Madrigal Weersink
weersinm
400244666

3X03 Assignment 4

04/12/2022

1.

```
function netbpfull
%NETBP_FULL
%   Extended version of netbp, with more graphics
%
%   Set up data for neural net test
%   Use backpropagation to train
%   Visualize results
%
% C F Higham and D J Higham, Aug 2017
%
%%%%%%% DATA %%%%%%%%%%
% xcoords, ycoords, targets
% x1 = [0.1,0.3,0.1,0.6,0.4,0.6,0.5,0.9,0.4,0.7];
% x2 = [0.1,0.4,0.5,0.9,0.2,0.3,0.6,0.2,0.4,0.6];
% y = [ones(1,5) zeros(1,5); zeros(1,5) ones(1,5)];

dataset = load('dataset.mat');
x1 = dataset.X(:,1)';
x2 = dataset.X(:,2)';
y = dataset.Y';
xlen = length(x1);
halflen = fix(xlen/2);

figure(1)
clf
a1 = subplot(1,1,1);
plot(x1(1:halflen),x2(1:halflen),'ro','MarkerSize',12,'LineWidth',4)
hold on
plot(x1(halflen+1:xlen),x2(halflen+1:xlen),'bx','MarkerSize',12,'LineWidth',4)
a1.XTick = [0 1];
a1.YTick = [0 1];
a1.FontWeight = 'Bold';
a1.FontSize = 16;
xlim([0,1])
ylim([0,1])

%print -dpng pic_xy.png

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize weights and biases
rng(5000);
W2 = 0.5*randn(10,2);
W3 = 0.5*randn(8,10);
W4 = 0.5*randn(2,8);
b2 = 0.5*randn(10,1);
b3 = 0.5*randn(8,1);
b4 = 0.5*randn(2,1);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Forward and Back propagate
% Pick a training point at random
eta = 0.25;
Niter = 1e6;
savecost = zeros(Niter,1);
saveaccuracy = zeros(Niter, 1);
for counter = 1:Niter
    k = randi(xlen);
    l = randi(xlen);
    m = randi(xlen);
    n = randi(xlen);
    x = [x1(k) x1(l) x1(m) x1(n); x2(k) x2(l) x2(m) x2(n)]; % increase batch size to 4
    % Forward pass
    a2 = activate(x,W2,b2);
    a3 = activate(a2,W3,b3);
    a4 = activate(a3,W4,b4);
    % Backward pass
    delta4 = a4.*(1-a4).*(a4-y(:,k));
    delta3 = a3.*(1-a3).*(W4'*delta4);
    delta2 = a2.*(1-a2).*(W3'*delta3);
    % Gradient step
    W2 = W2 - eta*delta2*x';
    W3 = W3 - eta*delta3*a2';
    W4 = W4 - eta*delta4*a3';
    b2 = b2 - eta*delta2;
    b3 = b3 - eta*delta3;
    b4 = b4 - eta*delta4;
    % Monitor progress
    [newcost, accuracy] = cost(W2,W3,W4,b2,b3,b4);   % display cost to screen
    savecost(counter) = newcost;
    saveaccuracy(counter) = accuracy;   % adds each accuracy to vector for plotting
    if accuracy == 1    % if all points are 97% accurate, exit loop to stop training
        fprintf('number of iterations: %d\n', Niter);
        break
    end
end
fprintf('Final accuracy: %f\n', accuracy);

figure(2)
clf
semilogy([1:1e4:Niter],savecost(1:1e4:Niter),'b-','LineWidth',2)
xlabel('Iteration Number')
ylabel('Value of cost function')
set(gca,'FontWeight','Bold','FontSize',18)
print -dpng pic_cost.png

%%%%%%%%%%% Display shaded and unshaded regions
N = 500;
Dx = 1/N;
Dy = 1/N;
xvals = [0:Dx:1];
yvals = [0:Dy:1];
for k1 = 1:N+1
    xk = xvals(k1);
    for k2 = 1:N+1
```

```
        yk = yvals(k2);
        xy = [xk;yk];
        a2 = activate(xy,W2,b2);
        a3 = activate(a2,W3,b3);
        a4 = activate(a3,W4,b4);
        Aval(k2,k1) = a4(1);
        Bval(k2,k1) = a4(2);
    end
end
[X,Y] = meshgrid(xvals,yvals);

figure(3)
clf
a2 = subplot(1,1,1);
Mval = Aval>Bval;
contourf(X,Y,Mval,[0.5 0.5])
hold on
colormap([1 1 1; 0.8 0.8 0.8])
plot(x1(1:halflen),x2(1:halflen),'ro','MarkerSize',12,'LineWidth',4)
plot(x1(halflen+1:xlen),x2(halflen+1:xlen),'bx','MarkerSize',12,'LineWidth',4)
a2.XTick = [0 1];
a2.YTick = [0 1];
a2.FontWeight = 'Bold';
a2.FontSize = 16;
xlim([0,1])
ylim([0,1])

print -dpng pic_bdy_bp.png

% adding in accuracy plot. follows same specs as other figures, especially
% fig 2 (since both have iteration number on the x axis)
figure(4)
clf
plot([1:1e4:Niter], saveaccuracy(1:1e4:Niter),'b-','LineWidth',2)
xlabel('Iteration Number')
ylabel('Accuracy')
set(gca,'FontWeight','Bold','FontSize',18)
ylim([0,1])


  function [costval, accuracy] = cost(W2,W3,W4,b2,b3,b4)
     accuracy = 0;
     costvec = zeros(xlen,1);
     for i = 1:xlen
         x =[x1(i);x2(i)];
         a2 = activate(x,W2,b2);
         a3 = activate(a2,W3,b3);
         a4 = activate(a3,W4,b4);
         costvec(i) = norm(y(:,i) - a4,2);

         for j = 1:length(y(:,i))
            if y(j,i) == 1
                % checks if each point is 97% accurate; if it is, it counts as
accurate
                if a4(j) >= 0.97
                    accuracy = accuracy + 1; % accuracy equals the number of 97%
```

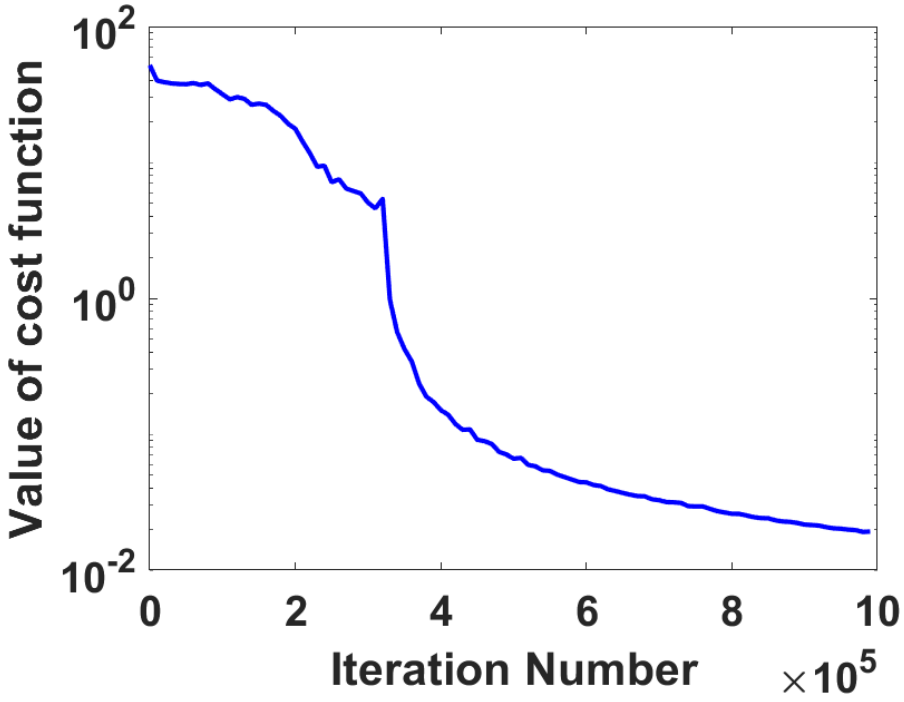
```
accurate points at the end of the outer loop
                end
            end
        end
    end
    costval = norm(costvec,2)^2;
    accuracy = accuracy/xlen; % divides accuracy by total number of points to get
percentage of accurate points
  end % of nested function
end
```

