Assignment 4

tranj52, Jame Tran, 400144141

Question 1

a)

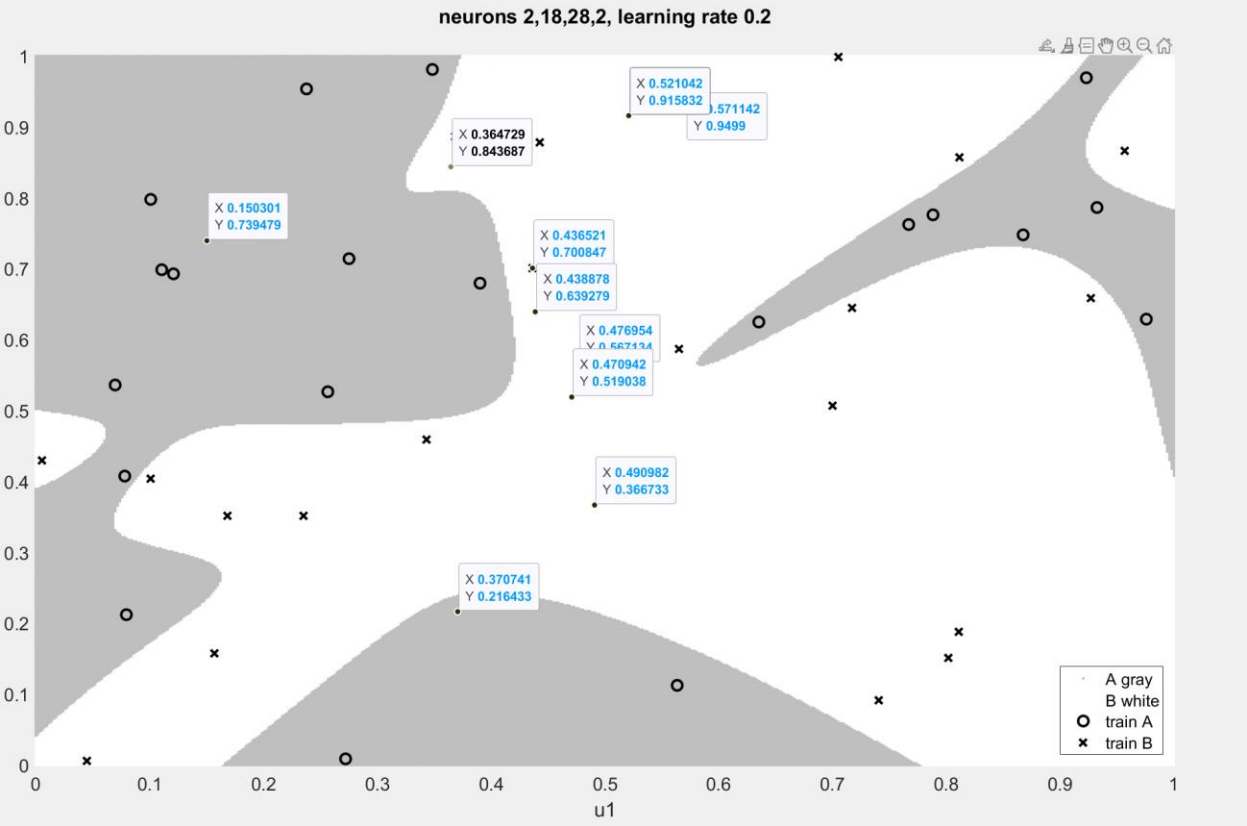
```
function cost = netbp2(neurons, data, labels, Niter,  lr, file)

%NETBP  Uses backpropagation to train a network

%%%%%%% DATA %%%%%%%%%%
%y = [ones(1,5) zeros(1,5); zeros(1,5) ones(1,5)]
x1 = data(1, :); %all data in first row is x
x2 = data(2, :); %all data in second row is y
numcol = size(x2,2);


w2_layer_num = neurons(1);
w3_layer_num = neurons(2);

% Initialize weights and biases
rng(5000);
W2 = 0.5*randn(w2_layer_num,2); W3 = 0.5*randn(w3_layer_num,w2_layer_num); W4 =
0.5*randn(2,w3_layer_num);
b2 = 0.5*randn(w2_layer_num,1); b3 = 0.5*randn(w3_layer_num,1); b4 = 0.5*randn(2,1);

% Forward and Back propagate
%lr = 0.05;                % learning rate
%Niter = 1e5;               % number of SG iterations
savecost = zeros(Niter,1); % value of cost function at each iteration
for counter = 1:Niter
    k = randi(numcol);         % choose a training point at random
    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-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 = new_cost(W2,W3,W4,b2,b3,b4) ;  % display cost to screen
```

