Problem 1:

Netbp2.m:

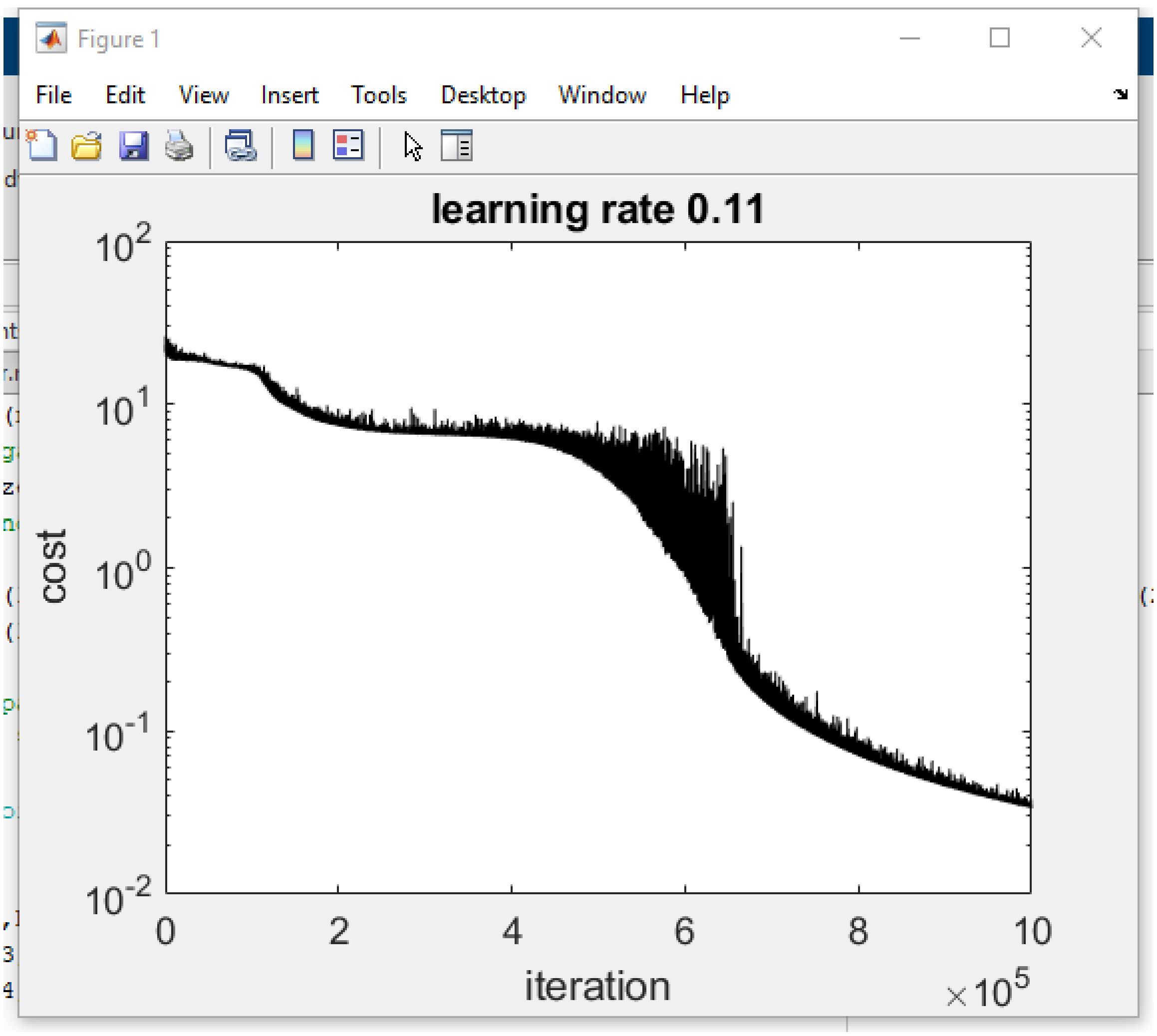
```
function cost = netbp2(neurons, data, labels, niter, lr, file)
%NETBP  Uses backpropagation to train a network
[~,num_of_points] = size(data);
% Initialize weights and biases
rng(5000);
W2 = 0.5*randn(neurons(1),2); W3 = 0.5*randn(neurons(2),neurons(1)); W4 = 0.5*randn(2,neurons(2));
b2 = 0.5*randn(neurons(1),1); b3 = 0.5*randn(neurons(2),1); b4 = 0.5*randn(2,1);

% Forward and Back propagate
cost = zeros(niter,1); % value of cost function at each iteration
for counter = 1:niter
    k = randi(num_of_points);         % choose a training point at random
    x = data(:,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-labels(:,k));
    delta3 = a3.*(1-a3).*(W4'*delta4);
    delta2 = a2.*(1-a2).*(W3'*delta3);
    % Gradient step
    W2 = W2 - lr*delta2*x';
    W3 = W3 - lr*delta3*a2';
    W4 = W4 - lr*delta4*a3';
    b2 = b2 - lr*delta2;
    b3 = b3 - lr*delta3;
    b4 = b4 - lr*delta4;
    % Monitor progress
    newcost = cost_function(W2,W3,W4,b2,b3,b4) ;  % display cost to screen
    cost(counter) = newcost;
    fprintf("i=%d  %e\n", counter, newcost);
end
```

