2020

Simulation & Intelligent Tracking of a Robot

201603033

IDREES MALIK

Introduction:

The outline of this report is to use various modelling techniques along with numerical methods in order to analyse and find the location of numerical errors using computational approximations; within the set domain space. To do this, it would require implementation of differential equations in our algorithms in order to correctly simulate the given problem.

Breakdown of ACW:

1. Simulate the robot's movements
2. Add noise to simulated results
3. Develop an intelligent agent to predict next position of the robot.

Part 1:

In part 1 we are given the logic for a simulated robot in a 1-dimensional space. The robot's movements in this model are based of the mathematical equation $\dot{x} = -2x + 2U$. The model states using the equation that as the time changes so does the displacement of the robot from its point of origin.

Individual Components Definition:

- H - Integration Step Size
- Time – Time
- U – Distance Travelled from Point of Origin
- X – Generalised Co-ordinates from the origin point
- K – Sample Number

$$U = \begin{cases} 2 \; for \; 0 < t \leq 5 \\ 1 \; for \; 5 < t \leq 10 \\ 3 \; for \; 10 < t \leq 15 \end{cases}$$

The logic behind the variable U, stipulates that at a given time after the robot has travelled from the origin, the distance the robot should have travelled in that time (U), should be a speicific value. The U value is dependant upon time as the robot travels at a set velocity making it easy to predict the given distance after a certain time, allowing for U to be calculated with such accuracy. Such a value of U can be calculated using a kinematics equation $S = vT$. S is displacement, being the same as U can be calculated using the fixed velocity (V) * Time (T).

Exact Solutions:

The exact solution of the model equation given below:

$$\dot{x} = -2x + 2U \qquad x(t) = U(t) - e^{-2t}$$

Stable & Unstable Simulations:

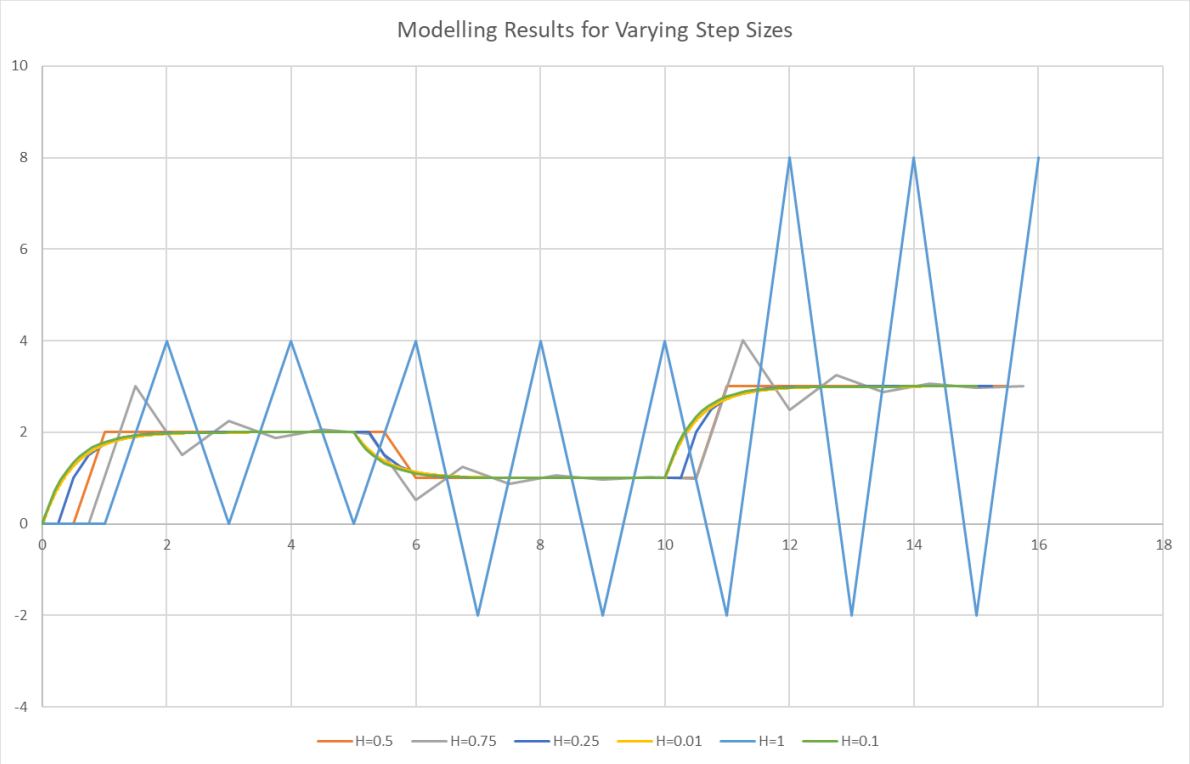
Within Euler's method, for a simulation to be considered stable it must meet the criteria that $|ha + 1| \leq 1$

