Computational Science

Simulation and Intelligent Tracking of a Robot

Word Count: 2198

February 2020

Student Number: 2017071831
Authored by: Onayo Moro

UNIVERSITY OF HULL

Contents

Introduction

The purpose of this project is to understand and develop the algorithmic techniques used in modelling and simulating an intelligent robotic tracking system. The report details the structure of the project in three parts. Simulating and ideal robot in 1 dimension, making it more realistic by adding random noise, and finally, training an intelligent agent to track said robot.

This report will lay out a comprehensive analysis on the location and origins of numerical errors and their imprecise nature regarding computational approximations. In the context of real-world applications, it is important to utilise the methods needed in modelling, simulation and abstraction to allow machines to become better problem solvers.

Part I

Simplified Robot Model

The function of the robot in this simulation is to respond to a series of commands that tell it where to move in 1 dimensional space. For the purpose of this simulation we can assume that this space is the floor. Equation 1 shows a simplified model of the robot being used in this project.

$$\dot{x} = -2x + 2U$$

Equation 1: Simplified Model of the Robot being used

- The $\boldsymbol{x}$ represents a set of generalised coordinates as a distance from the origin.
- $\boldsymbol{U}$ represents the distance from the origin that the robot has travelled and the value of $\boldsymbol{U}$ changes with time represented by $\boldsymbol{t}$ which is measured in seconds to the second decimal place.

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

Equation 2: How U Changes Relative to t

Euler Method

There are various methods to reduce the error between the numerical calculation and the true value for dynamic systems, one of them being the Runge-Kutta method. For this project however, the Euler method will be used. Compared to Runge-Kutta, Euler offers a more straightforward approach as it estimates the next point based on the rate of change at the current point. Therefore, the Euler method is more often than not used in time discretisation as opposed to spatial discretisation making it more suitable for the purpose of this project.

On the other hand, Runge-Kutta operates by taking slope calculations at multiple steps at or between the current and next discrete time values. Then, a weighted value of said multiple steps is taken. As this project relies on analysing values against a constant sampling interval, the Euler method is more appropriate.

Integration Step

The Euler method makes use of an integration step size represented by $\boldsymbol{h}$ (also known as a sampling interval) which essentially dictates the frequency of error measurements within a given $\boldsymbol{t}$ (time). In the case of this simulation, it also dictates the error in the integration. The smaller the value of $\boldsymbol{h}$, the greater the accuracy of the error measurement. As its value continues to diminish towards 0, the error between the current point and the desired point decreases. However, as $\boldsymbol{h}$ increases, the error accuracy decreases as the interval between error measurements becomes greater as larger step sizes are prone to overshooting, therefore making the simulation unstable.

$$||\boldsymbol{h}|| \leq \boldsymbol{h_l}.$$

Equation 3: Determine range of h

$$|\boldsymbol{ha} + \mathbf{1}| \leq \mathbf{1}$$

Equation 4: Unstable Simulation Equation.

Whilst the equation in ***Equation 4*** represents an unstable simulation, there is a difference between an unstable simulation, and an unstable simulation represented by the equation. A stability can be affected by other variables such as the precision of the measurements or other variables as well as the method itself. Another more accurate method is the midpoint method, which essentially derives an average, represented by a percentage, from a quantity which is like the Runge-Kutta method mentioned earlier.

Simulation Results Comparison

During the simulation the value of $\mathbf{U}$ will change as the time increments. The Euler method also makes use of forward difference. This allows for the prediction of the next value of $\mathbf{x(k)}$ ($\mathbf{k}$ being the sample number) in which the current position of the robot is held. With the same algorithm, it is then possible to calculate $\mathbf{x(t)}$ which will be on the y axis shown in **Figure 1: Comparison of Different Integration Step Sizes**. $\mathbf{x(t)}$ represents where the robot has moved to at a certain time.

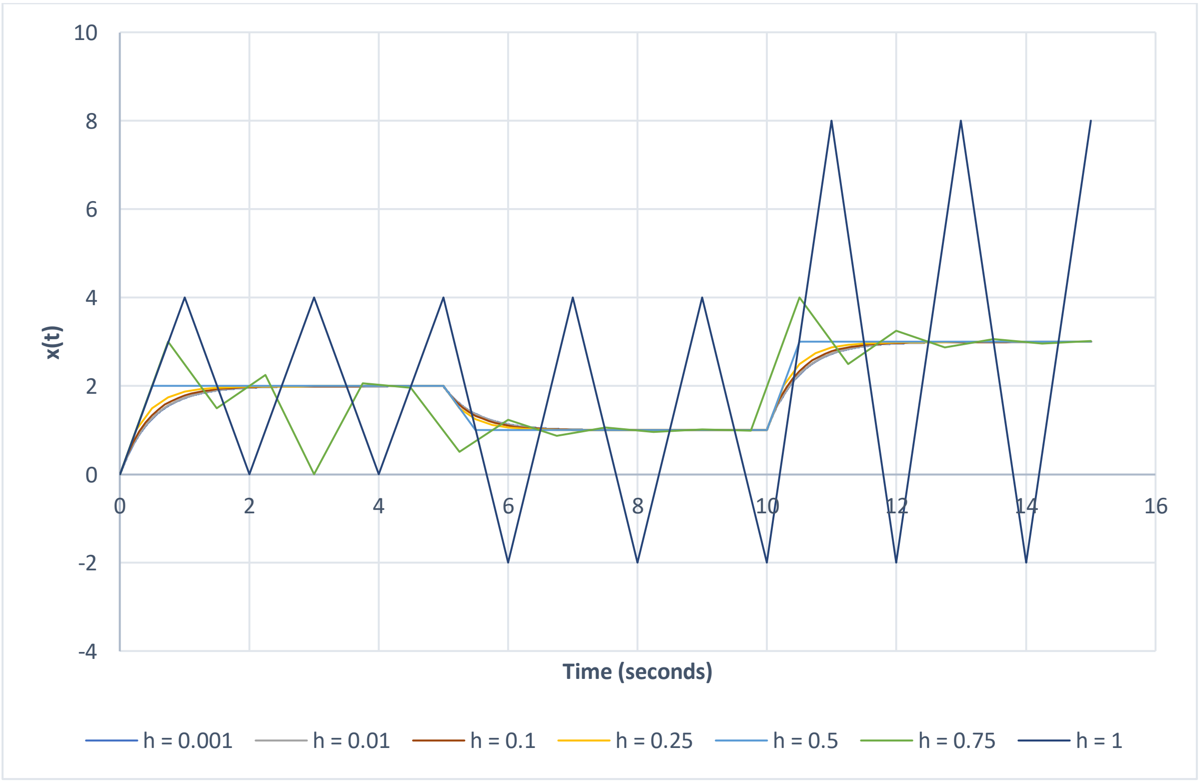


Figure 2: Comparison of Different Integration Step Sizes

As different values of $\boldsymbol{h}$ influence the accuracy of the error as well the interval for error measurements, the value of $\boldsymbol{x(t)}$ was directly affected. Based on the evidence shown in ***Figure 3: Comparison of Different Integration Step Sizes***, the upper limit of $\boldsymbol{h}$ that showed the least amount of overshoot was 0.5. Unlike $\boldsymbol{h}$ = 0.75 and $\boldsymbol{h = 1}$, there appears to be minimal oscillatory patters on the graph signifying minimal overshooting. This evidence suggests that for the parameters of this simulation, $\boldsymbol{h = 0.5}$ provides accurate enough error measurements between regular enough intervals for the robot model to operate effectively. This is also supported by the unstable simulation equation shown in ***Equation 4: Unstable Simulation Equation***. In this simulation, the value $\boldsymbol{a}$ is equal to -2. Therefore, $\boldsymbol{|ha + 1| = |0.5(-2)| = 0}$. This result is not only negative but also less than 1 signifying a stable simulation.

