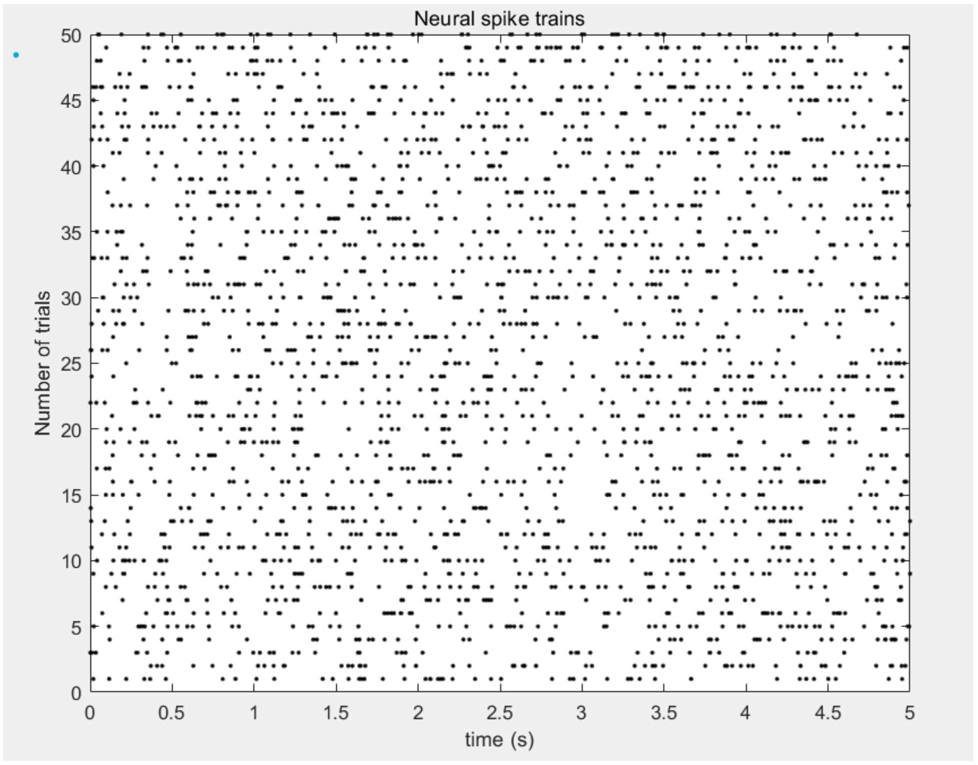
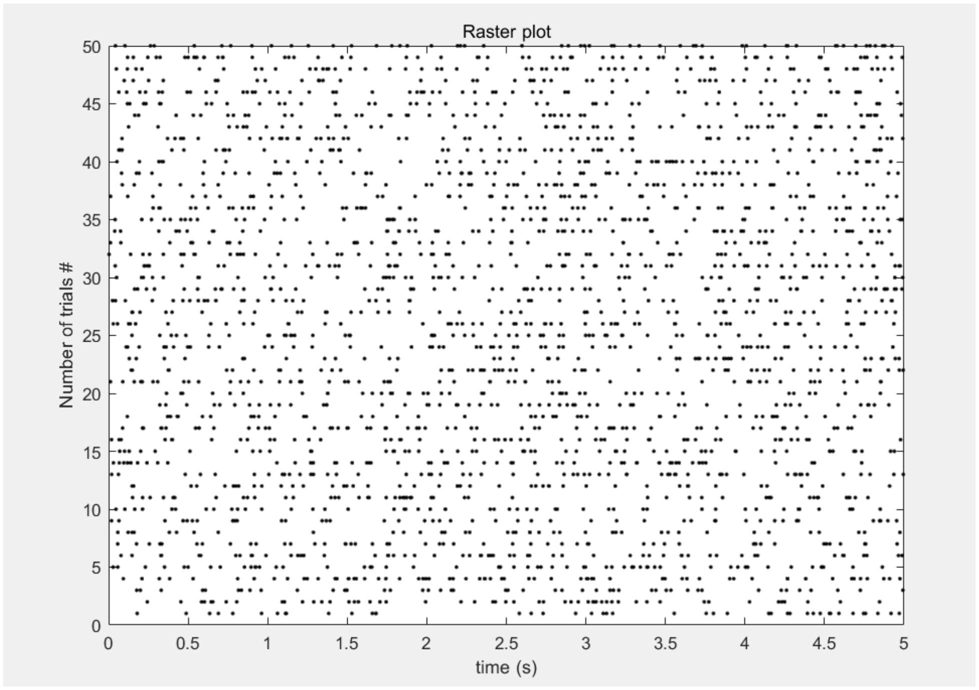
Part A

1.


Neural spike trains
Number of trials
time (s)

2.


Raster plot
Number of trials #
time (s)

3.