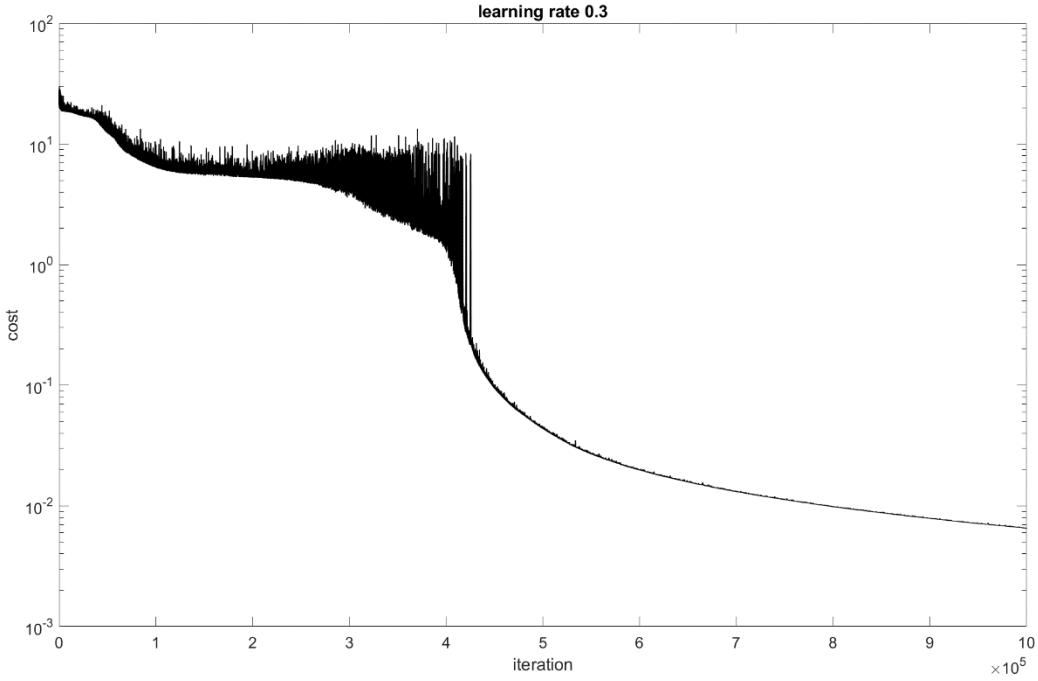
```
    cost(counter) = newcost;
    fprintf("i=%d  %e\n", counter, newcost);
end

% Show decay of cost function
save costvec
semilogy([1:1e4:Niter],savecost(1:1e4:Niter))

    function costval = new_cost(W2,W3,W4,b2,b3,b4)
     costvec = zeros(numcol,1);
     for i = 1:length(data)
         x =[x1(i);x2(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

function category = classifypoints(file, points)
    load(file);
    for i = 1:size(points,2)
        a2 = activate(points(:, i),W2,b2);
        a3 = activate(a2,W3,b3);
        a4 = activate(a3,W4,b4);

        if a4(1) >= a4(2)
            category(i) = 1;
        else
            category(i) = 0;
        end
    end
end
```

b)

- The learning interval size seems to have an important impact on the classification. The initial value of 0.05 was too high, and I experimented with dropping it to 0.01. However, at that learning interval it took too long to run. A value of 0.02 worked well to produce a good classification in a reasonable amount of time
- The number of neurons in the hidden layers played a large part in not just the classification ability but also the runtime. This makes sense since the number of neurons in the hidden layer increases the dimensions of the matrix in W and B, and increases the computation complexity.
- The stochastic gradient descent method to me was very interesting. To me, the idea of randomly readjusting the weights based on a random point or a random subset of the points to maintain a good tradeoff between performance and accuracy

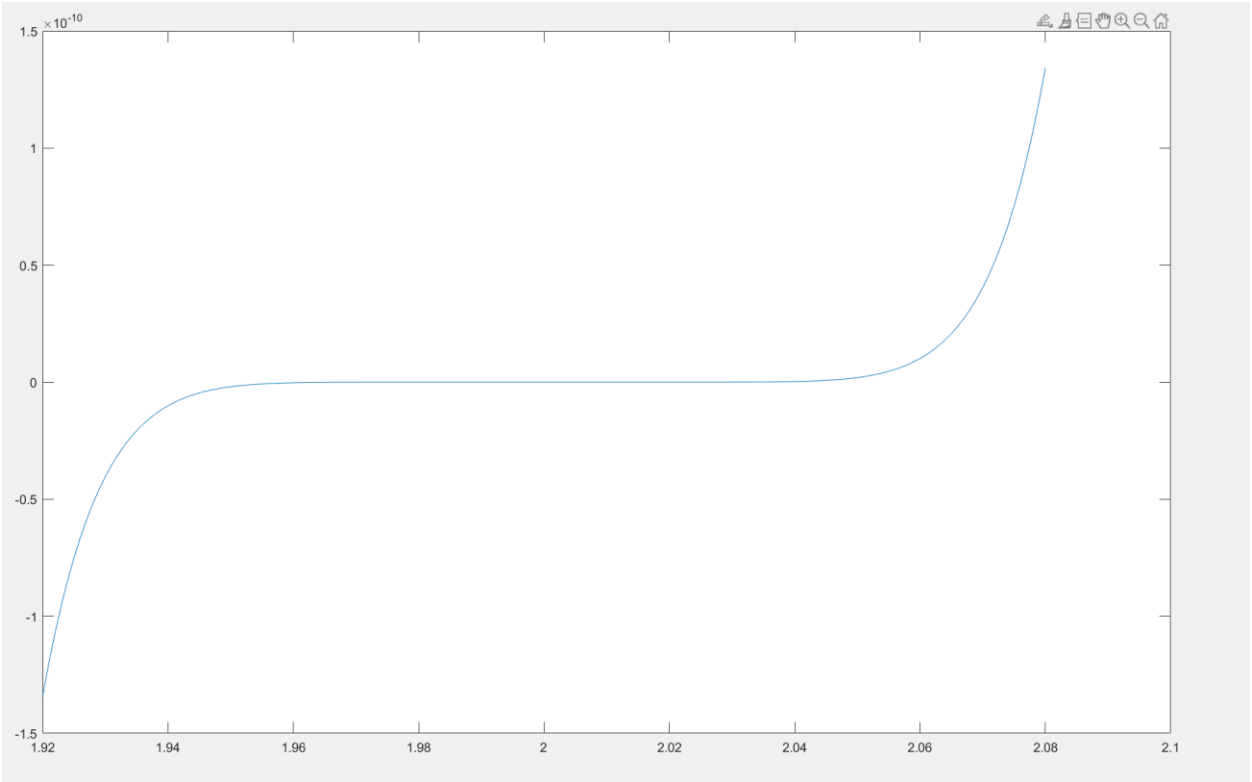
- It was very fun to learn that neural networks and the classification problem was an extension of the linear and logistic regression we learned before, but applied to classify datasets that are more randomly distributed. Giving a real world example (determining oil sites) made it a very applicable answer to our future careers