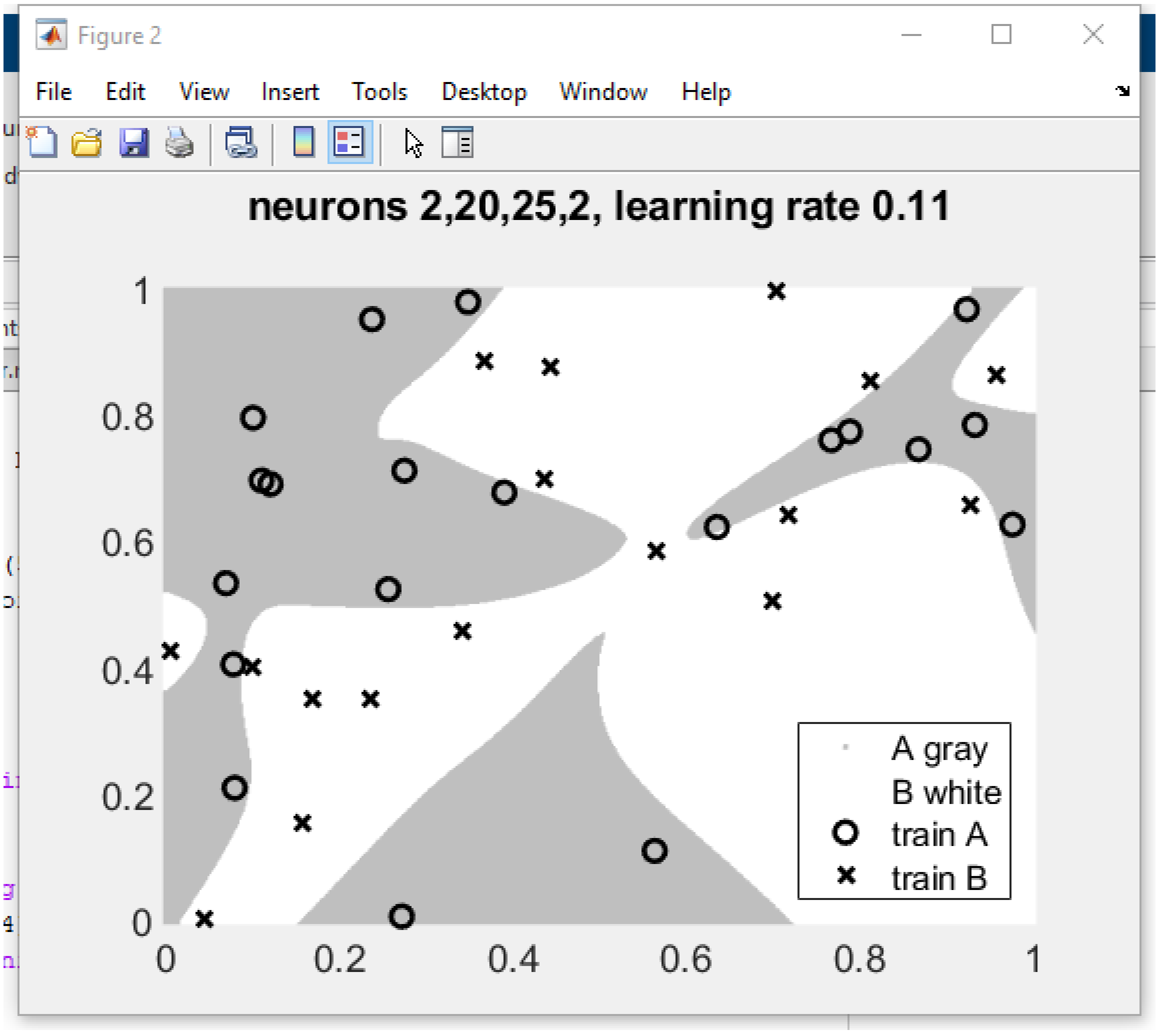
```
end
% Show decay of cost function
save costvec
semilogy([1:1e4:niter],cost(1:1e4:niter))

  function costval = cost_function(W2,W3,W4,b2,b3,b4)
     costvec = zeros(num_of_points,1);
     for i = 1:num_of_points
         x = data(:,i);
         a2 = activate(x,W2,b2);
         a3 = activate(a2,W3,b3);
         a4 = activate(a3,W4,b4);
         costvec(i) = norm(labels(:,i) - a4,2);
     end
     costval = norm(costvec,2)^2;
  end % of nested function
save(file, 'W2','W3','W4','b2','b3','b4');
end
```

