over a range of given lambda values

function [u_norms, myLambs] = getNorms(n,u0Guess,lambda0,ds,d_lambda)

% STEP1: Call newtons method to converge solution at u0 for lambda =
% lambda0
u0 = NewtonsMeth(n,u0Guess,lambda0);

%calculate the l2 norm of the result
d1 = u0.^2;
d2 = sum(d1(:));
n0 = sqrt(d2);

% STEP2: Use Analytic Continuation to converge at u1 with lambda1 =
% lambda0 + dLambda
u1Guess = analytic_continuation(u0,d_lambda,n,lambda0);

% STEP3: Call Newtons Method Again to converge at u1
lambda1 = lambda0 + d_lambda;
u1 = NewtonsMeth(n,u1Guess,lambda1);

%calculate the l2 norm of the result
d1 = u1.^2;
d2 = sum(d1(:));
n1 = sqrt(d2);

% STEP3: Use Arc Length Continuation to get new solution
[u2Guess,lambda2] = arc_length_continuation(u1,u0,lambda1,lambda0,n,ds);

% STEP4: Call Augmented Newton Function to converge
%run augmented newtons function
[u2,lambda2] = NewtonsMethAug(u2Guess,u1,lambda2,lambda1,n,ds);

%calculate the l2 norm
d1 = u2.^2;
d2 = sum(d1(:));
n2 = sqrt(d2);

%store everything in a vector for ploting
u_norms = [n0 n1 n2];
myLambs = [lambda0 lambda1 lambda2];

% loop over lambda range using arclength continuation and newtons augmented
% method stating with lamba2

while lambda2 < 60 && lambda2 > 6

[u3Guess,lambda3] = arc_length_continuation(u2,u1,lambda2,lambda1,n,ds);

[u3,lambda3] = NewtonsMethAug(u3Guess,u2,lambda3,lambda2,n,ds);

%calculate the l2 norm
d1 = u3.^2;
d2 = sum(d1(:));
```

```
n3 = sqrt(d2);

u_norms(end+1) = n3;
myLambs(end+1) = lambda3;
%disp(lambda3)
lambda1 = lambda2;
lambda2 = lambda3;

u1 = u2;
u2 = u3;

end
```