Figure 4: Comparison of Different Integration Step Sizes also signifies that the lower limit of $\boldsymbol{h}$ is 0.01. When compared to $\boldsymbol{h = 0.001}$, the results show negligible change in the accuracy of error measurements. Therefore, lowering the value of $\boldsymbol{h}$ beyond 0.01 would likely offer diminishing improvements in error accuracy.

Computational efficiency is also an important consideration regarding error measurements. Whilst lower values of $\boldsymbol{h}$ undoubtably offer more accurate error measurements, there higher frequency of measurements may not be worth the meagre increased in accuracy. Furthermore, the system would need to work faster to handle the increased number of error calculations in a certain amount of time potentially overworking the system.

Value of h	$\boldsymbol{h = 0.001}$	$\boldsymbol{h = 0.01}$	$\boldsymbol{h = 0.1}$	$\boldsymbol{h = 0.25}$	$\boldsymbol{h = 0.5}$	$\boldsymbol{h = 0.75}$	$\boldsymbol{h = 1}$
$\boldsymbol{\lvert ha + 1\rvert \leq 1}$	$\boldsymbol{0.001(-2) + 1 = 0.998}$	$\boldsymbol{0.01(-2) + 1 = 0.98}$	$\boldsymbol{0.1(-2) + 1 = 0.8}$	$\boldsymbol{0.25(-2) + 1 = 0.5}$	$\boldsymbol{0.5(-2) + 1 = 0}$	$\boldsymbol{0.75(-2) + 1 = -0.5}$	$\boldsymbol{1(-2) + 1 = -1}$
Stability	Stable	Stable	Stable	Stable	Stable	Unstable	Unstable
Computational Efficiency	Not Efficient	Efficient	Efficient	Efficient	Efficient	Efficient	Efficient

Table 1: Value of h Stability and Computational Efficiency

Table 2: Error Values at Different Time Intervals with Different Values of h

Time interval (seconds)	$\boldsymbol{h = 0.001}$	$\boldsymbol{h = 0.01}$	$\boldsymbol{h = 0.1}$	$\boldsymbol{h = 0.25}$	$\boldsymbol{h = 0.5}$	$\boldsymbol{h = 0.75}$	$\boldsymbol{h = 1}$
3	$\boldsymbol{e = 1.995}$	$\boldsymbol{e = 1.995}$	$\boldsymbol{e = 1.998}$	$\boldsymbol{e = 2}$	$\boldsymbol{e = 2}$	$\boldsymbol{e = 1.875}$	$\boldsymbol{e = 4}$
5	$\boldsymbol{e = 1.992}$	$\boldsymbol{e = 2}$	$\boldsymbol{e = 2}$	$\boldsymbol{e = 2}$	$\boldsymbol{e = 2}$	$\boldsymbol{e = 0.516}$	$\boldsymbol{e = 4}$
8	$\boldsymbol{e = 1.002}$	$\boldsymbol{e = 1.002}$	$\boldsymbol{e = 1.001}$	$\boldsymbol{e = 1}$	$\boldsymbol{e = 1}$	$\boldsymbol{e = 0.97}$	$\boldsymbol{e = -2}$
10	$\boldsymbol{e = 1}$	$\boldsymbol{e = 1}$	$\boldsymbol{e = 1}$	$\boldsymbol{e = 1}$	$\boldsymbol{e = 1}$	$\boldsymbol{e = 0.992}$	$\boldsymbol{e = -2}$
12	$\boldsymbol{e = 2.964}$	$\boldsymbol{e = 2.965}$	$\boldsymbol{e = 2.977}$	$\boldsymbol{e = 2.992}$	$\boldsymbol{e = 3}$	$\boldsymbol{e = 3.251}$	$\boldsymbol{e = -2}$
15	$\boldsymbol{e = 3}$	$\boldsymbol{e = 3}$	$\boldsymbol{e = 3}$	$\boldsymbol{e = 3}$	$\boldsymbol{e = 3}$	$\boldsymbol{e = 3.016}$	$\boldsymbol{e = 8}$

Part II

Random Process

This next stage in the project involves adding random values, also known as white noise, to the $\boldsymbol{x(t)}$ values that were already recorded. To add context to the purpose of this step, the assumption that a robot using a camera with an intelligent predictive agent is made. Its purpose is locating the robot from Part 1. As the robot's camera is prone to being affected by noise experienced in the real world, to simulate this, white noise will be generated using the Box-Muller method.

The Box-Muller method generates random numbers in a normal distribution. This process uses a mean $\boldsymbol{\mu = 0.0}$ and a standard deviation $\boldsymbol{\sigma = 0.01}$. The method first generates one random number in a uniform distribution $(\boldsymbol{z1})$ between 0 and $\boldsymbol{2(\pi)}$. Standard deviation is then calculated by using $\boldsymbol{B = \sigma(\sqrt{-2(\log(random\ number\ between\ 0\ and\ 1))}})$. With the values of $\boldsymbol{B}$ and $\boldsymbol{z1}$, two other random values ($\boldsymbol{z2}$ and $\boldsymbol{z3}$) are generated.

- $\boldsymbol{z2 = B(\sin(z1)) + \mu}$
- $\boldsymbol{z3 = B(\cos(z1)) + \mu}$

With the random numbers $\boldsymbol{z2}$ and $\boldsymbol{z3}$ generated, after each iteration of the Box-Muller algorithm either the of the random variables is then added to $\boldsymbol{x(k)}$.

The advantage of using the Box-Muller method is that, unlike randomly generated numbers, the values generated accurately represent the modelled data. This method creates two normally distributed random numbers generated from one random number in a uniform distribution, the white noise generated is able to conform around the distribution of the original data. In other words, the white noise clusters around the average.