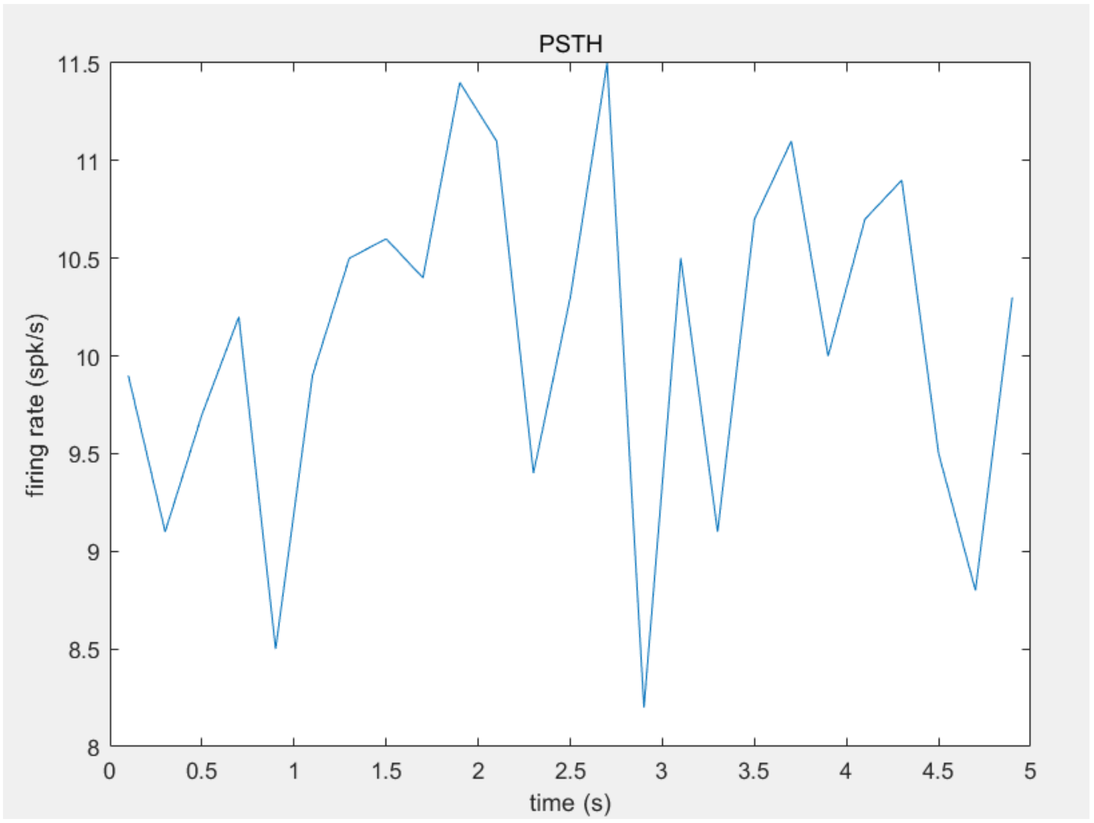
```
N = 50;
lambda = 10;
T = 5;
n = 2*lambda*T;
S = zeros(N,2*lambda*T);
% number of total spikes across trials
sum_f = 0;
% average firing rate
a_f = 0;
for i=1:N
   u = rand(n,1);
   ISI = -log(u)/lambda;
   t = cumsum(ISI);
   f = 0;
   for j=1:n
       S(i,j) = t(j);
       if S(i,j)>T
           if f == 0
               f = j-1;
               sum_f = sum_f+f;
           end
       end
   end
end
S(S>T) = NaN;
a_f = sum_f/(N*T);
```

I ran this code four times and got this answer each time:

```
a_f =   10.0240

>> a_f
a_f =   10.1160

>> a_f
a_f =    9.7320

>> a_f
a_f =    9.9320
```

Since all these answers are close to the theoretical firing rate λ= 10, we can say that our estimated firing rate a_f is very close to the theoretical firing rate λ.

4.



5.