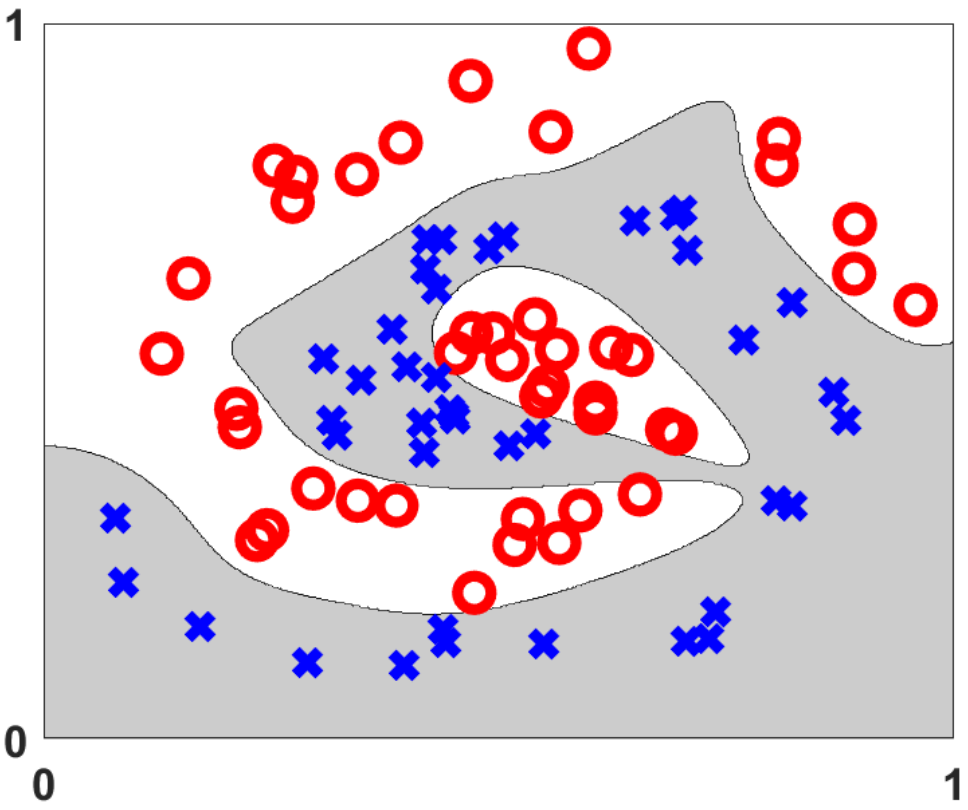
b)

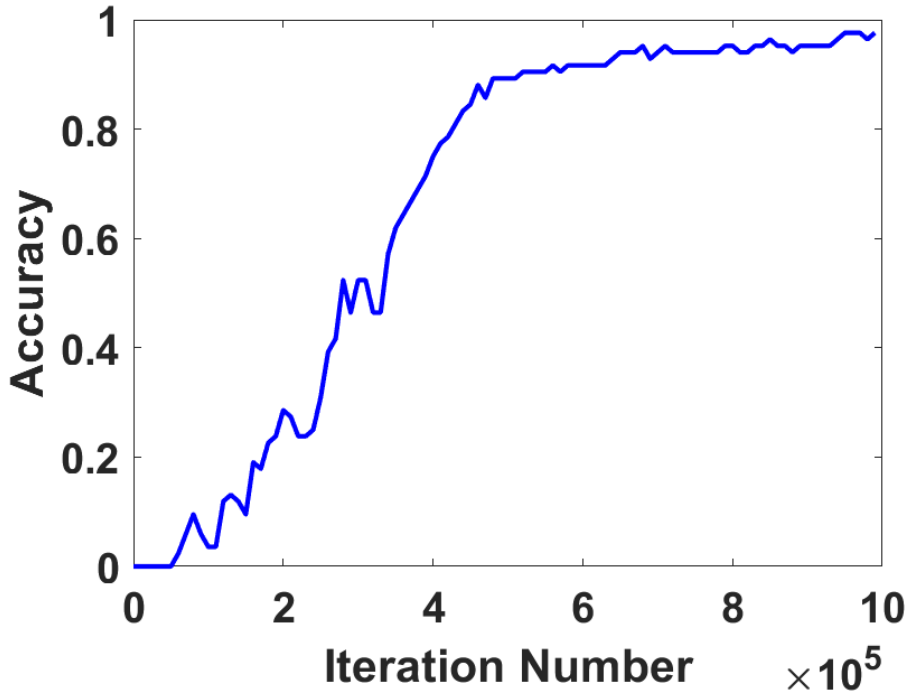
Parameters: 10, 8, and 2 neurons with a learning rate of 0.25

Niter: 1e6, though training will stop if all points achieve 97% accuracy

Final accuracy: 0.988095 (98.8% of points are at least 97% accurate - overall accuracy 95.8%)

