|  |  |
| --- | --- |
|  | Computational Science: Predicting a Robot in 1D Space |
|  | James Duncan  Word Count: 2085  201709001 |

Contents

[Introduction 2](#_Toc36542760)

[Part 1 – Euler’s Algorithm 2](#_Toc36542761)

[Part 2 – Creating Noise Using Box Muller Algorithm 4](#_Toc36542762)

[Part 3 – Perceptron learning Algorithms 5](#_Toc36542763)

[Conclusion 9](#_Toc36542764)

[Appendix 10](#_Toc36542765)

[Code 10](#_Toc36542766)

[Part 1: Euler’s Algorithm, Euler’s Equation, Method for generating actual results and Integral Calculation 10](#_Toc36542767)

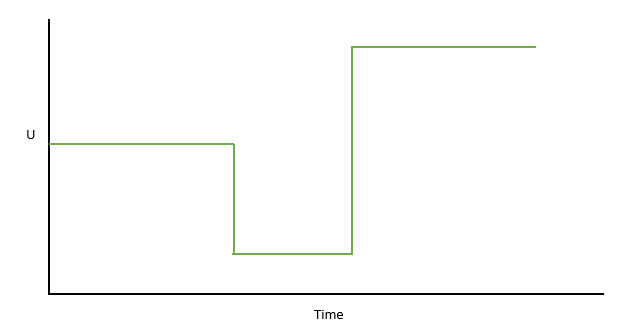
[Part 2: Box Muller Algorithm & Normalising 12](#_Toc36542768)

[Part 3: Tuple Conversion, Step Perceptron, Sigmoid Perceptron and Epoch Selection. 13](#_Toc36542769)

# Introduction

This report covers the simulation of a robot moving in 1D space across time. To simulate its movements, 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 based on previous locations of the robot.

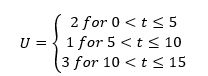
I expect the path of the robot to look something like figure 1 (Defined as *“𝑥(𝑡) = 𝑈(𝑡) − 𝑒 −2t”*) with the perceptron fitting the line but showing clear errors in the progress. I further expect as the step size decreases and the amount of points increases the accuracy of both Euler’s and the perceptron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



***Figure 1: Expected Path of Robot***

# Part 1 – Euler’s Algorithm

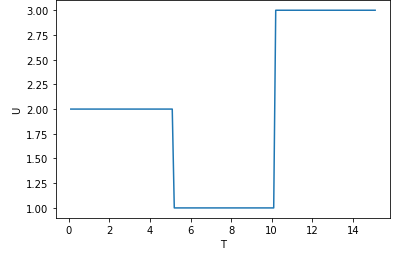
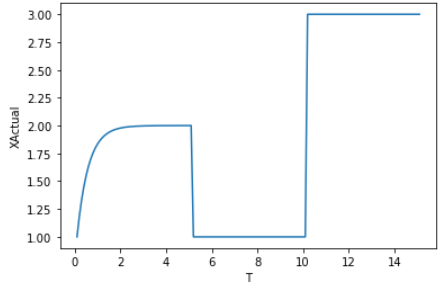
The robot has two values that we can measure. The First of which is “U” being the distance the robot has to travel that changes with time based on what the current time of the simulation is (details of which can be found below in figure 2). For example if the current time in the simulation is 6 we can 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.

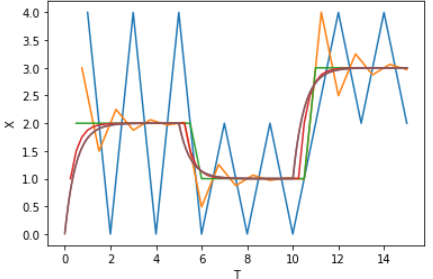
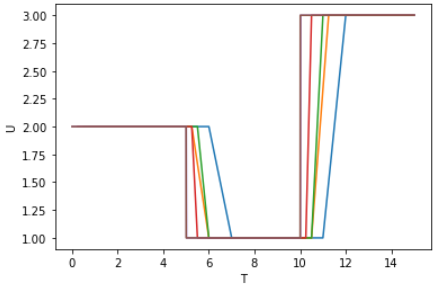
***Figures 3: Actual Graphs of X and U Variables***



