Numerical Analysis Project

Part 1

Question 1

A process for implementing the Newton-Raphson method for N non-linear equations in N unknowns (x_1, x_2… x_n):

Our existing knowledge of Newton-Raphson method allows us to solve one equation in one unknown. This is accomplished by iterating

$$g(x) = x - \Phi(x)*f(x)$$

where

$$\Phi(x) = 1/f'(x)$$

This method is based conditionally on $f'(x) \neq 0$. This method requires an initial guess for x, and is very sensitive to this guess.

To attempt to solve N equations with N unknowns simultaneously we will be using a method analogous to this. In our method we will re-define the function g(x) as

$$G(X) = X - J^{-1}(X)*F(X)$$

where X is a column vector of the n unknowns, and J(X) is the Jacobian matrix of the system.

The method involves evaluating G(X) at each iteration and checking to see if G(X) –X is within a given tolerance. A major drawback to this is the need to evaluate J(X) at each step. The method will require an initial 'guess' for X in order to work.

The method works under the assumption that a sufficiently accurate starting value for X is known, and that the inverse of the Jacobian exists. For the latter assumption to be upheld J(X) must be non-singular at the fixed G(X).

The general method follows below:

$$G(x) = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{N-1} \\ x_n \end{bmatrix} - \frac{1}{\left\| \begin{bmatrix} \partial f_1 / \partial f_{x_1} & \partial f_1 / \partial f_{x_2} & \dots & \partial f_1 / \partial f_{x_N} \\ \partial f_2 / \partial f_{x_1} & \partial f_2 / \partial f_{x_2} & \dots & \partial f_1 / \partial f_{x_N} \\ \dots & \dots & \dots & \dots \\ \partial f_N / \partial f_{x_1} & \partial f_N / \partial f_{x_2} & \dots & \partial f_N / \partial f_{x_N} \end{bmatrix} \right\|^{adj}} \begin{bmatrix} f_1(x_1, x_{2,} \dots, x_N) \\ f_2(x_1, x_{2,} \dots, x_N) \\ \dots \\ f_N(x_1, x_{2,} \dots, x_N) \end{bmatrix}$$

Question 2

The algorithm is as follows:

1. Input equations, initial guesses (= to number of equations), tolerance, max iterations.
2. Create loop from 1 to Nmax.
    a. Calculate all partial derivatives, using initial guesses.
    b. Input derivatives into Jacobian matrix.
    c. Make sure inverse Jacobian is non singular.
    d. Evaluate $G(X) = X – J^{-1}(X)*F(X)$, where X is initial guesses.
    e. If $G(X) – X$ is less than Tol output X.
    f. Else $X = G(X)$, go back to a.
3. End

Code is:

```
% Project#1 of part1
% Group Alpha

% since we cannot index our function we use an example for N = 3
% Assumptions:
% Inverse of jacobian has to exist
% sensitive to initial guess
% 3-point approximation for partial derivative is valid


%temporary


x(1) = -10;
x(2) = 500;
x(3) = -20000;
x(4) = 10;
x(5) = 28000;
x(6) = -72000;
x(7) = 72000;
x(8) = 1000;

w(1,:) = x;
X = x';
Y = x';
Z = x';
A = x';
B = x';
h =.01;
Nmax = 50;
N=8;
tol = 0.01;
for k = 2:Nmax

  %computing the jacobian

  for j = 1:N

    X(j,:) = X(j,:) - 2*h;
    Y(j,:) = Y(j,:) - h;
    Z(j,:) = Z(j,:) + h;
    A(j,:) = A(j,:) + 2*h;



    J(j,:) = (1/(12*h))*(feval('projectq3a',X) - 8*feval('projectq3a',Y) + 8*feval('projectq3a',Z) - feval('projectq3a',A));

    X(j,:) = B(j,:);
```

```
        Y(j,:) = B(j,:);
        Z(j,:) = B(j,:);
        A(j,:) = B(j,:);
    end


    if det(J) == 0
        fprintf('determinant is zero, will not work')
        break
    end

    w(k,:) = w(k-1,:) + (inv(J)*(-feval('projectq3a',(w(k-1,:))))')'   ;


    if norm((w(k,:) - w(k-1,:)),inf) < tol
        fprintf('YES!!!!, the root is');

        break
    end
end

check = feval('projectq3a',(w(k,:)))
w(k,:)
```

Where projectq3a is:

```
function input = projectq3a(x);
input(1) = 3*x1-cos(x2*x3)-0.5;
input(2) = x1^2-81*(x2+0.1)^2+sin(x3)+1.06;
input(3) = exp(-x1*x2)+20*x3+(10*pi-3)/3;
```

Question 3

Part a

The Jacobian in question 8 sections 10.2 will simply be the coefficients in each equation. We know that the Jacobian will not be singular as long as no equation is the linear combination of another.

$$G(P) = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{bmatrix} - \frac{1}{\begin{vmatrix} a_{11} & a_{12} & a_{31} & a_{41} & a_{51} & a_{61} & a_{71} & a_{81} \\ a_{21} & a_{22} & a_{32} & a_{42} & a_{52} & a_{62} & a_{72} & a_{82} \\ a_{31} & a_{23} & a_{33} & a_{43} & a_{53} & a_{63} & a_{73} & a_{83} \\ a_{41} & a_{24} & a_{34} & a_{44} & a_{54} & a_{64} & a_{74} & a_{84} \\ a_{51} & a_{25} & a_{35} & a_{45} & a_{55} & a_{65} & a_{75} & a_{85} \\ a_{61} & a_{26} & a_{36} & a_{46} & a_{56} & a_{66} & a_{76} & a_{86} \\ a_{71} & a_{27} & a_{37} & a_{47} & a_{57} & a_{67} & a_{77} & a_{87} \\ a_{18} & a_{28} & a_{38} & a_{48} & a_{58} & a_{68} & a_{78} & a_{88} \end{vmatrix}^{adj}} \begin{bmatrix} f_1(p_1, p_{2,} \ldots, p_8) \\ f_2(p_1, p_{2,} \ldots, p_8) \\ \ldots \\ f_N(p_1, p_{2,} \ldots, p_8) \end{bmatrix}$$ Where P is the initial guess

We created a vectorized code that will work for any number of equations. As our results show, even when using a fairly large tolerance, it takes very few iterations to get extremely precise results. We used a 5-point difference computation to calculate our Jacobian, which yields very accurate derivative results which will help our overall precision. Unfortunately there is no free lunch; the five point difference is a slower method than less precise algorithms such as midpoint or 3-point formula.

