# A4

December 3, 2022 3:38 PM

1)

```
function netbp_full
close all; %%%%%%%%%%%%%%%%%%%%%%%%
tic
%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)];

data = load('dataset.mat');

x1 = data.X(:,1);
x2 = data.X(:, 2);
y = data.Y';
y([1, 2],:)=y([2,1],:);

figure(1)
clf
a1 = subplot(1,1,1);
plot(x1(1:42),x2(1:42),'ro','MarkerSize',12,'LineWidth',4)
hold on
plot(x1(43:84),x2(43:84),'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(5,2); %
W3 = 0.5*randn(3,5); %%
W4 = 0.5*randn(2,3);
b2 = 0.5*randn(5,1); %
b3 = 0.5*randn(3,1);
b4 = 0.5*randn(2,1);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Forward and Back propagate
% Pick a training point at random
eta = 0.05;
```

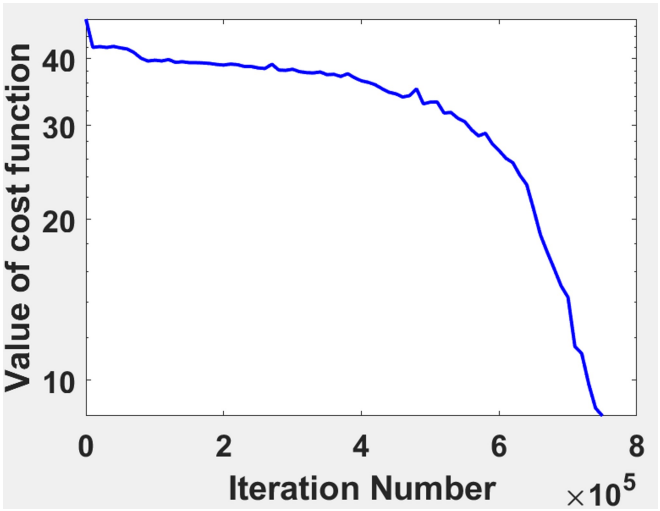
```
eta = 0.05;
```

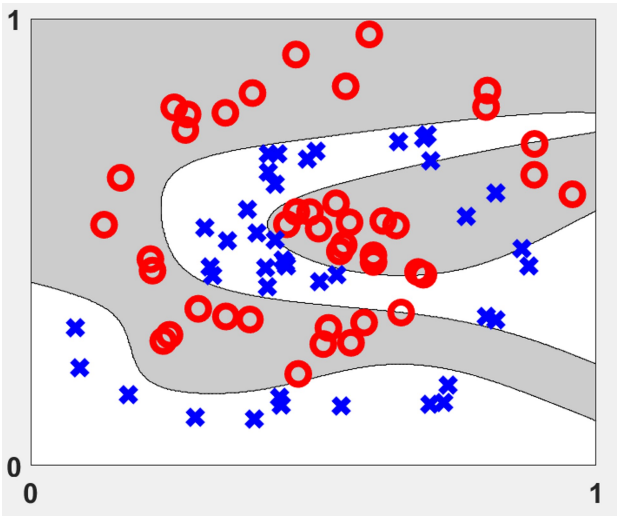
```
Niter = 1e6;
batches = 4; %%%%%%%%%
savecost = zeros(Niter,1); %%%%%%%%
accuracies = zeros(1, Niter); %%%%%%%%%
for counter = 1:Niter %%%%%%%%%%%
    for batch=1:batches
        k = randi(84);
        x = [x1(k); x2(k)];
        % Forward pass
        a2 = activate(x,W2,b2);
        a3 = activate(a2,W3,b3);
        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
        accuracies(counter) = accuracy;%%%%%%%%%%%%%%
        savecost(counter) = newcost;
        if (accuracy > 0.97)
            break
        end
    end
    if (accuracy > 0.97) %&& (accuracy < 1)           %%%%%%%%%%%%
        fprintf("*** break ***\n"); %%%%%%%%%%
        break                       %%%%%%%%%%%%
    end                             %%%%%%%%%%%%
end
%newcost = newcost   %%%%%%
%accuracy = accuracy %%%%%%
fprintf("Iterations: %i\n", counter); %%%%%%%%%
%accuracies = accuracies(1:counter); %%%%%%%%%%

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];
```