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 results (Current Time, X Calculated and U Calculated). In the code there’s 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)

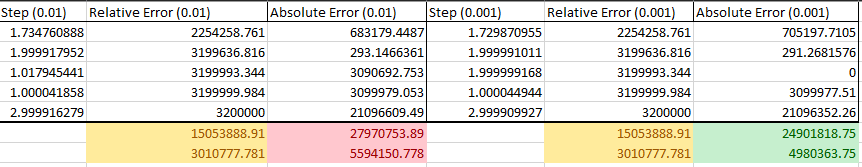
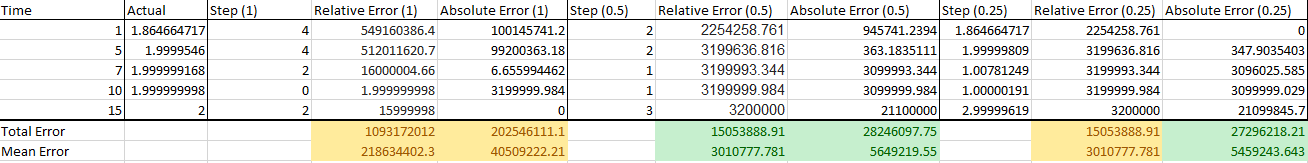


***Figures 4: Different Step sizes***

*Step Sizes*

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

As the step sizes decrease the computational power for the system increases as you are generating more and more results as each step is calculated, but the error to the real answer seems to halve 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 versus 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***

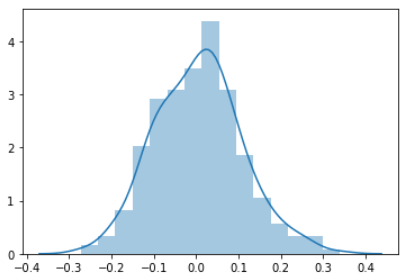
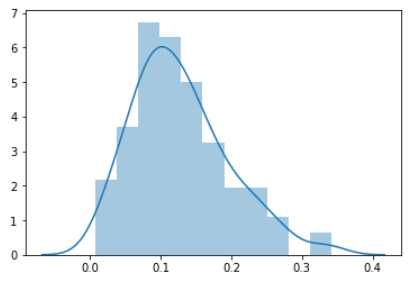
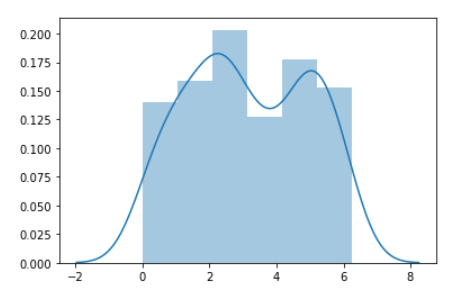
*Yellow (Initial or no change), Green (Error Decreased), Red (Error Increased)*

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 significant. The relative error however, improves at first where the simulation goes from unstable to stable - but stays stagnant though the remaining 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

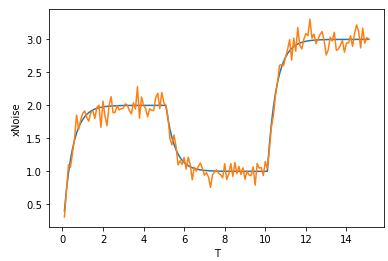
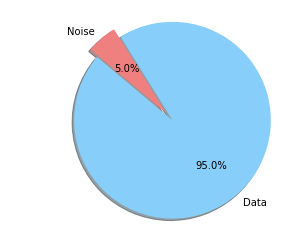
In our simulation it assumes that there is nothing disrupting the robot in its path, however, an addition of a secondary robot could create some disruption in our signal, this is not ideal but could happen in reality. Therefore, adding white noise will make our simulation take in that the real world 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 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 (Figures 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 (Figures 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 halve the number after the algorithm has completed ensuring to check they still remain in a standard distribution.