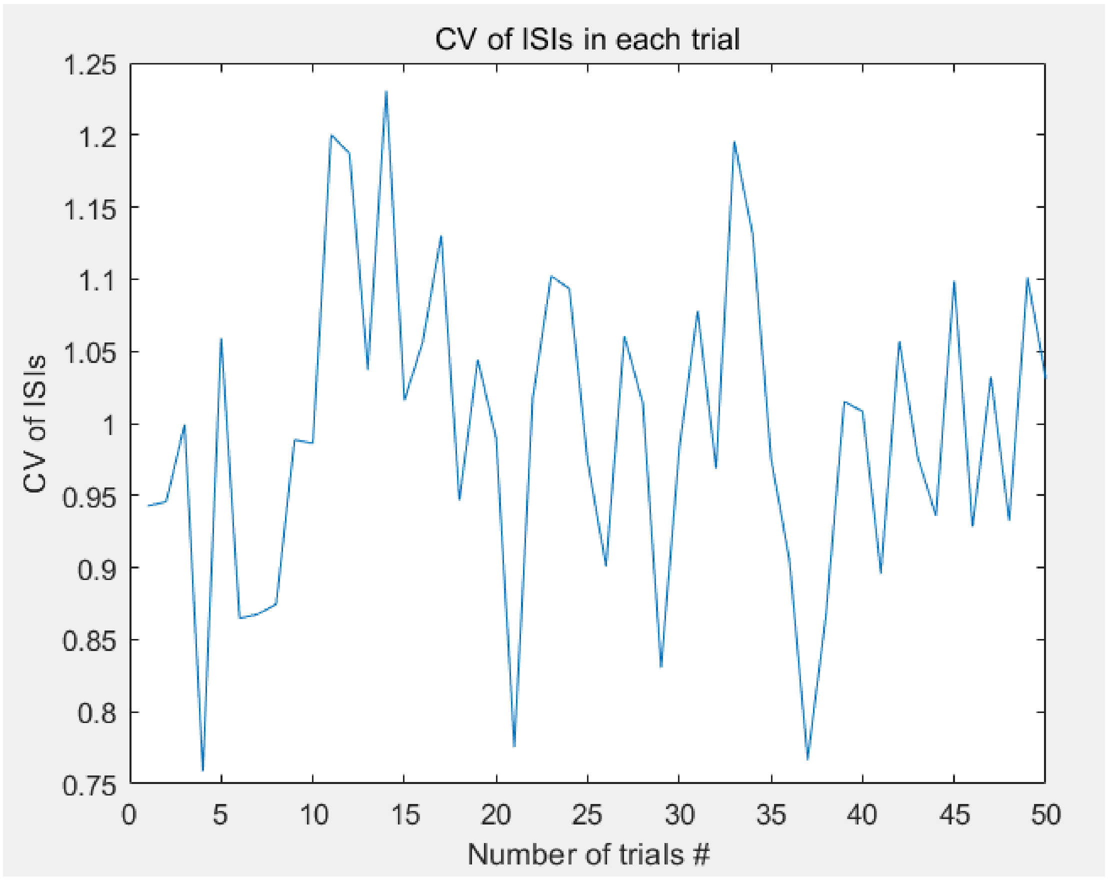
```
% let λ = lambda;
N = 50;
lambda = 10;
T = 5;
n = 2*lambda*T;
S = zeros(N,2*lambda*T);
% CV
CV = zeros(N,1);
trial = zeros(N,1);
a_CV = 0;
for i=1:N
   trial(i) = i;
   u = rand(n,1);
   ISI = -log(u)/lambda;
   t = cumsum(ISI);
   f = 0;
   for j=1:n
       S(i,j) = t(j);
   end
   t(t>T) = NaN;
   nn = ~isnan(t);
```

```
    n_s = length(find(nn));
    t1 = t(1:n_s);
    CV(i) = std(diff(t1))/mean(diff(t1));
end
S(S>T) = NaN;
a_CV = 1/N*(sum(CV));
plot(trial, CV)
xlabel('Number of trials #'); ylabel('CV of ISIs')
title('CV of ISIs in each trial')
```

I ran this code four times and got this answer each time:

```
a_CV =    0.9976
>> a_CV
a_CV =    0.9597
>> a_CV
a_CV =    0.9956
>> a_CV
a_CV =    0.9931
```

Since all these answers are close to the theoretical CV of Poisson spike trains = 1, we can say that our estimated <CV>, a_CV, ≈ 1



6.

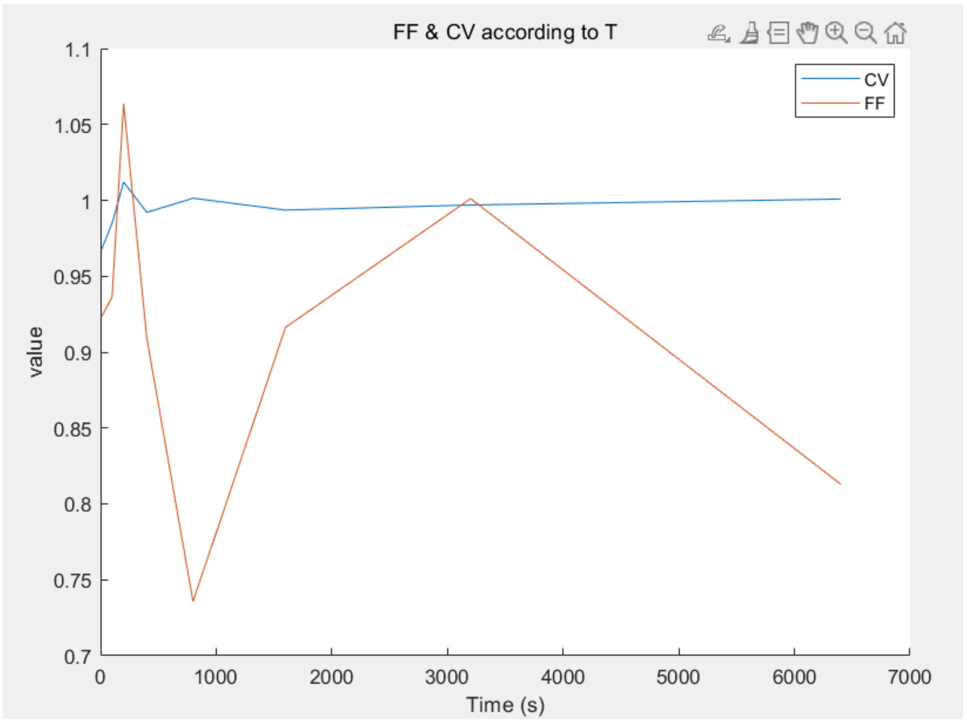
```
% let λ = lambda;
N = 50;
lambda = 10;
T = 5;
n = 2*lambda*T;
S = zeros(N,2*lambda*T);
% number of total spikes across trials
sum_f = 0;
% firing factor of all trials
all_f = zeros(N,1);
for i=1:N
   u = rand(n,1);
   ISI = -log(u)/lambda;
   t = cumsum(ISI);
   f = 0;
   for j=1:n
       S(i,j) = t(j);
   end
   t(t>T) = NaN;
   nn = ~isnan(t);
   n_s = length(find(nn));
   all_f(i) = n_s;
end
S(S>T) = NaN;
FF = var(all_f)/mean(all_f);
```