Results

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} -72 \\ 2520 \\ 27720 \\ 138600 \\ -360360 \\ 504504 \\ -360360 \\ 102960 \end{bmatrix} \quad \text{where,} \quad \begin{bmatrix} f_1(X) \\ f_2(X) \\ f_3(X) \\ f_4(X) \\ f_5(X) \\ f_6(X) \\ f_7(X) \\ f_8(X) \end{bmatrix} = 10^{-10} \bullet \begin{bmatrix} .4547 \\ -.2910 \\ -.5275 \\ -.6730 \\ .1273 \\ -.3092 \\ .3365 \\ .0364 \end{bmatrix}$$

With a tolerance of .01 it takes 61 iterations.

Part b

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8.771289 \\ .2596975 \\ -1.372304 \end{bmatrix} \quad \text{where} \quad \begin{bmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \end{bmatrix} = 10^{4} \bullet \begin{bmatrix} .045746 \\ .208522 \\ .580556 \end{bmatrix}$$

With a tolerance of .001 it takes 7 iterations to converge to the roots.

Part 2

Question 1

Part a

We created a function 'oscillator', which was a vectorized version of the input equations for part a, here it is:

```
function input = oscillator(t,y);
input(1) = y(1)-t^2+1;
input(2) = 0;
```

The code for the forward Euler method in part a, is:

```
*please note alpha is a vector which takes in alpha and beta
function y_prime = forwardeuler(a,b,N,alpha)
clc
format long

t(1) = a;
w(1,:) = alpha;
tol = 10^-6;

%Calculate w

h = (b-a)/N;
for i = 2:N+1
   w(i,:) = w(i-1,:) + h*feval('oscillator',t(i-1),w(i-1,:));
   t(i) = a + (i-1)*h;
end
```

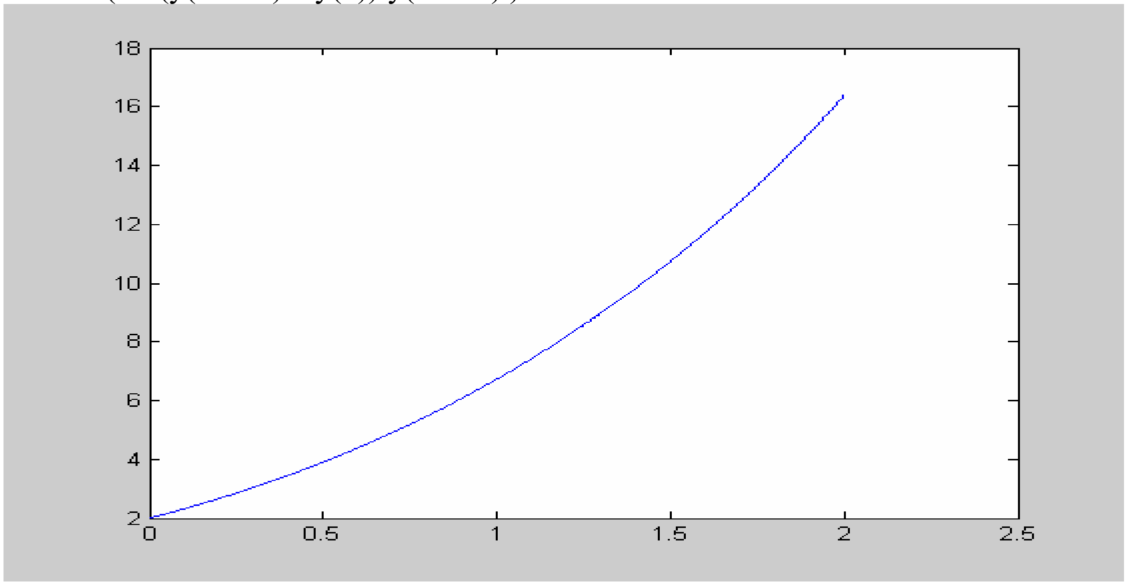
```
if abs(w(i,:)-16.38905610) < tol
   plot(t,w(1,:))
   N

end

fprintf('you are done');
```

Forward Euler gets only two decimal places correct with 100 000 iterations. We tried to run it for 500 000, but our computer crashed.

We chose our order two method to be midpoint. The code for the midpoint method in part a, is:

```
function out = Midpoint(a,b,N,alpha)
clc
format long
value = 16.38905610;
tol = 10^-6;
% N = 500;


True =1;
false=0;
OK = false;


w(1,:) = alpha;
t(1) = a;

N = 4032;
   h = (b-a)/N;
   if OK==True
      break
   end
   for i = 1:N

      w(i+1,:) = w(i,:)+h*feval('oscillator',t(i)+h/2,w(i,:)+h/2*feval('oscillator',t(i),w(i,:)));
      t(i+1) = t(i) + h;
      if abs(w(i+1,:)-value) < tol
         fprintf('the number of iterations needed :%d',i)
         w(i+1)
         OK = True;
         break
      end
   end

fprintf('\ndone');
plot(t,w(:,1))
```

