Indian Institute of Technology
Gandhinagar

ES 332 Control Theory

Course Project

Performing Complex Control using Spiking Neural Network

Authors:
Anand Yadav 15110015
Arik Pamnani 15110032
Gandhi Meet Bankim 15110049
Ravi Shrimal 15110102
Shivdutt Sharma 15110125

Supervisor:
Prof. Babji Srinivasan

April 26, 2018

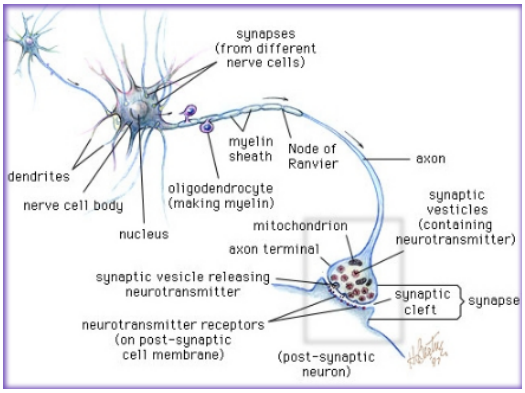
Abstract

Spiking Neural Networks (SNN) are third generation Neural Networks which were designed to simulate the working of human brain. The main difference between the earlier generation Neural Networks and SNN is that SNN uses temporal features for training whereas the earlier Neural Networks used spatial features for training. This makes SNN a good candidate for simulation of various complex phenomena. This document explains different models of Spike Generation in an SNN, the approach of Spike Time Dependent Plasticity for training an SNN and also present a simulation of a complex control system.

1 A Neuron and its Models

1.1 A Biological Model of the Neuron

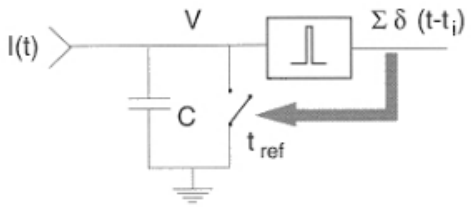
- In a neuron cell there are axons and dendrites.
- There is a transmission of ions which takes place through walls of neurons. There are ions that can pass through semipermeable wall of the cell. In this case these are sodium and potassium ions. They are called neurotransmitters.
- The above ions are received at dendrites and released at axons. The accumulation of these neurotransmitters create potential at the cell membrane boundary.
- When the potential exceeds threshold voltage then it releases an spike. And transmit signal from neuron to other.



http://www.terrapsych.com/neuron.jpg

1.2 Integrate and Fire Model

Almost similar dynamics can be realised using some electrical components. One of the very famous spiking neuron model is integrate and fire model. This is implemented using a capacitor, switch and an spike generating box.

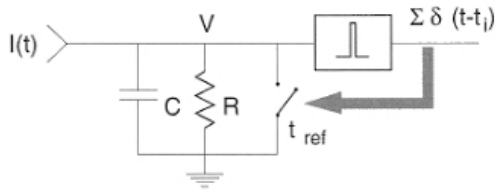


- Here the input is a current source. The capacitor resembles a charge storing cell membrane. With time the current is stored in capacitor in form of charges. Thus its voltage increases with time.
- When its voltage exceeds some threshold value then the rectangular box generate an impulse. At the same instant it turn on the switch thus all potential difference across the capacitor goes to zero and charges are neutralised. After that witch is turned off and again the capacitor starts charging. Thus it repeat periodically.
- The dynamics of the system is given by -

$$C\frac{dV}{dt} = I(t)$$

1.3 Leaky Integrate and Fire Model

The integrate and fire in not a typical model and doesn't exist in real life. In the real life there is also leakage of charges while charging of the cell membrane. And actually that corresponds to leakage resistance of the capacitor. So, the this topology includes a resistance in parallel with the capacitor. Thus while charging it leaks charges too. Understanding is same as the integrate and fire topology.



- The governing equation is -

$$C\frac{dV}{dt} + \frac{V(t)}{R} = I(t)$$

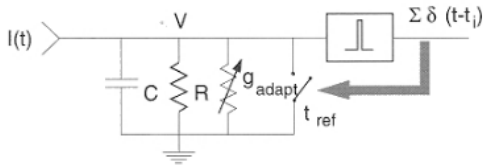
- Assuming zero initial charge on the capacitor. The voltage of the capacitor comes out to be in time domain as:

$$V(t) = RI(t)(1 - e^{\frac{-t}{\tau}})$$

- All these dynamics are discussed for a single neuron. Here the input is taken as external input. But if these neurons in neural network the there will be another input that will be sum of output of previous layer. So the dynamics equation there will be another term that will be introduced in next part in Hodgkin-Huxley model.

- There is one enhanced version of this model in the same thread called adapting integrate and fire model.

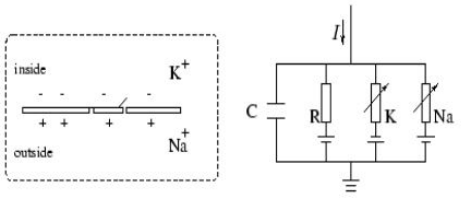
- In this model the value of resistance in parallel and switch are dynamically operated. Thus having more control over these spikes can be generated for better response. Since spiking and neural network in temporal rather than spacial. Thus the adapting integrate and fire model is more useful and has better performance.