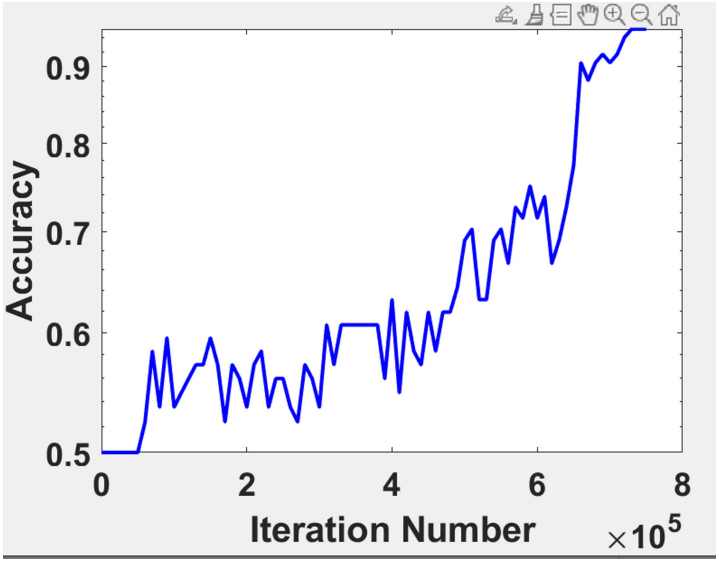
```
yvals = [0:Dy:1];
for k1 = 1:N+1
    xk = xvals(k1);
    for k2 = 1:N+1
        yk = yvals(k2);
        xy = [xk;yk];
        a2 = activate(xy,W2,b2);
        a3 = activate(a2,W3,b3);
        a4 = activate(a3,W4,b4);
        Aval(k2,k1) = a4(1);
        Bval(k2,k1) = a4(2);
     end
end
[X,Y] = meshgrid(xvals,yvals);

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(1:42),x2(1:42),'ro','MarkerSize',12,'LineWidth',4) %%%%%%%%
plot(x1(43:84),x2(43:84),'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

figure(4) %%%%%%%%%%%%%%%%%%
clf
semilogy([1:1e3:Niter],accuracies(1:1e3:Niter),'b-','LineWidth',2)
title('Iterations vs Accuracy (step size of 100)')
xlabel('Iteration Number')
ylabel('Accuracy')
set(gca,'FontWeight','Bold','FontSize',12)



toc

  function [costval, accuracy] = cost(W2,W3,W4,b2,b3,b4)

     costvec = zeros(84,1);
     total_pts = 84;                                    %%%%%%%
     class_pts = 0;                                     %%%%%%%%
     accuracy = 0;
     for i = 1:84
         x =[x1(i);x2(i)];
         a2 = activate(x,W2,b2);
```

```
            a3 = activate(a2,W3,b3);
            a4 = activate(a3,W4,b4);
            y1 = a4(1); y2 = a4(2);
            costvec(i) = norm(y(:,i) - a4,2);

            if (y1 > y2) && isequal(y(:,i),[1;0])          %%%%%%%%
                class_pts = class_pts + 1;                                  %%%%%%%%
            elseif (y1 < y2) && isequal(y(:,i),[0;1])      %%%%%%%%
                class_pts = class_pts + 1;                                 %%%%%%%%
            end                                            %%%%%%%%
            %total_pts = total_pts + 1;                     %%%%%%%%
        end
        costval = norm(costvec,2)^2;
        accuracy = class_pts/total_pts;                    %%%%%%%%

    end % of nested function

end
```

b)