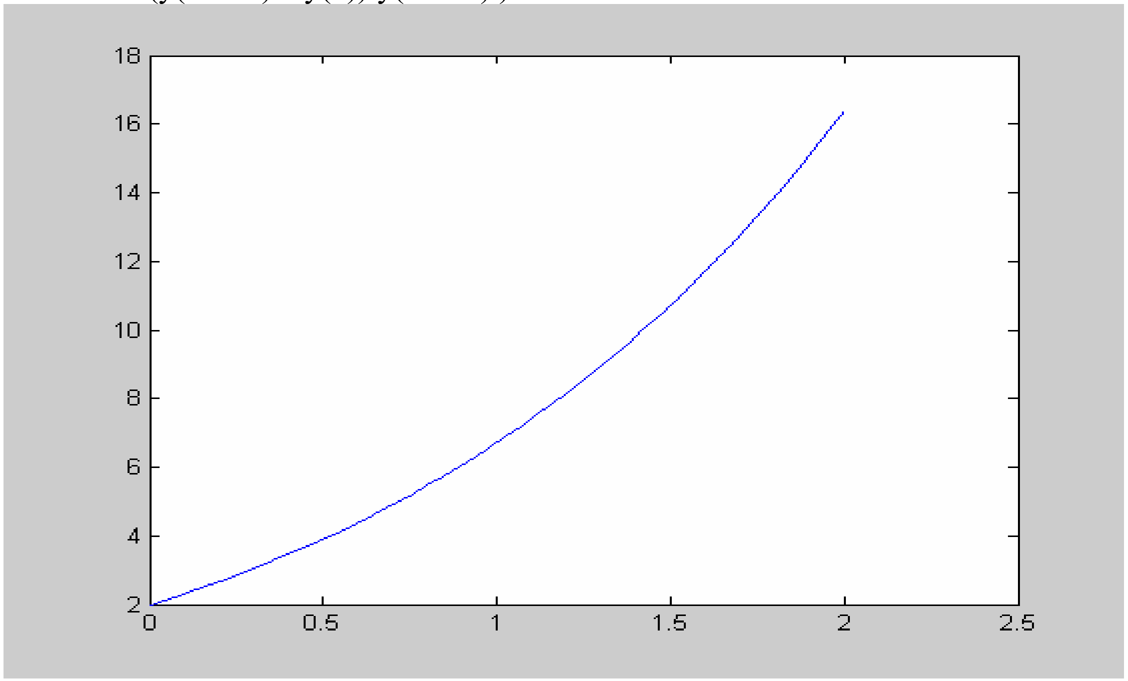
Midpoint takes 4032 iterations (N=4032) and returns y(2) = 16.38905510023065
Error = (abs(y(actual) – y(2))/y(actual))*100 = 6.1E-6 %



The famous Runge-Kutta code for part a is:

```
function out = runge-kutta(a,b,N,alpha)

% Inputing h,a,b,alpha
clc
format long
value = 16.38905610;
tol = 10^-6;
% N = 500;



w(1,:) = alpha;
t(1) = a;

 N = 45;
   h = (b-a)/N;

for i = 1:N
   k1 = h*feval('oscillator',t(i),w(i,:));
   k2 = h*feval('oscillator',t(i)+h/2,w(i,:)+.5*k1);
   k3 = h*feval('oscillator',t(i)+h/2,w(i,:)+k2/2);
   k4 = h*feval('oscillator',t(i)+h,w(i,:)+k3);
   w(i+1,:) = w(i,:) + 1/6*(k1+2*k2+2*k3+k4);
   t(i+1) = t(i) + h;
   if abs(w(i+1,:)-value) < tol
      fprintf('the number of iterations needed :%d',i)
            w(i+1)

   end

end
fprintf('\ndone');
% fprintf('the value of y(b) is %2.4f\n',w(N+1,:));
plot(t,w(:,1))
```

Runge-Kutta takes 45 iterations and returns y(2) = 16.38905512797664
Error = abs(y(actual) – y(2))/y(actual))*100 = 6.0E-6 %

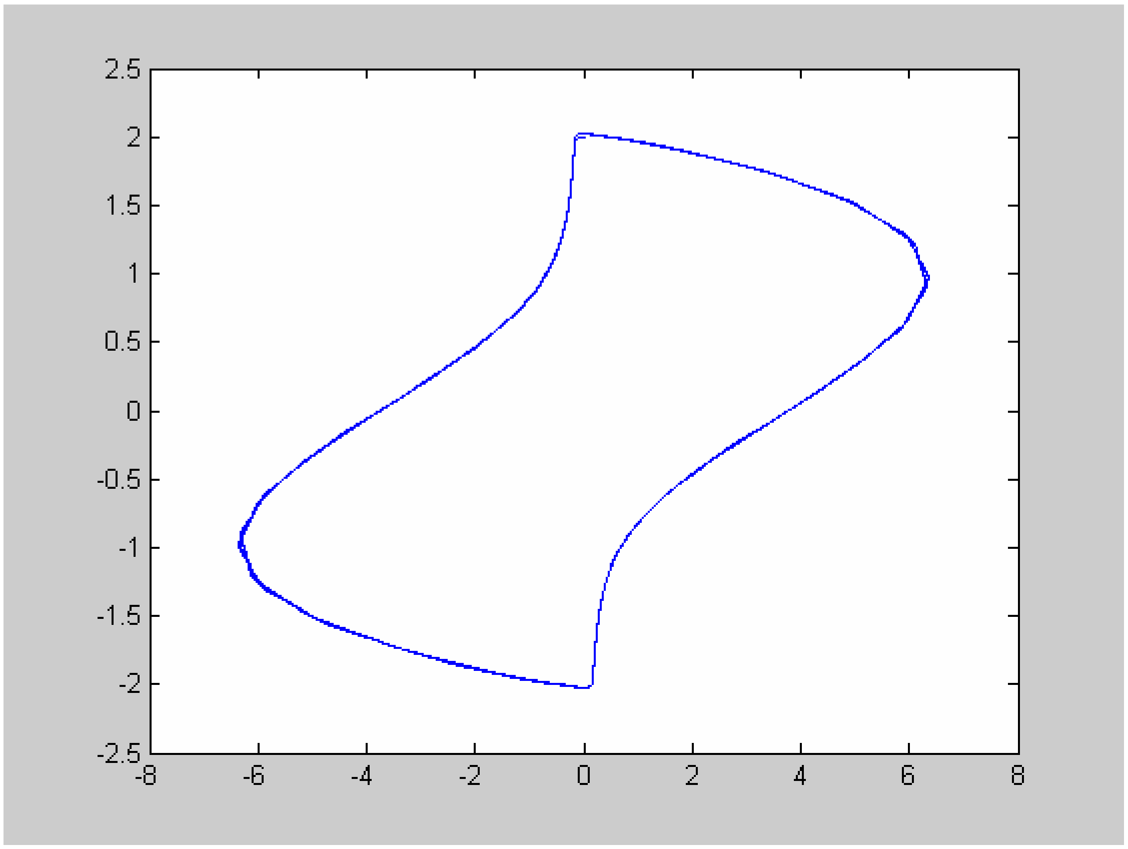


As we can see the RK4 algorithm is far superior to any other. Midpoint did a satisfactory job of solving the ODE. The forward Euler is a very poor method of solving ODEs, taking 100,000 iterations to converge to an answer that was correct to only 2 decimal places.