***Figures 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 robot’s next position. Consider it as a camera trying to view the robot, it’s 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 as it increases the variance of the data points. Limiting the data that 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 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***

This data should also be normalised to get the most accurate results with the step and sigmoid function otherwise the data might saturate very quickly, and the step perceptron will just cap out as most of the values predicted will be above 1.

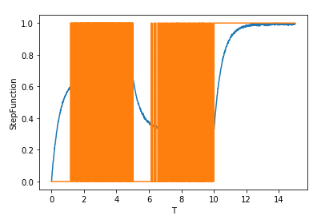
The step perceptron should not work for this solution as this perceptron can only return binary values, this is perfect for linearly separable problems but not this one. The activation function can be changed later in order to get better results; however, this perceptron can still be simulated.

The perceptron works by taking a few values of the data (a Tuple in this scenario), a learning rate and the number of epochs you want to use (amount of iterations around the data). Before any prediction is made the perceptron initialises a weight for each entry in the tuple and a bias weight randomly (between 0 and 1) as we don’t know the most efficient weight to use at the beginning. The perceptron then loops round each epoch and around each entry of the data and performs a mathematical summation of the weights and value of “X”. This value is then passed to the activation function (Step) which returns its predicted result. After the predicted result is returned the error between the actual result and the predicted result can be calculated (Delta) which is then used in parallel with the learning rate to recalculate the weight (New Weight = learning rate \* Actual \* Delta) ready for the next tuple of results. For each iteration around the inputs the weights will update, and the perceptron will “learn”. This can be shown in figure 8 which shows the flow of data through a perceptron.

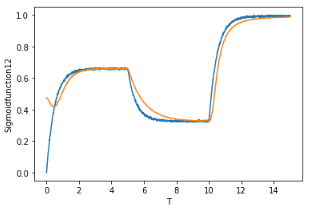


***Figure 8: Flow of Data through Perceptron***

The Step function unfortunately, as mentioned above, cannot accurately predict the correct results of the data as the output is always 0 or 1, in this simulation as the perceptron cannot output the values between 0 and 1 it fluctuates and creates a very unstable observation. (The code for this can be found in the appendix and figure 9 shows the results of the step perceptron).

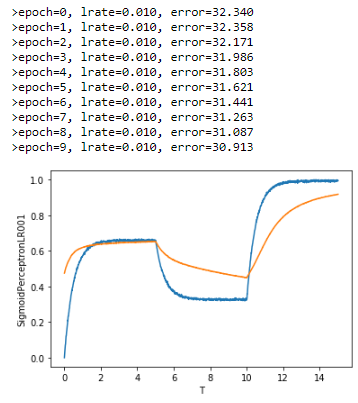
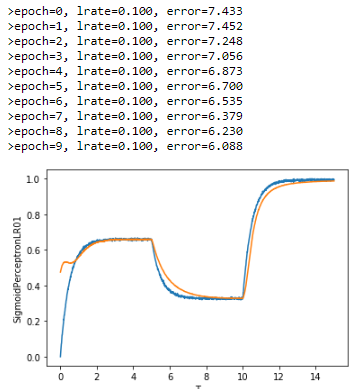


***Figure 9: Step Perceptron Results***

Replacing the step function with sigmoid (*1(/1+exponential(-x)*) allows the perceptron to calculate all values on a linear curve meaning the function can plot, 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***