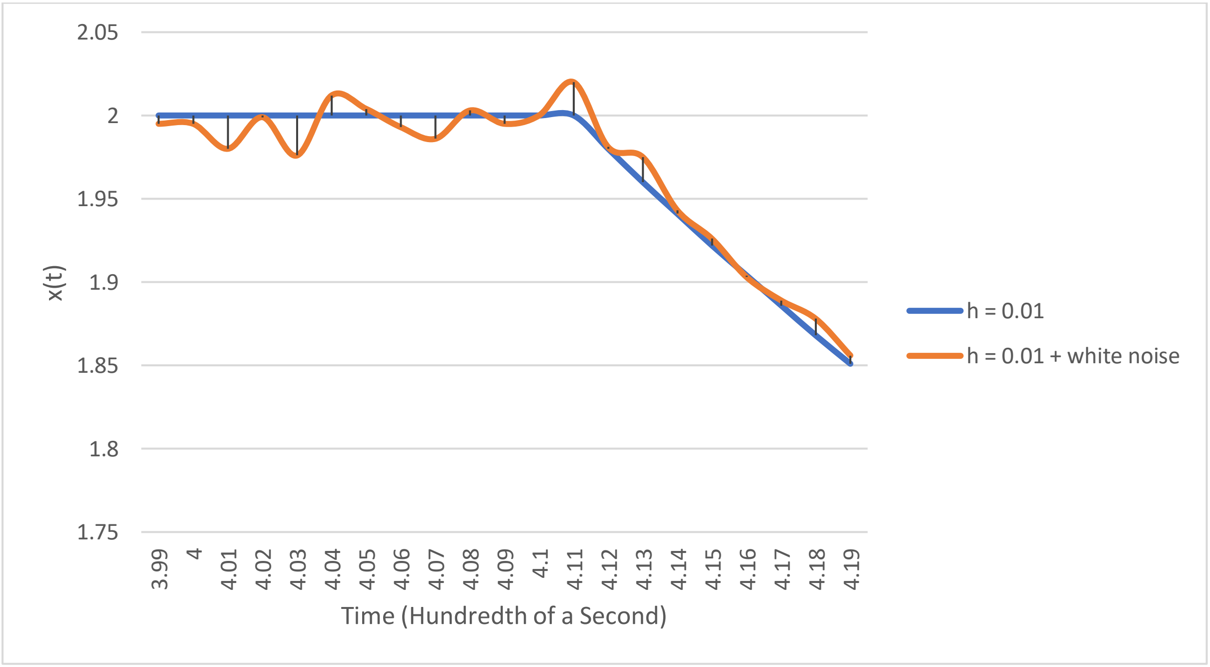


Figure 5: Sample Comparing Modelled Results to White Noise Results

Mean Value

Regarding the mean value, it is set to $\boldsymbol{\mu = 0.0}$. This is due to the purpose of the simulation being to simulate the noise of the intelligent agent's environment. Giving the mean a value would simulate the noise within the agent's system.

Part III

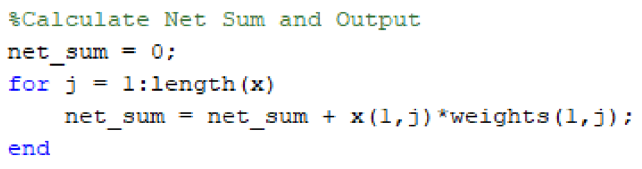
Perceptron Model

Whilst in Part I a simplified model of a robot was created to move through 1 dimensional space, in Part III, an intelligent agent will be created to predict the movements of the simplified robot model. The noise generated data (simulated sensor readings) from Part II will serve as the data set to train the agent. The data set will be normalised to allow for easier comparison with the perceptron's outputs. A perceptron model with a single neuron will be used to give the agent the capability of learning the robot's movements in order to predict its next position.

$$y(k+1) = Sigmoid\left(\sum_{i=1}^{m} b_i y(k-i)\right)$$

Equation 3: Perceptron Model

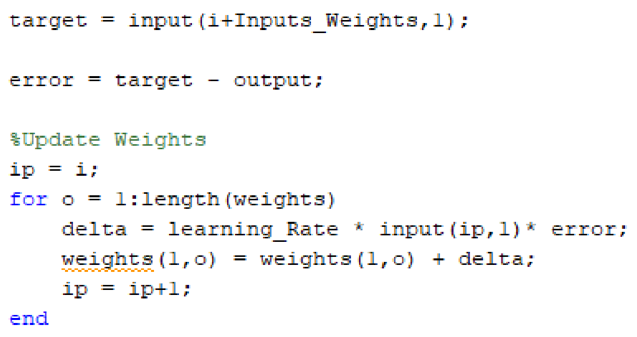
For the perceptron to begin training, the inputs are multiplied by their respective weights. These are then added together to find the net sum of the weights and the inputs.



```
%Calculate Net Sum and Output
net_sum = 0;
for j = 1:length(x)
    net_sum = net_sum + x(1,j)*weights(1,j);
end
```

Figure 6: Calculate Net

Next, the output must be generated using an activation function which returns a value between 1 and 0. With this output, the error can be found by subtracting the error from the target. With this error, the weights can be updated by adding the multiplication of the learning rate, error and the weights respective input.



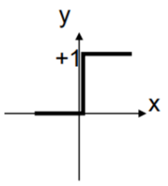
```
target = input(i+Inputs_Weights,1);

error = target - output;

%Update Weights
ip = i;
for o = 1:length(weights)
    delta = learning_Rate * input(ip,1)* error;
    weights(1,o) = weights(1,o) + delta;
    ip = ip+1;
end
```

Figure 7: Error and Weight Updating

Activation Functions

Before and training takes place, the data set is normalised, so its values are scaled between 1 and 0. This allows easy comparison between the output of the activation function and the input values.

The perceptron model uses a step function. This works by comparing the net sum, represented by x in ***Equation 4: Step Activation Function***, to a threshold. If x is greater than or equal to the threshold, then the output, y, is equal to 1. Otherwise, the output is equal to 0.



$$y = \begin{cases} 1 & if \quad x \geq 0 \\ 0 & if \quad x \prec 0 \end{cases}$$

Equation 4: Step Activation Function

```
if net_sum >= thresh
    output = 1;
else
    output = 0;
end
```

Figure 8: Step Activation Function Implementation

When using this activation function, the perceptron was not able to accurately predict the position of the robot. As the gradient of the step activation function is 0, it is less than ideal for backpropagation when the output is sent back for error calculations. This means that the perceptron cannot optimise its weights, preventing any useful learning from taking place.