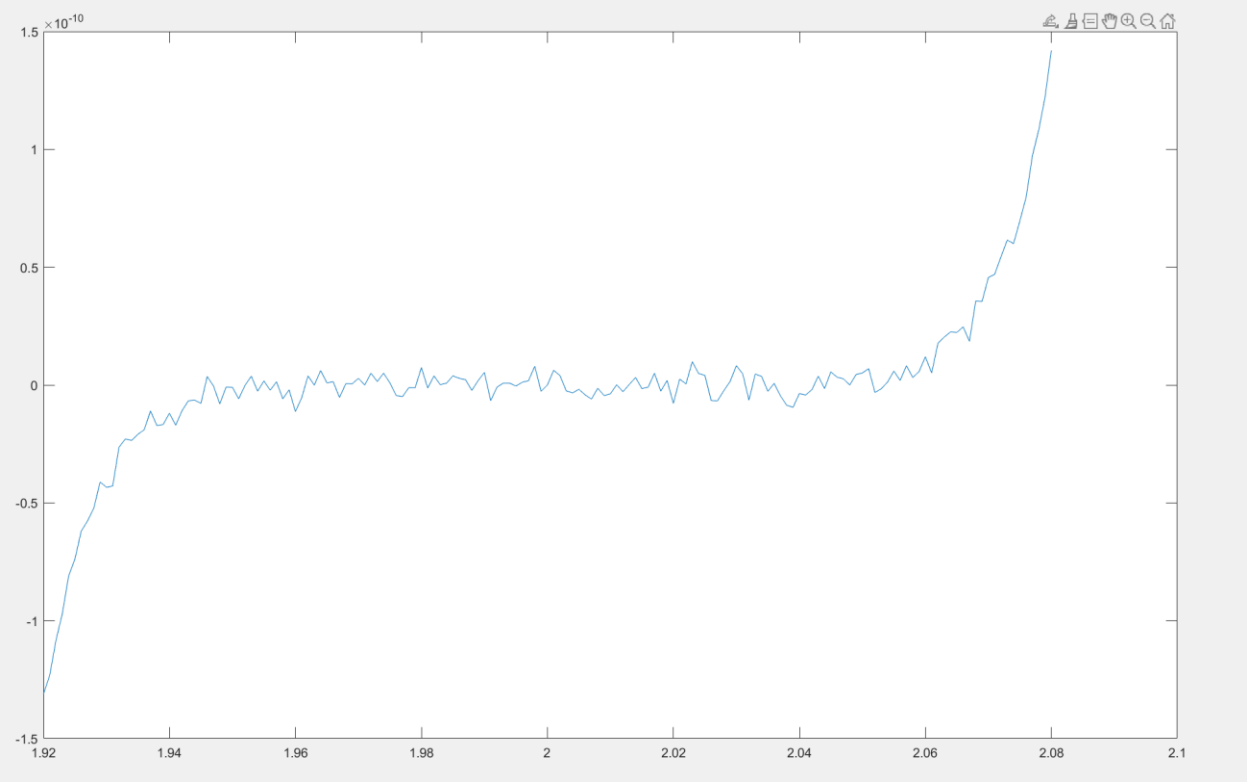




Question 2

a)





b) We cannot get a root that satisfies |r-2| < 10e-6 using Horner's Method and bisection method

c) This occurs because Horner's Method produces many cancellation errors when adding the terms between its evaluations. Direct evaluation on the other hand produces a much smoother result, because there is only one cancellation error that are not scaled by large coefficients.