2 Hodgkin-Huxley Model

2.1 Introduction

- A semipermeable membrane separates the interior of the cell from the extracellular liquid and acts as a capacitor.
- An input current $I(t)$ is injected into the cell.
- Adds some charge on the capacitor or leaks through the channels in the cell membrane.



2.2 Model parameters and variables

- $0.04v^2 + 5v + 140 - u + I$
- $u' = a(bv - u)$
- With the auxiliary after spike resetting
- If $v \geq 30$ mV, then
- $v = c$
- $u = u + d$
- Here, v and u are dimensionless variables, and a,b,c and dimensionless parameters and $' = d/dt$, where t is the time. The variable v represents the membrane potential of the neuron and u represents a membrane recovery potential variable, which accounts for the activation of K+ ionic currents and inactivation of Na+ ionic currents, and it provides negative feedback to v

2.3 A spiking neuron model

A neuron is a particular type of cell in our body and like all cells it is surrounded by a thin membrane. This membrane controls what goes in and out of the cell. There is a lot of water both in and outside the cell and this there are all kind of different substances. But for our model we will consider only:

- Sodium
- Potassium
- Concentration of sodium ions is higher outside the cell than inside the cell.
- The cell membrane is impermeable to sodium ions. But there are specific channels(specific to sodium ions) that allow sodium ions to go through.
- If channels(specific to sodium ions) are open $\rightarrow$ sodium ions will flow inside $\rightarrow$ increase in voltage.
- Concentration of potassium ions is higher inside the cell than outside the cell. The cell membrane has channels specific to potassium ions.
- If channels(specific to potassium ions) are open $\rightarrow$ potassium ions will flow outside $\rightarrow$ decrease in voltage.

Hodgkin and **Huxley** in 1952 described some properties of these channels.

- When cell is at rest , both types of channels are mainly closed.
- We know opening of sodium channels leads to increase in voltage, but the reverse is also true, increase in voltage $\rightarrow$ opening of channels. It is a positive feedback loop and could lead to things getting out of control.

- At rest , positive feedback loop is not activated. There is a threshold which must be reached in order to trigger the positive feedback loop. If the voltage is below the threshold, the voltage goes back down to resting levels but somehow if you manage to kick the voltage above threshold then positive feedback loop will be triggered.

2.4 Dynamical system model of a neuron((only considering sodium ions now))

- State variable : v(t) = voltage of neuron at time t.
- Resting potential = -70mV (voltage inside is lower)
- But for simplicity we will consider resting potential equal to 0.
- Maximum voltage = 1mV
- Threshold voltage = a
- Requirement for model
- If v < a, then v $\to$ 0
- If v > a, then v $\to$ 1

2.5 The Model

$\frac{dv}{dt} = f(v)$ We want stable equilibrium at v = 0 and v = 1 and unstable equilibrium at v = a between them. $f(v)$ should be positive for a<v <1 Simplest form of f(v) $f(v) = -v(v-a)(v-1)$
So far we have built a one dimensional autonomous differential equation model that just incorporates the effects of sodium. Now considering potassium channels.
Potassium channel opens $\to$ voltage inside the cell decreases. Potassium channels tend to open when the voltage inside the cell increases. In this case, since potassium channels bring the voltage down, the potassium channels act as a negative feedback.
Potassium channels differ with sodium channels in one more way

- Sodium channels are fast, opening quickly in response to increased voltage.Potassium channels are a bit more sluggish. When the voltage increases, potassium channels take their time in opening allowing the fast positive feedback of the sodium channels to cause the voltage to shoot up quickly. A millisecond later potassium channels open up $\to$ voltage goes back down.

Another state variable- Number of open potassium channels $\approx$ w(t) $\frac{dw}{dt} = v - w, parameter > 0$ As potassium channels are small $\rightarrow \frac{dw}{dt}$ should be small $\rightarrow$ small positive number
To complete the negative feedback effect of the potassium channels. We have to include in our equations the influence of potassium on voltage.
Remember, when the potassium channels open, they cause the voltage to go down.
$\frac{dv}{dt} = -v(v-a)(v-1) - w$
$\frac{dw}{dt} = \epsilon(v - \gamma w)$

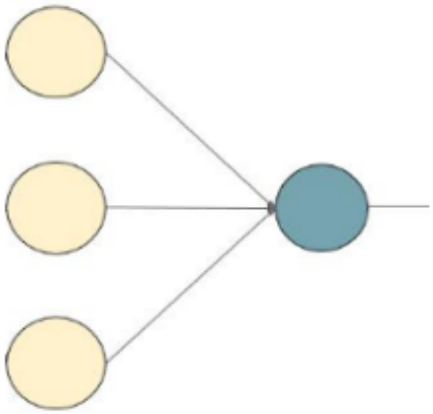
These equations for the activity of a neuron are called Fitzhugh - Nagumo equations , a simplified versions of the Hodgkin - Huxley equations.