I ran this code four times and got this answer each time:

```
FF =    0.9643
>> FF
FF =    1.1274
>> FF
FF =    1.2320
>> FF
FF =    0.9621
```

Since all these answers are close enough to the theoretical Poisson spike trains FF=1, we can confirm that FF =1.

7.

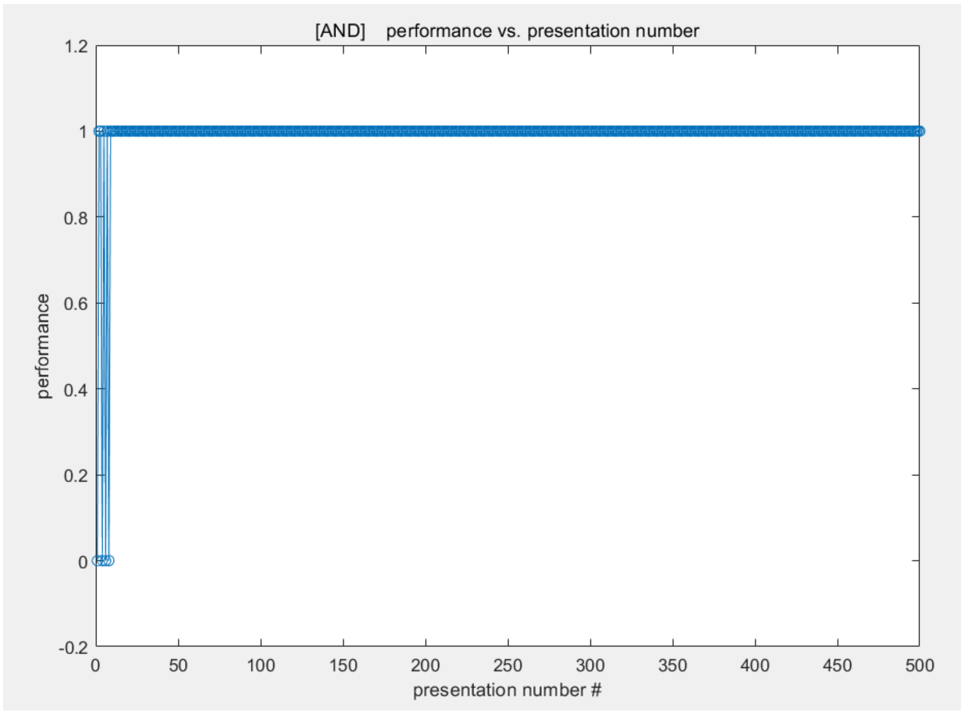


As we can see from the graph above,
CV visibly converges to 1 as T increases. However, FF does not converge to 1 as T increases.