This equation states that the modulo of the step size (h) multiplied by a, in which x = ax. If the value of h is above a certain value leading to the model becoming unstable then the results generated for that given value of h, would not show any trends and in extreme cases lead to osscilating results. Using varying values of H we can deduce the point at which the simulation starts to become unstable as the results generated from the simulation start act in an unpredictable manner leading to an unstable simulation.

Analysing Results:

Analysing the figure below it is a reasonable assumption to make that as the values of H approaches the exact solution the more accurate the results become, and therefore the closer the results are to the true values. The opposite can be said for the results as the step size increases as the less accurate the results become and the more error prone the results are. The reason for the increase in error in results as H increases is because the instability of the simulation due to H increases leading to less accurate results and less accurate simulations. Meaning the smaller the step size the more accurate the outcome and the larger the step size the less accurate assumption. With the value H = 1.0 the model continuously over-estimates the results causing the outcome to be an oscillation; whereas when H = 0.75 the model over-estimates the results and overtime corrects the error allowing 0.75 to follow the correct trend of the result to a given extent.

Figure 1 Comparison of Different Step Sizes



Time	True Results	H = 0.01	H = 0.25	H = 0.50	H = 0.75	H = 1.00
3.00	1.99752	1.997667	1.99975	2.00000	1.93700	4.00
6.00	1.13533	1.135320	1.12499	1.00000	0.49601	-2.00
9.00	1.00033	1.000315	1.00003	1.00000	0.98650	4.00
12.00	2.98168	2.964106	2.98437	3.00000	2.51806	-2.00
15.00	2.99995	2.999916	2.99998	3.00000	2.98826	8.00

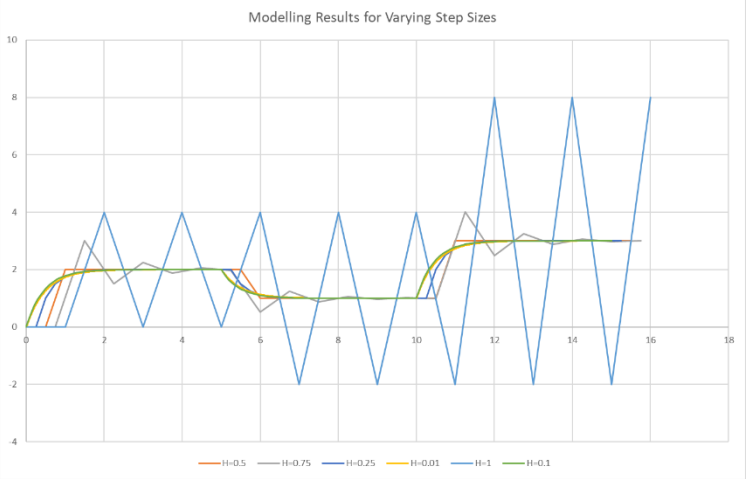
Upper & Lower Value of H:

$$|ha + 1| \leq 1$$

In order to calculate the whether the simulation is stable for the given equation, the graphs created in step 1 should be an exponential. As even using the equation for stability for a given value of h, the model maybe mathematically considered to be stable, but when observing the graph created it becomes apparent that the graph is not an exponential and the results don't show any of the patterns shown in the graphs of lower H values.