3 Spike-Time Independent Plasticity Algorithm (STDP)

3.1 Introduction

- STDP Algorithm is used to "train" our SNN.
- Our network is composed of several layers. Each layer consists of a certain number of neurons. Neurons of different layers are connected to each other through synapses.
- These synapses have a weight associated with them (we will soon see the significance of the weight later).
- STDP Algorithm essentially updates the weight associated with each synapse. This is analogous to the Back Propagation Algorithm (BPA) that is used to train conventional neural networks.
- Infact, the human brain also uses the STDP algorithm to modify the strength of connections between neurons.

3.2 The Algorithm



https://www.learnopencv.com/understanding-feedforward-neural-networks/

- The above image shows 2 layers of a neural network. 3 neurons of one layer are connected to 1 neuron of the other layer by 3 synapses.
- Each of the 3 neurons in the 1^{st} layer (let us call them presynaptic neurons) fires spikes at a certain rate. These spikes are sent to the neuron in the 2^{nd} layer (let us call this postsynaptic neuron). (This is similar to the forward propagation algorithm in conventional neural networks).
- However, the intensity with which the presynaptic neurons send the spikes depends on the strength of the connection (weight of the synapse) they have with the postsynaptic neuron.
- The membrane potential of the postsynaptic neuron increases as it receives spikes from the presynaptic neurons. As soon as the membrane potential of the postsynaptic neuron reaches a certain threshold, it too fires a spike.
- Our STDP algorithm keeps a check on those presynaptic neurons which contribute to the firing of the postsynaptic neuron and strengthen the corresponding synapse (or, increase its weight).

- Those presynaptic neurons whose time of spike is close to the postsynaptic neuron's time of spike are given a large weight as compared to other presynaptic neurons.

3.3 Understanding STDP

- We have used STDP algorithm to train our main Spiked Neural Network (controller for tracking a trajectory).
- However, to understand the STDP algorithm better, we have tries to train it for a simple Machine Learning problem, classifying handwritten digits.
- For this task, we have used the code from - `https://github.com/Shikhargupta/Spiking-Neural-Network` This repository implements a binary classification SNN. We explain a model of a SNN which uses STDP to classify a '0' or a 'X'

4 An SNN Model to Understand STDP (Binary Classification)

Code Reference - to generate the following results - `https://github.com/Shikhargupta/Spiking-Neural-Network/`

We have the following components in the model -

4.1 Neuron

Returns a membrane potential vector of size 400, i.e for time 400ms -

- Generates a random spike train of size 400
- If current potential is greater than threshold potential, we add a spike in the current potential.

Parameters:

- Threshold Potential - If membrane potential is greater than threshold potential a spike will occur.

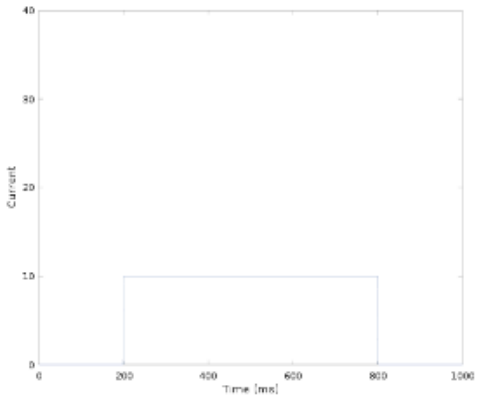
- Minimum Potential - If membrane potential is less than minimum potential, membrane potential will be set to minimum potential.
- Spike Potential - If membrane potential is greater than threshold potential, membrane potential will be sum of membrane potential and spike potential.
- Resting potential - If time is less than threshold time, membrane potential will be set to resting potential.

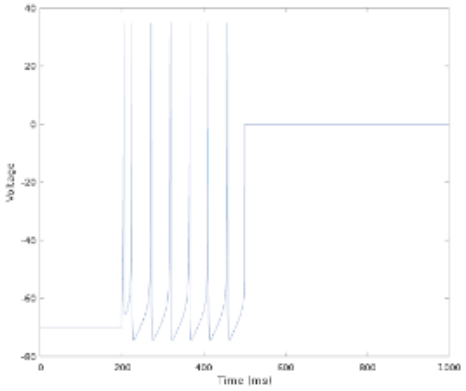
Firing a Single Neuron

- v = membrane potential
- u = recovery potential
- vth = threshold potential
- We observe the variation of single neuron for 1 sec -
  - Apply a step current between 0.2 s and 0.8 s
  - Apply a step current between 0.3 s and 0.6 s
- Hodgkin-Huxley model

  If v<vth
  $v' = .04v2 + 5v + 140 - u + I$
  $u' = a(bv - u)$
  else
  v $\rightarrow$ c
  u $\rightarrow$ u + d

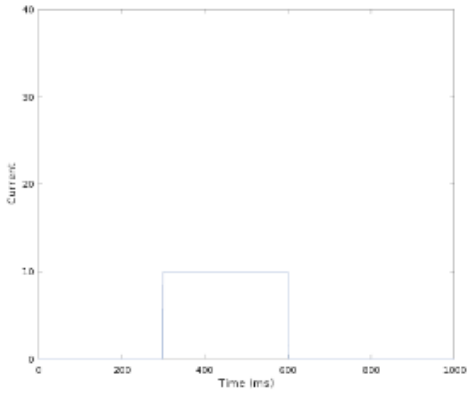
We get the following results for firing a single neuron **Code Reference** to generate the following results - `https://www.izhikevich.org/publications/spikes.pdf`