Part b) i)
For RK4 we used the same general function as in part a, but the oscillator became:

```
function input = oscillator(t,y);
input(1) = y(2);
input(2) = 4*(1-y(1)^2)*y(2)-y(1);
RK4 with N = 1000
```

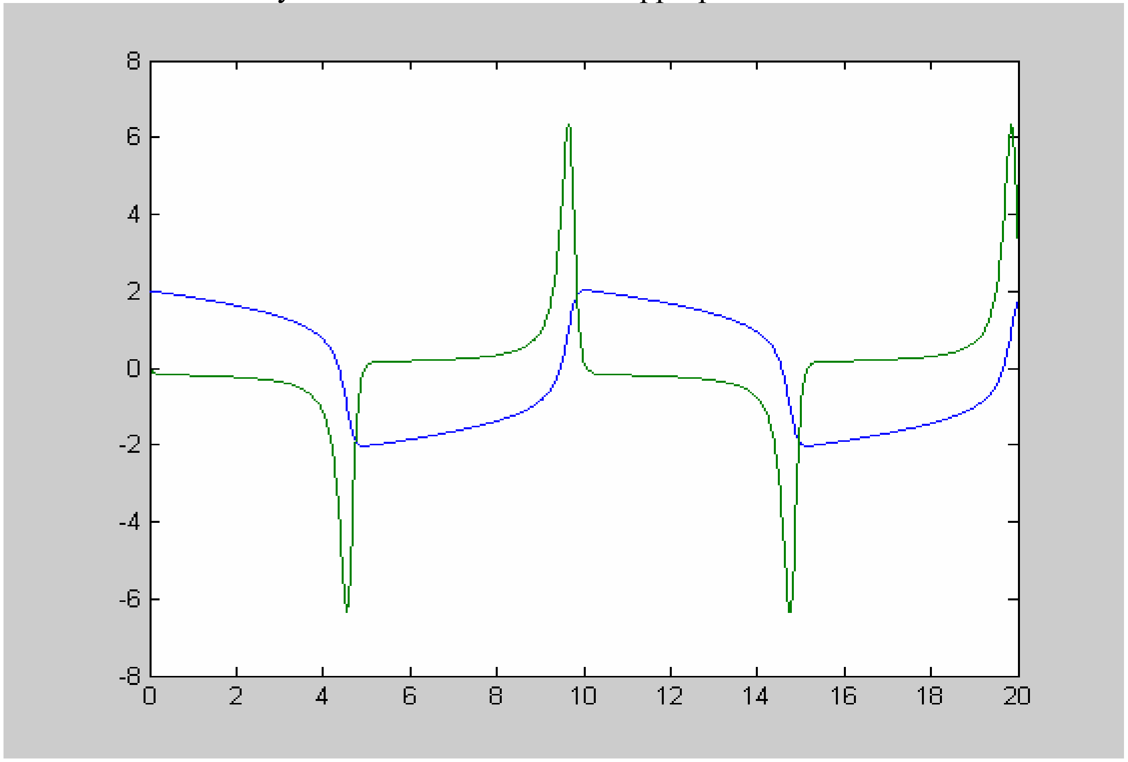
The value of q(50) is 1.4192375258
The value of p(50) is -0.6514740545
The graph looks like



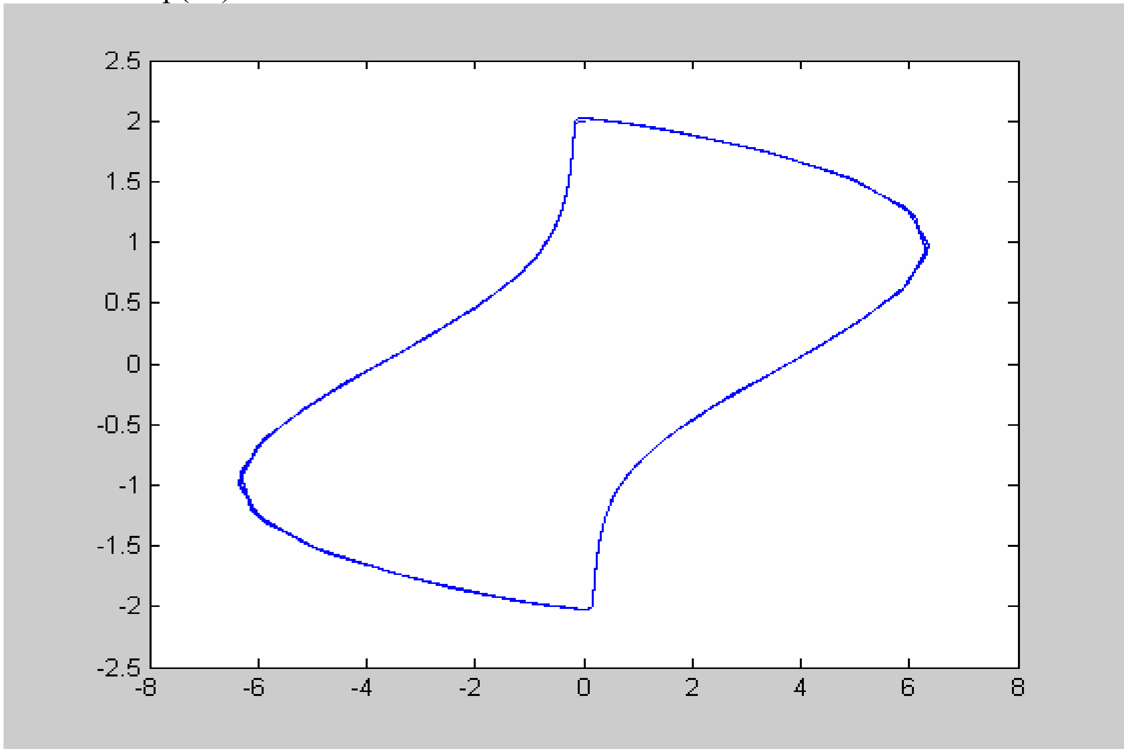
N=10000
The value of q(50) is 1.4301974948
The value of p(50) is -0.6475138083
(Graph is identical)

N=20000
the value of y(b) is 1.4301987830
the value of y(b) is -0.6475133440
(graph is identical)

