James Duncan

Computational Science

Contents

[Introduction 2](#_Toc34933156)

[Part 1 – Euler’s Algorithm 2](#_Toc34933157)

[Part 2 – Creating Noise Using Box Muller Algorithm 4](#_Toc34933158)

[Part 3 – Perceptron learning Algorithms 5](#_Toc34933159)

[Conclusion 6](#_Toc34933160)

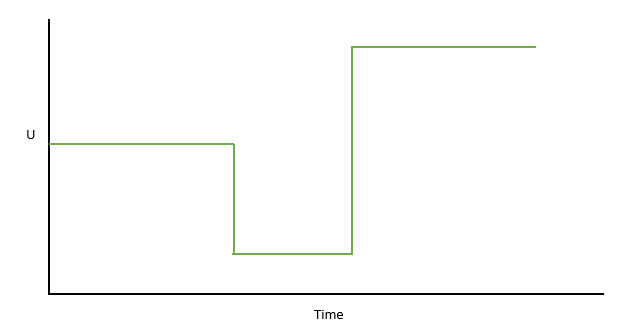
[Appendix 6](#_Toc34933161)

[Code 6](#_Toc34933162)

# Introduction

This report covers the simulation of a robot moving in 1D space across time. To simulate the movements of the robot, Euler’s method for solving differential equations at different time steps is implemented and compared against the actual result. This gives a dataset that can be used in further sections of the report. Noise is added to the robot to simulate a different robot moving in 1D space disrupting the signal of our robot. This noisy value is then passed to a machine learning perceptron which attempts to predict the correct position of the robot based on previous locations of the robot.

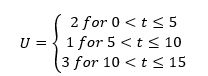
I expect the path of the robot to look something like figure 1 (Defined as *“𝑥(𝑡) = 𝑈(𝑡) − 𝑒 −2t”*) with the perceptron fitting the line but showing clear errors in the progress. I also Expect as the step size decreases and the amount of points increases the accuracy of both Euler’s and the perceptron to go increase.



*Figure 1: Expected Path of Robot*

# Part 1 – Euler’s Algorithm

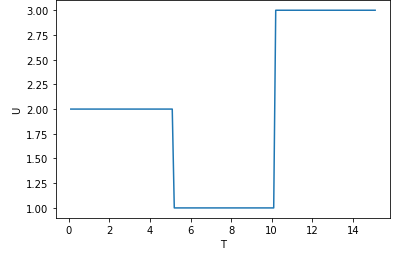
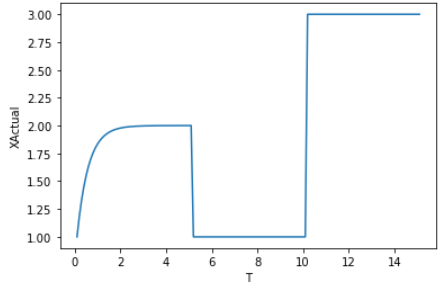
The Robot has two values that we can measure. The First of which is “U” which is the distance the robot has to travel that changes with time based on what the current time of the simulation is (details of which can be found below in figure 2) for example if the current time in the simulation is 6 we can expect “U” to be 1. Secondly the “X” Value which is a set of general co-ordinates from the origin position (usually 0,0) where “X” is on the Y Axis and time is on the X Axis. For example, at time 1 “X” could be 1.34.



*Figure 2: Changing of value U over time*

Given the actual mathematical notation (*“𝑥(𝑡) = 𝑈(𝑡) − 𝑒 −2t”)* U and X can be plotted with 100% accuracy as seen below in figures 3.

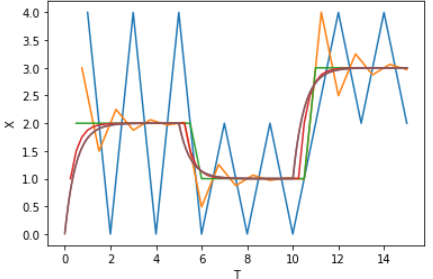
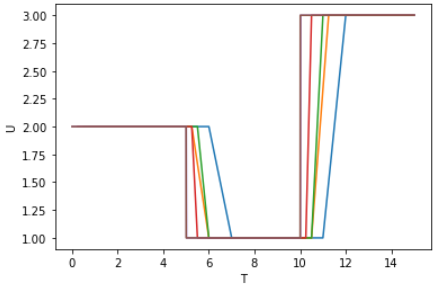
*Figures 3: Actual Graphs of X and U Variables*



In order to calculate different positions of X and U over time a method is required to do so. Euler’s Algorithm which can be found in the appendix does exactly this by calculating the value of “X” based on the previous input. The Method takes the following inputs, Time which is the maximum time , step size which is the amount of time that is incremented on until the max time (1,2,3…Time) the initial X (current X position) and initial Y (current time position) Co-ordinates.