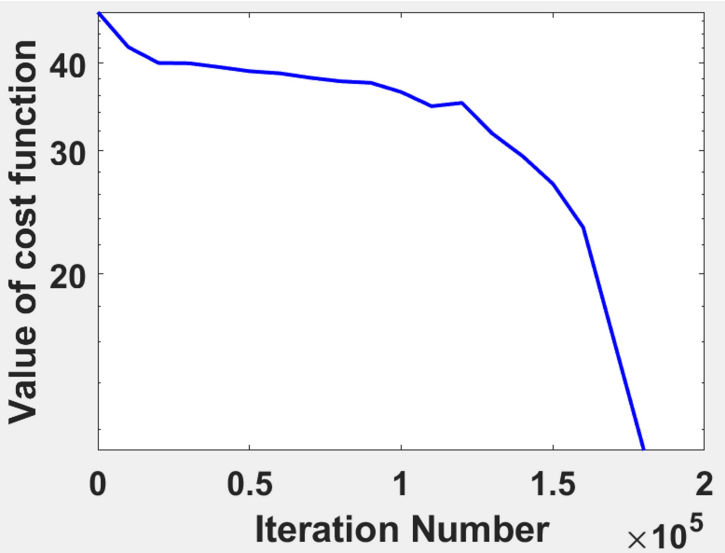



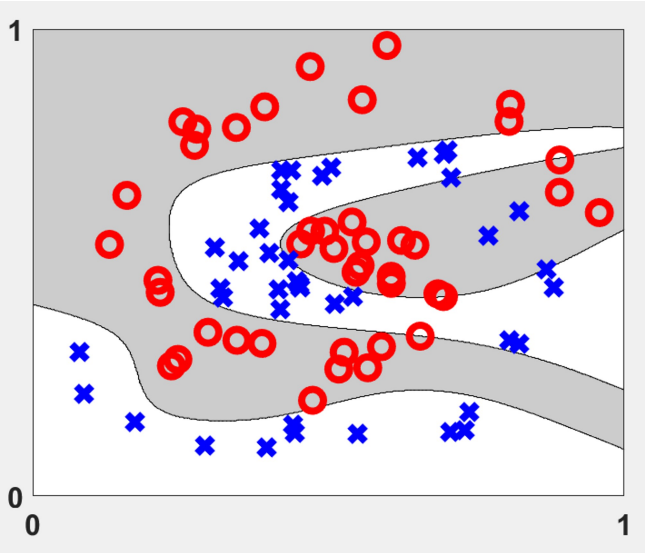



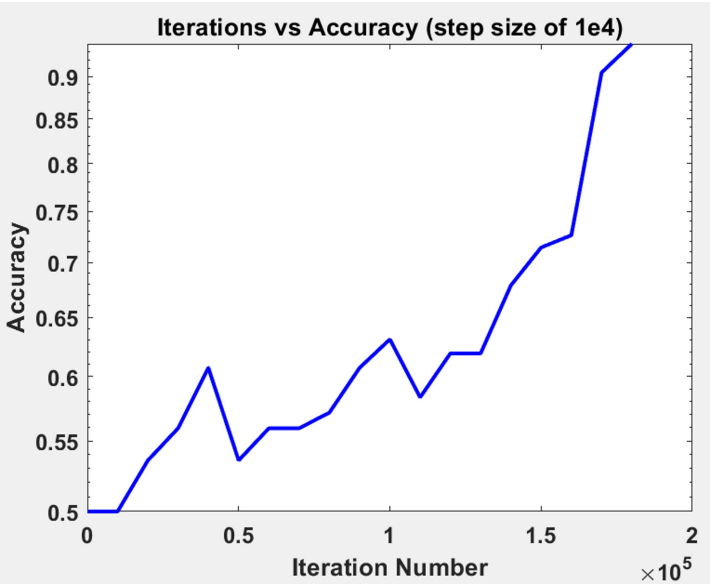

```
>> netbpfull
*** break ***

newcost =

    8.4609


accuracy =

    0.9762

Iterations: 751167
Elapsed time is 64.672652 seconds.
```