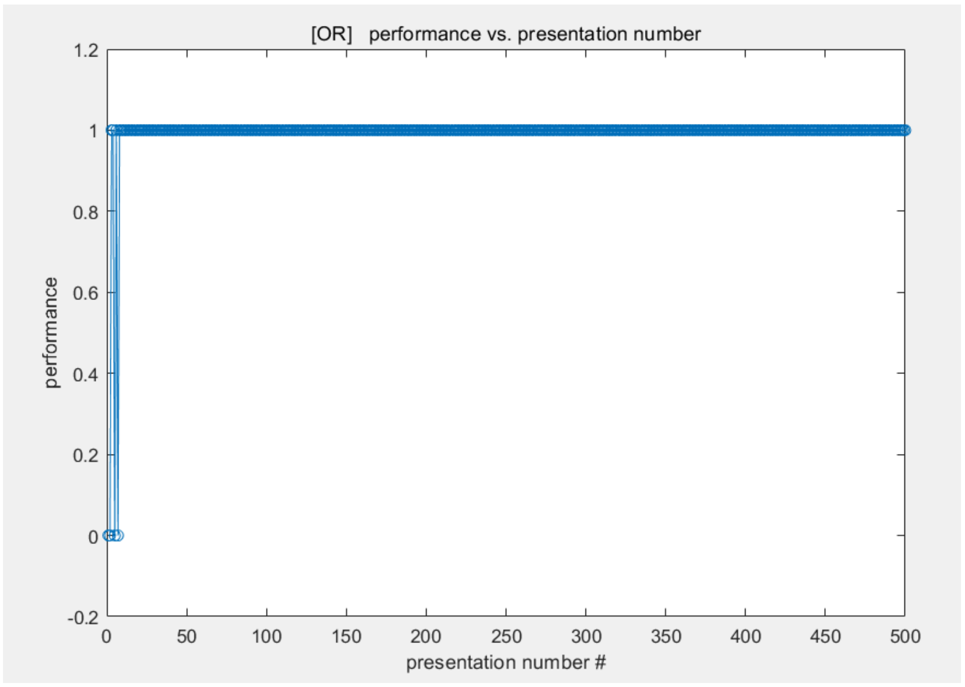
Part B

1 -1.

```
%%%% AND
```



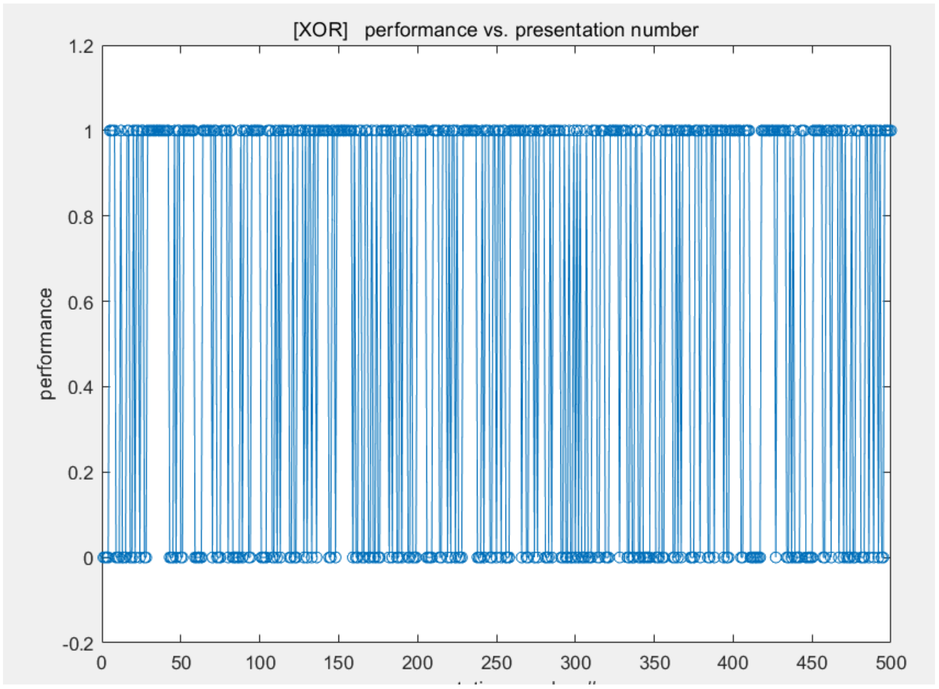
```
%%%% OR
```



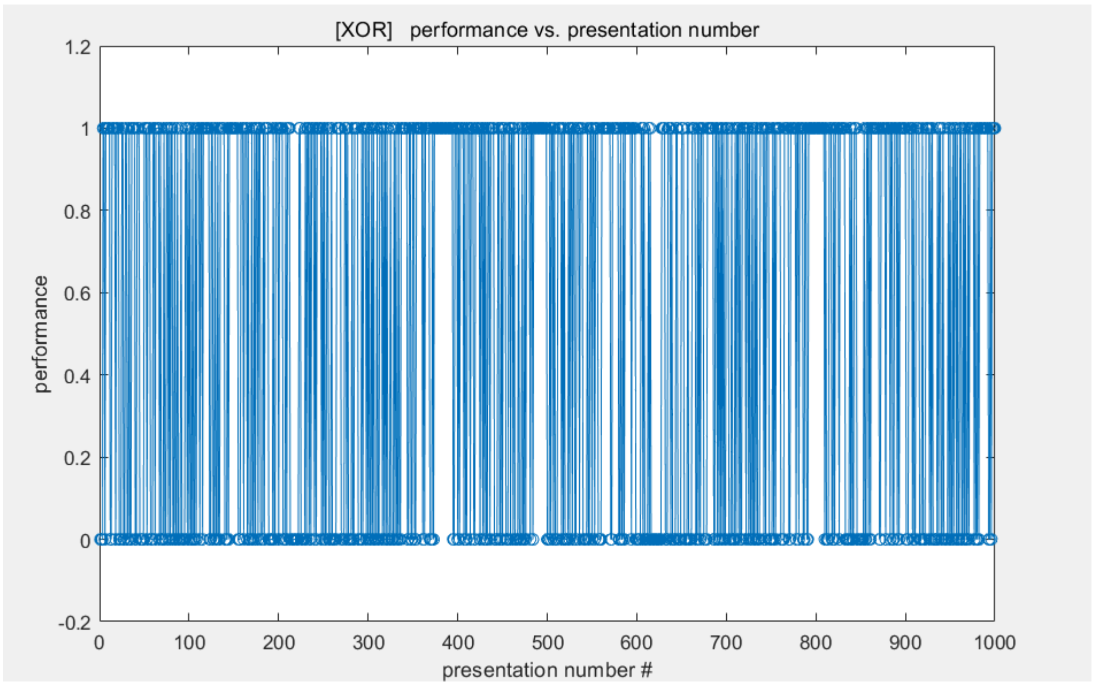
I ran AND and OR codes six times each. These are the number of steps it took to converge to the solution.
AND: 14, 14, 15, 9, 2, 20
OR: 13, 2, 21, 10, 2, 21
Thus, approximately, we can say it usually takes from 2 to 30 steps to converge to the solution.