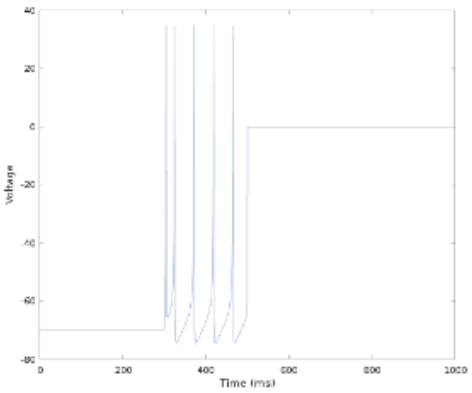




Step input (200 ms - 800 ms)

Output from a Neuron





Step input (300 ms - 600 ms)

Output from a Neuron

4.2 Receptive Field

- The receptive field of a neuron is the region in a image in which stimulus will modify the firing of the neuron.
- In the image we can take the receptive region as n x n window.
- Receptive Field -
  - Input: Image of size m x n

  - Parameters: window size c x c, weight matrix of size c x c
  - Output: return matrix 'A' of size m x n

- **Algorithm**
  - Iterate in the image with a window step
  - For each window, convolve window with weight matrix
  - Return result for each window

4.3 Encoding the Resultant matrix A

- **Encode**
  - Input: A(m x n)
  - Output: spike train vector of size (m x n, T)
  - Explanation: There are m x n neurons in the first layer. For each neuron we find pattern of neuron activation for time T.

4.4 SNN on MNIST

- Similar to Unsupervised learning - We are classifying digits using Spiking Neural Network. For this we are using MNIST dataset. Size of one mnist image is x 28. To classify the images we have 4 neurons in second layer. 10 layers to learn digits 0-9 and 11th reserved for noise. So our Spiking neural network has 2 layers. First one has 16 x 16 neurons and second one has 4 neurons.
- All ten neurons learn different patterns for different digits.
- Layer 1: 16 x 16 neurons
- Layer 2: 4 neurons

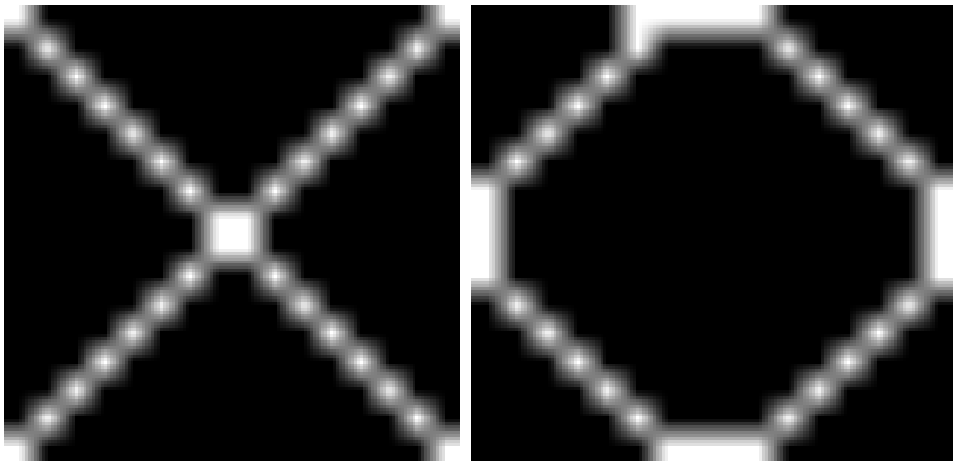
4.5 Training

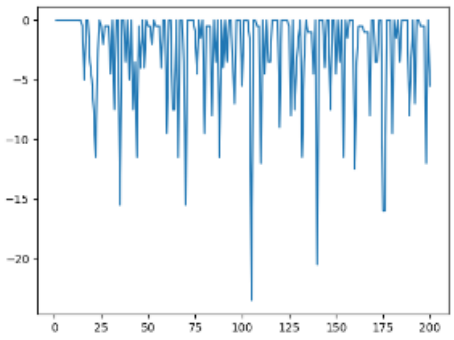
- Take N samples of digit images.
- For each image, find receptive field of that image.

- Encode the receptive field with the encode function.
- Iterate from time 0 to time T -
- Iterate from neuron 1 to neuron 4 -
- If at any instant t, neuron x has membrane potential greater than threshold potential, then a spike will occur in x at t.