Classifypoints.m:

```
function category = classifypoints(file, points)
    load(file);
    [~,num_of_points] = size(points);
    category = zeros(1, num_of_points);
    for i = 1:num_of_points
        x = points(:,i);
        a2 = activate(x,W2,b2);
        a3 = activate(a2,W3,b3);
        a4 = activate(a3,W4,b4);
        if a4(1) >= a4(2)
            category(i) = 1;
        end
    end
end
```

Two plots:





Params:

```
neurons = [20 25];
learning_rate = 0.11;
niter = 1e6;
```

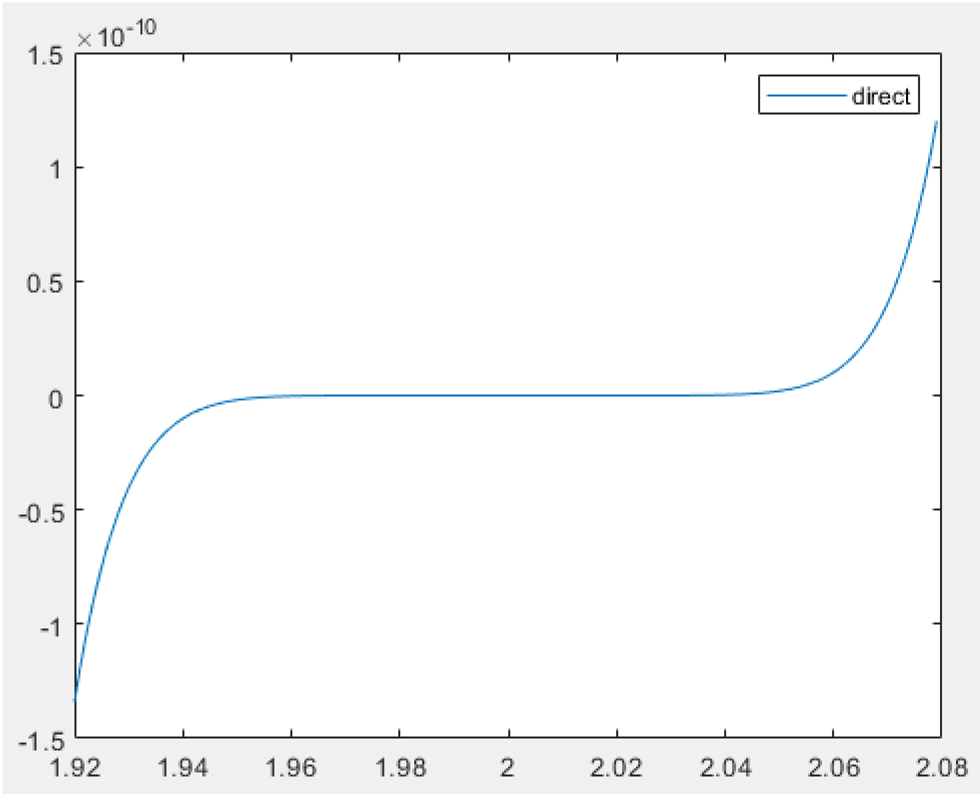
What I found:

1. More neurons will reduce the speed of training.
2. More iterations do not necessarily lead to a better learning and classification outcome.
3. Minimizing the cost will lead to a better learning outcome.
4. We need enough number of iterations to ensure the neural network can minimize the cost function.