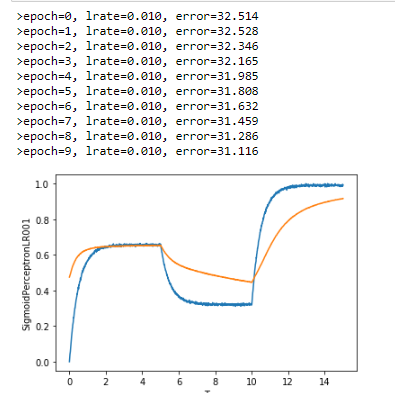
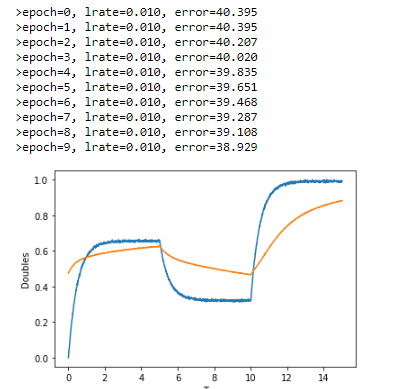
The sigmoid perceptron is affected by a changing learning rate, it seems to be that when the learning rate increases the error per epoch reduces as a result of the weights changing to fit the line more closely. However, if the learning rate is too high it can result in the model overfitting which is where the predictions fit to the noise. This is undesirable because the model will ignore any additional data points that may fall outside of this line which can happen in the real world. This can be illustrated in figures 11.



***Figures 11: Effects on Learning Rate on Sigmoid Activation***

*Left(0.001),Middle (0.01),Right(2)*

Results of the data can show a clear amount of lag between the predicted and actual results for the step size (learning rate 0.001) this means that the perceptron although it is predicting the correct pattern, its not predicting very accurately. This can be solved by tweaking the learning rate further or by increasing the epochs for the simulation. Increasing the learning rate allows more computational efficiency as it will take less epochs to get an accurate result. Additionally, adding more data gives a more accurate result with fluctuations lowering the higher amount of data you put in. This makes sense as the perceptron has more data to base its prediction on but not too much to choke the simulation, this can be shown in figures 12 where lowering the amount of data increases the error.



***Figures 12: Error Increase as less data supplied (Right: Tuples Left: Doubles)***

The perceptron’s coded can work with any activation function leading to a level of generalisation but it can also take any amount of data assuming you initialise the weights and make small changes. It also might be possible to plot velocity alongside the position of x by taking the difference between the values of x positions. A position that has not changed that much might be a result of the robot heading in the same direction which may increase the speed of the robot, on the other hand a position that has changed a lot might mean the robot had to slow down for a turn. Velocity could be predicted using a new set of data and another perceptron or by using a multiple layer perceptron which would increase the accuracy and allow for more complciated problems such as plotting the velocity.

# Conclusion

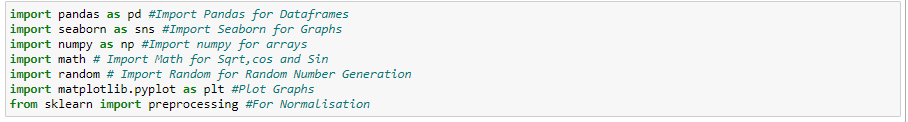
In conclusion the simulation went as expected with the robot following a linear path and the neuron (with sigmoid activation) making an accurate prediction in terms of the trend. Although, there is some error with the simulations it is necessary to ensure we don’t overfit. If there is a lag it can be improved by adjusting the learning rate of the simulation or increasing the epochs.