1-2.

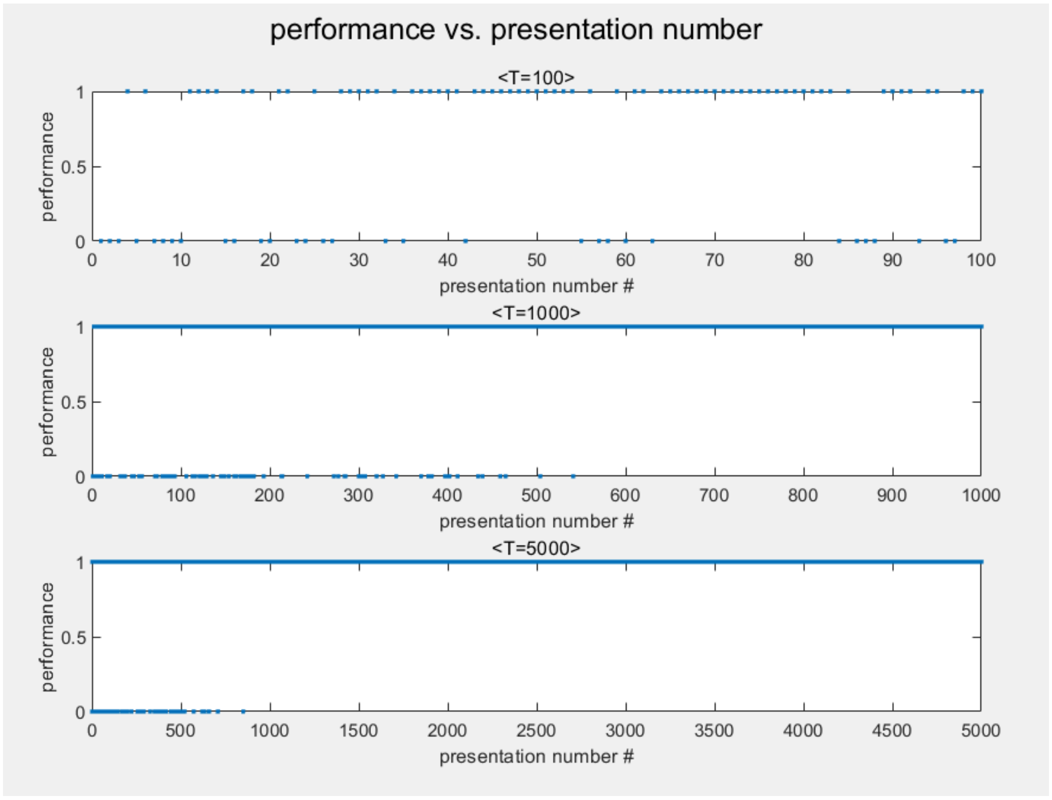
```
%%%% XOR
```



As we can see in the graph above, the algorithm does not converge to a perfect performance,1, for XOR. It doesn't appear to be converging, but it keeps repeating the correct and incorrect classification.
In the graph below, I increased the number of pattern suggestions to 1000, but it still does not converge to perfect performance. Therefore, it can be confirmed that the divergence of XOR is not due to the insufficient number of presentations. The real reason is that the linear form of a perceptron cannot solve the XOR problem, which is not linearly separable.



2-1.