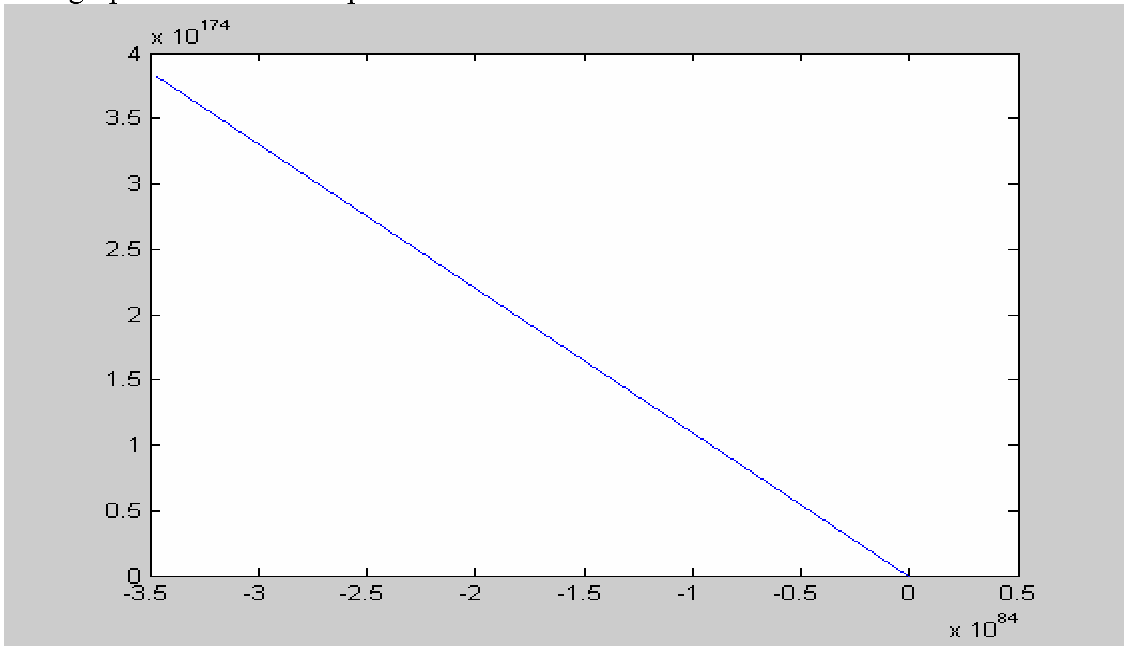
As we can see when we plot q vs t and p vs t get periodic functions that **oscillate** back and forth. I would say Van Der Pol oscillator is appropriate name for this function.

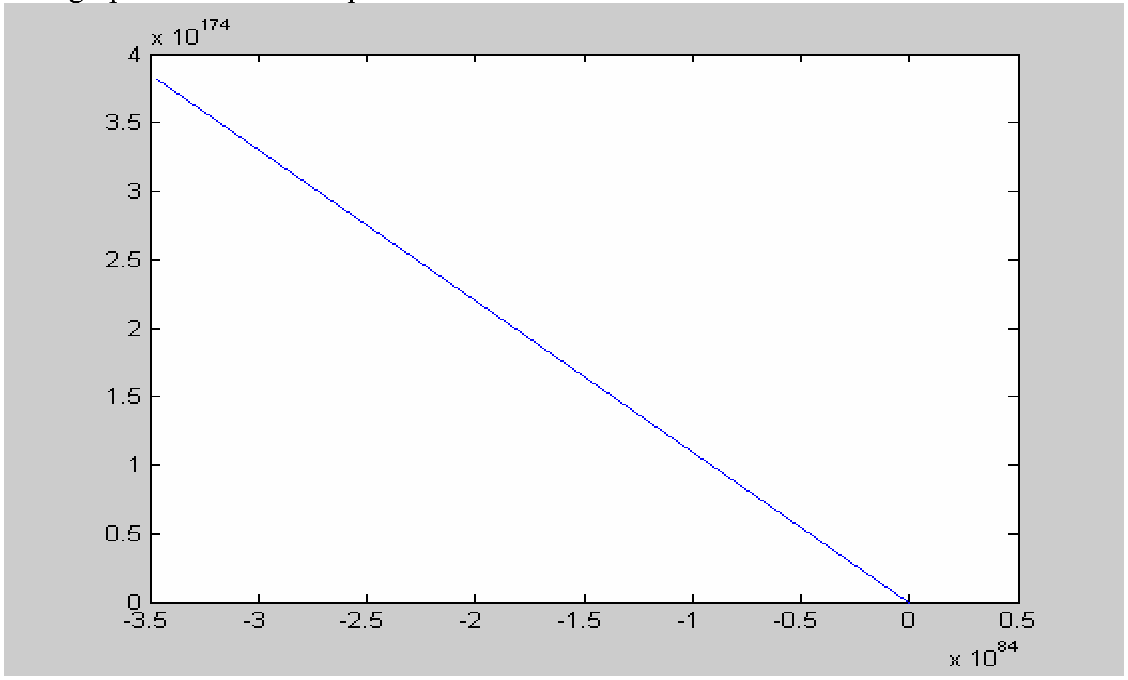


With Forward Euler N=1000, the value of q(50) is -0.1480621931
the value of p(50) is 2.1428676639



With Forward Euler N=10000, the value of q(50) is 0.3991609514 the value of p(50) is -1.2720972733

With Forward Euler N=20000, the value of q(50) is 0.6143908173
the value of p(50) is -1.0516826113
Part a) ii)
With Forward Euler μ= 500 it does not converge for any reasonable values of N.
This graph is the failed output from the forward Euler.


With RK4 μ= 500 it does not converge for any reasonable values of N.
This graph is the failed output from the RK4.


Question 1c

The backward Euler method is an implicit algorithm meaning it requires a non-linear method to be solved. It has two major advantages, it works very well for a large range of h and it can be used to solve stiff functions. We will implement our Newton Method from part 1, (assuming that the determinant of the Jacobian is non-zero) along with our backward Euler to solve this problem. The algorithm is as follows:

1. Input a, b, N (number of iterations) and alpha (initial values).
2. Start loop for t with i from 1 to N.
3. Calculate t and w (value of each point)
    a. $w_{i+1} = w + hf(t_{i+1}, w_{i+1})$
    b. this requires a modified Newton, which means the jacobian has to exist.
        i. $w_{i+1} = w_i + inv(J)*F(w_i)$
4. output w

Question 1d

Backward Euler calls, Newton, which calls test (the function Newton solves), which calls oscillator (which is the functions we want to solve)

Backward Euler

```
 function y_prime = backwardeuler(a,b,N,alpha)


clc
h = (b-a)/N;
t(1) = a;
w(1,:) = alpha;

%Calculate w
for i = 1:N
t(i+1) = a + (i)*h;
%    calculating w(i+1) using newton
    w(i+1,:) = feval('modnewton',2,h,t(i+1),w(i,:),[1,1]);

%    w(i+1,:) = w(i,:) + h*feval('oscillator',t(i+1),w(i+1,:));


end
plot(w(:,1),w(:,2))
fprintf('\ndone\n');
fprintf('the value of y(b) is %2.10f\n',w(N+1,:));
```