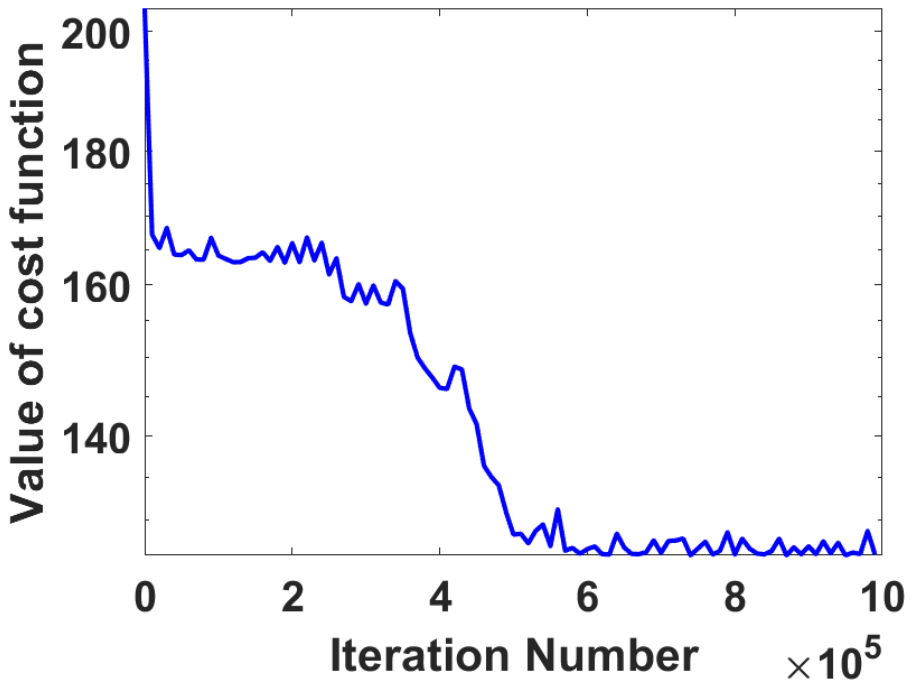


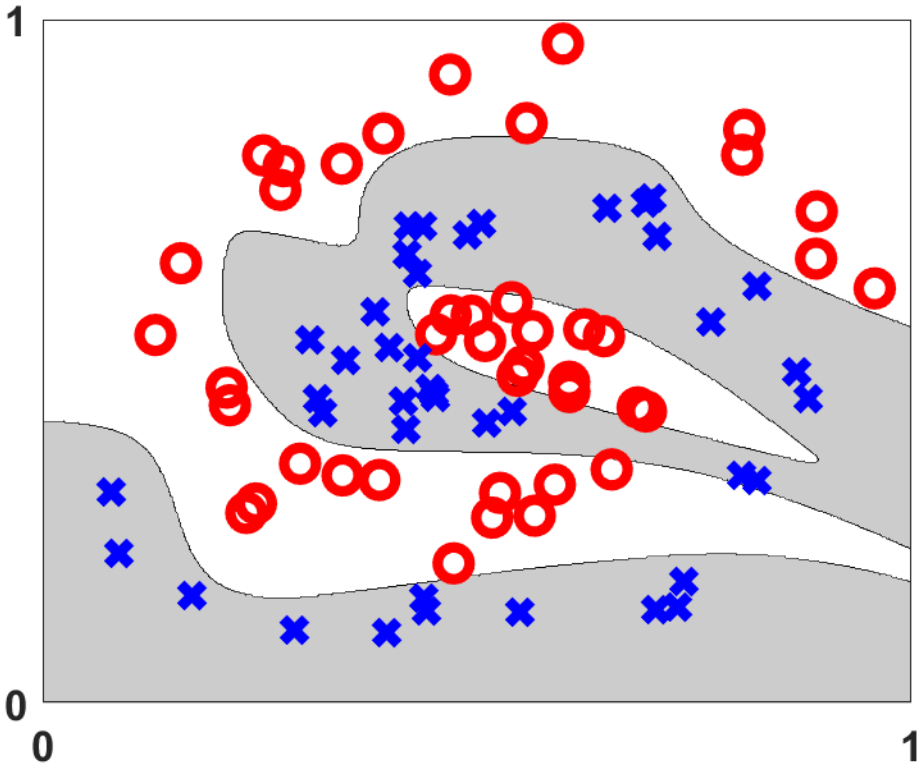


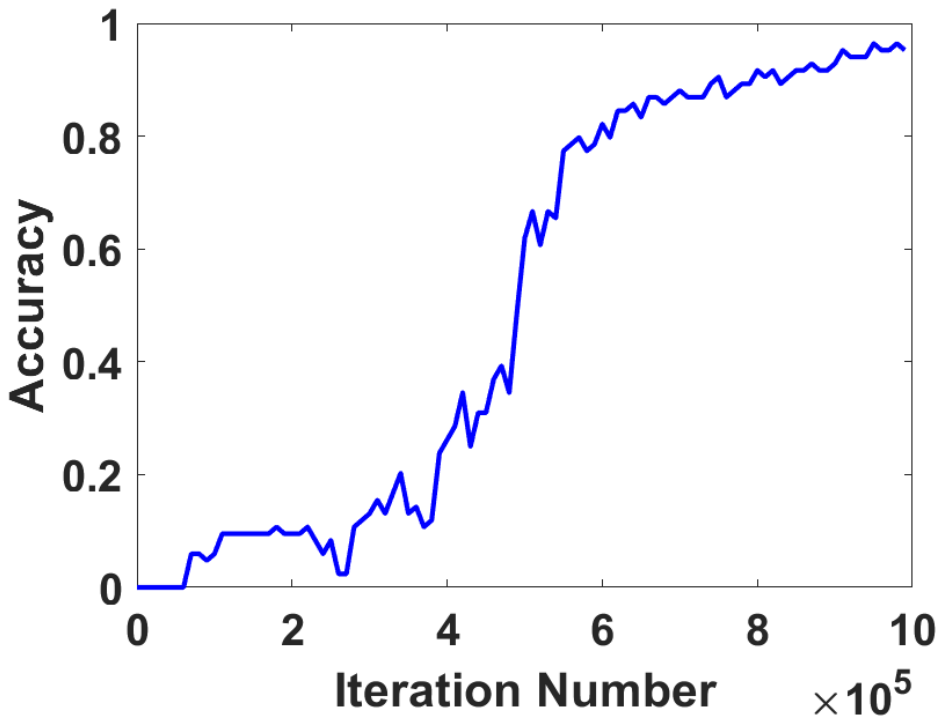


c)

Final accuracy: 0.964286 (96.4% of points are at least 97% accurate - overall accuracy 93.5%)


Value of cost function
200
180
160
140
0
2
4
6
8
10
Iteration Number
$\times 10^5$


1
0
0
1



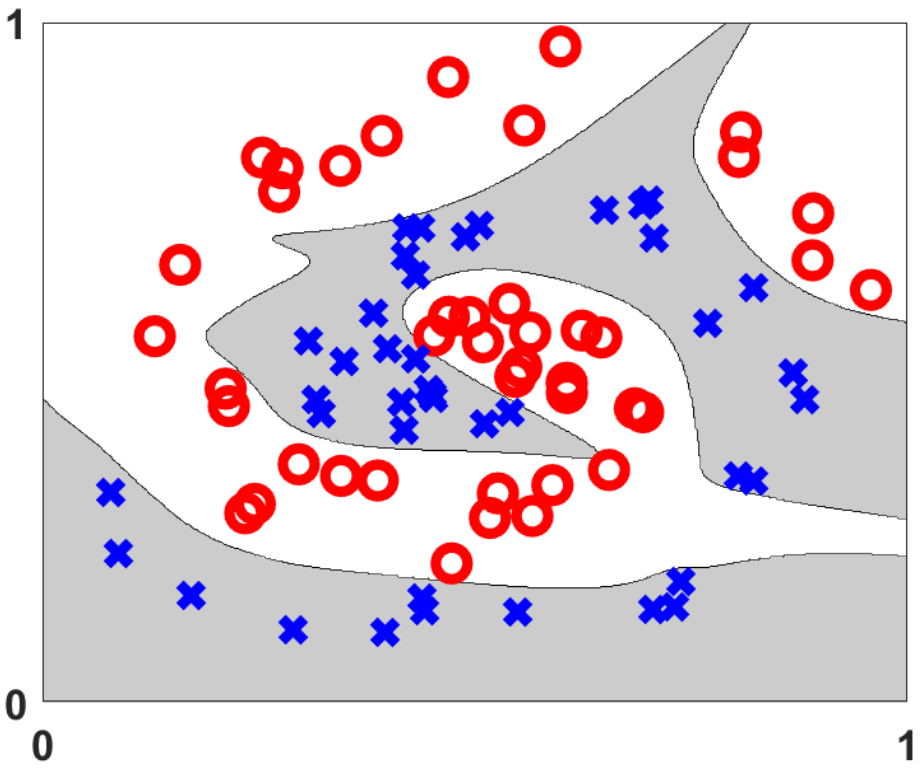
The training time does not improve with a batch size of 4, it actually gets much slower. The accuracy does not improve, and the program is doing 4 times the calculations, so it gets slower per iteration without actually reducing the number of iterations. It's possible that if I lowered the accuracy threshold within the cost function, it would run faster. However, if I look at the accuracy graphs in b) and c), the program with a batch of one reaches a higher accuracy significantly faster than the program with a batch of four.