# Appendix

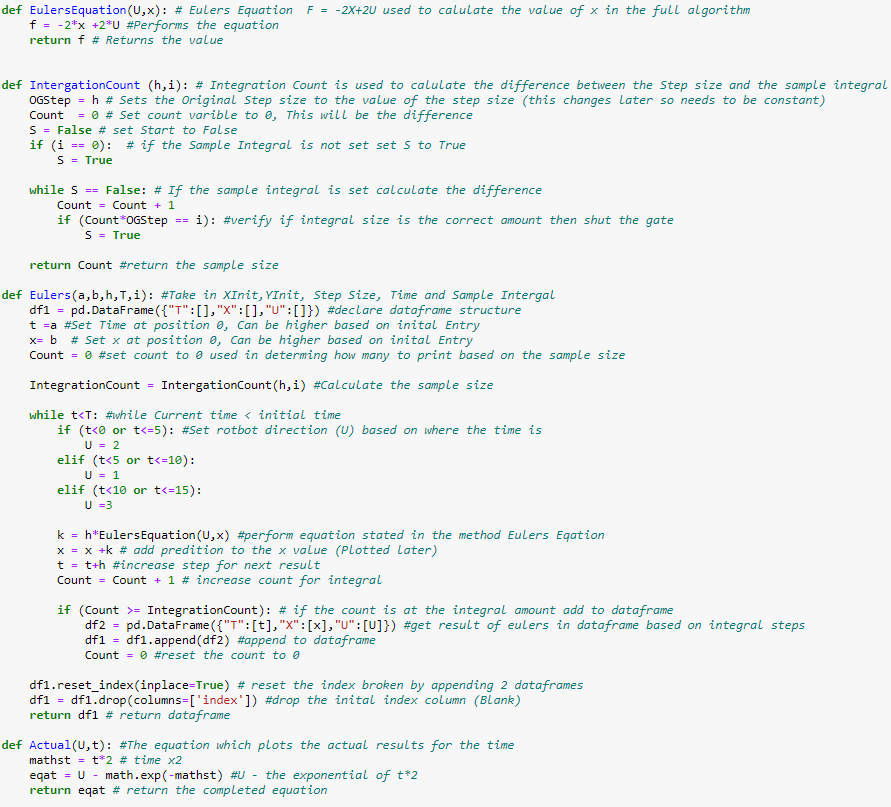
## Code

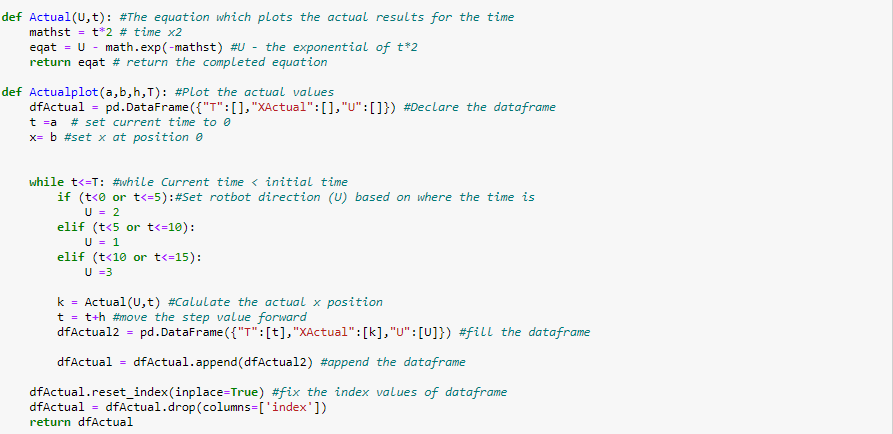
### Part 1: Euler’s Algorithm, Euler’s Equation, Method for generating actual results and Integral Calculation