Modified Newton

```
function out = newton(N,h,t,w,x)

% Group Alpha

% Assumptions:
% Inverse of jacobian has to exist
% sensitive to initial guess
% 3-point approximation for partial derivative is valid


%temporary
clc


y(1,:) = x;
```

```
X = x';
Y = x';
Z = x';
A = x';
B = x';
Nmax = 50;
tol = 0.0001;
for k = 2:Nmax

   %computing the jacobian

   for j = 1:N

      X(j,:) = X(j,:) - 2*h;
      Y(j,:) = Y(j,:) - h;
      Z(j,:) = Z(j,:) + h;
      A(j,:) = A(j,:) + 2*h;



      J(j,:) = (1/(12*h))*(feval('test',X,w,t,h) - 8*feval('test',Y,w,t,h) + 8*feval('test',Z,w,t,h) - feval('test',A,w,t,h));

      X(j,:) = B(j,:);
      Y(j,:) = B(j,:);
      Z(j,:) = B(j,:);
      A(j,:) = B(j,:);
   end


   if det(J) == 0
      fprintf('determinant is zero, will not work');
      break
   end

   y(k,:) = y(k-1,:) + (inv(J)*(-feval('test',y(k-1,:)',w,t,h))')'   ;


   if norm((y(k,:) - y(k-1,:)),inf) < tol
      fprintf('YES!!!!, the root is');
      y(k,:)
      break
   end
end

out(1,:) = y(k,:)
```

Test

```
function input = test(x,w,t,h)
input = w + h*feval('oscillator',t,x)-x';
```

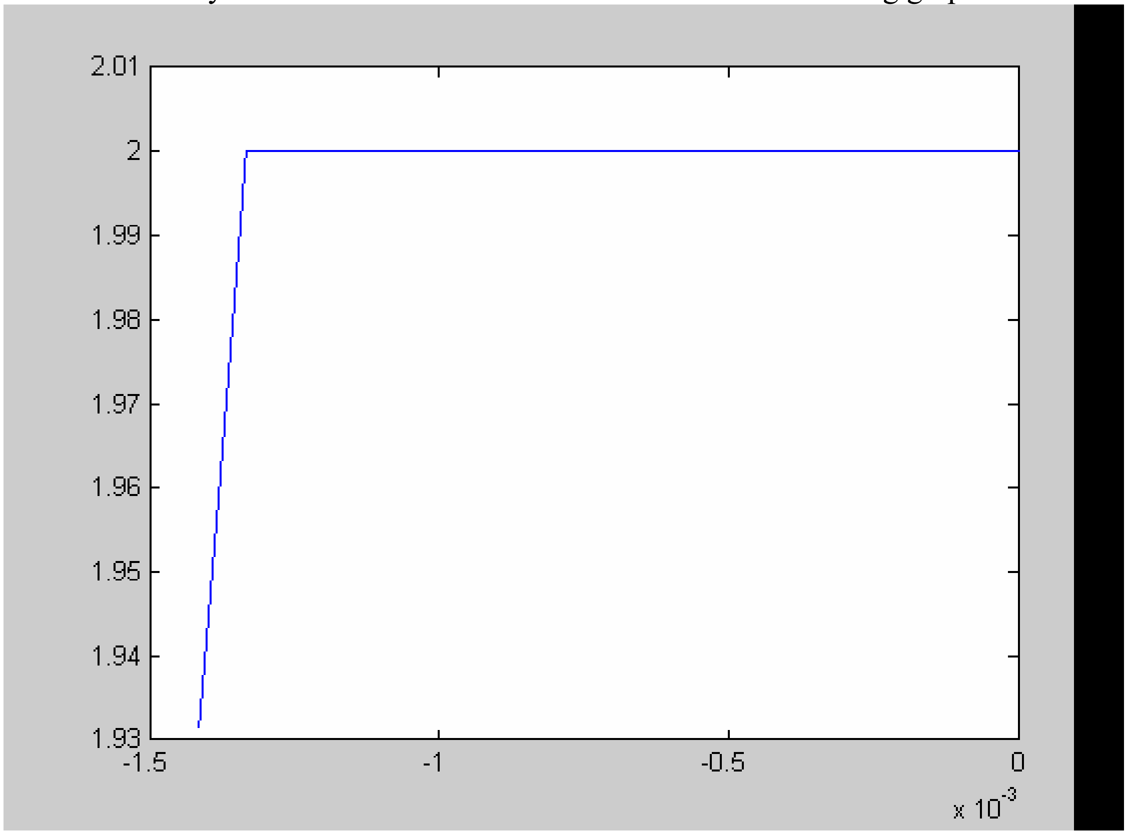
Oscillator

```
function input = oscillator(t,y);
input(1) = y(2);
input(2) = 500*(1-y(1)^2)*y(2)-y(1);
```

Question 2

Part a) i)

We ran our code on the Van Der Pol oscillator. We were able to get the exact same graph when μ = 4. We expected backward Euler to work well on the stiff function (μ = 500), however we only obtained useless results. We obtained the following graph:



Part ii)

$$y' = \lambda y$$

$$k_1 = h\lambda w_i$$

$$k_2 = h\lambda(w_i + \frac{h\lambda w_i}{2})$$

$$k_3 = h\lambda(w_i + \frac{h\lambda}{2}\left[w_i + \frac{h\lambda w_i}{2}\right]$$

$$k_4 = h\lambda\left[w_i + h\lambda\left[w_i + \frac{h\lambda}{2}\left[w_i + \frac{h\lambda}{2}(w_i + \frac{h\lambda}{2}w_i)\right]\right]\right]$$

$$w_{i+1} = w_i\left[1 + \frac{1}{6}\left[h\lambda + 2h\lambda(1 + \frac{h\lambda}{2}) + 2h\lambda\left[1 + \frac{h\lambda}{2}(1 + \frac{h\lambda}{2})\right] + h\lambda\left[1 + h\lambda\left[1 + \frac{h\lambda}{2}\left[1 + \frac{h\lambda}{2}(1 + \frac{h\lambda}{2})\right]\right]\right]\right]\right]$$