Bonus:

```
function nlsrun
%NLSRUN
%
% Set up data for neural net test
% Call nonlinear least squares optimization code to minimze the error
% Visualize results
%
% C F Higham and D J Higham, August 2017
%
%%%%%%% DATA %%%%%%%%%%
% coordinates and targets
% x1 = [0.1,0.3,0.1,0.6,0.4,0.6,0.5,0.9,0.4,0.7];
% x2 = [0.1,0.4,0.5,0.9,0.2,0.3,0.6,0.2,0.4,0.6];
% y = [ones(1,5) zeros(1,5); zeros(1,5) ones(1,5)];

dataset = load('dataset.mat');
x1 = dataset.X(:,1)';
x2 = dataset.X(:,2)';
y = dataset.Y';
xlen = length(x1);
```

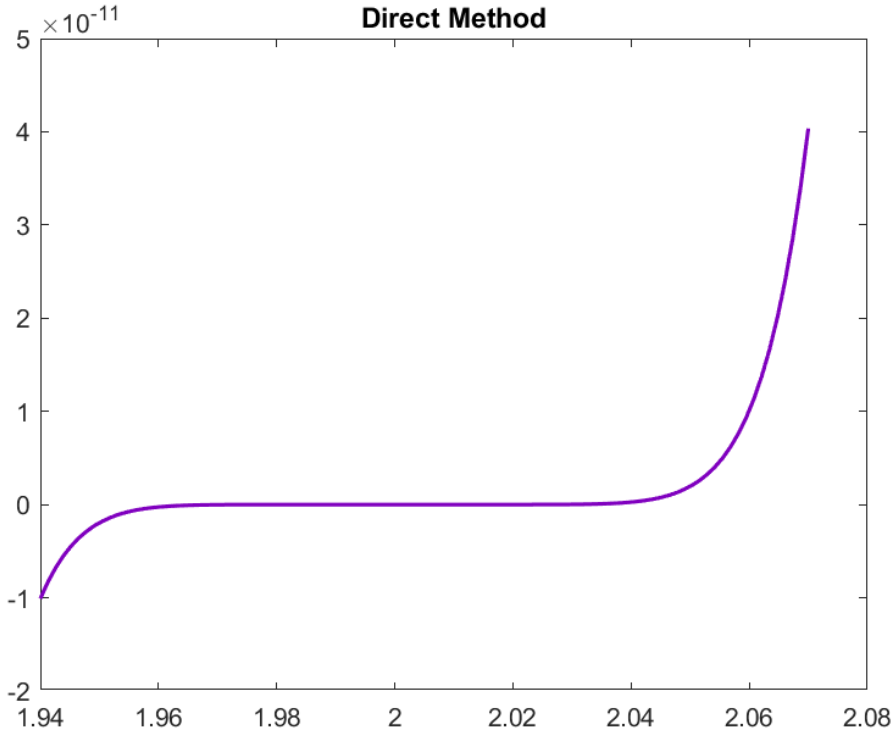
```
halflen = fix(xlen/2);

figure(1)
clf
a1 = subplot(1,1,1);
plot(x1(1:halflen),x2(1:halflen),'ro','MarkerSize',12,'LineWidth',4)
hold on
plot(x1(halflen+1:xlen),x2(halflen+1:xlen),'bx','MarkerSize',12,'LineWidth',4)
a1.XTick = [0 1];
a1.YTick = [0 1];
a1.FontWeight = 'Bold';
a1.FontSize = 16;
xlim([0,1])
ylim([0,1])

print -dpng pic_xy.png

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize weights/biases and call optimizer
rng(5000);
Pzero = 0.5*randn(135,1);

global finalW2;
global finalW3;
global finalW4;
global finalb2;
global finalb3;
global finalb4;

options = optimoptions('lsqnonlin','MaxFunctionEvaluations', 1e6, 'MaxIterations',
1e4);
lb = -1.*Inf(size(Pzero));
ub = Inf(size(Pzero));
[finalP,finalerr] = lsqnonlin(@neterr,Pzero, lb, ub, options);

% grid plot
N = 500;
Dx = 1/N;
Dy = 1/N;
xvals = [0:Dx:1];
yvals = [0:Dy:1];
for k1 = 1:N+1
    xk = xvals(k1);
    for k2 = 1:N+1
        yk = yvals(k2);
        xy = [xk;yk];
        a2 = activate(xy,finalW2,finalb2);
        a3 = activate(a2,finalW3,finalb3);
        a4 = activate(a3,finalW4,finalb4);
        Aval(k2,k1) = a4(1);
        Bval(k2,k1) = a4(2);
     end
end
[X,Y] = meshgrid(xvals,yvals);
```

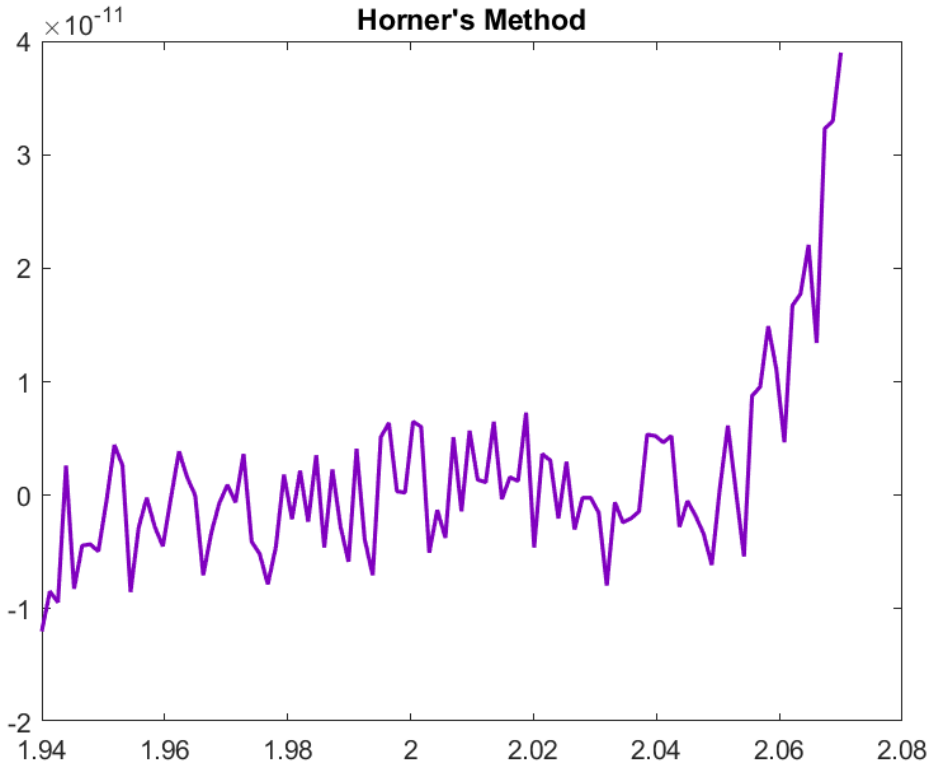
```
figure(2)
clf
a2 = subplot(1,1,1);
Mval = Aval>Bval;
contourf(X,Y,Mval,[0.5 0.5])
hold on
colormap([1 1 1; 0.8 0.8 0.8])
plot(x1(1:halflen),x2(1:halflen),'ro','MarkerSize',12,'LineWidth',4)
plot(x1(halflen+1:xlen),x2(halflen+1:xlen),'bx','MarkerSize',12,'LineWidth',4)
a2.XTick = [0 1];
a2.YTick = [0 1];
a2.FontWeight = 'Bold';
a2.FontSize = 16;
xlim([0,1])
ylim([0,1])

print -dpng pic_bdy.png

    function [costvec] = neterr(Pval)
    % return a vector whose two-norm squared is the cost function

     % changed values to match those in netbpfull
     W2 = zeros(10,2);
     W3 = zeros(8,10);
     W4 = zeros(2,8);
     W2(:) = Pval(1:20);
     W3(:) = Pval(21:100);
     W4(:) = Pval(101:116);
     b2 = Pval(117:126);
     b3 = Pval(127:134);
     b4 = Pval(134:135);

     costvec = zeros(xlen,1);
     for i = 1:xlen
         x = [x1(i);x2(i)];
         a2 = activate(x,W2,b2);
         a3 = activate(a2,W3,b3);
         a4 = activate(a3,W4,b4);
         costvec(i) = norm(y(:,i) - a4,2);
     end
     finalW2 = W2;
     finalW3 = W3;
     finalW4 = W4;
     finalb2 = b2;
     finalb3 = b3;
     finalb4 = b4;
     end % of nested function
end
```