H Value	|Ha + 1| Value
0.01	|0.01(-2) + 1| = 0.98 – Stable
0.1	|0.1(-2) + 1| = 0.8 – Stable
0.25	|0.25(-2) + 1| = 0.5 – Stable
0.5	|0.5(-2) + 1| = 0.0 – Stable
0.75	|0.75(-2) + 1| = 0.5 – Unstable
1.0	|0.01(-2) + 1| = 1.0 - Unstable

Figure 2 Different Step Sizes & Comparing Instability of Results



From the values shown in the table and the graphical results for each value of H it becomes evident that the upper value of H is 0.5. This is because this is the highest value for H where the results show the least overshoot; any value of H above 0.5 seems to over-approximate the results leading to an increase in the error of the results. The results shown for a H = 0.75 is not oscillating but the results are initially over-estimating and under-estimating the results by a considerable amount leading to an increase in the error.

The lower limit is 0.01 as any value below this value of H is essentially identical meaning there is no change in the error. Also reducing the value of H below this value increases the computational complexity without allowing for there to an increase in the accuracy of the results.

Part 2:

Box-Muller Method & Random Noise:

Part 2 of the ACW was to generate and add noise in a random process with a mean = 0.0 and a standard deviation = 0.001 to the results from part 1, using the Box-muller method.

The box-muller method works by generating one random number in a uniform distribution and returns two random numbers normally distributed given the mean = 0.0 and standard deviation = 0.001.

The box muller method can be implemented in three steps:

- Step 1 – Generate a uniformly distributed random number between [0, 2 * π].
- Step 2 – compute b = sigma sqrt (-2 * ln [0, 1]) – ln[0, 1] finds the natural log of a random number between 0 – 1.
- Step 3 – compute X1 = b*sin(a) + mean, X2 = b*cos(a) + mean.

The purpose for using random numbers generated using the box-muller method is because the random numbers generated are normally distributed meaning, they don't affect the mean or SD of the current model and they also accurately represent the data from part 1. The box-muller method for generating white noise is better than just generating a random number as the numbers generated by the box-muller method conforms to the distribution of the data from part 1 and clusters around an average. Adding white noise to our simulation allows us to see how noise encountered by the robot affects the model and is a more realistic simulation.