#### Imports

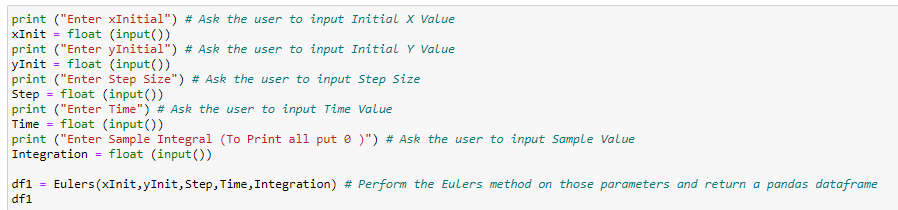


#### Methods

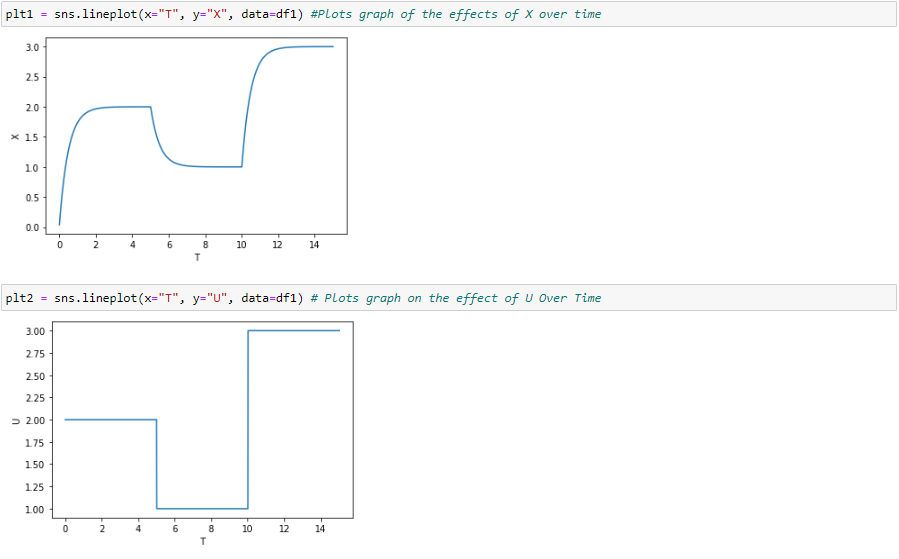




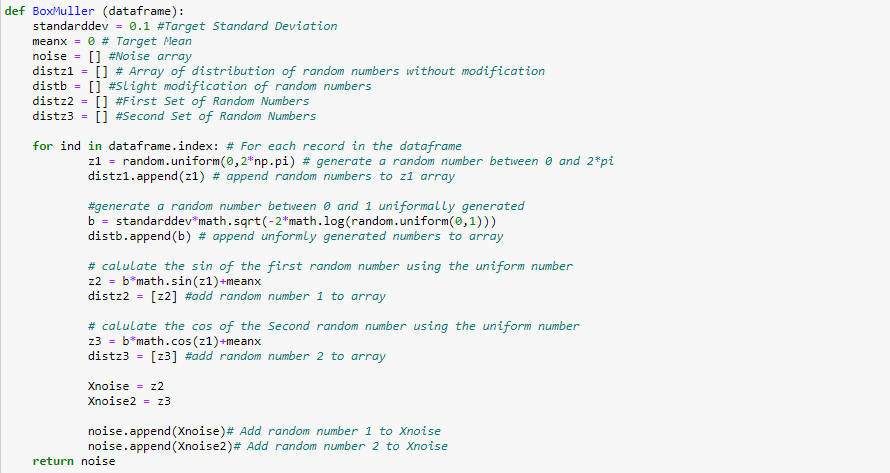
#### Ask for datapoints and generate data