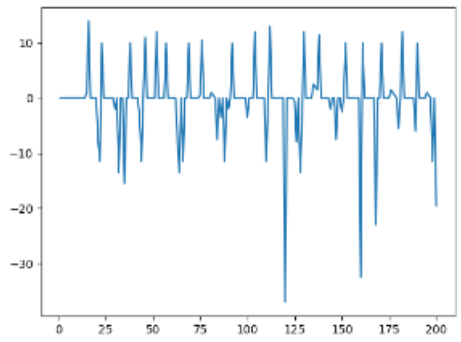
4.6 Results

Test Data

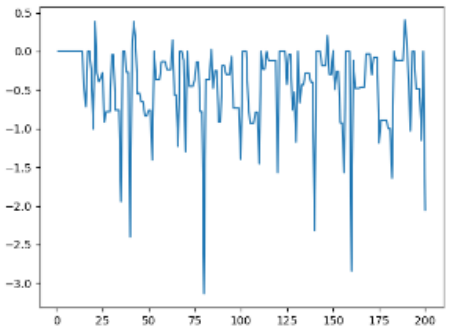




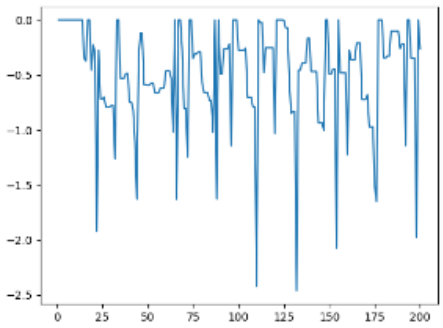
Neuron-1 corresponds to class-1



Neuron-2 corresponds to class-2



Neuron-3 corresponds to noise



Neuron-4 corresponds to noise

5 Simulation of a Complex Control System

5.1 Objective of our Complex Control System

We plan to devise a control system such that an entity can travel from a predefined starting point to a predefined destination point. The entity initially has the knowledge of just moving in a straight path along its initial orientation direction. This can be thought of as a locomotive robot with just DC Motors connected to its axles, hence the robot will move just in a straight path using the DC Motor. However now if a microcontroller like Arduino is connected, we can perform different control operations on the robot. Hence our complex control system is to make a locomotive robot autonomously go from the predefined starting point to the predefined ending point without using any human wisdom which can program a microcontroller to do so.

5.2 Approach

- We plan to train a Spiking Neural Network which can direct the entity to move from a predefined starting point to a predefined destination point.
- It is assumed that the entity initially has the knowledge of just moving along its initial orientation direction.
- SNN would be given as an input the current coordinates and orientation

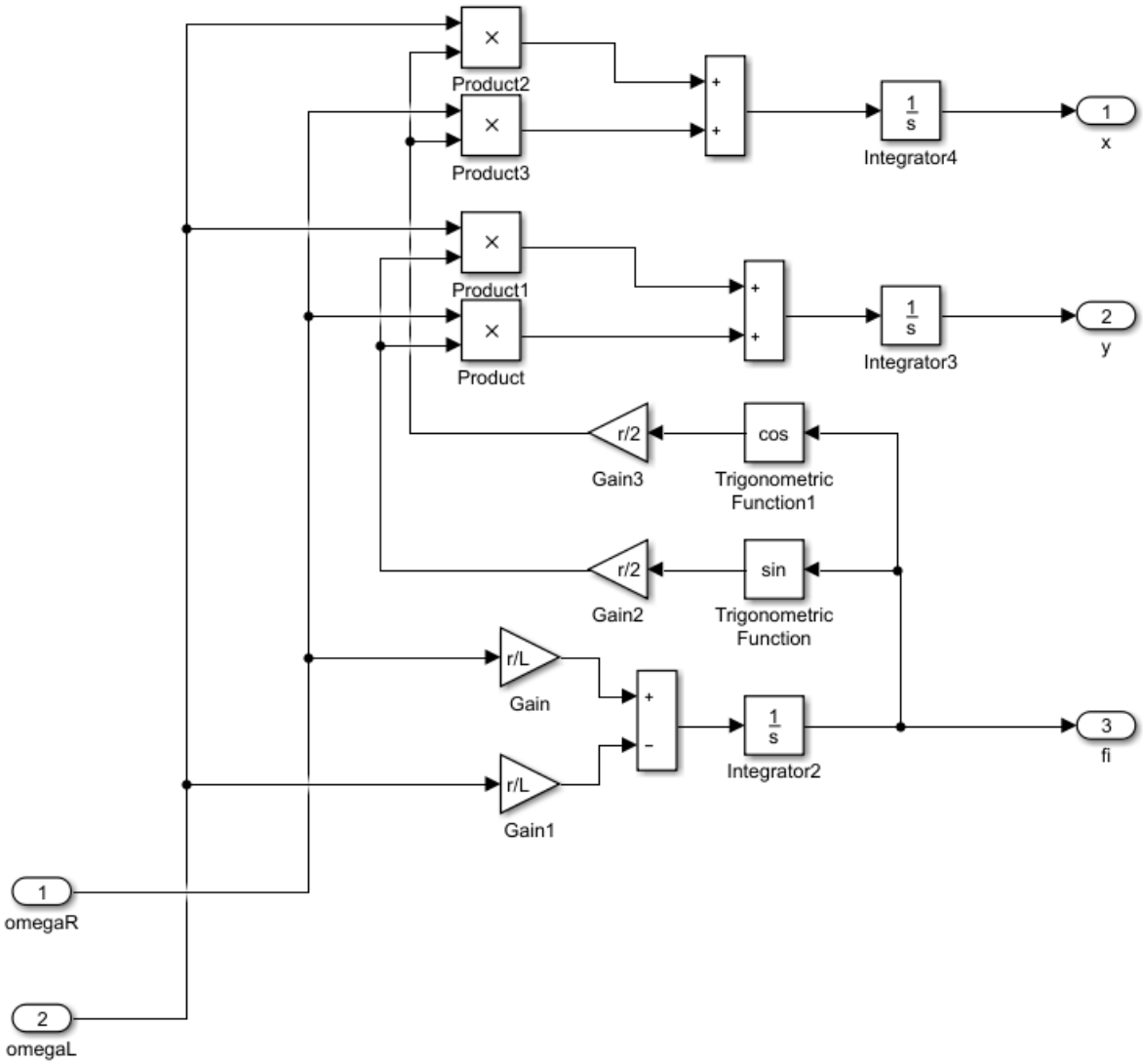
of the entity. Using the above information SNN will try to generate two optimized constants ω_L and ω_R.