Using these inputs, it calculates U based on the logic found in figure 2 above then uses the Euler’s equation (*f=2x+2U*) using the U and X values supplied. The method will then update the X and Step value ready for the next iteration before outputting the results (Current Time, X Calculated and U Calculated). In the code there is also a function which writes to file only a small amount of points given a number, for example if the step size was 0.01 and the integral was 0.1 it would write every 10th value.

Now that we can take different step sizes we can test how accurate the step sizes are to the real answers. This is visualised below in figures 4, however you will notice that some simulations using higher step sizes tend to be unstable suggesting the maximum step size of 0.75 with complete instability at step size 1. In these cases, they do not follow the correct pattern at all and tend to have large fluctuations in the “X” value. (The following Values were generated using the max time as 15 and initial X and Y co-ordinates set to 0 with varying step sizes)

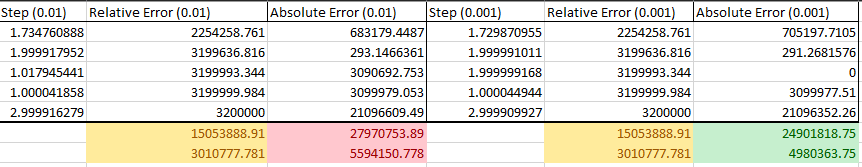
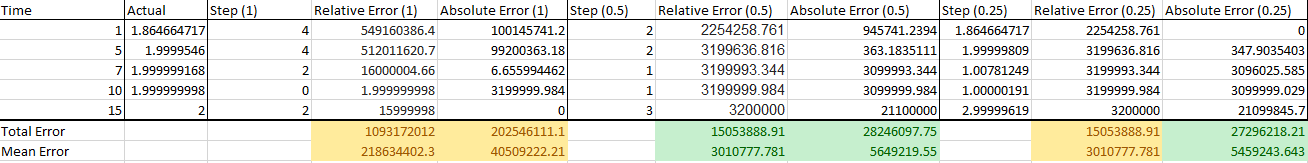


*Figures 4: Different Step sizes*

*Step Sizes*

*Blue: 1, Orange: 0.75, Green: 0.5,Red: 0.25,Purple 0.01,Brown0.001*

As the step sizes decreases the computational power for the system gets higher as you are generating more and more results as you calculate each step, but the error to the real answer seems to half for a while fading off at higher lower values of the step size. This can be measured by taking the actual values of the simulation and comparing them to the values generated by the step size. Relative and Absolute error can be used to show how much error is between these two values. This can be shown in figures 5



*Figures 5: Error of Step Sizes*

*Yellow (Initial or no change)*

*Green (Error Decreased)*

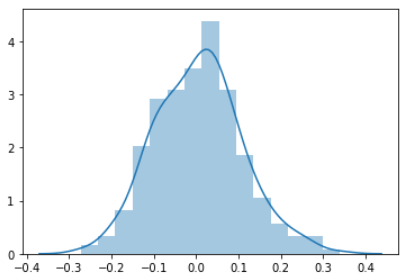
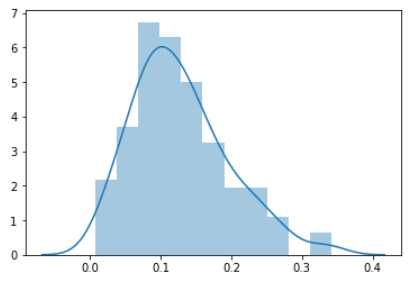
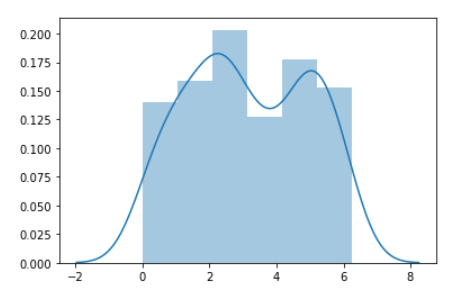
*Red (Error Increased)*

As shown above the absolute error tends to improve significantly as you lower the step size however, this falls off as you get to the lower values suggesting that the difference between the step sizes is not that big. The relative error however improves at first where the simulation goes from unstable to stable but stays stagnant though the rest of the step sizes. This suggests that the step size changing does not matter which is contradictory to both the graphs in figures 4 and the absolute error. Given that the graphs and the absolute error show improvement to decreasing step size we can assume that decreasing the step size does decrease the error at the price of computational power.