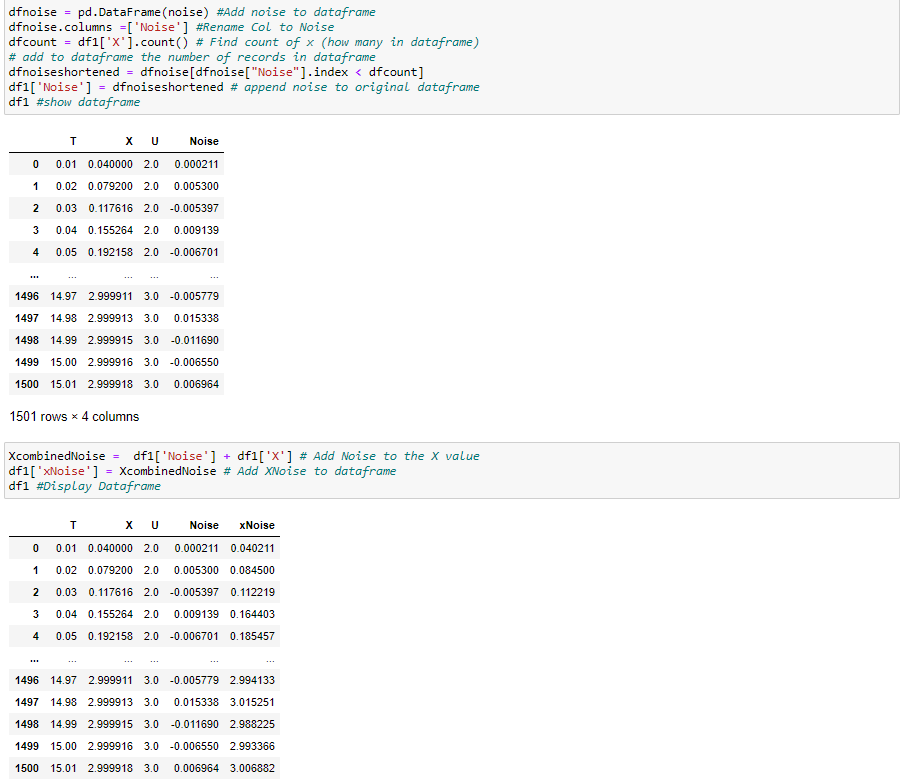
#### Plot data



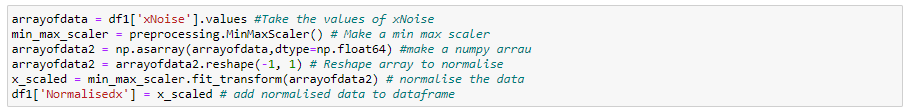
### Part 2: Box Muller Algorithm & Normalising



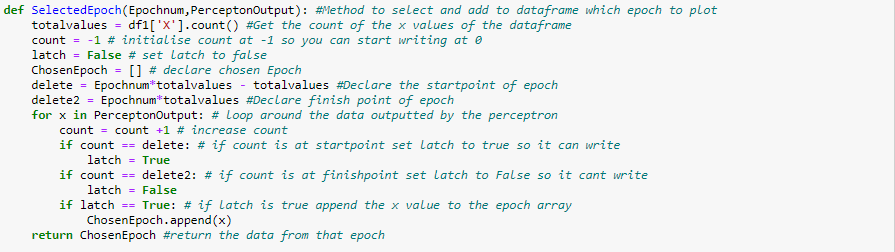
#### Append Noise to Data frame deleting additional records



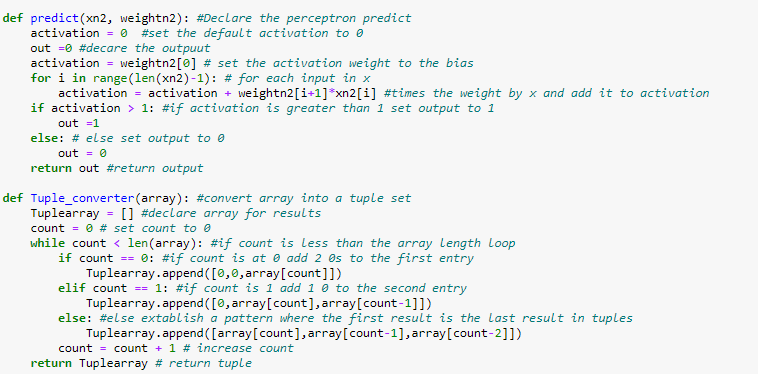
#### Normalise Data



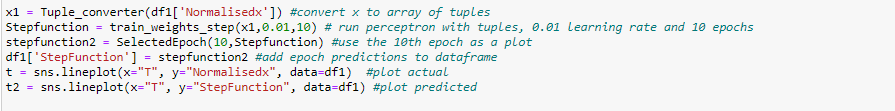
### Part 3: Tuple Conversion, Step Perceptron, Sigmoid Perceptron and Epoch Selection.



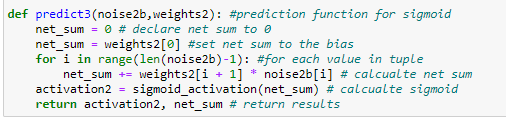




#### How to use defined step perceptron and functions







#### How to use sigmoid perceptron