- These optimized constants would then be passed through a Simulink controller to get the new coordinates and orientation of the entity which would then be passed again to SNN to get new values of two optimized constants ω_L and ω_R such that the entity can reach the predefined destination point.

- For our Simulation, Spiking Neural Network would be trained using Spike Time Dependent Plasticity (STDP) approach which is based on human brain synapse. STDP would be used to update the weights of the connections between the neurons between different layers of SNN. Hence by updating the weights efficiently, SNN can be trained through STDP approach. This trained SNN would then output two optimized constants ω_L and ω_R at each step of the simulation. This STDP approach is analogous to the Back propagation approach used in Convolutional Neural Networks.

- While training a CNN, we require an optimized like Adam. Hence for our simulation while training a SNN to optimize various Spike Time Dependent Plasticity Approach Parameters, we would be using Genetic Algorithm Paradigm.

5.3 Simulink Controller Design



Above Simulink Controller takes as input constants ω_L and ω_R . These constants were outputs of the Spiking Neural Networks which was trained using Spike Time Dependent Plasticity (STDP) Paradigm. The parameters of STDP during training SNN were optimized using Genetic Algorithm. Above Simulink Controller outputs the coordinates and orientation of the entity at the end of a step of the entire simulation. Hence at the end of the step, the entity is at a new position and orientation with the goal of reaching the pre-

defined destination point. These new coordinates and orientation are then again passed through SNN and the Control System is repeated again until the entity reaches the predefined destination point.

5.4 Codes

Reference: Viceriel. "Viceriel/Spiking-Neural-Network-Controler." GitHub, github.com/Viceriel/Spiking-neural-network-controler.
Note that each and every code is heavily commented, hence I would recommend you to go through the code in order to understand our approach.

Main.m MATLAB Script file performs the entire process of training an SNN as well as simulating the complex control system.

5.4.1 Main.m

```
%clc
%clear all
%close all

% Our goal is to use SNN to train the entity to take a path or
% trajectory such that it reaches a predefined destination assigned by the
% user beforehand

% position of the entity after one training/epoch and then those
% trained parameters being passed through the controller
% designed in Simulink titled mobileSim to get the resultant x (xVec)
% and y (yVec) coordinates of the entity as well as the orientation of the
% entity (fiVec) after being trained for that
% epoch

% defining global variables x_1, y_1 and fi_1 describing the initial
% coordinates of the entity and also the initial orientation of the entity
% global variables omegaR, omegaL trained parameters/constants from SNN
% which are passed through the Simulink Controller to get the (X,Y)
% coordinates of the entity as well the orientation of the entity fi

global x_1
```

```
global y_1
global omegaR
global omegaL
global fi_1
global fi

% constructing a Spiking Neural Network with 3 layers
% First layer which is the input layer contains 6 neurons as there are 6
% inputs to SNN (explained more in RunSim MATLAB Script File)
% Second Hidden layer of SNN containds 5 neurons
% Third Layer which is the output layer contains 2 neurons representing two
% outputs omegaR and omegaL from SNN (explained more in RunSim MATLAB
% Script File)
% Second argument is the connection matrix which establishes which
% neuron of each layer is connected to with neurons
% Third and Fourth arguments are the maximum and minimum values of Synapse
% and hence the weights of SNN

net = SpikeNetwork([6 5 2], [0 0 0; 1 0 0; 1 1 0], 1, 0);

% Refer to the RunSim MATLAB Script File to know more about the arguments
% of RunSim function
RunSim(net, 0.1, 0.1, 10);

% Refer to the GeneticAlgorithm MATLAB Script File to know more about the
% arguments of GeneticAlgorithm function
% Briefly, 0.05 is the mutation rate, 30 is the population size initially,
% and the rest arguments are SNN structure parameters

EA = GeneticAlgorithm(0.05, 30, [6 5 2], [0 0 0; 1 0 0; 1 1 0]);

% Refer to the GeneticAlgorithm MATLAB Script File to know more about the
% arguments of Evolve function
% Briefly, 45 is the total number of generations GeneticAlgorithm should go
% through and EA is the GeneticAlgorithm class object

% Here Genetic Algorithm is used for the optimization of Spike Time
% Dependent Plasticity (STDP) Learning Parameters which is used for
```

```
% updating the weights of different connections in SNN and hence ultimately
% training SNN (for more info refer to the comments of SpikeNetwork MATLAB
% Script File)

EA = Evolve(EA, 45);
```