2. a)

Matlab code: p2_main.m, horner_binomial.m





The direct method clearly produces a much smoother line. This is because the equation from Horner's method lends itself to much more error than the direct method. The direct method consists of 1 subtraction operation and then 1 power operation, whereas the Horner method has 9 addition/subtraction operations and 8 multiplication operations, giving significant opportunity for rounding errors to affect the final value.

b)

Using this bisection method:

```
function root = bisection_method(f, a, b, tol)
    while(1)
        c = (a + b)/2;                      % start with c as the average of a and b
        if double(f(c)) == 0 || (b - a)/2 < tol     % loop will run until we find c
such that f(c) is zero or (b - a)/2 is less than tol
            disp(f(c))
            root = c;                       % when we find such c, it is the root
            break
        end
        if sign(f(a)) == sign(f(c))         % if we haven't found our root, we update
a or b, moving closer to the desired value
            a = c;
        else
            b = c;
        end
    end
end
```

I found that the function would always exit at r = 2.0101 because f(c) would evaluate to zero, likely due to rounding errors. As such, I was unable to reach $|r-2| < 10^{-6}$ using this program.

3. a)

```
No solution found.

fsolve stopped because the problem appears regular as measured by the gradient,
but the vector of function values is not near zero as measured by the
value of the function tolerance.

<stopping criteria details>
Newton's Method:  x1: 5  x2: 4   n: 43
fsolve:           x1: 15 x2: -2
```

b)

```
No solution found.

fsolve stopped because the problem appears regular as measured by the gradient,
but the vector of function values is not near zero as measured by the
value of the function tolerance.

<stopping criteria details>
Newton's Method:  x1: 1.666667  x2: -0.666667  x3: 1.333333  n: 57
fsolve:           x1: 1.366025  x2: -0.366025  x3: 1.732051
```

c)

```
Warning: dx contains NaN elements. Program cannot continue.

No solution found.

fsolve stopped because the problem appears regular as measured by the gradient,
but the vector of function values is not near zero as measured by the
value of the function tolerance.

<stopping criteria details>
Newton's Method:  x1: Inf  x2: Inf  x3: Inf  x4: Inf  n: 1
fsolve:           x1: 1.000000  x2: 2.000000  x3: 1.000000  x4: 1.000000
```

d)

```
Warning: dx contains NaN elements. Program cannot continue.

No solution found.

fsolve stopped because the problem appears regular as measured by the gradient,
but the vector of function values is not near zero as measured by the
value of the function tolerance.

<stopping criteria details>
Newton's Method:  x1: Inf  x2: Inf   n: 2
fsolve:           x1: 1.800000e+00 x2: 0
```

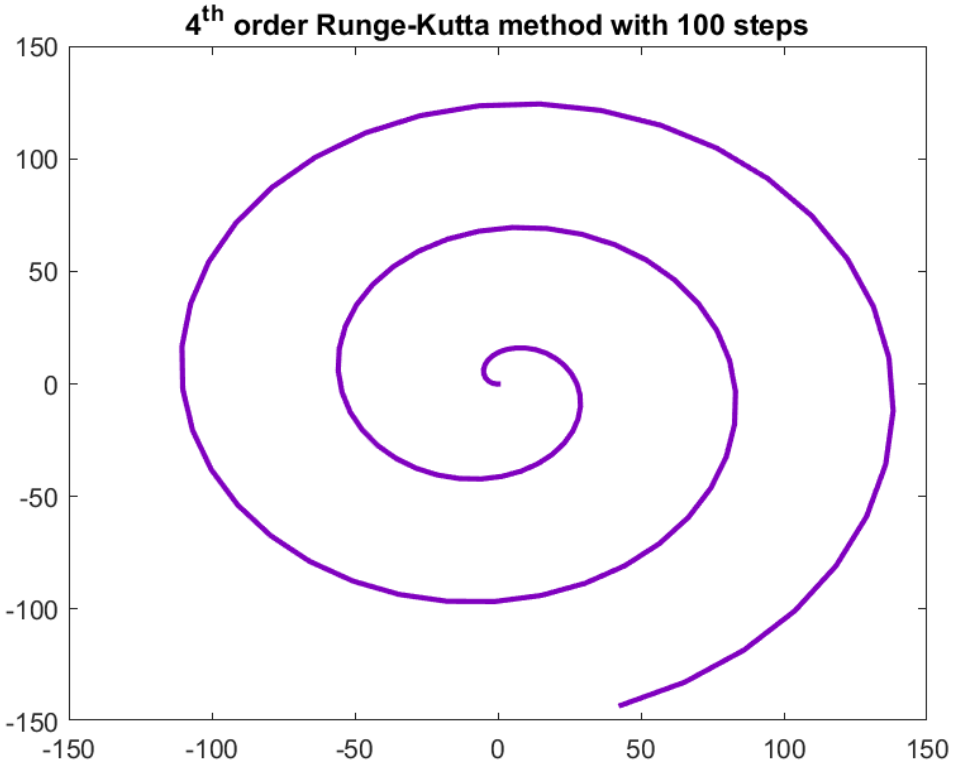
I had troubles with fsolve throughout this question, and consistently received a "No solution found" response. However, my own implementation of Newton's Method found the accurate