Figure 3Noise Values vs Actual Values with SD = 0.001

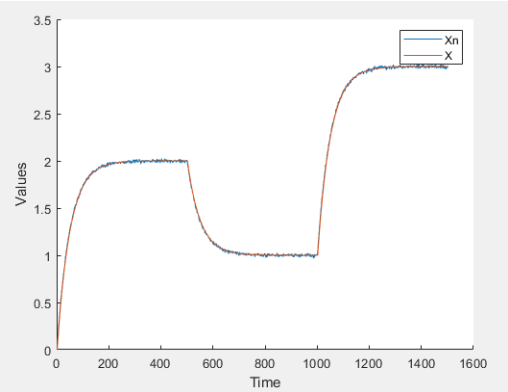


Figure 4Noise Values vs Actual Values with SD = 0.001

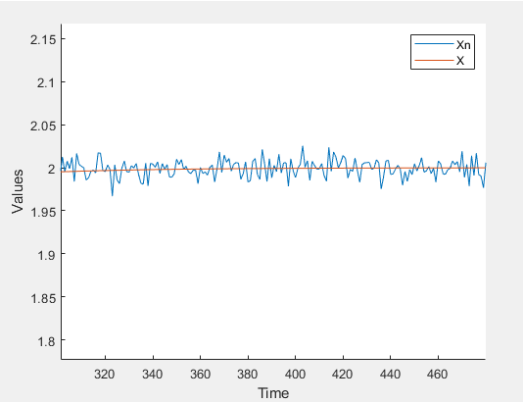


Figure 5 Noise Values vs Actual Values with SD = 0.01 and H = 0.01

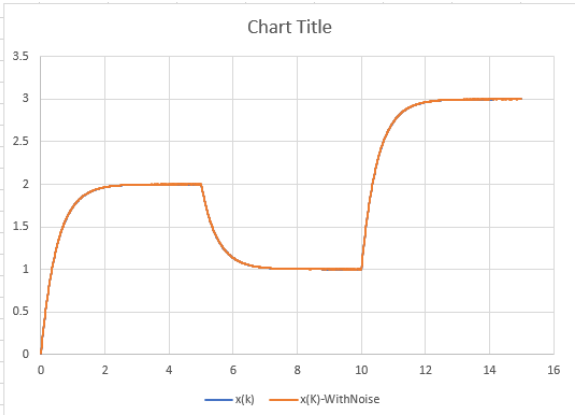
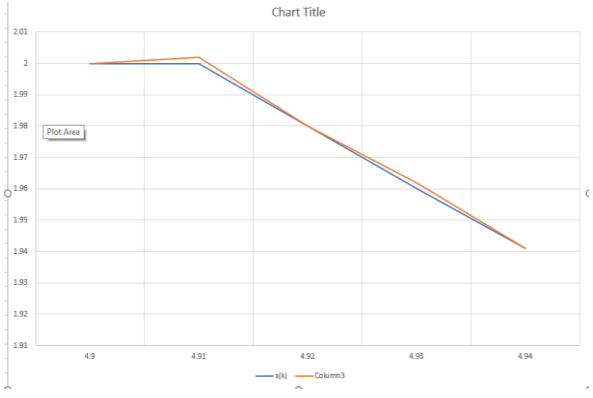


Figure 6 Noise Values vs Actual Values with SD = 0.01 and H = 0.01



Part 3:

Creating a perceptron to predict the next position of the robot:

The purpose of task 3 is to create an intelligent system in a perceptron which is able to predict the next position of the robot using training data from previous tasks. The perceptron will make use of the noise generated in task 2 using the randomly generated noise as training data for the given perceptron.

The process of creating a perceptron can be broken down into the given stages:

- Calculate the sum, which is the input values multiplied by the given weights at each instance.
  - {Constant * W0 + X1 * W1 + X2 + WN * XN}
- Calculate the output, the value caluclated in the previous step then enters the activation function for which it's compared against the threshold value assigning it a 0 or 1 depending on the type of activation function used.
- Update the weights, the delta value is calculated based on the given output for which the weight's are updated.
  - {Weights[i] = Weights[i] + (learning_rate*Delta*input[i])}

When creating my perceptron I decided to initialise 2 random weights with a threshold value all of which was randomly caluclauted between -0.5 – 0.5, with a bias value of 1.0 and a learning rate of 0.0001.

The table below shows the process of calculating the error of the perceptron for both activation functions.

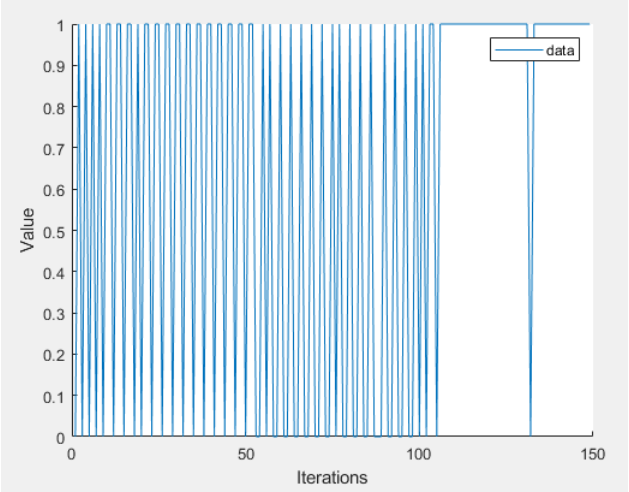
Iterations/Epoch	Input(1)	Input(2)	Output of Perceptron (step activation)	Ouput of Perceptron (Sigmoid Activation)	Error
1	0.0000	0.1306	0.0000	0.4035	0.2371 – 0.4035
2	0.1306	0.2371	0.0000	0.3769	0.3233 – 0.3769
3	0.2371	0.3233	0.0000	0.3558	0.3836 – 0.3558
4	0.3233	0.3836	0.0000	0.3401	0.4403 – 0.3401
5	0.3836	0.4403	0.0000	0.3279	0.4877 – 0.3279

Step Activation Function

Whilst using the step activation function the perceptron is unable to learn as the with the step activation function is not linearly seperable as it cannot be differentiated meaning the weight update isn't able to learn as the weight update uses a gradient. The gradient is the first derivative of the error squared function, and as the step function doesn't have a first derivative that means the perceptron isn't able to learn the weight update and also do any back propagation as it has no useful derivative. As a result the when using the step activation the perceptron cannot differentiate the change in

values. As a result the perceptron is unable to predict with any accuracy the next position of the robot as the step function only has 2 ouptuts, 1 & 0.