5.4.2 SpikeNetwork.m

```
classdef SpikeNetwork    % defining a class for Spike Neural Network (SNN)


    properties

        neural;          % neuron layers
        weights;         % synapse layers
        connections;     % connection matrix
        layers;          % count of layers
        time;            % time in network
        max;             % max value of spike generated at synapse
        min;             % min value of spike generated at synapse
        net_output;      % output vector of network

    end

    methods

        % Network constructor
        % structure- topology of network
        % connections- connections between layers
        % ma, mi- synapse bounding

        function obj = SpikeNetwork(structure, connections, ma, mi)

            % initializing various Hyper Parameters of SNN

            obj.max = ma;
```

```
obj.min = mi;
obj.time = 1;
len = length(structure);
obj.layers = structure;
obj.neural = {};
obj.weights = {};
obj.net_output = 0;

% creating connections between layers according to connection
% argument every neuron(except input layer) has synapse vector
% in net topology so there are four cycles,
% one for layer, one for neuron in layer
% and another two for same reason(see comment two) i.e. synapse

for i = 2 : len

    obj.weights{i} = {};

    for j = 1 : obj.layers(i)

        obj.weights{i}{j} = {};

        for k = 1 : len

            obj.weights{i}{j}{k} = {};

            for l = 1 : obj.layers(k)

                  obj.weights{i}{j}{k}{l} = Synapse();

                  if connections(i, k) == 1
                     obj.weights{i}{j}{k}{l}.value =
                     (0.1 - 0.001)*rand(1, 1) + 0.001;
                  end

            end

        end
```

```
        end

    end

    % creating layers of neurons
    % first layer is created from input neurons
    % rest of layers contain Spiking Neurons

    for i = 1 : len

        neurons = {};

        for j = 1 : obj.layers(i)

            if i == 1

                neurons{j} = InputNeuron();

            else

                neurons{j} = SpikeNeuron();

            end

        end

        obj.neural{i} = {neurons};

    end

end

% running network with input data
% obj- network
% data- vector of input data- obsolete because input neurons are
% injected in RunSim
% return- network
```

```
function obj = Run(obj, data)

    len = length(obj.layers);

    % running through first layer
    % if some neuron from first layer emitted spike, spike time is
    % recorded
    % if spike isn't emitted, spike time is zero and time from
    % spike is incremented

    for i = 1 : obj.layers(1)
        obj.neural{1}{1}{i} = ComputeOutput(obj.neural{1}{1}{i});
        if obj.neural{1}{1}{i}.output == 1
            obj.neural{1}{1}{i}.spike_time = obj.time;
            for j = 2 : len
                 for k = 1 : obj.layers(j)
                    obj.weights{j}{k}{1}{i}.time_from_spike = 0;
                    obj.weights{j}{k}{1}{i}.spike_time = obj.time;
                 end
            end
        else
          for j = 2 : len
            for k = 1 : obj.layers(j)
                if obj.weights{j}{k}{1}{i}.spike_time == 0
                   obj.weights{j}{k}{1}{i}.time_from_spike =
                   obj.weights{j}{k}{1}{i}.time_from_spike + 0.1;
                end
            end
          end
        end
    end

    % passing through remaining layers
    % after next layer, time is incremented

    for i = 2 : len
        obj.time = obj.time + 0.1;
```

```
% computing input for neurons(output x synapse)

for j = 1 : obj.layers(i)
    input = 0;
    for k = 1 : len
        for l = 1 : obj.layers(k)
            input = input +
            (obj.neural{k}{1}{l}.output*
            obj.weights{i}{j}{k}{l}.value);
        end
    end

    obj.neural{i}{1}{j} = OutputCompute(obj.neural{i}{1}{j},
    input);

    % if neuron emits spike, spike time is recorded

    if obj.neural{i}{1}{j}.output == 1
        obj.neural{i}{1}{j}.spike_time = obj.time;

        % recording spike time of neuron to synapses of
        % neurons in next layer

        if i ~= len
           for x = i + 1 : len
               for y = 1 : obj.layers(x)
                    obj.weights{x}{y}{i}{j}.spike_time
                    = obj.time;
                    obj.weights{x}{y}{i}{j}.time_from_spike = 0;
               end
           end
        end

         % passing through all synapses of actual neuron
         % applying STDP based on spike times
         % STDP function defined later in the code
         % Here STDP stands for Spike Time Dependent
```

```
        % Plasticity

        for m = 1 : len
           for n = 1 : obj.layers(m)
               if obj.weights{i}{j}{m}{n}.value ~= 0
                   if obj.weights{i}{j}{m}{n}.spike_time ~= 0
                       obj = STDP(obj,
                       obj.weights{i}{j}{m}{n}.spike_time -
                       obj.neural{i}{1}{j}.spike_time, [i j],
                       [m n], obj.neural{m}{1}{n}.rule);
                   else
                       obj = STDP(obj,
                       obj.weights{i}{j}{m}{n}.time_from_spike,
                       [i j], [m n], obj.neural{m}{1}{n}.rule);
                       obj.weights{i}{j}{m}{n}.time_from_spike =
                       0;
                   end
               end
           end
        end

       % reset of spike times

       for x = i : -1 : 1
           for y = 1 : obj.layers(x)
               obj.weights{x}{y}{i}{j}.spike_time = 0;
               obj.weights{i}{j}{m}{n}.time_from_spike = 0;
           end
       end
   else % if no spike
        if i ~= len

         % incrementing of time distance from last spike
         % to synapse of next layer neuron

          for x = i + 1 : len
              for y = 1 : obj.layers(x)
                  if obj.weights{x}{y}{i}{j}.spike_time == 0
```

```
                                    obj.weights{x}{y}{i}{j}.time_from_spike =
                                    obj.weights{x}{y}{i}{j}.time_from_spike +
                                    0.1;
                                end
                            end
                        end
                    end
                end
            end

            % this is because in first pass through the network,
            % times in layers are 0.1, 0.2, 0.3
            % however in second samples we want 0.2, 0.3, 0.4 (chosen)

            obj.time = obj.time - 0.1;

            % collecting  network output

            for i = 1 : obj.layers(len)

                obj.net_output(i) = obj.neural{len}{1}{i}.output;

            end
        end

        % Spike time dependent plasticity rule for updating synapses
        % parameters of STDP which are optimized by Genetic algorithm
        % Code for Genetic Algorithm in another script file

        function obj = STDP(obj, spike_differences, post, pre, rule)

            % if using STDP

            if rule == 1
                if spike_differences < 0
                    obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value =
                    obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value +
```

```
                    obj.neural{post(1)}{1}{post(2)}.A1*exp(spike_differences /
                    obj.neural{post(1)}{1}{post(2)}.t);
                else
                    obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value =
                    obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value -
                    obj.neural{post(1)}{1}{post(2)}.A2*exp(-spike_differences /
                    obj.neural{post(1)}{1}{post(2)}.t);
                end
            else % ANTI STDP
                if spike_differences < 0
                    obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value =
                    obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value -
                    obj.neural{post(1)}{1}{post(2)}.A2*exp(-spike_differences /
                    obj.neural{post(1)}{1}{post(2)}.t);
                else
                    obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value =
                    obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value +
                    obj.neural{post(1)}{1}{post(2)}.A1*exp(spike_differences /
                    obj.neural{post(1)}{1}{post(2)}.t);
                end
            end

            % bounding of weight change

            if obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value > obj.max
                obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value = obj.max;
            elseif obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value < obj.min
                obj.weights{post(1)}{post(2)}{pre(1)}{pre(2)}.value = obj.min;
            end
        end

    end

end

%done commenting, for any doubts contact: gandhi.meet@iitgn.ac.in
```