d) Fsolve to both functions with an initial value of 1.9 gives back 1.9

Question 3

```
parta_x =

     5     4


parta_count =

    43


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>

parta_res =

   11.4128   -0.8968
```

```
partb_x =

    1.6667   -0.6667    1.3333


partb_count =

    57


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>

partb_res =

    1.0000    0.0000    2.0000
```

```
partc_x =

   NaN   NaN   NaN   NaN


partc_count =

     2


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>

partc_res =

   -0.0027    0.0003    0.0004    0.0004
```

```
partd_x =

   NaN   NaN


partd_count =

     3


No solution found.

fsolve stopped because the relative size of the current step is less than the
value of the step size tolerance squared, but the vector of function values
is not near zero as measured by the value of the function tolerance.

<stopping criteria details>

partd_res =

   22.8321    0.0000
```

We get inaccurate results in some of these systems (such as b, c and d) since the Jacobian matrix is singular, which gets inaccurate results.

Question 4

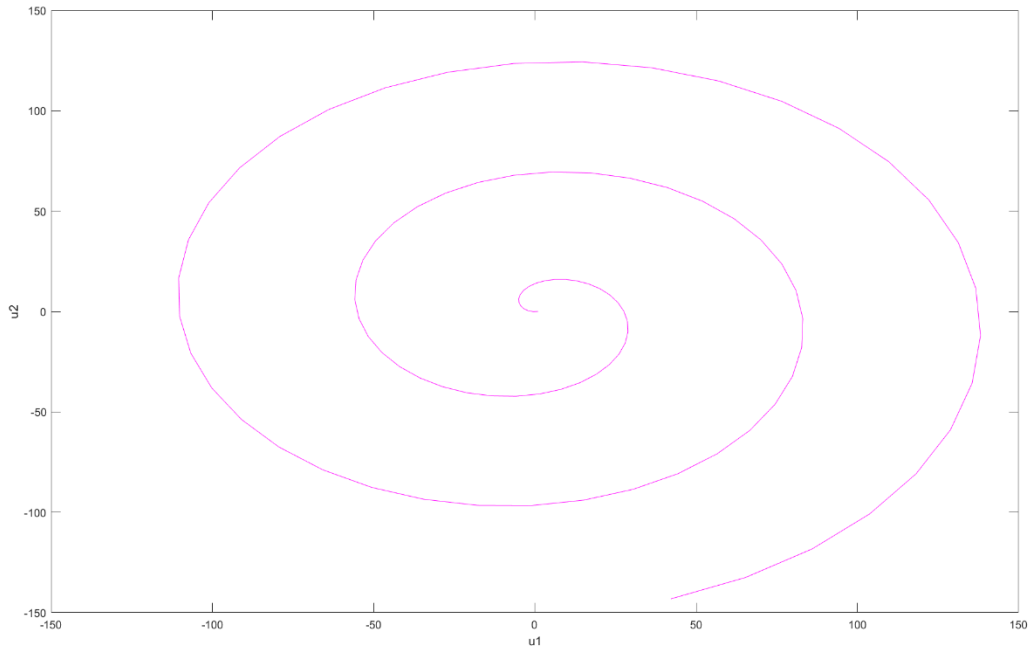


Figure 1: n = 100

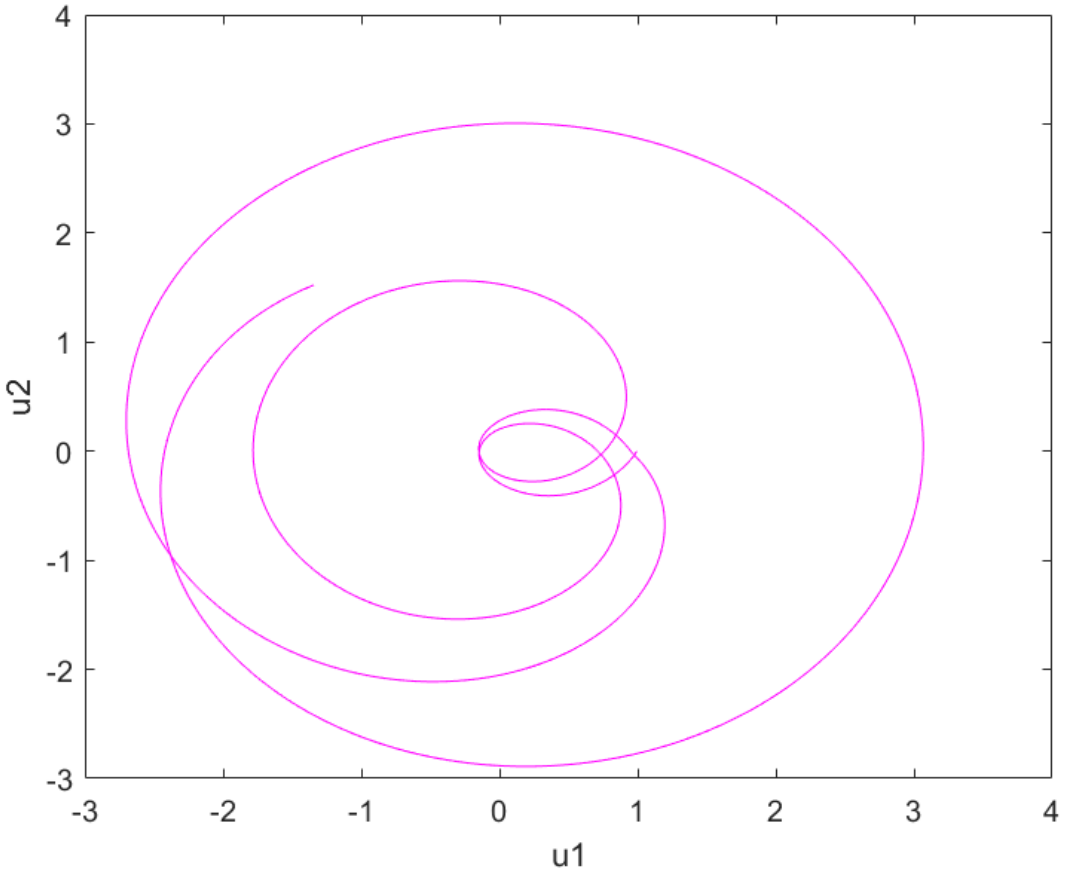


Figure 2: n=1000

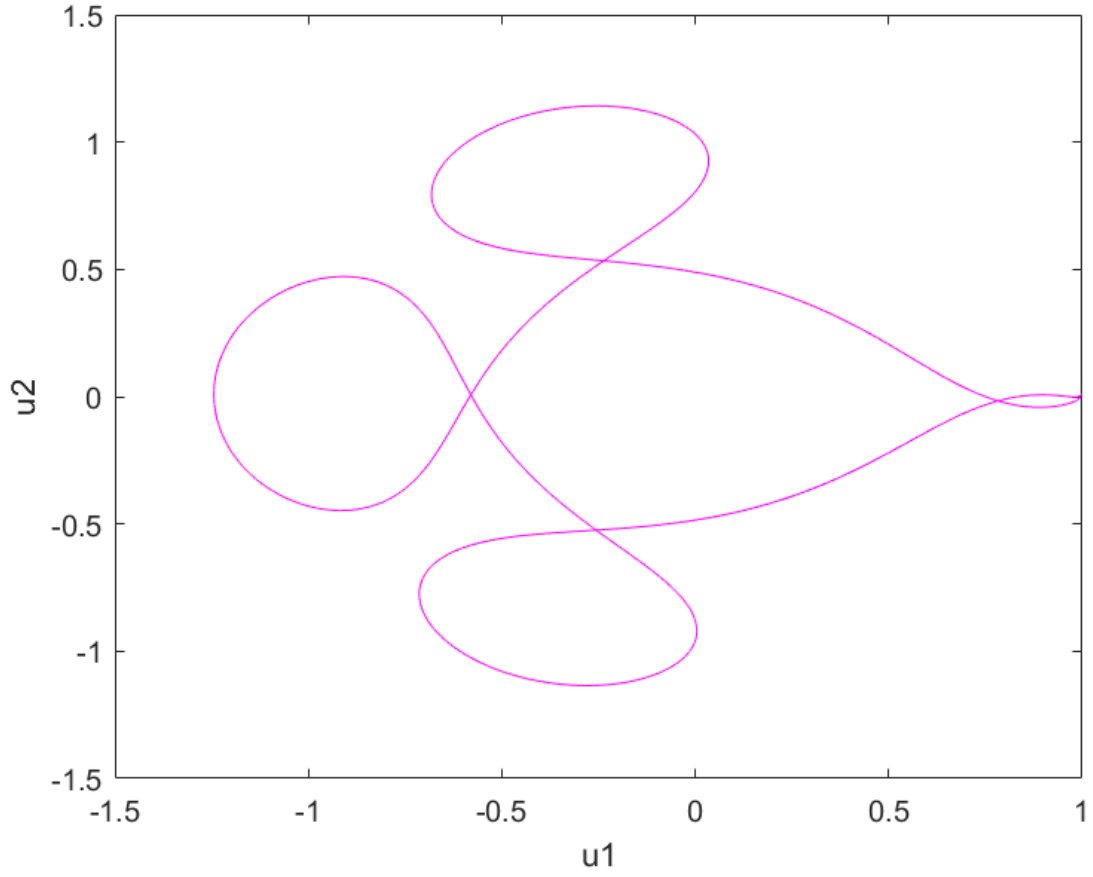


Figure 3: n = 10000

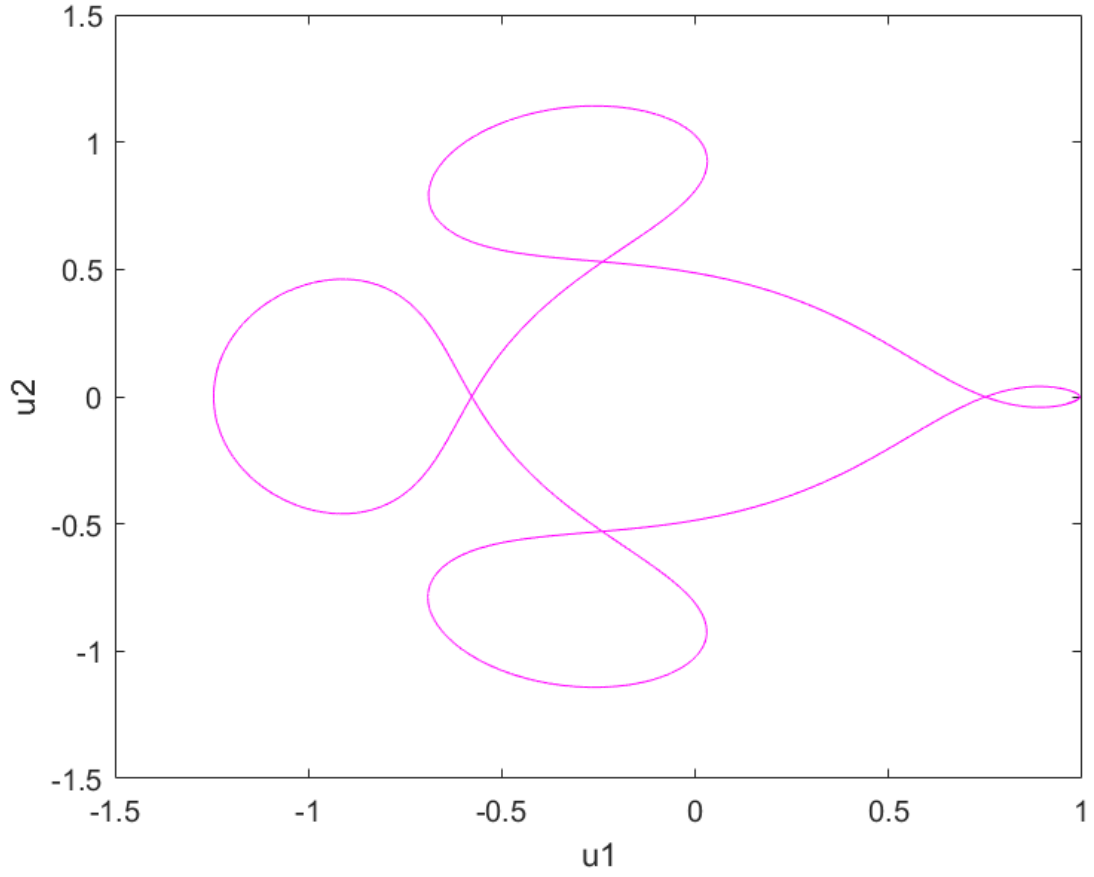


Figure 4: n = 20000

We get a result that looks qualitatively correct at n= 10000

Question 5

solver	CPU	steps	failed_steps	function_evaluation
"ode23"	13.526	1.2018e+06	0	3.6055e+06
"ode45"	0.5041	39006	0	2.3404e+05
"ode78"	0.29108	4462	66	76646
"ode89"	0.22185	3697	59	78522
"ode113"	0.58481	16479	161	33120

The most efficient ODE solve would be “ode89” since it has the lowest CPU time and the lowest number of steps.

Question 6

solver	CPU_Time	steps	failed_steps	function_eval	LU_decomposition	nonlinear_solves
"ode23s"	0.1479	303	0	3336	303	303
"ode15s"	0.012986	200	22	435	7	55
"ode45"	0.14557	10381	641	66133	0	0

`

We can see that ode15s is the most efficient. The stiff solvers perform much better on the problem because the HIRES problem is a stiff ODE problem. The function ode15s is better than ode23s, which is also stiff, in terms of function evals and CPU time and non linear solves.