# Part 2 – Creating Noise Using Box Muller Algorithm

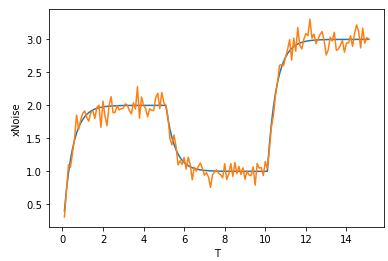
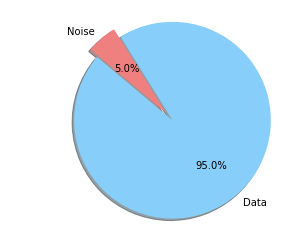
In our simulation it assumes that there is nothing disrupting the robot in its path however an addition of a secondary robot could create some disruption in our signal, this is not ideal but could happen in real life. Therefore, adding white noise will make our simulation take in that the real world is not perfect. In computational problems such as simulating the robot it is best to add noise in a gaussian/normal distribution so we can specify the mean and the standard deviation we want the numbers to have in our simulation. To do this the Box muller Algorithm can be used, code for this can be found in the appendix.

Box Muller takes first creates a random number (in the code below number between 0,2pi) which it will use later (Figure 6(Left)). The algorithm then Generates another random number between the target mean and standard deviation which forms a standard distribution (Figure 6 (Right) )that can then be used with the original set of numbers. The algorithm then maps the original set of numbers using the cos and sin functions to the standard distribution (Figure 6 (Bottom)). Due to both cos and sin functions being used to map the numbers, two numbers are generated and outputted by the algorithm. To get around this you can either feed the algorithm less numbers or half the number after the algorithm has completed ensuring to check they still remain in a standard distribution.



*Figure 6: Standard Distribution of Random Numbers*

These white noise values generated can then be added to the values of “X” to create a “xNoise” variable and depending on the standard deviation, a different percentage of noise to real “X” value can be generated. Using 0.01 standard deviation the amount of noise was not noticeable (0.5%) so the value of the target standard deviation was increased to 0.1 which returns a much more noticeable 5% noise. The impact of the noise can be shown in figures 7.



*Figures 7: Noise Impact on X*

# Part 3 – Perceptron learning Algorithms

Now that we have a set of noisy data it is now possible to use a perceptron to take the input and predict the location of the robots next position. Think of it as a camera trying to look at the robot, its always looking one step ahead trying to predict where the robot will be next to get the best shot.

Before any learning can commence it is necessary to preformat the data into tuples this is because the predictions learning on the whole dataset often leads to unstable predictions of the data especially in unseen areas of “X” (Every time U fluctuates in the graph). Limiting the data, the perceptron can see solves this problem. This table also shows how the model predicts using unseen data as it does not take the whole dataset and only bases its results on only a section of data. The tuples can be preformatted to the pattern in table 1.

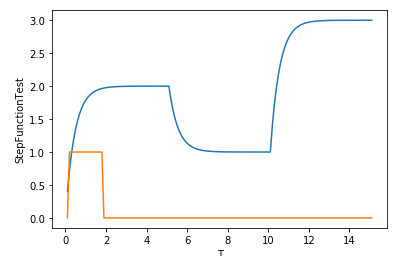
|  |  |  |  |
| --- | --- | --- | --- |
| Value 1 | Value 2 | Value 3 | Output of Perceptron |
| 0 | 0 | X(1) | Predicted X(2) |
| 0 | X(1) | X(2) | Predicted X(3) |
| X(1) | X(2) | X(3) | Predicted X(4) |
| X(2) | X(3) | X(4) | Predicted X(5) |
| X(3) | X(4) | X(5) | Predicted X(6) |

*Table 1: Tuple Format*

At first we can code a step function perceptron; this perceptron should not work for this solution as this perceptron can only return binary values. The activation function can be changed later in order to get better results; however, this perceptron can still be simulated.