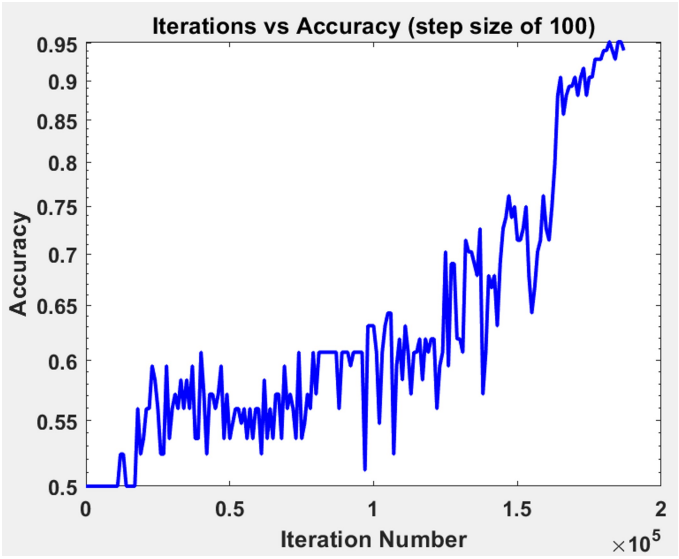
```
W2 = 0.5*randn(5,2);
W3 = 0.5*randn(3,5);
W4 = 0.5*randn(2,3);
b2 = 0.5*randn(5,1);
b3 = 0.5*randn(3,1);
b4 = 0.5*randn(2,1);
%%%%%%%%%%%%%%%%%%%%%%%%
```

c)



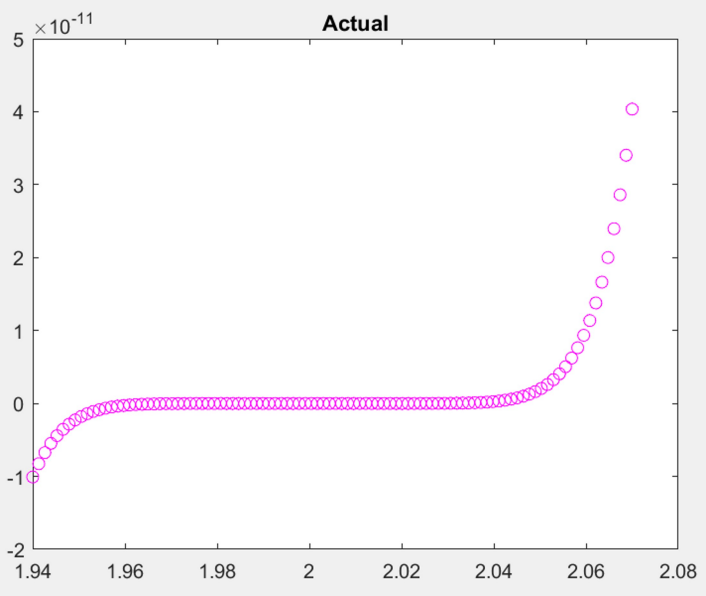



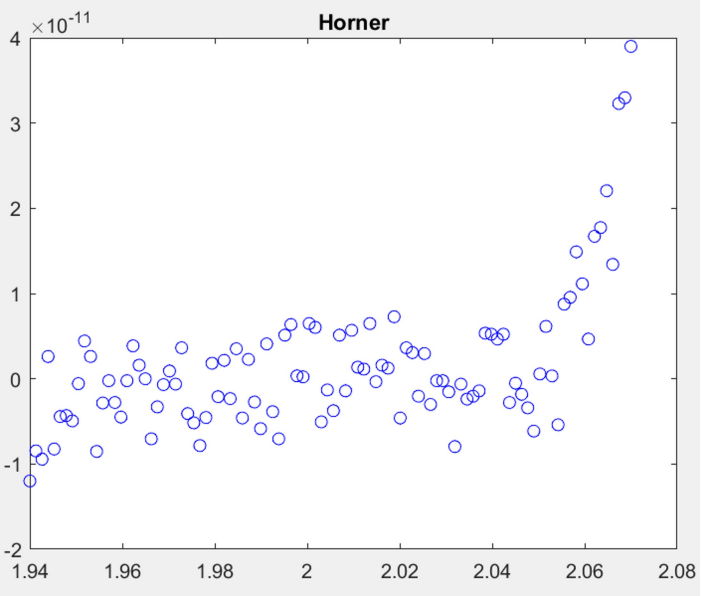

```
newcost =

    8.4609


accuracy =

    0.9762

Iterations: 187792
Elapsed time is 104.273810 seconds.
```