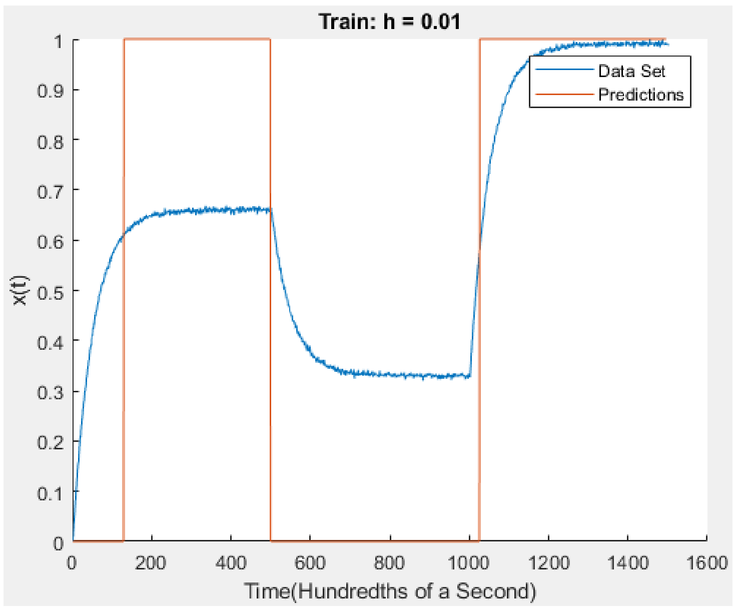


Figure 9: Step Activation Function Train

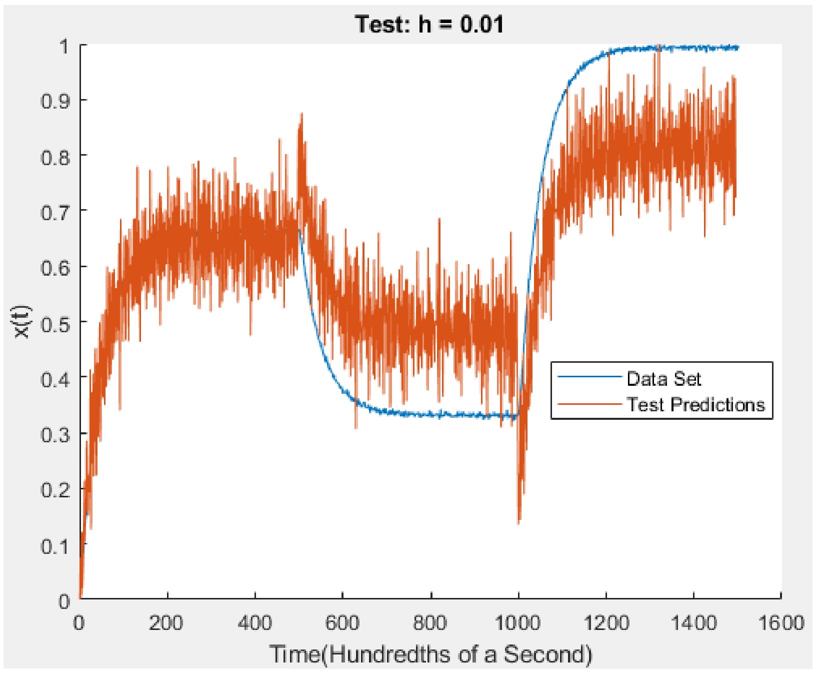
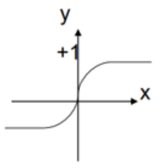


Figure 10: Step Activation

Logistic sigmoid, on the other hand, has the advantage of using a gradient based algorithm to return a real number between 1 and 0. Sigmoid returns the gradient which is the first derivative of the error^2 function. This implies that sigmoid is differentiable.



$$y = \frac{1}{1+e^{-x}}$$

Equation 5: Logistic Sigmoid

```
output = 1/(1 + exp(1)^-net_sum);
```

Figure 11: Logistic Sigmoid Implementation

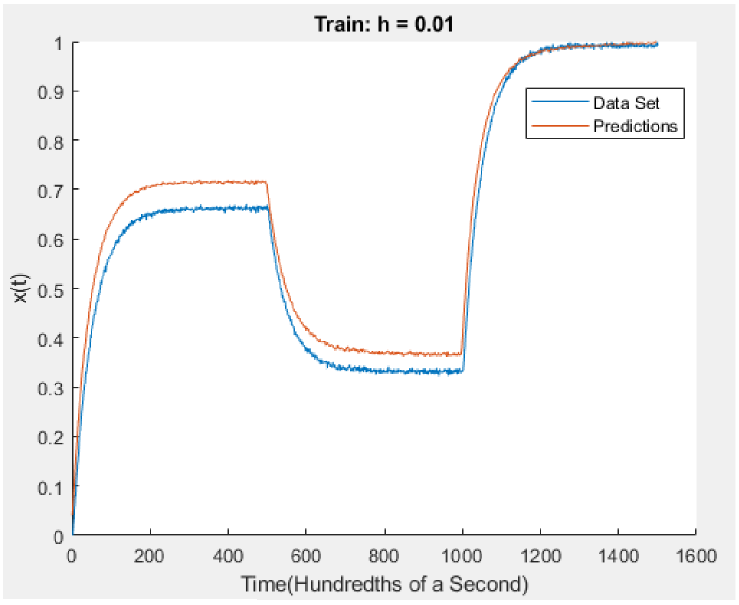


Figure 12: Sigmoid Train

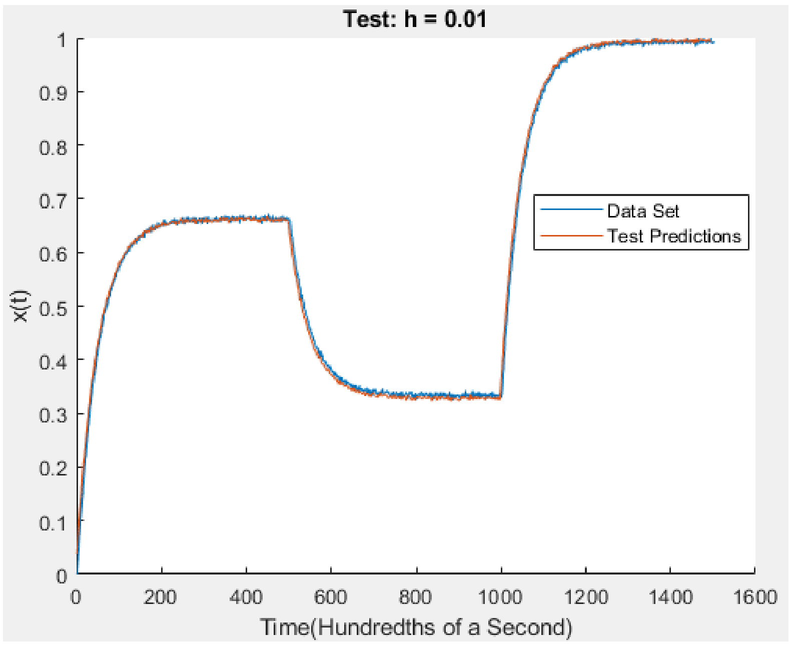


Figure 13: Sigmoid Test

Test for Unseen Data

For the perceptron to be able to test on unseen data, a new data set was generated by running the same Box-Muller code but with a different set of randomised values. Furthermore, the integration step size may have also been changed to further differentiate the test data set from the train data set. The weights for the test code are the last updated values from the perceptron's training. This ensures that the test script uses the most optimised weights for predicting.

Using a different data set for the test ensures that the perceptron tests the concepts it learned during training and can apply those concepts to data it has not seen before within the same problem domain. The better the perceptron is at doing this, the more proficient it will be at generalising. This also reduces the chances of the perceptron overfitting and underfitting.