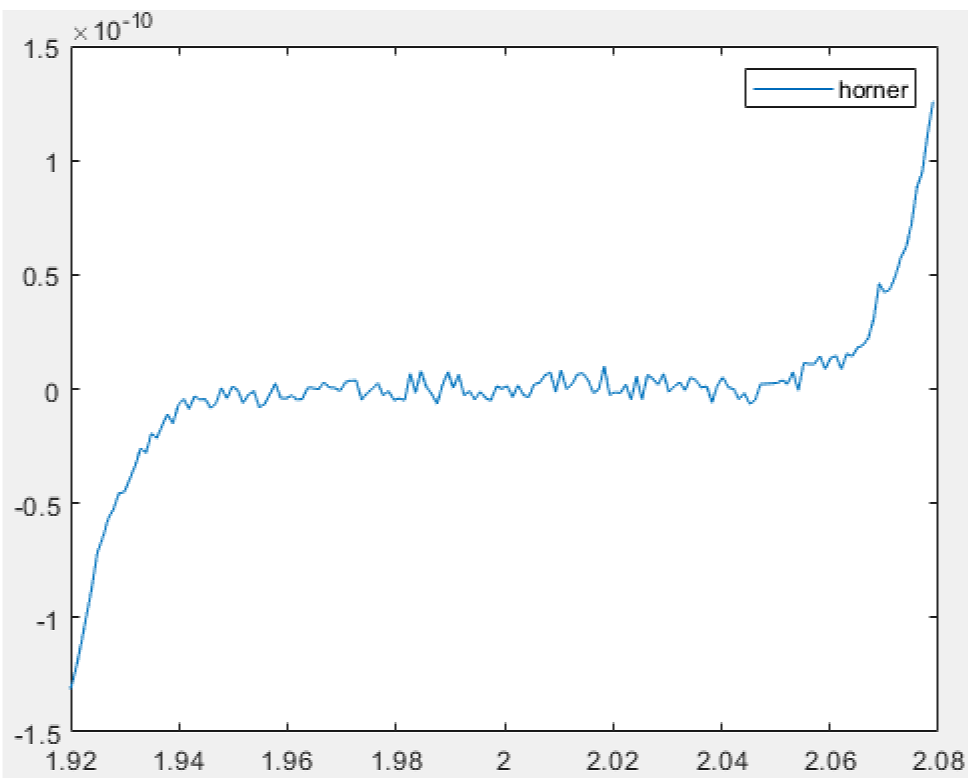
Problem 2:

a)

Direct evaluation:



Horner's method:



We can easily tell that Horner's method appeared to have more visible errors in the plot. This is because we expanded our function and thus introduced more calculations in the evaluation process. Therefore, the errors were larger.

So, when the function is not transformable into a less complicated expression, Horner's method has a better performance because it involves the least amount of calculation possible. However, if the whole expression can be transformed into a simpler one, the simpler expression outperforms Horner's method.

b) Applying bisection to (x-2)^9 ended up returning the result 1.999999 which is within 1e-6 error with the correct result. Applying bisection to horner's ended up returning the result 1.964358 which isn't within the error tolerance.

c) Because the accuracy of horner's method is limited. As you can see from the plot, horner's evaluation's plot was jagged, although the overall shape is very close. Because of this jagged nature of our data, bisection is determining the root based on the signs of two function evaluations, which is obviously very sensitive to the Jagged data that horner's evaluation had.

For example, f(a) and f(b) may actually have the same sign, but in horner's plot because of the inaccuracy of our data, they might even have the same sign and lead the algorithm to proceed with another interval of a and b that doesn't actually contain the correct root.

d)

```
f_fsolve =

    1.9000


Equation solved at initial point.

fsolve completed because the vector of function values at the initial point
is near zero as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>

expanded_f_fsolve =

    1.9000
```

Both terminated at 1.9 as solution.