No, the training does not improve with a batch size of 4. Though the iterations of the outer loop (counter=1:Niter) have been reduced, the total iterations of both loops (4*187792) remains the same. Therefore it can be said that a batch size of 4 does not change the total iterations needed to achieve the threshold accuracy. Additionally, both processes arrive at the same final accuracy and newcost, however, the runtime for the batch size of 4 is much longer than the batch size of 1 (~65 seconds vs ~104 seconds).

2)

a)




As can be observed from the graphs above, the points acquired via horner fluctuate a lot more than the actual plotted function, indicating inaccuracies in the horner evaluation process. It is also worth noting that the error seems to be varying, as some points deviate more/less from the actual points than others. This is likely due to floating point error during the addition of two numbers with different exponents, which happens very often in the horner's evaluation. When adding and subtracting numbers of different exponents, bits from the mantissa of the smaller exponent will be discarded (cancellation).

b)

```
ans =

    2.0160
```

Because of the large inaccuracies present in the horner's evaluation, the resulting r-value is not accurate within 10^-6 of the actual root (2). So, while the bisection method is accurate within a tolerance of 10^-6 for the root of the horner's evaluation, the horner's evaluation of (x-2)^9 is NOT necessarily accurate in estimating the root of (x-2)^9 within a tolerance of 10^-6.

3)

There are a few scenarios that may cause newton's method not to converge or to converge to a point other than a solution. If the initial guess is very far from the actual root then newton's method may fail to converge. Newton's method will also diverge if the derivative of the given function evaluated at the initial guess is zero or close to zero.

a)

```
*** Question A ***
Newton's method implementation: x1 = 5, x2 = 4
Number of iterations: 42
Fsolve implementation: x1 = 1.141278e+01, x2 = -8.968053e-01
```

b)

```
*** Question B ***
Newton's method implementation: x1 = 1.666667e+00, x2 = -6.666667e-01, x3 = 1.333333e+00
Number of iterations: 56
Fsolve implementation: x1 = 1.000000e+00, x2 = 1.338358e-09, x3 = 2.000000e+00
```

c)

```
*** Question C ***
Newton's method implementation: x1 = NaN, x2 = NaN, x3 = NaN, x4 = NaN
Number of iterations: 0
Fsolve implementation: x1 = -2.672986e-03, x2 = 2.672986e-04, x3 = 4.073372e-04, x4 = 4.073372e-04
```

In this case, the some of the derivatives of each expression in the system were close to or equal to zero, which may have caused divergence as described above.

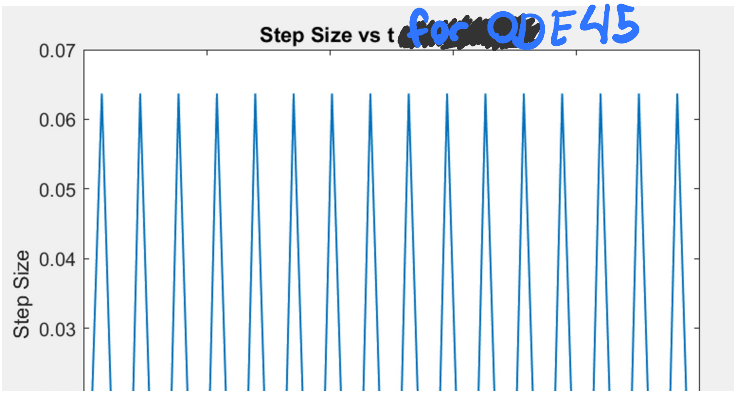
d)