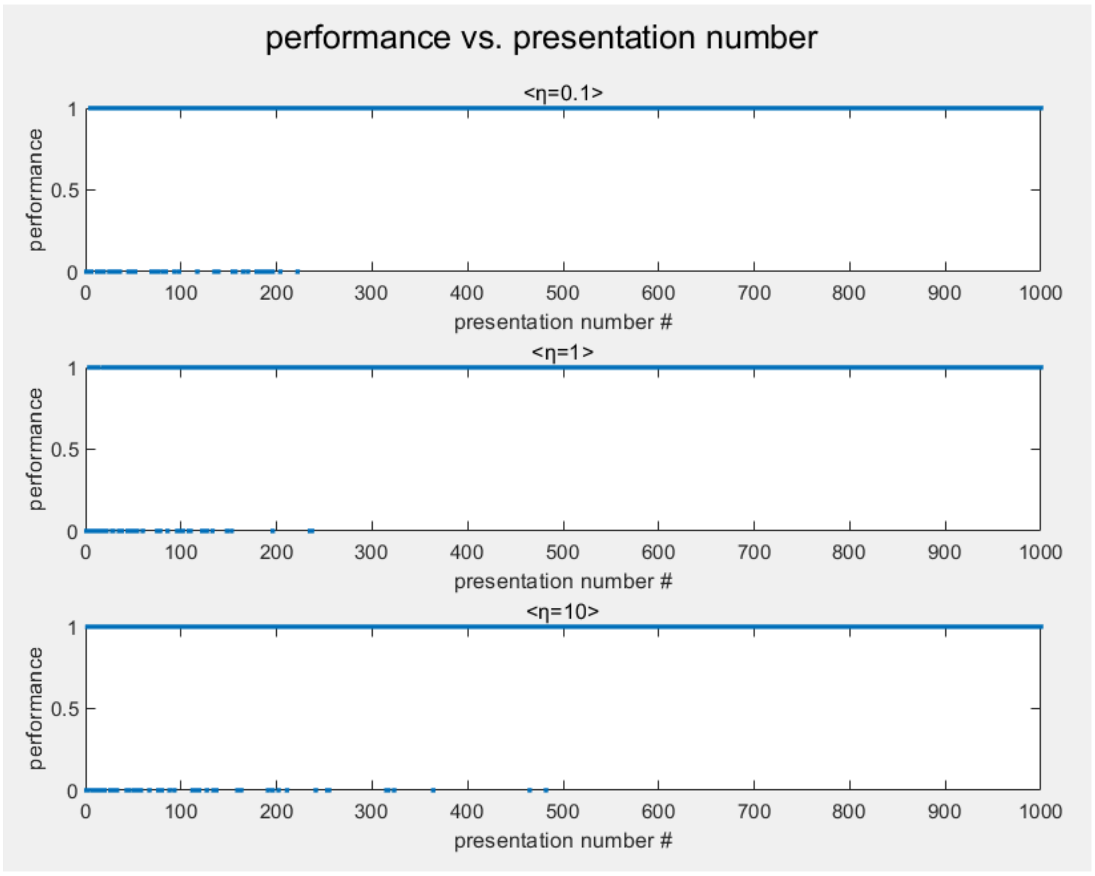


From the graphs above, we can see that the perceptron usually learns to separate numbers in the desired classes in about 600 steps.

2-2.

If the random vectors are linearly separable, by running the perceptron in enough steps, we will eventually get correct continuous outputs, in this code, performance=1. This means that by increasing the number of pattern presentations, we can separate these random vectors into straight lines. However, because they are random vectors, there can be cases where random vectors cannot be linearly separated. In these cases, the perceptron algorithm cannot perform classification perfectly even if an unlimited number of steps are applied, and the above graph repeats oscillation.

2-3.

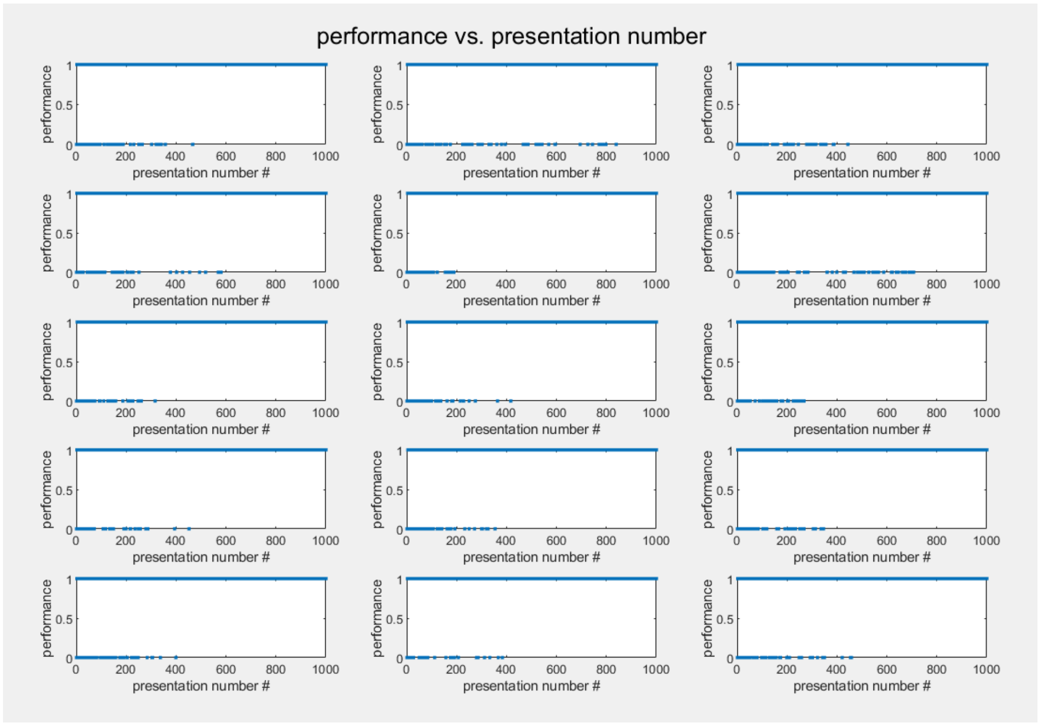


Yes, the results change if I use a different learning rate. This is because, learning rate, eta, affects the modification of synaptic weights, ‘w’. As a result, a learning rate that is too high may result in long oscillation steps, while a learning rate that is too low may result in slower convergence of results.