Figure 7 Perceptron Output Using Step Activation Function



Logistic Sigmoid

When using the logistic sigmoid the perceptron is able to learn and develop a very accurate prediction of the movements of the robot. This is because the logistic sigmoid isn't limited to only 2 ouptut values like the step activation function. Instead the logistic sigmoid which is represented by $1/ (1 + e^{-x})$ is able to learn and predict the next location of the perceptron with accuracy as it's able to deal with real numbers as well as the logistic sigmoid equation is differentiable everywhere allowing the weights to update using a gradient whilst the perceptron is learning. This is important as because the perceptron is able to update it's weights whilst running this allows it to accuractely predict the next movement of the robot.

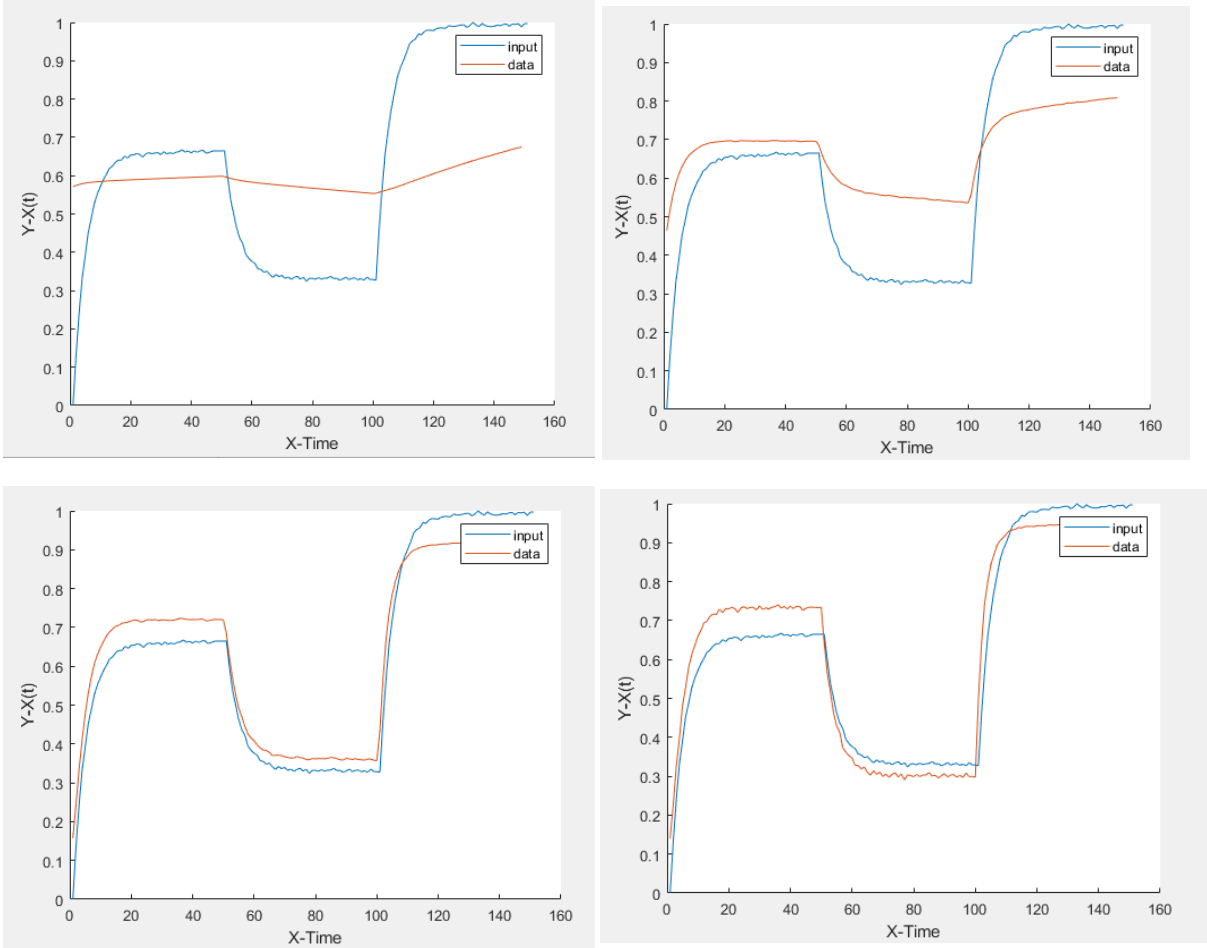
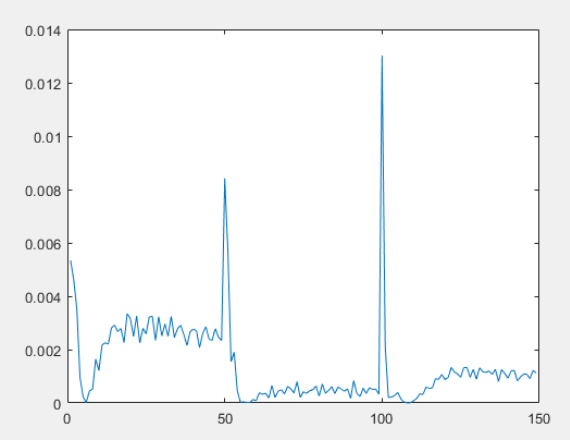