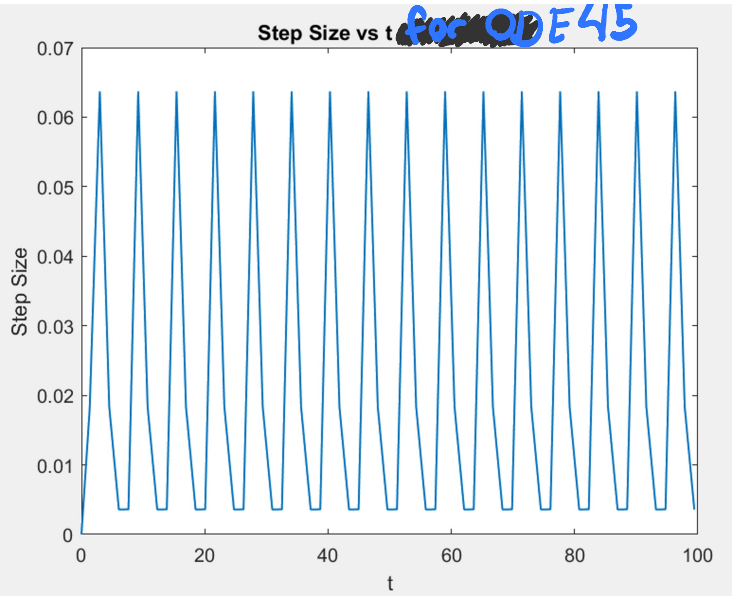
```
*** Question D ***
Newton's method implementation: x1 = 0, x2 = NaN
Number of iterations: 1
Fsolve implementation: x1 = 0, x2 = -3.161825e-04
```

In this case, the first equation (x1 = 0) could be causing problems during newton's method. Additionally, the derivative of the system evaluated at the initial guess is exactly equal to zero.

5) ODE 45

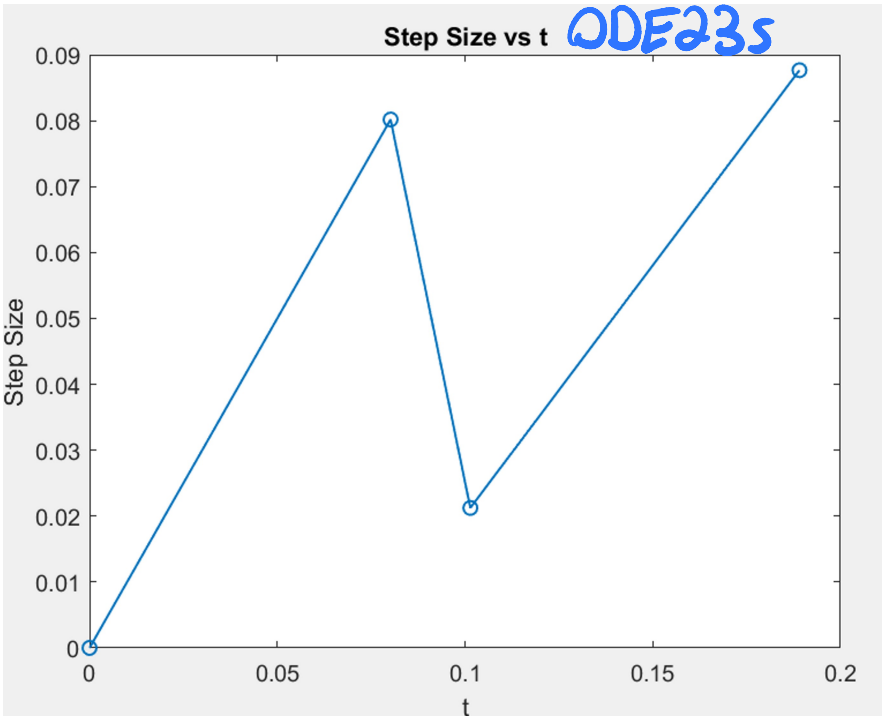
| Tol | # of Evals | error @ t=100 | avg step sz |
|---|---|---|---|
| 1.0e+04 * | | | |
| 0.000010000000000 | 2.415700000000000 | 0.000097881576292 | 0.000000413958687 |
| 0.000001000000000 | 2.415700000000000 | 0.000097881576292 | 0.000000413958687 |
| 0.000000100000000 | 2.415700000000000 | 0.000097881576292 | 0.000000413958687 |
| 0.000000010000000 | 2.415400000000000 | 0.000097881576292 | 0.000000414010102 |
| 0.000000001000000 | 2.413600000000000 | 0.000097881576292 | 0.000000414318860 |
| 0.000000000100000 | 2.332900000000000 | 0.000097881576292 | 0.000000428651035 |
| 0.000000000010000 | 0.644100000000000 | 0.000097881576292 | 0.000001552553951 |
| 0.000000000001000 | 0.000600000000000 | 0.000097881576292 | 0.000018364846957 |
| 0.000000000000100 | 0.000200000000000 | 0.000097881576292 | 0.000019397930468 |
| 0.000000000000010 | 0.000200000000000 | 0.000097881576292 | 0.000019397930468 |
| 0.000000000000001 | 0.000100000000000 | 0.000097881576292 | 0 |
| 0.000000000000000 | 0.000100000000000 | 0.000097881576292 | 0 |