Question 8

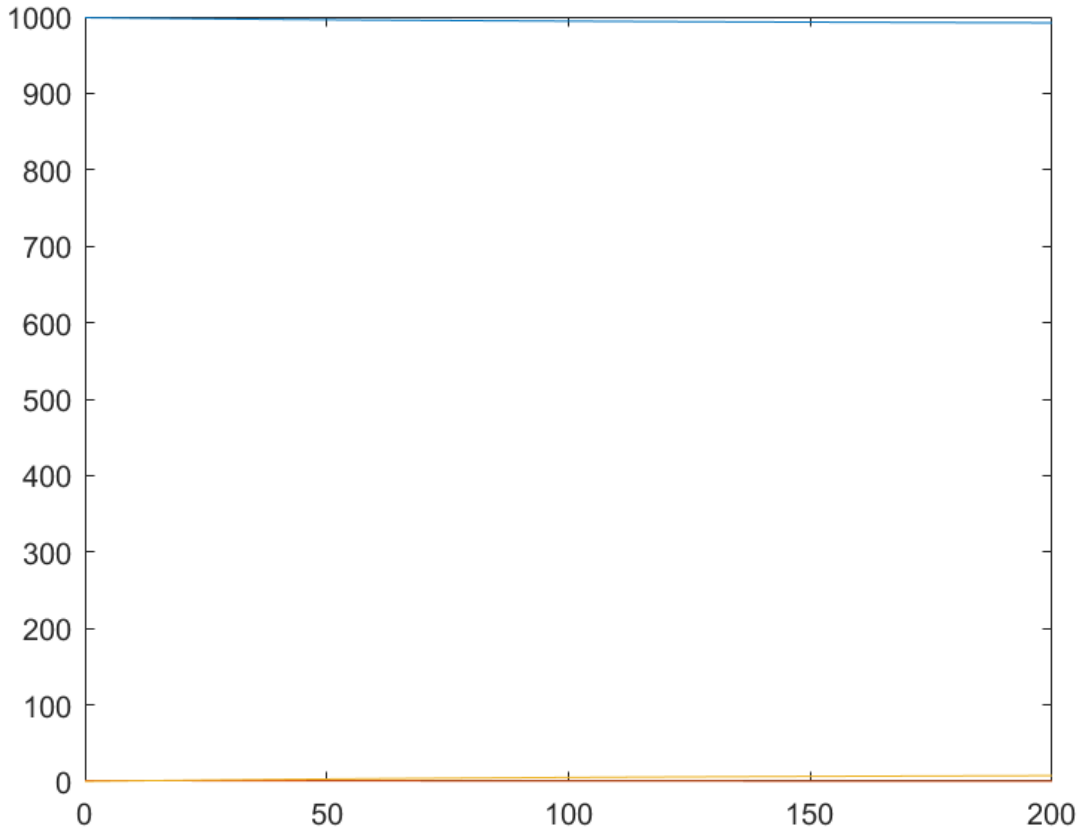


Figure 5 ODE 0.9

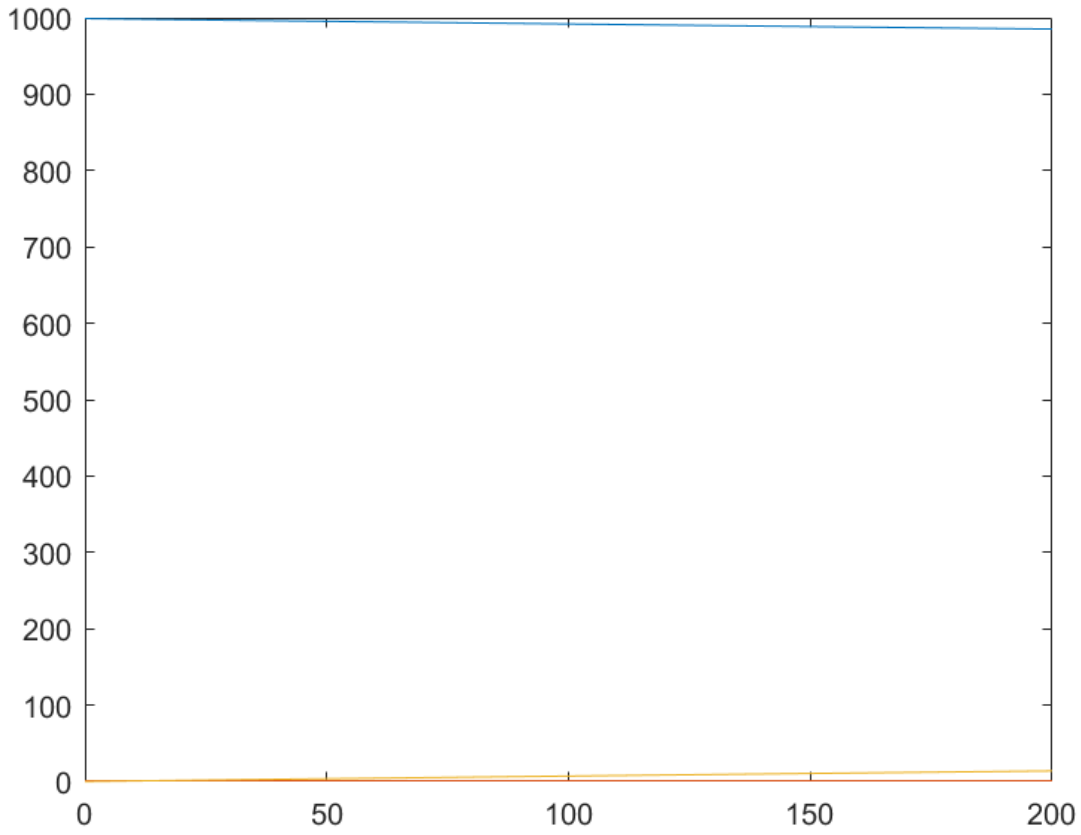