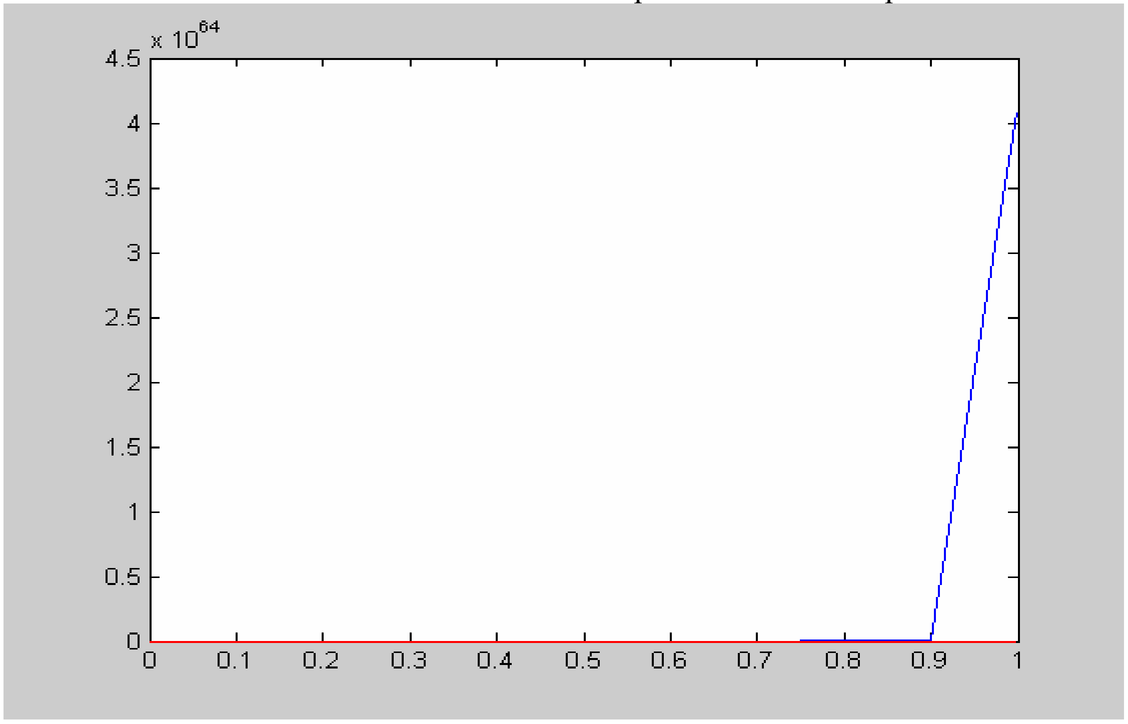
Doing tedious algebra we obtain:

$$w_{i+1} = w_i\left[1 + h\lambda + \frac{1}{2}(h\lambda)^2 + \frac{1}{6}(h\lambda)^3 + \frac{1}{24}(h\lambda)^4\right]$$

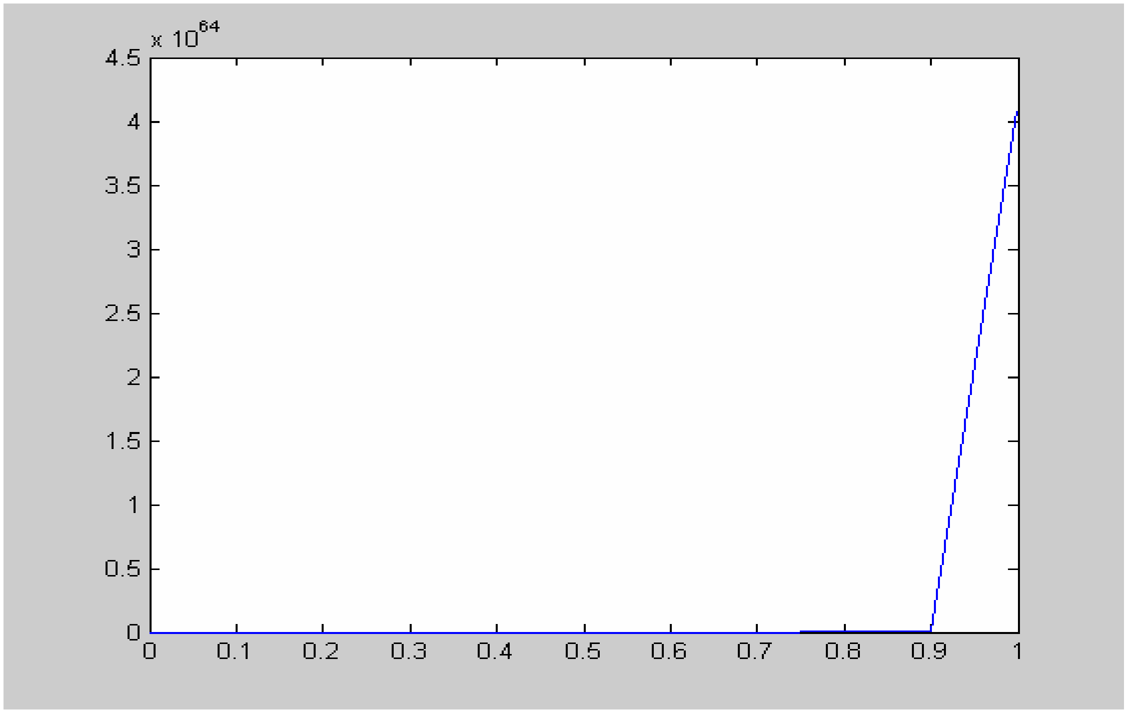
Using the following oscillator function:

```
function input = oscillator(t,y);
input(1) = -900*y(1);
```

and RK4 with h = .1. We can see that for this step size RK4 blows up.