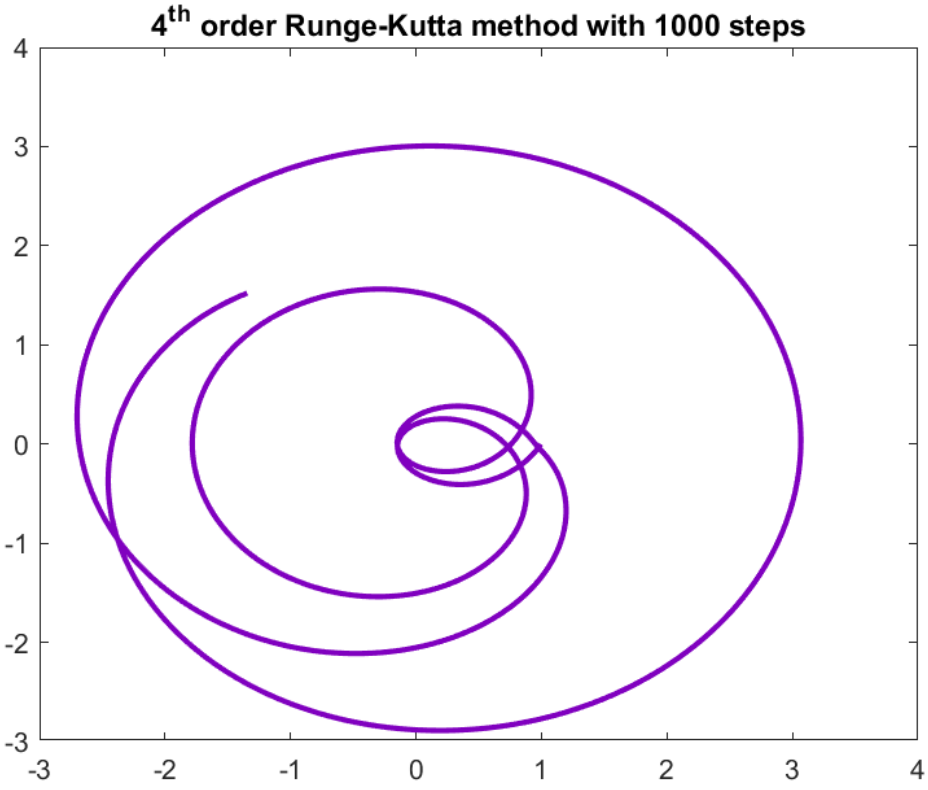
values for parts a) and b) in 43 and 57 steps, respectively.
It did not work for parts c) and d), likely because the values were too small for the program to handle. I found that NaN values would appear in my matrices, and then the function would infinitely iterate, so I built in a break if any NaN values appeared in dx. The function would then return all infinite values, so the user doesn't mistake them for the actual answer.
I have included my Newton's Method function and the code for part a) below.

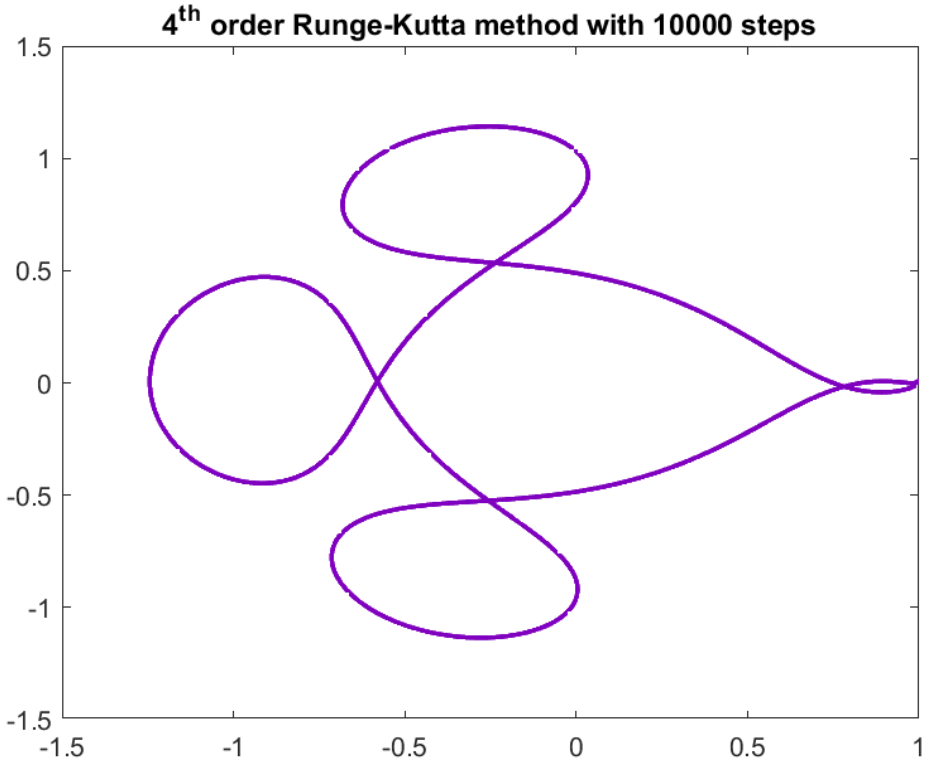
```
function [x1, n] = newtons_method(x0, f, df, tol)
    flag = 0;
    n = 1;
    while n < 10000
        y = zeros(1, length(f));
        dy = zeros(size(df, 1), size(df, 2));
        for i = 1:length(f)
            y(i) = f{i}(x0{:});
        end
        for i = 1:size(df, 1)
            for j = 1:size(df, 2)
                dy(i, j) = df{i, j}(x0{:});
            end
        end
        dx = dy\y';
        for i = 1:length(dx)
            j = dx(i);
            if isnan(j)
                fprintf('Warning: dx contains NaN elements. Program cannot
continue.\n');
                x1 = Inf(1, length(x0));
                flag = 1;
                break
            end
        end
        if flag
            break
        end
        if max(abs(dx)) < tol
            x1 = cellfun(@minus, x0, num2cell(dx'), 'Un', 1);
            break
        else
            x0 = num2cell(cellfun(@minus, x0, num2cell(dx'), 'Un', 1));
        end
        n = n + 1;
    end
    if n == 10000
        disp(dx)
        fprintf("Reached max number of iterations. Current tolerance is %f, with x
values of:", max(abs(dx)));
        x1 = x0;
        disp(x1)
    end
end

% a)
funA = @partA;

x0 = {15 -2};
```

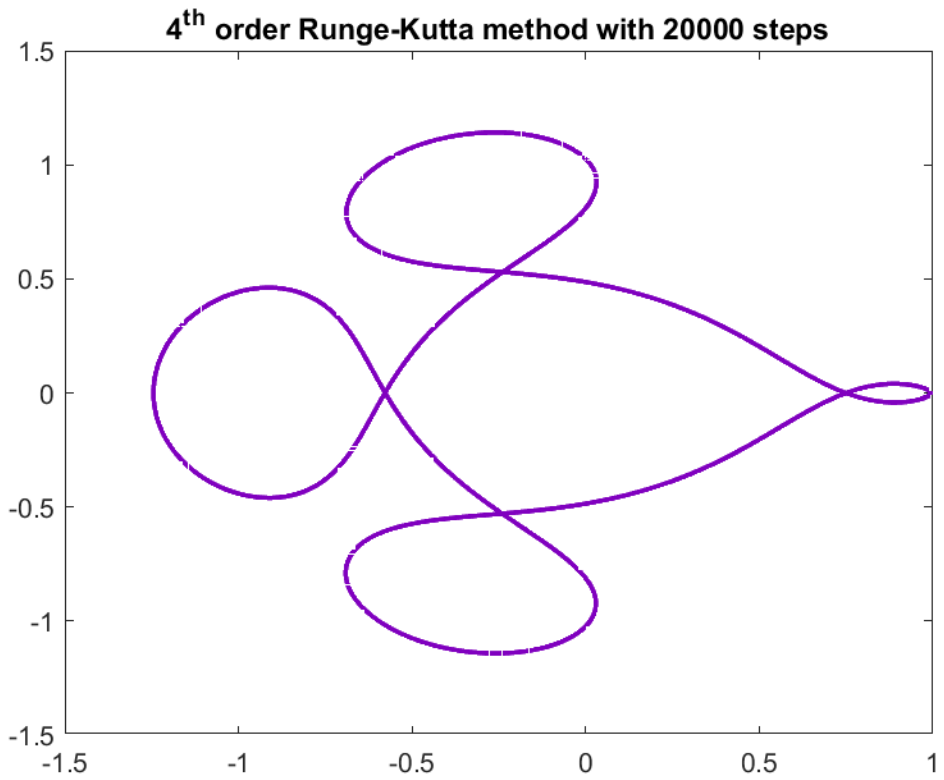
```
f = {@(x1, x2) x1 + x2.*(x2.*(5 - x2) - 2) - 13; @(x1, x2) x1 + x2.*(x2.*(1 + x2) -
14) - 29};                                % initializing each function to equal zero
df = {@(x1, x2) 1 @(x1, x2) -x2.*(x2.*2 - 5) - x2.*(x2 - 5)-2; @(x1, x2) 1 @(x1, x2)
x2.*(x2.*2 + 1) + x2.*(x2 + 1) - 14};  % initializing each partial derivative

[x_ans, n] = newtons_method(x0, f, df, 10^-6);
x1 = 15;
x2 = -2;
x0 = [x1 x2];
[x_fsolve, ~] = fsolve(@(x) partA(x1, x2), x0);
fprintf("Newton's Method:\tx1: %d  x2: %d   n: %d\nfsolve:\t\t\tx1: %d x2: %d\n\n",
x_ans(1), x_ans(2), n, x_fsolve(1, 1), x_fsolve(1, 2));

function F = partA(x1, x2)
    F(1) = x1 + x2*(x2*(5 - x2) - 2) - 13;
    F(2) = x1 + x2*(x2*(1 + x2) - 14) - 29;
end
```