Figure 6 ODE 1

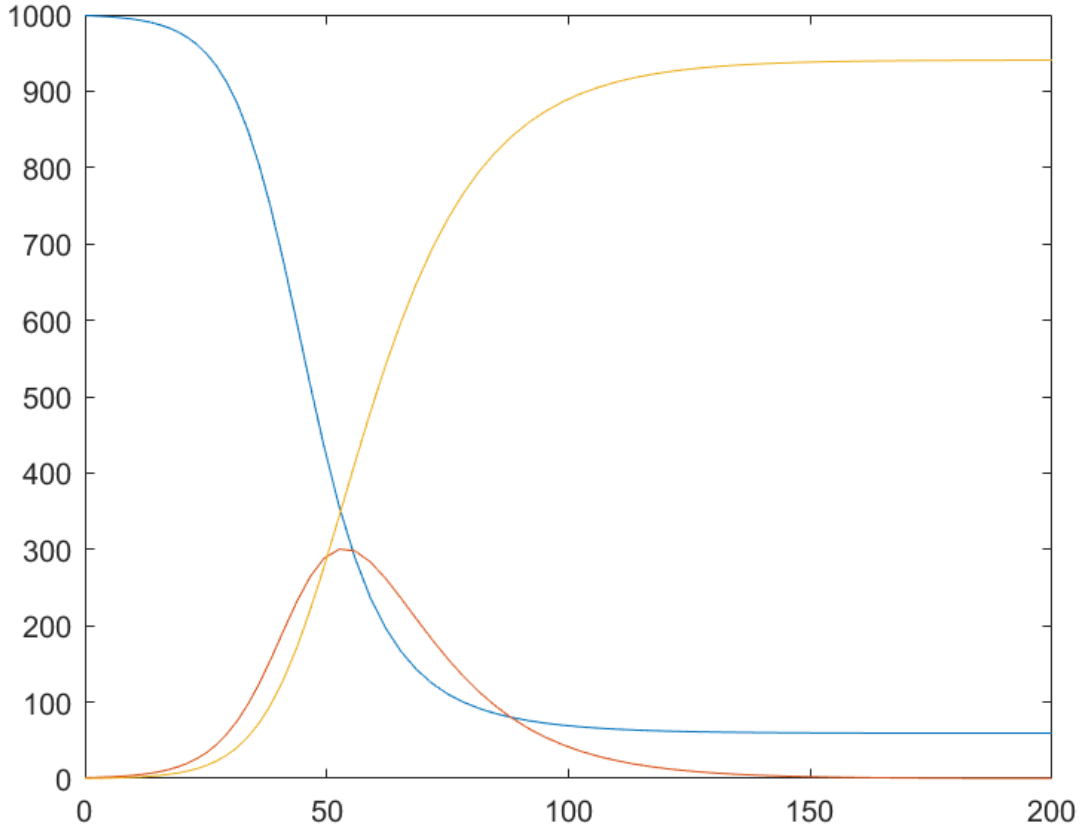


Figure 7 ODE 3

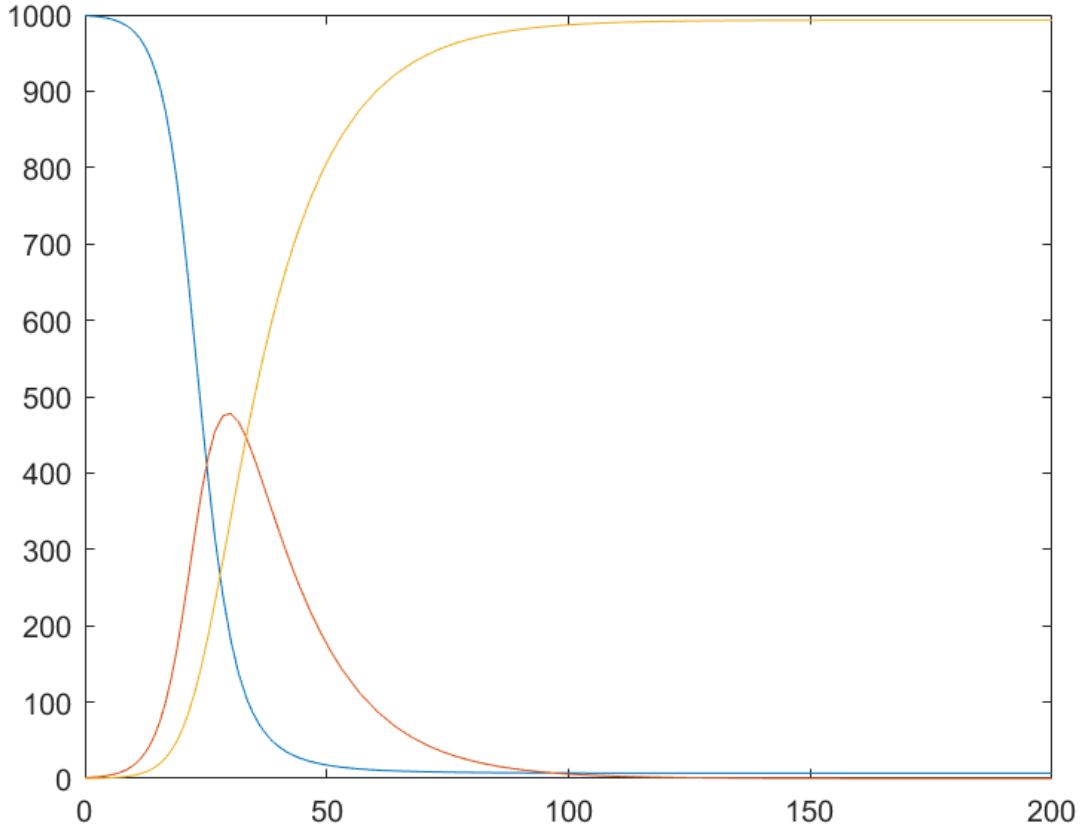


Figure 8 ODE 5