Figure 14 and 15 show the perceptron test results that used a data set with an integration step size of $\boldsymbol{h = 0.01}$. In contrast, the training data set was generated using $\boldsymbol{h = 0.1}$. As one can see in these figures, the weights trained for one data set were able to generalise for another during testing.

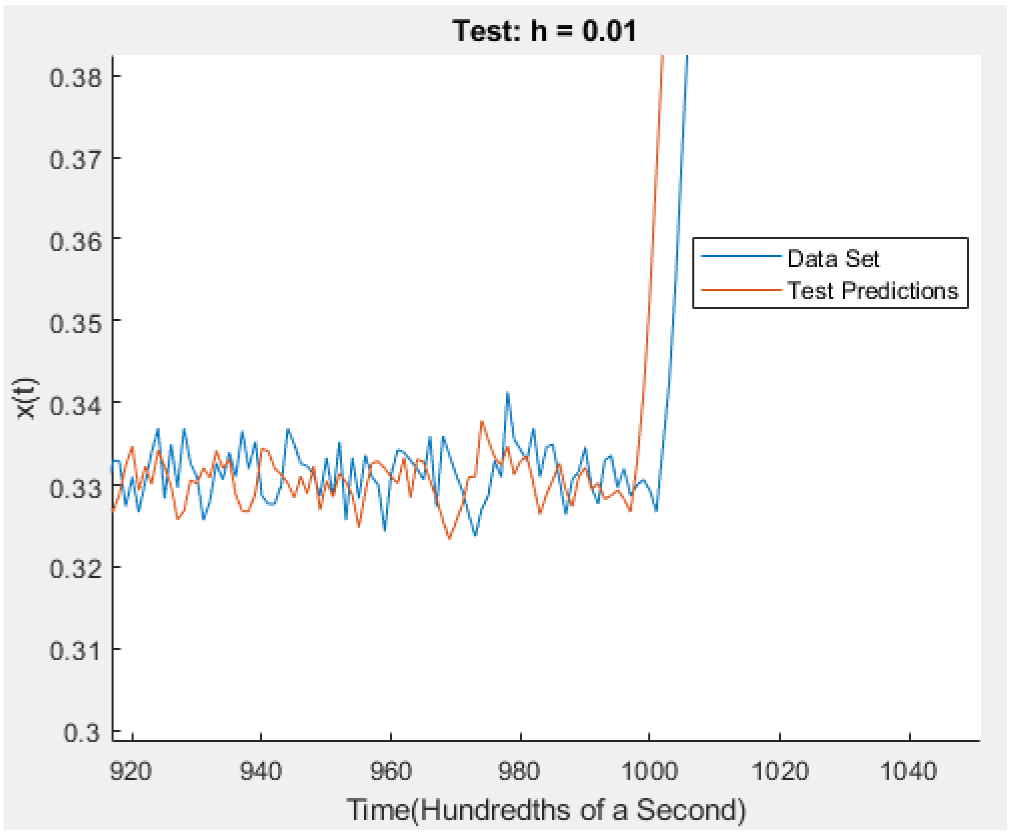


Figure 14: Test results Zoomed

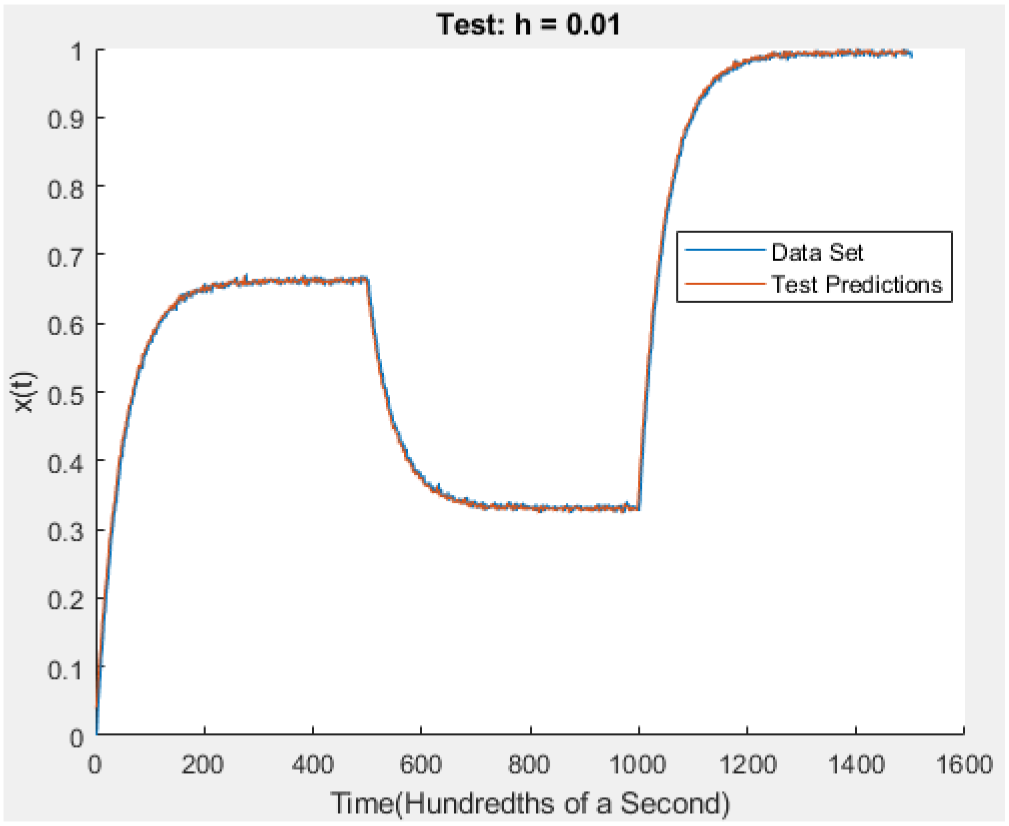


Figure 15: Test Results

Prediction Accuracy

In training and testing the perceptron it was found that as the value of $\boldsymbol{U}$ changed in the data set, there would be a gap in timing between the perceptron's predictions and the actual values. The predictions produced may have been accurate in terms of position, however a latency in time was certainly present. This is true in both the training and testing.