The RK4 error plot looks exactly like the actual RK4 plot because y is essentially 0 for all x > .0001

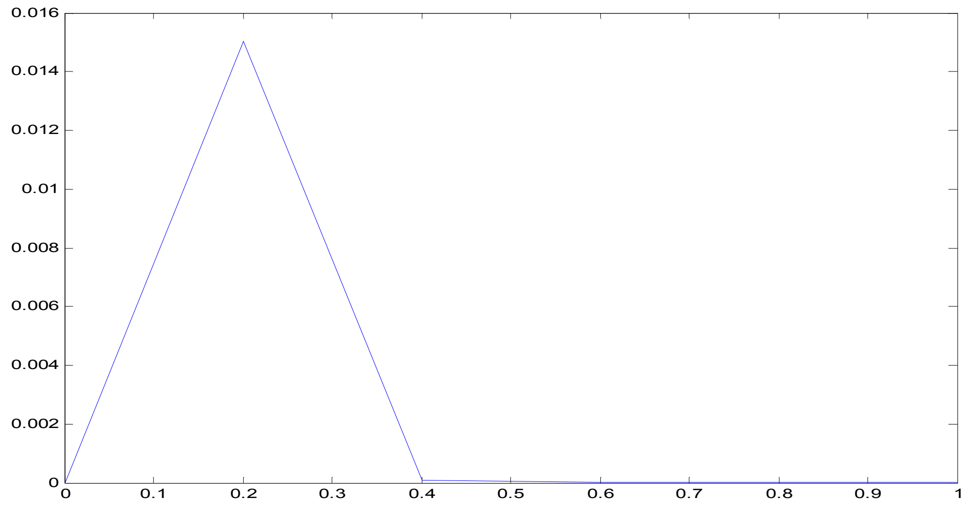


Part ii)
When h is 0.001 RK4 converges.

Part iv)
Here is the code with input RK4(0,1,.001,exp(1))

```
function out = RK4(a,b,h,alpha)

% Inputing h,a,b,alpha
clc
format long
% format short
w(1,:) = alpha;
t(1) = a;

N = (b-a)/h;

for i = 1:N
   k1 = h*feval('y_prime',t(i),w(i,:));
   k2 = h*feval('y_prime',t(i)+h/2,w(i,:)+.5*k1);
   k3 = h*feval('y_prime',t(i)+h/2,w(i,:)+k2/2);
   k4 = h*feval('y_prime',t(i)+h,w(i,:)+k3);
   w(i+1,:) = w(i,:) + 1/6*(k1+2*k2+2*k3+k4);
   t(i+1) = t(i) + h;
end


fprintf('\ndone\n');
fprintf('the value of y(b) is %2.10f\n',w(i+1,:));



for i = 1:length(t)

   y(i) = exp(1-(900*(t(i))));
end


plot(t,w(:,1),'b',t,y,'r')
```

Using the backward Euler code, with h = .2, the method converges. We evaluated the function and calculated the Error.



The error plotted against the t-values

Part v)
The backward Euler works for larger values of h compared to the RK4. Values 200 times bigger and achieve results 200 times less iterations.

Part 3

Zeeman Cusp Catastrophe

In this problem we will attempt to solve a series of non-linear BPVs. The problem that is posed is variant in both space (x,y) and in time (t) with periodic boundary conditions. The general concept which we will apply to solve this section is to divide the space and time interval into Nt elements (w.r.t t) and N (w.r.t x). The problem also involves the first derivative of y with respect to time and the second derivative with respect to x. We will apply a *Centered Difference* to solve in space and then a *Backward Euler* to solve in time. The Backward Euler is necessary due to the fact that the problem posed is a stiff one. These two algorithms will have to be implemented simultaneously.

Part A

$$\frac{\partial y}{\partial t} = -\frac{1}{\varepsilon}(y^3 + ay + b) + \sigma\frac{\partial^2 y}{\partial x^2}$$
$$\frac{\partial a}{\partial t} = b + 0.07v + \sigma\frac{\partial^2 a}{\partial x^2}$$
$$\frac{\partial b}{\partial t} = (1-a^2)b - a - 0.4y + 0.035v + \sigma\frac{\partial^2 b}{\partial x^2}$$

Now with Centered difference of $\frac{\partial^2 y_i}{\partial x^2} = \frac{(y_{i-1} - 2y_i + y_{i+1})}{h^2}$ where

$$h^2 = \frac{(b-a)^2}{N^2}\ ,\ (b-a) = 1$$
$$D = \sigma N^2,\ \varepsilon = 0.0001$$

$$y_i' = -10^4(y_i^3 + a_i y_i + b_i) + D(y_{i-1} - 2y_i + y_{i+1})$$

and correspondingly

$$a_i' = b_i + 0.07v_i + D(a_{i-1} - 2a_i + a_{i+1})$$
$$b_i' = (1-a_i^2)b_i - a_i - 0.4y_i + 0.035v_i + D(b_{i-1} - 2b_i + b_{i+1})$$

Part B

We will be solving 9 ODEs. There are 11 but the first two are the same as the last two.

Algorithm.

Step 1 : divide x into 11 steps where $\Delta x = \dfrac{(l-0)}{N}$, where $x_i = i\Delta x$

Apply this to the periodic boundary conditions

Pick a step size for t → Δt, where $t_j = j\Delta t$

Step 2 : For each x_i use a *Centered difference* to solve $\dfrac{\partial^2 y}{\partial x^2}$ for a specific t_j

Step 3 : Now use this information to solve the PDE in terms of (y,a,b) using a *Backward Euler* combined with a *Modified Newton* at i and j (Backward Euler and modified Newton described in part 2, question 3c)

Step 4 : Do this for all of i and j

Part C

The RK4 method exploded due to the fact that the problem is stiff. A stiff problem requires an implicit method to properly solve it, which the RK4 is not.
The code for RK4 in this question is:

```
ia = 0;
ib = 1.1;
Nt = 1000;
N=10;
for i = 1:N
   y(i,1) = 0;
   a(i,1) = -2*cos(2*pi*i/N);
   b(i,1) = 2*sin(2*pi*i/N);

end


h = (ib-ia)/Nt;
t = ia:h:ib;

for j = 1:Nt
   time = t(j);
   for i = 1:N

      if i == 1
         n = N;
         m = i+1;
```

```
        elseif i==N
            n = i-1;
            m = 1;
        else
            n = i-1;
            m = i+1;
        end
        ddydx = (y(n,j) -2*y(i,j) + y(m,j));
        ddadx = (a(n,j) -2*a(i,j) + a(m,j));
        ddbdx = (b(n,j) -2*b(i,j) + b(m,j));

        w(i,:) = feval('RK4',time,h,[ y(i,j) a(i,j) b(i,j) ], [ddydx ddadx ddbdx]);
        y(i,j+1) = w(i,1);
        a(i,j+1) = w(i,2);
        b(i,j+1) = w(i,3);
    end
end
plot(t,y(1,:))
```

Part D

Our backward Euler did not perform well on this function. Once again we obtained surprising results considering backward Euler is designed to work on stiff functions. Our backward Euler code worked extremely well on non stiff functions, (up to 200 hundred times faster then RK4). However, it yielded unexpected results for stiff functions and therefore was useless to us on this question.

*Please note we could not output graphs for the next questions due to lack of values obtained from this function.