4.




4th order Runge-Kutta method with 1000 steps
4
3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3
4


4th order Runge-Kutta method with 10000 steps
1.5
1
0.5
0
-0.5
-1
-1.5
-1.5
-1
-0.5
0
0.5
1



The orbit appears to be qualitatively correct at 10,000 steps, as it stays consistent between 10,000 and 20,000 steps. This is also where the plot becomes a closed loop, which is generally expected in an orbit.

To check whether the plot would stay consistent with an even higher number of steps, I graphed it using 100,000 steps. The plot did stay consistent, making me more confident that this is the final plot that I'm looking for. I also plotted the function with 5000 steps to see if 10,000 was the actual lower limit, or it was less. At 5000 steps, the plot was starting to resemble the final plot, but did not form a closed loop. I then plotted at 6000, 8000, and 9000 steps, and found that at 8000 and 9000 steps, the plot was very close to the final plot, and the loop was nearly closed, but I would say that 10,000 steps is where it really looked like an actual closed-loop orbit.

Code:

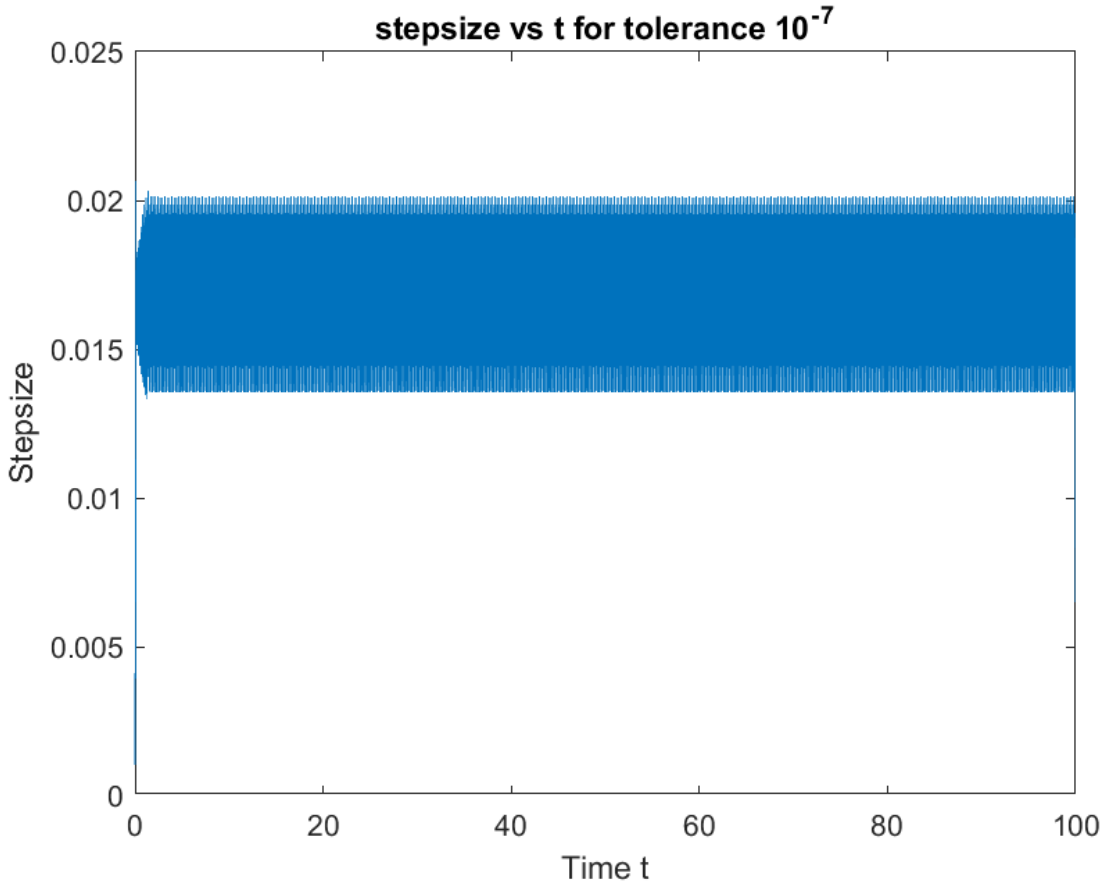
```
mu = 0.012277471;
mu_hat = 1 - mu;

fy_1 = @(t, u1, u2, y1, y2, z1, z2) z1;
fz_1 = @(t, u1, u2, y1, y2, z1, z2) u1 + 2*y2 - mu_hat*(u1 + mu)/(((u1 + mu)^2 +
u2^2)^(3/2)) - mu*(u1 - mu_hat)/(((u1 - mu_hat)^2 + u2^2)^(3/2));
fy_2 = @(t, u1, u2, y1, y2, z1, z2) z2;
fz_2 = @(t, u1, u2, y1, y2, z1, z2) u2 - 2*y1 - mu_hat*u2/(((u1 + mu)^2 +
u2^2)^(3/2)) - mu*u2/(((u1 - mu_hat)^2 + u2^2)^(3/2));

steps_vec = [100 1000 10000 20000];
for i = 1:length(steps_vec)
```

```
    steps = steps_vec(i);
    h = 17.1/steps;
    t = 0:h:17.1;

    u1 = zeros(1,length(t));
    u2 = zeros(1,length(t));
    y1 = zeros(1,length(t));
    y2 = zeros(1,length(t));
    z1 = zeros(1,length(t));
    z2 = zeros(1,length(t));
    u1(1) = 0.994;
    u2(1) = 0;
    y1(1) = 0;
    y2(1) = -2.00158510637908252242053786 22;

    for j = 1:(length(t)-1)
        k1_y1 = fy_1(t(j), u1(j), u2(j), y1(j), y2(j), z1(j), z2(j));
        k1_z1 = fz_1(t(j), u1(j), u2(j), y1(j), y2(j), z1(j), z2(j));
        k1_y2 = fy_2(t(j), u1(j), u2(j), y1(j), y2(j), z1(j), z2(j));
        k1_z2 = fz_2(t(j), u1(j), u2(j), y1(j), y2(j), z1(j), z2(j));

        k2_y1 = fy_1(t(j) + h/2, y1(j) + h/2*k1_y1, z1(j) + h/2*k1_z1);
        k2_z1 = fz_1(t(j) + h/2, y1(j) + h/2*k1_y1, z1(j) + h/2*k1_z1);
        k2_y2 = fy_2(t(j) + h/2, y1(j) + h/2*k1_y1, z1(j) + h/2*k1_z1);
        k2_z2 = fz_2(t(j) + h/2, y1(j) + h/2*k1_y1, z1(j) + h/2*k1_z1);


        k3_y = fy_1(t(j) + h/2, y1(j) + h/2*k2_y1, z1(j) + h/2*k2_z1);
        k3_z = fz_1(t(j) + h/2, y1(j) + h/2*k2_y1, z1(j) + h/2*k2_z1);
        k4_y = fy_1(t(j) + h, y1(j) + h*k3_y, z1(j) + h*k3_z);
        k4_z = fz_1(t(j) + h, y1(j) + h*k3_y, z1(j) + h*k3_z);
        y1(j + 1) = y1(j) + h/6*(k1_y1 + 2*k2_y1 + 2*k3_y + k4_y);
        z1(j + 1) = z1(j) + h/6*(k1_z1 + 2*k2_z1 + 2*k3_z + k4_z);
    end
end

%fy_2=@(x,y,z) z;
fz=@(x,y,z) 0.1*(1-x^2)*z-x;
t(1)=0;
z1(1)=1;
y1(1)=t;
h=0.1;
xfinal=10;
N=ceil((xfinal-t(1))/h);
for j=1:N
    k1_y1 = fy_1(t(j), y1(j), z1(j));
    k1_z1 = fz_1(t(j), y1(j), z1(j));
    k2_y1 = fy_1(t(j) + h/2, y1(j) + h/2*k1_y1, z1(j) + h/2*k1_z1);
    k2_z1 = fz_1(t(j) + h/2, y1(j) + h/2*k1_y1, z1(j) + h/2*k1_z1);
    k3_y = fy_1(t(j) + h/2, y1(j) + h/2*k2_y1, z1(j) + h/2*k2_z1);
    k3_z = fz_1(t(j) + h/2, y1(j) + h/2*k2_y1, z1(j) + h/2*k2_z1);
    k4_y = fy_1(t(j) + h, y1(j) + h*k3_y, z1(j) + h*k3_z);
    k4_z = fz_1(t(j) + h, y1(j) + h*k3_y, z1(j) + h*k3_z);
    y1(j + 1) = y1(j) + h/6*(k1_y1 + 2*k2_y1 + 2*k3_y + k4_y);
    z1(j + 1) = z1(j) + h/6*(k1_z1 + 2*k2_z1 + 2*k3_z + k4_z);
end
```