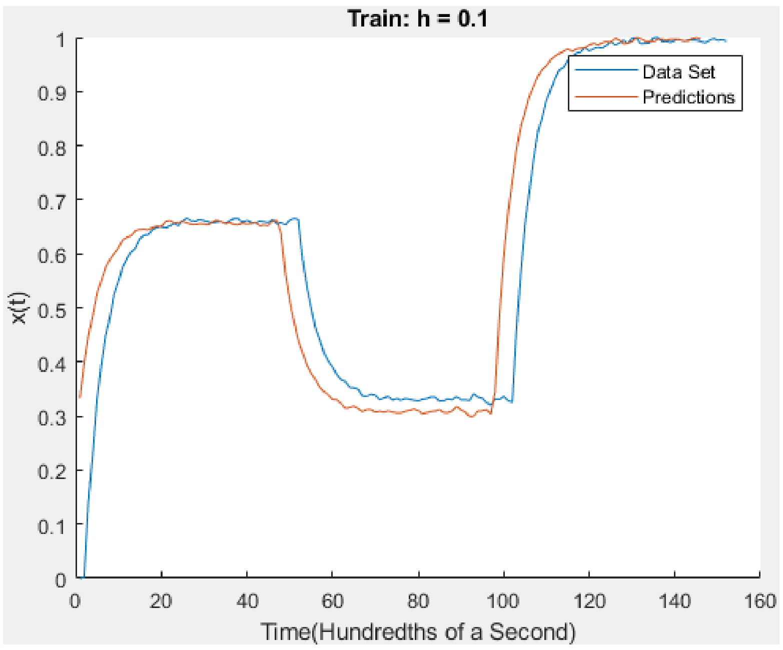


Figure 16: Latency in Training

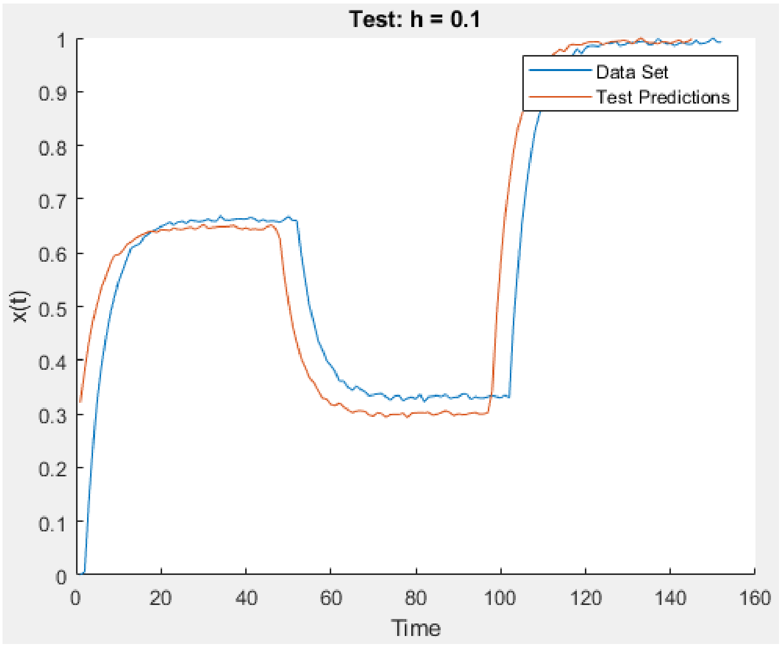


Figure 17: Latency in Testing

Increasing the value of $\boldsymbol{h}$, therefore increasing the complexity of the training data, the perceptron was able to more accurately predict the robot's location at a given time. The latency between the predictions and the actual values saw a decrease as a result. This is shown in the figures bellow where the step size was set to $\boldsymbol{h} = \mathbf{0.01}$ as opposed to the data shown in figure 16 and figure 17 where the step size is $\boldsymbol{h} = \mathbf{0.1}$.