2-4.

average = 439.6667

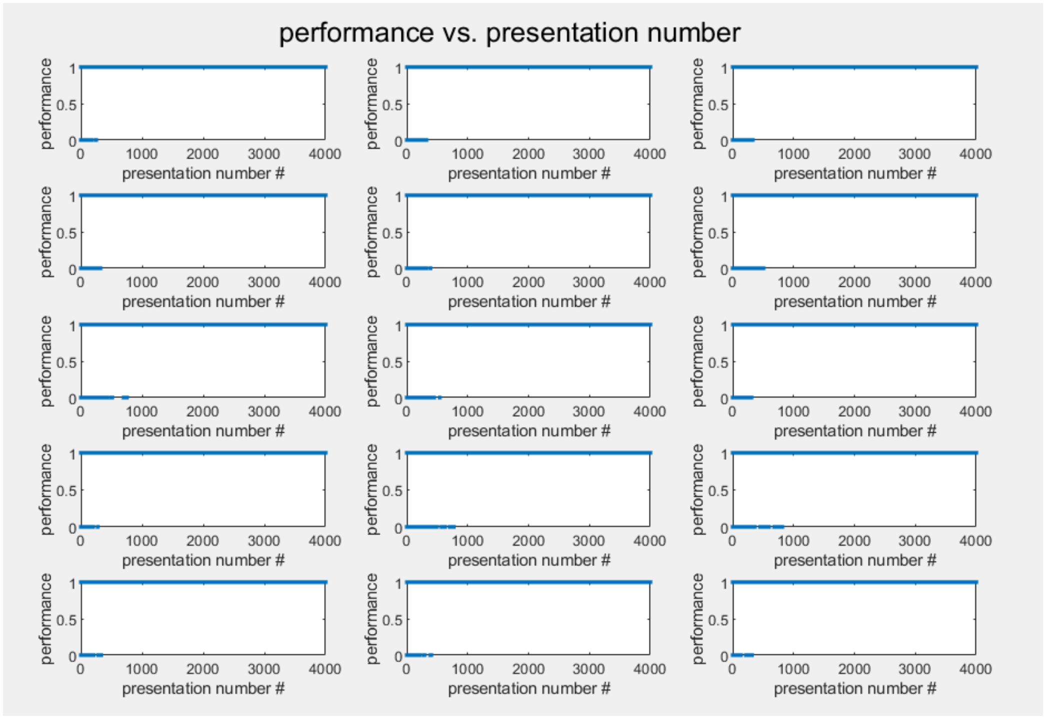
So in these 15 cases, on average, convergence will not be reached if we run less than 440 steps.



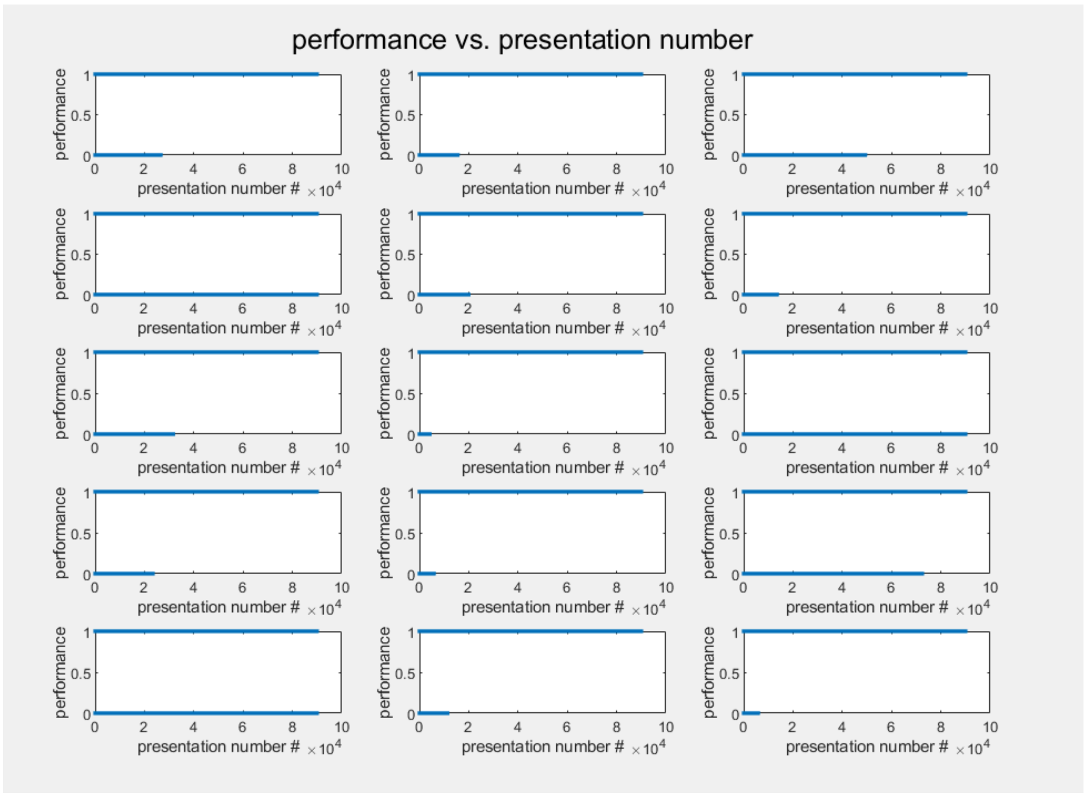
2-5.

100 × M presentations for M = 40

mean_pattern =   11.1167



1, 000 × M presentations for M = 90
mean_pattern = 407.0970



By repeating the simulation using different values of M, we can see that as more random vectors are added, the algorithm requires on average many more steps to converge to perfect performance.