# ODE 23s

| Tol | # of Evals | error @ t=83 | avg step sz |
|---|---|---|---|
| 0.100000000000000 | 83.000000000000000 | 0.000001596522745 | 1.204819277108434 |
| 0.010000000000000 | 83.000000000000000 | 0.000001596522745 | 1.204819277108434 |
| 0.001000000000000 | 83.000000000000000 | 0.000001596522745 | 1.204819277108434 |
| 0.000100000000000 | 46.000000000000000 | 0.000001596522745 | 2.173913043478261 |
| 0.000010000000000 | 23.000000000000000 | 0.000001596522745 | 4.347826086956522 |
| 0.000001000000000 | 7.000000000000000 | 0.000001596522745 | 0.027010119069732 |
| 0.000000100000000 | 4.000000000000000 | 0.000001596522745 | 0.047267708372030 |
| 0.000000010000000 | 2.000000000000000 | 0.000001596522745 | 0.050708133621518 |
| 0.000000001000000 | 1.000000000000000 | 0.000001596522745 | 0 |
| 0.000000000100000 | 1.000000000000000 | 0.000001596522745 | 0 |
| 0.000000000010000 | 1.000000000000000 | 0.000001596522745 | 0 |
| 0.000000000001000 | 1.000000000000000 | 0.000001596522745 | 0 |



ODE45

- Step size periodically changes as t increases
- Average step size remains the same throughout
- Step size peaks at ~0.065

ODE23s

- Step size drastically and suddenly fluctuates as t increases
- Step size is increasing on average, line of best fit for the graph would be sloped upwards
- At tol=10^-7 only 4 steps are taken, however the steps are increasing in magnitude over time

I think that ODE23s is more efficient for solving an ODE given a relatively large tolerance, since it solves the system in significantly less evaluations (i.e. for tol = 10^-1, ODE45 takes 24157 evaluations, while ODE23s only requires 83 evaluations), and also results in much smaller error (~1.6e-6 for ODE23s, versus 9.8e-1 for ODE45).

However, when observing the results for smaller tolerances such as 10^-8, the errors and step sizes are still in the favour of ODE23s, but the # of evaluations are similar between the two methods.

Finally, for tolerances of 10^-9 and smaller, ODE45 is proven to be more effective, since floating point error occurs in ODE23s causing step size to become zero and only one evaluation to be made: indicating that a solution has not been found.

It is worth noting that this same problem occurs for ODE45 for tolerances of 10^-11 and smaller.