Problem 3:

a)

```
No solution found.

fsolve stopped because the last step was ineffective. However, the vector of function
values is not near zero, as measured by the value of the function tolerance.

<stopping criteria details>
system a) fsolve result: [1.141278e+01, -8.968053e-01]
system a) my implementation result: [5.000000e+00, 4.000000e+00]
system a) fsolve num of iterations: 39
system a) my implementation result: 42
```

Because the system is in newton's form and involve more calculations and potential errors.

b)

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  4.344298e-18.
> In Newtown_system_solver (line 15)
In main_newton (line 19)

system b) fsolve result: [1.000000e+00, 1.338358e-09, 2.000000e+00]
system b) my implementation result: [1.666667e+00, -6.666669e-01, 1.333333e+00]
system b) fsolve num of iterations: 6
system b) my implementation result: 56
```

Results are different because the system might have multiple solutions or the fsolve only ends at a root that is close enough to 0 but not actually zero mathematically.

c)

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
system c) fsolve result: [2.036462e-03, -2.036368e-04, 2.036457e-03, 2.036462e-03]
system c) my implementation result: [-8.877841e-04, 8.877841e-05, -8.877841e-04, -8.877841e-04]
system c) fsolve num of iterations: 2
system c) my implementation result: 10
```

Results are different because the system might have multiple solutions or the fsolve only ends at a root that is close enough to 0 but not actually zero mathematically.

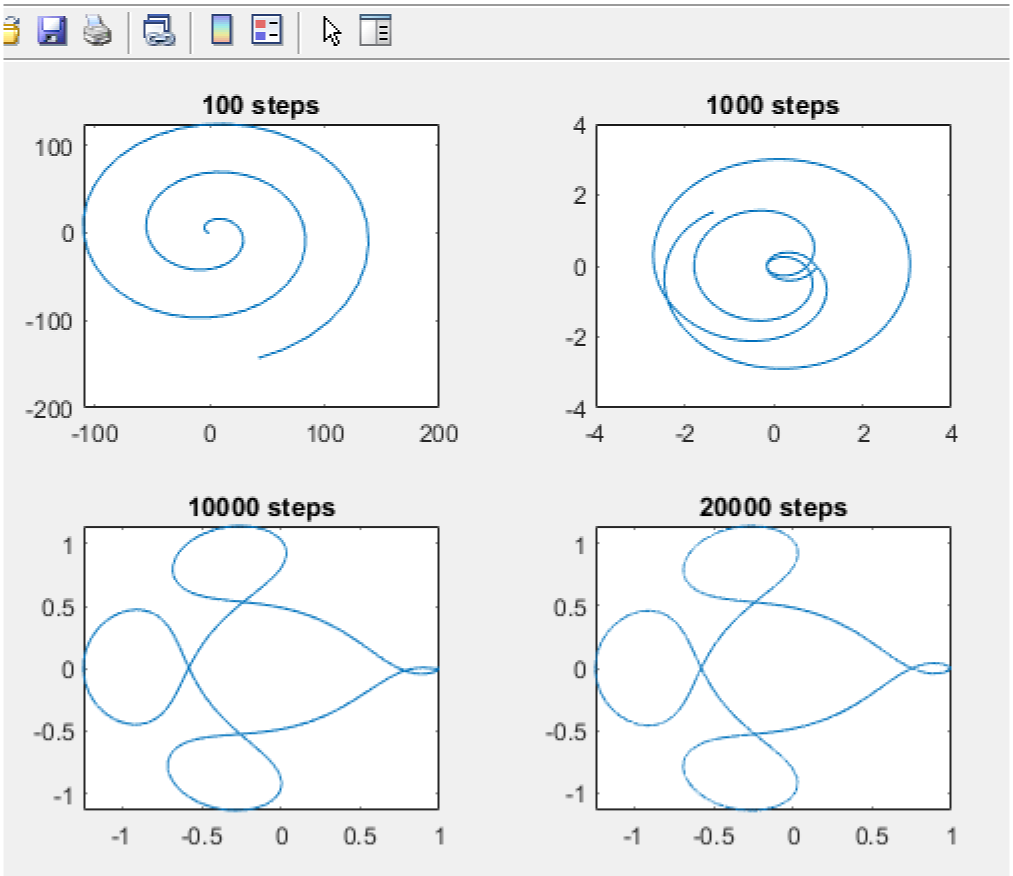
d)

```
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>
Warning: Matrix is singular to working precision.
> In Newtown_system_solver (line 15)
In main_newton (line 41)

Warning: Matrix is singular to working precision.
> In Newtown_system_solver (line 15)
In main_newton (line 41)

system d) fsolve result: [1.004816e-02, 1.004816e-02]
system d) my implementation result: [NaN, NaN]
system d) fsolve num of iterations: 11
system d) my implementation result: 2
```

Reason that it is not working is the matrix itself is singular to working precision.