5.

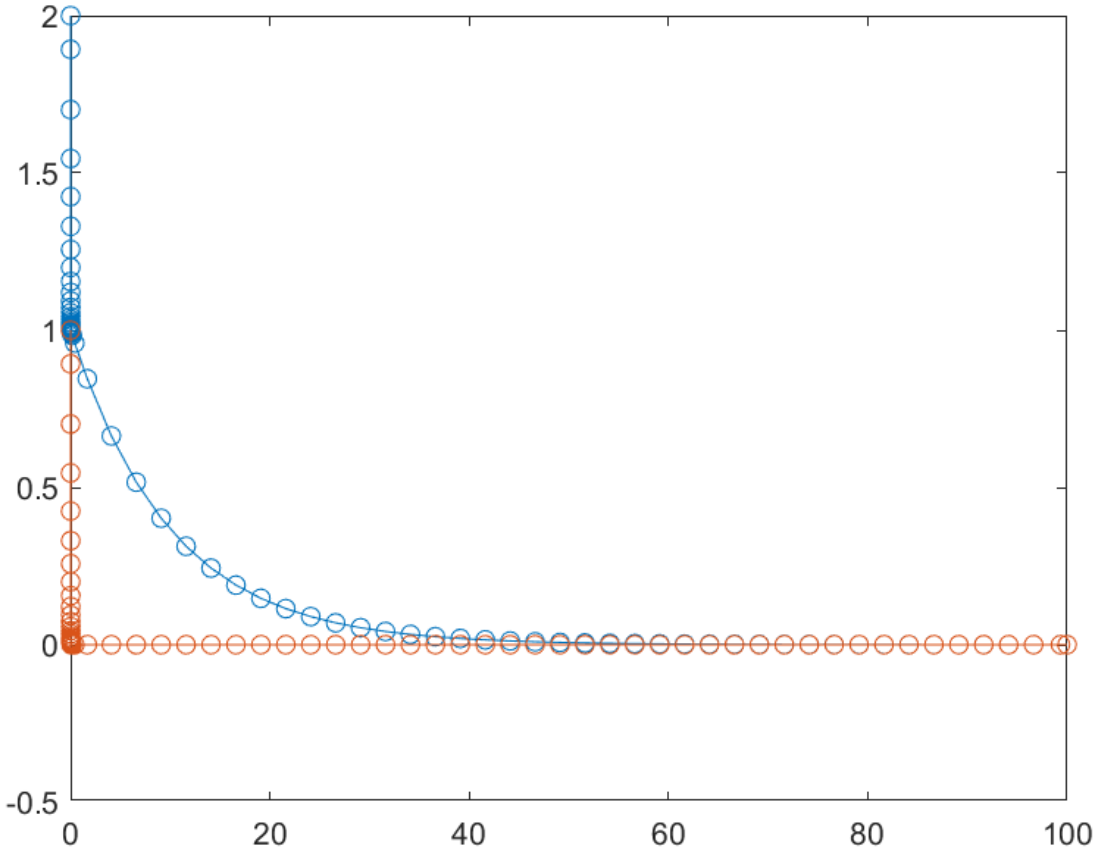
Table

'tol'	'nfevals'	'err_y1'	'err_y2'	'avg stepsize'
0.1000	38593	8.7830e-08	8.7830e-08	0.0166
0.0100	38593	8.7830e-08	8.7830e-08	0.0166
1.0000e-03	38617	8.7830e-08	8.7830e-08	0.0166
1.0000e-04	38581	8.7830e-08	8.7830e-08	0.0166
1.0000e-05	38593	8.7830e-08	8.7830e-08	0.0166
1.0000e-06	38617	8.7830e-08	8.7830e-08	0.0166
1.0000e-07	38635	8.7830e-08	8.7830e-08	0.0166
1.0000e-08	38647	8.7830e-08	8.7830e-08	0.0165
1.0000e-09	38395	8.7830e-08	8.7830e-08	0.0165
1.0000e-10	38689	8.7830e-08	8.7830e-08	0.0165
1.0000e-11	38701	8.7830e-08	8.7830e-08	0.0165
1.0000e-12	38725	8.7830e-08	8.7830e-08	0.0165



The stepsize appears to oscillate between 0.02 and ~0.013. This is because some parts of the function vary more quickly, so they need smaller stepsizes than others. The stepsize stays small throughout due to the possibility of solutions that vary quite rapidly.

Plot of the IVP using ode23s



number of function evals: 485

This solver is much more efficient than ode45 for this ODE. ode45 used at least 38,000 function evaluations for the system, even with a high tolerance. ode23s, on the other hand, uses only 485. This is because the equations we are solving are stiff. They vary relatively slowy, but there are other solutions nearby that vary much more rapidly, so a non-stiff method must use very small steps to get a good result. ode23s, the stiff method, will likely do more calculations per step, but it can take far, far fewer steps than ode45 because this ODE is well suited to it.