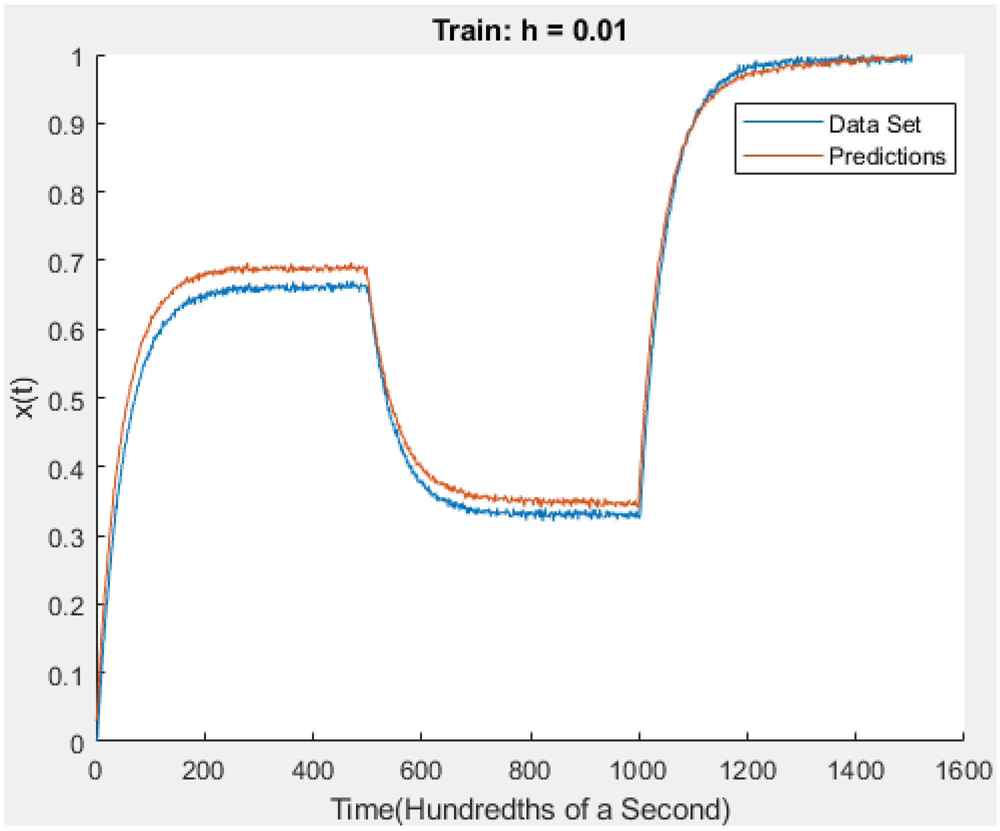


Figure 18: Latency Decrease in Training

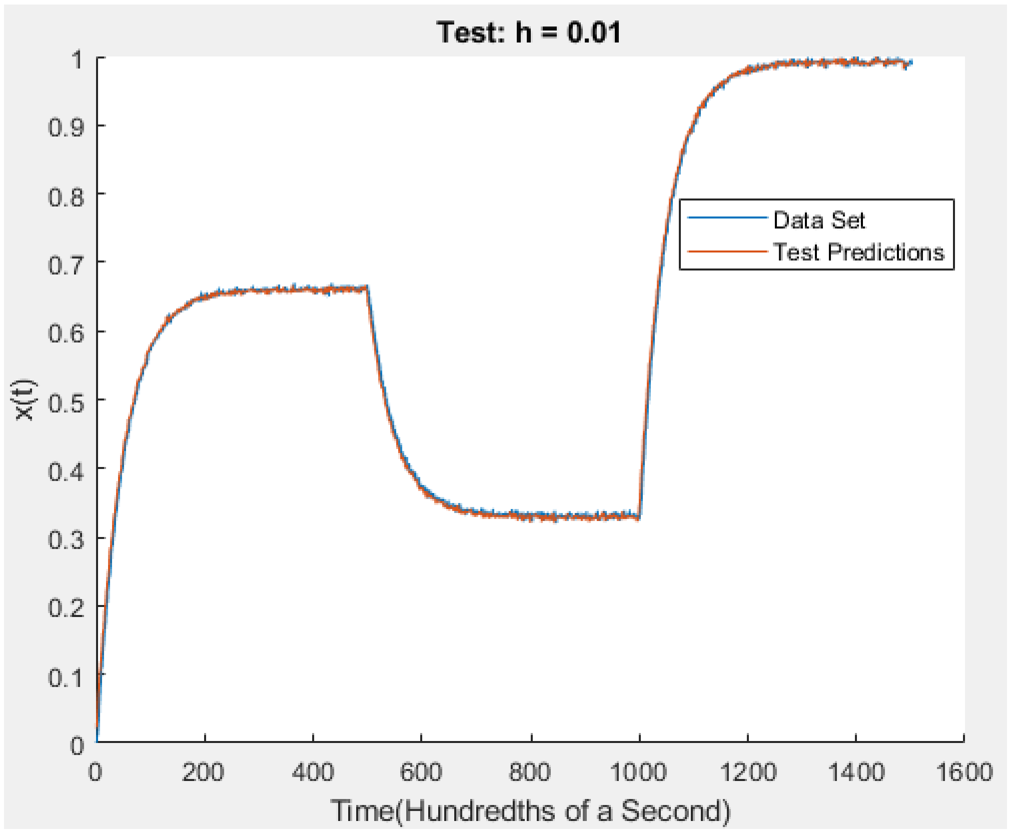
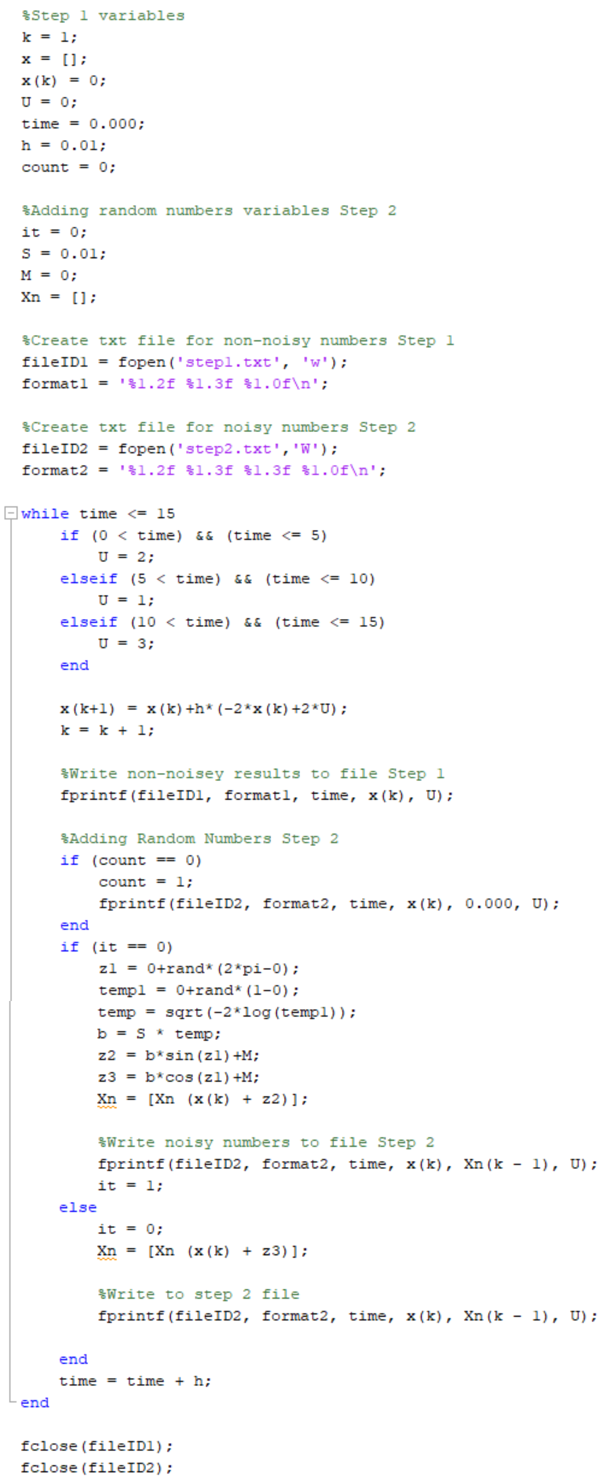