Problem 4:

About 10000 steps and more are needed.



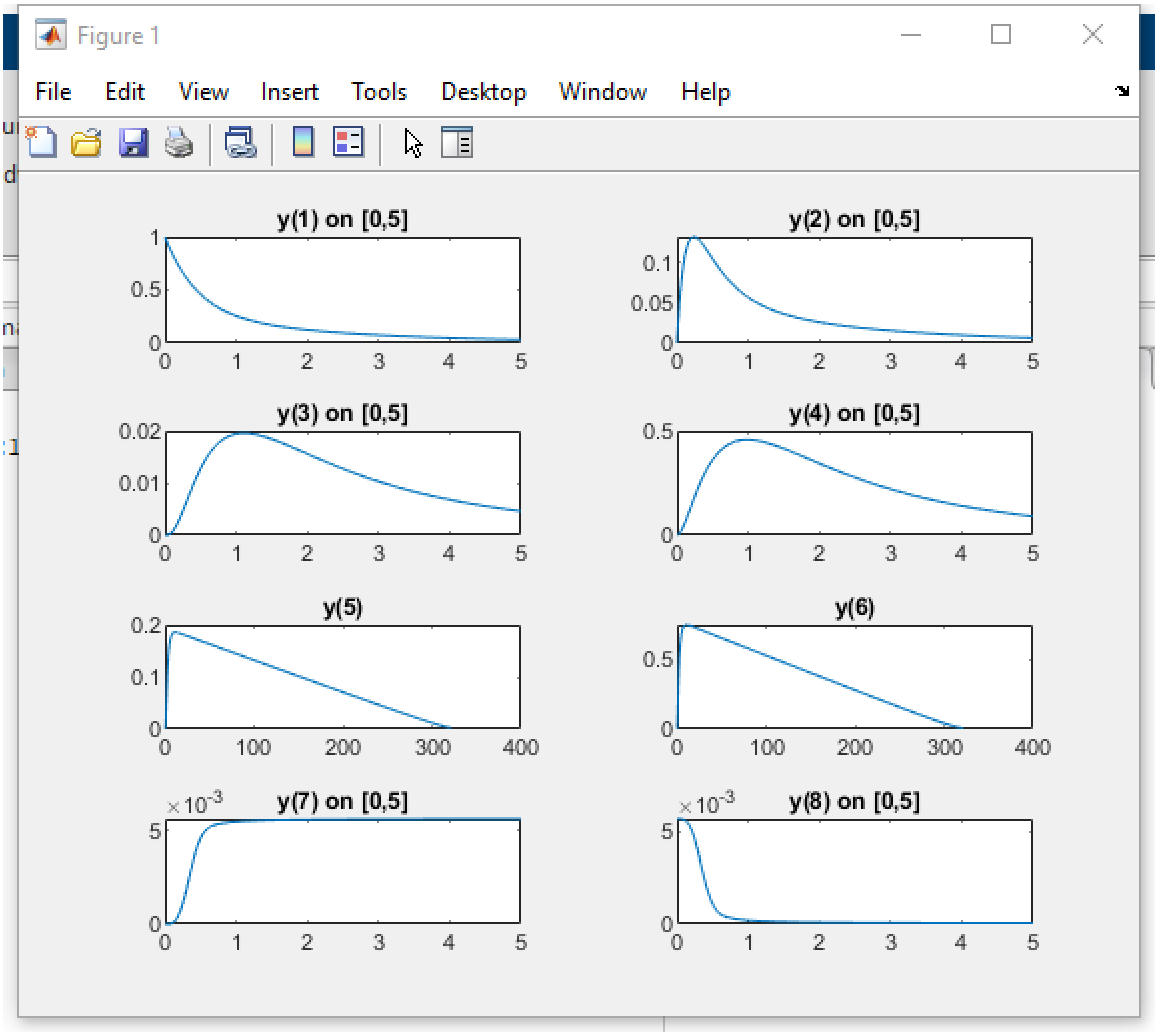
Problem 5:

Table

		number of		
solver	CPU time	steps	failed steps	function evaluations
ode23	1.6330e+01	1198181	0	3594544
ode45	6.0485e-01	43825	2	262963
ode23s	1.4540e+01	153152	708	1073488
ode15s	1.0300e+00	50169	843	66811
ode113	8.9711e-01	25628	378	51635

Ode45 most efficient.

Problem 6:



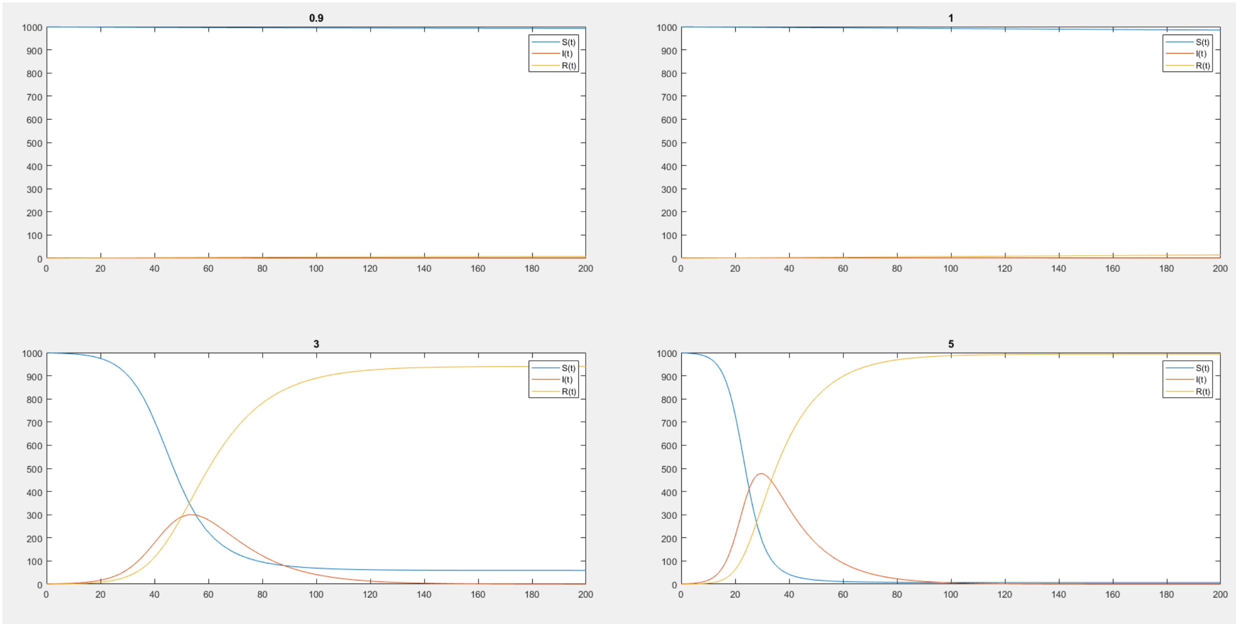

```
>> main_bonus
                                                         number of
                    --------------------------------------------------------------------------------
solver   CPU time    steps   failed steps   function evaluations   LU decompositions   nonlinear solves
ode23s   8.1774e-02  303          0                3336                  303                  909
ode15s   2.0482e-01  200         22                 435                   55                  370
ode45    2.6702e-01  10381      641               66133
```

Conclusion: When solving stiff problems, stiff solvers like ode23s and ode15s are much faster than solvers like ode45. We should always choose the most suitable ode solvers depending on the problem that we are dealing with.

Problem 7:

a) Words: Interpolate the functions for x,y,z respectively using the x,y,z values read from the data file. Then find the first roots of the interpolated functions respectively as periods for x,y,z. To determine the period, take the largest period among the x,y,z periods. If either of xyz doesn't have another root, then return -1 as period.

Problem 8:


0.9
1
3
5
S(t)
I(t)
R(t)