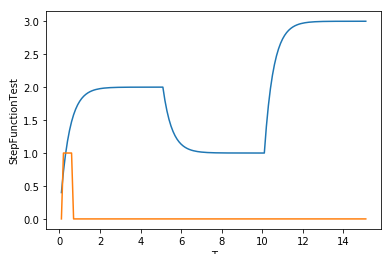
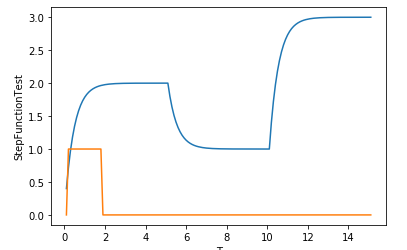
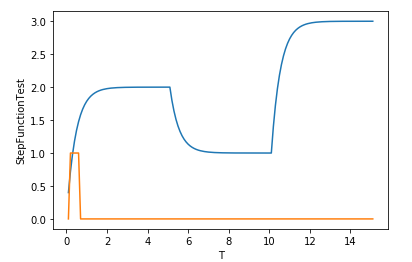
The Perceptron works by taking a few values of the data (pre-formatted into Tuples) to use as the data for each time iteration. Before any prediction is made the perceptron initialises a bias weight and a standard weight for each entry fed into the perceptron. The perceptron then loops round each Epoch (amount of iterations around the data) and around each entry of the data (each tuple) . The Perceptron will then use its current activation method (Step) and return the result of that iteration of “xNoise”. From there the perceptron will calculate the delta (Target Data (xNoise) – Predicted Value) and update each weight based on the Delta. For each iteration around the input the weights will update, and the Perceptron will “learn”.

The Step function works by setting the activation to equal the bias weight and looping round each entry in the tuple subtracting the learning rate and adding the weight to the activation and multiplying it by that value of “X”. The activation will return either 1 or 0 based on if the activation is greater than or less than 0 (threshold). The code for this can be found in the appendix and figure 8 shows the results of the step perceptron.



*Figure 8: Step Perceptron Results*

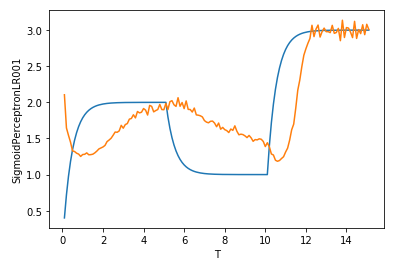
Changing the learning rate of this function doesn’t really affect the results of the data in the step function, this is shown in figures 9.



*Figures 9: Affect of Learning Rate on Step Perceptron*

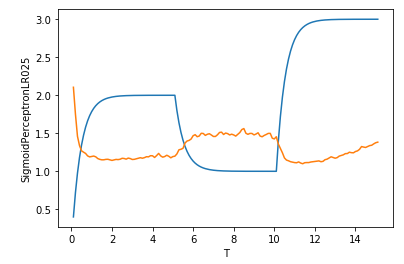
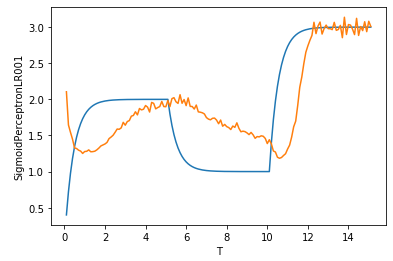
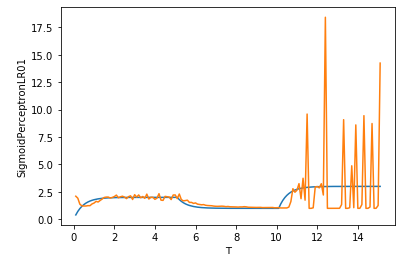
*Left (0.2) , Middle (0.01), Right (0.1)*

Replacing the Step function with sigmoid ( 1/1+exponential(-x)) allows the perceptron to calculate values between 0 and 3 inclusive allowing the perceptron space to trace the location of “x”, This function allows the perceptron to “learn” this is shown by the result in figure 10.



*Figure 10: Output of Sigmoid Perceptron*

Unlike the step perceptron the sigmoid perceptron is affected by learning rate. It seems to be that when the learning rate increases the predictions become more unstable as the weights become larger and when the learning rate is lowered the simulation becomes less accurate. This can be shown in figure 11.



*Figure 11: Affects on Learning Rate on Sigmoid Activation*

*Left(0.001),Middle (0.01),Right(0.1)*

Results of the data show a clear amount of lag between the predicted results and the actual results for the step size this means that the perceptron although its predicting the correct pattern, its not predicting very accurately.

Discuss MLP

# Conclusion

# Appendix

## Code