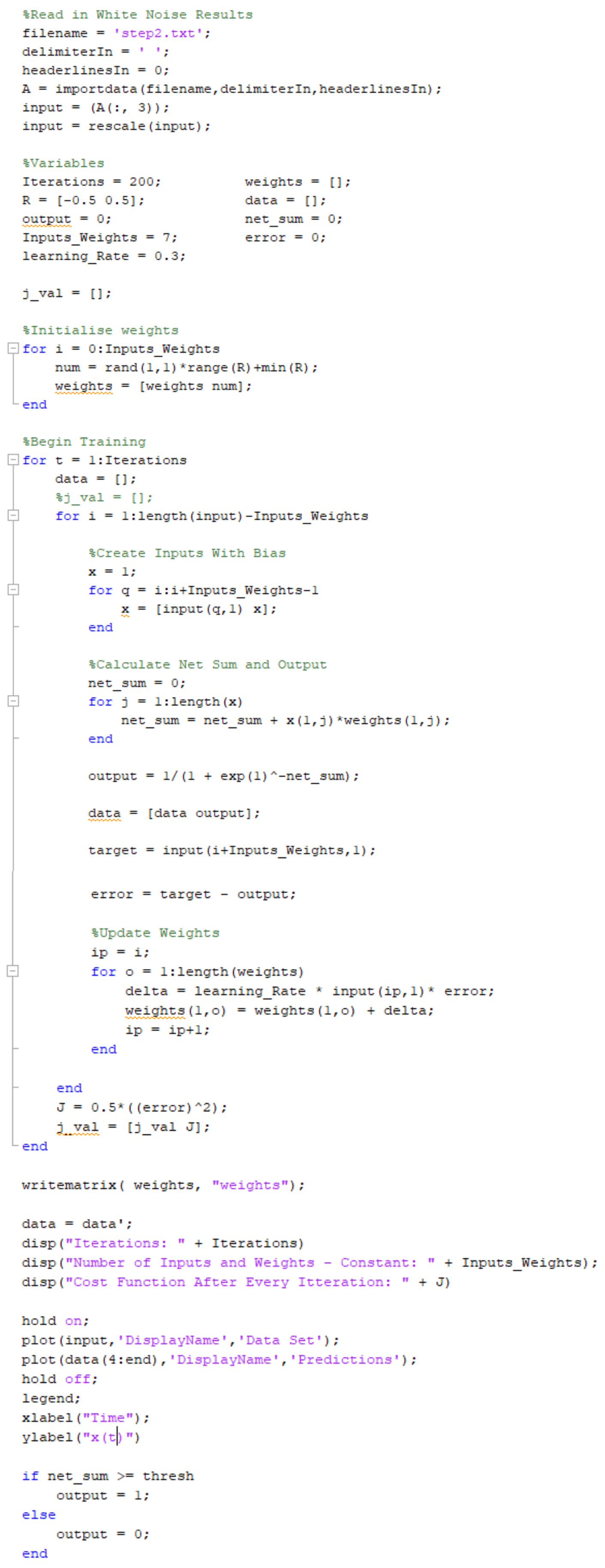
Figure 19: Latency Decrease in Testing

Appendices A Code



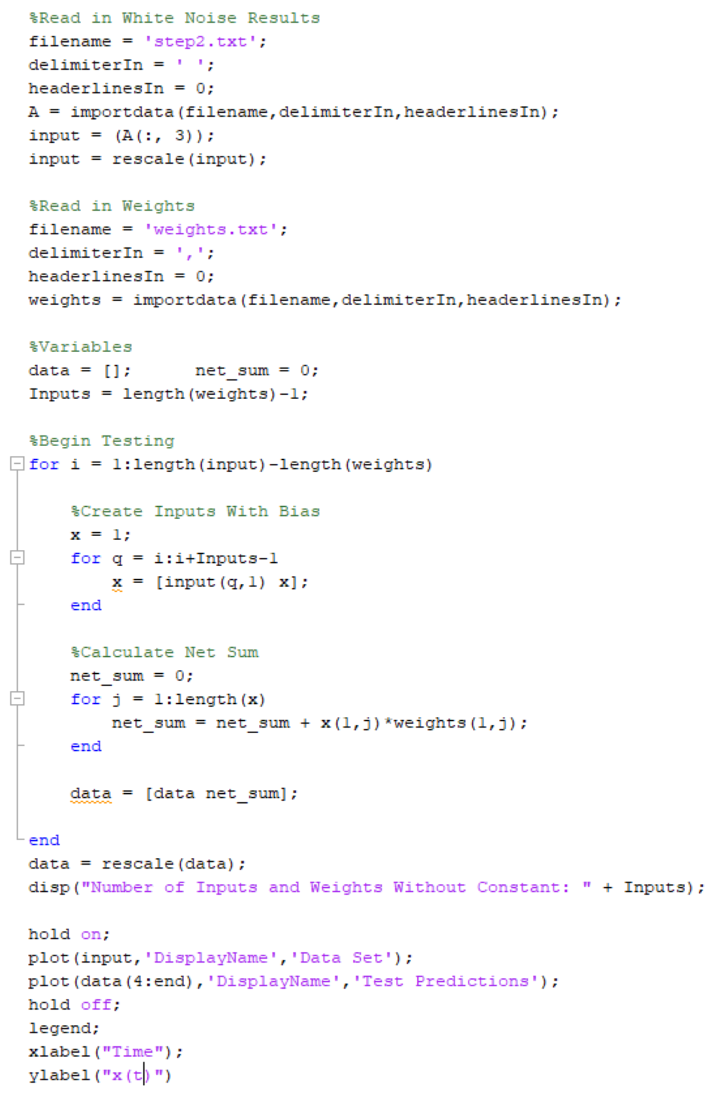
```
%Step 1 variables
k = 1;
x = [];
x(k) = 0;
U = 0;
time = 0.000;
h = 0.01;
count = 0;

%Adding random numbers variables Step 2
it = 0;
S = 0.01;
M = 0;
Xn = [];

%Create txt file for non-noisy numbers Step 1
fileID1 = fopen('step1.txt', 'w');
format1 = '%1.2f %1.3f %1.0f\n';

%Create txt file for noisy numbers Step 2
fileID2 = fopen('step2.txt','W');
format2 = '%1.2f %1.3f %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

    x(k+1) = x(k)+h*(-2*x(k)+2*U);
    k = k + 1;

    %Write non-noisey results to file Step 1
    fprintf(fileID1, format1, time, x(k), U);

    %Adding Random Numbers Step 2
    if (count == 0)
        count = 1;
        fprintf(fileID2, format2, time, x(k), 0.000, U);
    end
    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 Step 2
        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

fclose(fileID1);
fclose(fileID2);
```

Figure 20: Code for Simulating Robot Movements and Adding White Noise



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

%Variables
Iterations = 200;          weights = [];
R = [-0.5 0.5];            data = [];
output = 0;                net_sum = 0;
Inputs_Weights = 7;        error = 0;
learning_Rate = 0.3;

j_val = [];

%Initialise weights
for i = 0:Inputs_Weights
    num = rand(1,1)*range(R)+min(R);
    weights = [weights num];
end

%Begin Training
for t = 1:Iterations
    data = [];
    %j_val = [];
    for i = 1:length(input)-Inputs_Weights

        %Create Inputs With Bias
        x = 1;
        for q = i:i+Inputs_Weights-1
            x = [input(q,1) x];
        end

        %Calculate Net Sum and Output
        net_sum = 0;
        for j = 1:length(x)
            net_sum = net_sum + x(1,j)*weights(1,j);
        end

        output = 1/(1 + exp(1)^-net_sum);

        data = [data output];

        target = input(i+Inputs_Weights,1);

        error = target - output;

        %Update Weights
        ip = i;
        for o = 1:length(weights)
            delta = learning_Rate * input(ip,1)* error;
            weights(1,o) = weights(1,o) + delta;
            ip = ip+1;
        end

    end
    J = 0.5*((error)^2);
    j_val = [j_val J];
end

writematrix( weights, "weights");

data = data';
disp("Iterations: " + Iterations)
disp("Number of Inputs and Weights - Constant: " + Inputs_Weights);
disp("Cost Function After Every Itteration: " + J)

hold on;
plot(input,'DisplayName','Data Set');
plot(data(4:end),'DisplayName','Predictions');
hold off;
legend;
xlabel("Time");
ylabel("x(t)")

if net_sum >= thresh
    output = 1;
else
    output = 0;
end
```

Figure 21: Code for Perceptron Training



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

%Read in Weights
filename = 'weights.txt';
delimiterIn = ',';
headerlinesIn = 0;
weights = importdata(filename,delimiterIn,headerlinesIn);

%Variables
data = [];      net_sum = 0;
Inputs = length(weights)-1;

%Begin Testing
for i = 1:length(input)-length(weights)

    %Create Inputs With Bias
    x = 1;
    for q = i:i+Inputs-1
        x = [input(q,1) x];
    end

    %Calculate Net Sum
    net_sum = 0;
    for j = 1:length(x)
        net_sum = net_sum + x(1,j)*weights(1,j);
    end

    data = [data net_sum];

end
data = rescale(data);
disp("Number of Inputs and Weights Without Constant: " + Inputs);

hold on;
plot(input,'DisplayName','Data Set');
plot(data(4:end),'DisplayName','Test Predictions');
hold off;
legend;
xlabel("Time");
ylabel("x(t)")
```

Figure 22: Code for Testing Perceptron