Figure 8 Cost Function During Runtime of Perceptron



In order to find if the agent was able to generalise or not, I ran the box-muller code from part 2 and generated a new set of data which had not been seen before. To further the unseen data, I also changed the standard deviation. This unseen data was inputted into the perceptron but this time I did not let the weights update. Instead I stored the weights from a previous run and then inputted those weights into the perceptron when running the unseen data. What I found was that the perceptron was still able to predict with reasonable accuracy the movement of the robot even with unseen data. This shows that the perceptron was able to generalise with reasonable accuracy. This process is also a lot faster as there is no weight training involved.

It is possible to predict the position and velocity using the perceptron. Using the kinematics equation of motion. Using the equation of S=VT where S = displacement, V = Velocity, T = Time. 1st the perceptron would need to the predict the location of the robot, which is already functional, as the code has already been implemented as part of the part 3. Now that the robot has the value S, at a given time T you would subtract the start Time from the end time in order to get the time difference this time difference is T, therefore velocity can be calculated simply by Velocity = Distance/Time.

Conclusion

During Task 1 I found that whilst generating the results the step size is crucial as anything above the upper limit leads to over-approximations and unstable plots, and also anything below the lower limit leads to increased computational complexity with not much increase in accuracy of results.

During Task 2 by adding noise using the methods described it is possible to cause a simulation to act as if it was in a real-world scenario especially in terms of the noise generated using the sensors.

During Task 3 our objective was to create a robot that essentially predicts the next position of the robot, using training data from task 2. During part 3 I gained a deeper knowledge of perceptron's and their individual components. Not only understanding the theory behind the perceptron but also understanding how the perceptron reacts to different learning rates and activation functions. As a result, I feel I have developed finding a way around any limitation and also adapting my approach to concepts I haven't encountered before.

Appendix Code:

Step 1 & Step 2 Code:

```
% Step 1, this process is just writing the code and getting the logic
% working
k = 1;
x = [];
x(k) = 0;
U = 0;
time = 0;
h = 0.01;
sampleInterval = 0.1;
%Adding random numbers variables
it = 0;
S = 0.1;
M = 0;
Xn = [];

%Create txt file for noisy numbers
fileID2 = fopen('step2.txt','W');
format2 = '%1.2f %1.3f %1.3f %1.0f\n';

%Create txt file for non - noisy numbers
fileID = fopen('step1.txt', 'w');
format = '%1.2f %1.3f %1.0f\n';

while time <= 15
    if (0 < time) && (time <= 5)
        U = 2;
    elseif (5 < time) && (time <= 10)
        U = 1;
    elseif (10 < time) && (time <= 15)
        U = 3;
    end
    if time == 0
```

```
while time <= 15
    if (0 < time) && (time <= 5)
        U = 2;
    elseif (5 < time) && (time <= 10)
        U = 1;
    elseif (10 < time) && (time <= 15)
        U = 3;
    end
    if time == 0
        fprintf(fileID, format, time, x(k), U);
    end
    %x(k) = U - exp(-2 * time);
    x(k+1) = x(k)+h*(-2*x(k)+2*U);
    k = k + 1;

    %Adding the numbers to the step1.txt
    fprintf(fileID, format, time, x(k), U);

    %Adding Random Numbers(step 2)
    if (it == 0)
        z1 = 0+rand*(2*pi-0);
        temp1 = 0+rand(1-0);
        temp = sqrt(-2*log(temp1));
        b = S * temp;
        z2 = b * sin(z1)+M;
        z3 = b * cos(z1)+M;
        Xn = [Xn x(k) + z2];
        %Write noisy numbers to file
        fprintf(fileID2, format2, time, x(k), Xn(k-1), U);
        it = 1;
    else
        it = 0;
```

```
    %Adding the numbers to the step1.txt
    fprintf(fileID, format, time, x(k), U);

    %Adding Random Numbers(step 2)
    if (it == 0)
       z1 = 0+rand*(2*pi-0);
       temp1 = 0+rand(1-0);
       temp = sqrt(-2*log(temp1));
       b = S * temp;
       z2 = b * sin(z1)+M;
       z3 = b * cos(z1)+M;
       Xn = [Xn x(k) + z2];
       %Write noisy numbers to file
       fprintf(fileID2, format2, time, x(k), Xn(k-1), U);
       it = 1;
    else
       it = 0;
       Xn = [Xn x(k) + z3];
       %Write to step 2 file
       fprintf(fileID2, format2, time, x(k), Xn(k-1), U);
    end
    time = time + h;
end
hold on;
plot(Xn,'DisplayName','Xn');
plot(x,'DisplayName', 'X')
hold off;
legend;
xlabel("Time");
ylabel("Values")
fclose(fileID);
fclose(fileID2);
```

Step 3 Code:

```
%Read in White Noise Results
filename = 'step2.txt';
delimiterIn = ' ';
headerlinesIn = 0;
A = importdata(filename,delimiterIn,headerlinesIn);
input1 = [];
input1 = [input1 (A(:, 3))];
input = [input1];
input = rescale(input);

%Initialise the weights and threshold value for the given two inputs
weights = [];
R = [-0.5 0.5];
Threshold = rand(1,1)*range(R)+min(R);
for i = 0:2
    weights = [weights (rand(1,1)*range(R)+min(R))];
end

%start the perceptron
J = 1;
output = 0;
learning_Rate = 0.01;
data = [];
CostList = [];
Error = 0;
for r = 0:10000
    data = [];
    CostList = [];
    for i = 1:length(input) -2
        x = [input(i) input(i + 1) 1];
        %Simulating the Perceptron
        net_sum = 0;
```

```
for r = 0:10000
    data = [];
    CostList = [];
    for i = 1:length(input) -2
        x = [input(i) input(i + 1) 1];
        %Simulating the Perceptron
        net_sum = 0;
        for j = 1:length(x)
            net_sum = net_sum + x(j) * weights(j);
        end
        % Calculate the sigmoid ouput for the given net_sum calculated
        % whilst simulating the perceptron

        % output value for the sigmoid activation function
        output = 1/(1 + (exp(1) ^ -net_sum));

        % output value for the Step Activation function
        %if net_sum >= 0
         %   output = 1;
        %else
        %    output = 0;
        %end

        %Adding output value to list
        data = [data (output)];

        %Calculating the error of the perceptron
        Target = input(i + 2);
        Delta = Target - output;
        for k = 1:length(weights)
            weights(k) = weights(k) + (learning_Rate * Delta * x(k));
        end
```

```
        % output value for the sigmoid activation function
        output = 1/(1 + (exp(1) ^ -net_sum));

        % output value for the Step Activation function
        %if net_sum >= 0
         %   output = 1;
        %else
        %    output = 0;
        %end

        %Adding output value to list
        data = [data (output)];

        %Calculating the error of the perceptron
        Target = input(i + 2);
        Delta = Target - output;
        for k = 1:length(weights)
           weights(k) = weights(k) + (learning_Rate * Delta * x(k));
        end
        Error = Delta;
        J = 0.5 * ((Error)^2);
        CostList = [CostList (J)];
    end
end
hold on;
plot(input,'DisplayName','input');
plot(data,'DisplayName','data');
hold off;
legend;
xlabel("X-Time");
ylabel("Y-X(t)")
```