5.4.3 InputNeuron.m

```
classdef InputNeuron % defining a class for Input Neurons

    properties

        % defining Input Neuron Object Attributes

        output;
        spike_time;
        A1;
        A2;
        t;
        rule;
        count;

    end

    methods

        function obj = InputNeuron()

            % Initializing Input Neuron Object Parameters

            obj.output = 0;
            obj.A1 = 0;
            obj.A2 = 0;
            obj.t = 0;
            obj.rule = 0;
            obj.count = 0;
        end

        function obj = ComputeOutput(obj)

            % obj.count defined in RunSim MATLAB Script File

            if obj.count > 0
                obj.output = 1;
```

```
                obj.count = obj.count - 1;
            else
                obj.output = 0;
            end

        end

    end

end
```

5.4.4 SpikeNeuron.m

```
classdef SpikeNeuron % defining a class for Spike Neurons

   properties

       % defining SpikeNeuron Class Attributes

        output;
        rule;
        tau;
        uprah;
        u
        R;
        C;
        A1;
        A2;
        t;
        spike_time;

    end

    methods

        % Initializing SpikeNeuron Class Attributes

        function obj = SpikeNeuron()
```

```
            obj.uprah = 0;
            obj.u = obj.uprah;
            obj.tau = 0.5;
            obj.R = 10;
            obj.C = obj.tau/obj.R;
            obj.A1 = 1;
            obj.A2 = 1;
            obj.t = 10;
            obj.rule = 1;
            obj.output = 0;
            obj.spike_time = 0;

        end

        function obj = OutputCompute(obj, thalamic_input)

            % Spike Neuron Output based on Biophysical Hodgkin-Huxley model
            % and its Mathematical simplification by Izhikevich
            % Various HyperParameters for the output of Spike Neuron were
            % selected from the standard Model of Izhikevich

           obj.u = obj.u + (obj.uprah/obj.tau) - (obj.u/obj.tau) +
           (obj.R*thalamic_input)/obj.tau;

           if obj.u >= 22
               obj.output = 1;
               obj.u = obj.uprah;

           else

               obj.output = 0;

           end
        end
    end
end
```

5.4.5 Synapse.m

```
classdef Synapse % defining a class for Synapse

    properties

        % defining Synapse Object Class Attributes

        value;
        spike_time;
        time_from_spike;

    end

    methods

        function obj = Synapse()

            % Initializing Synapse Object Class Attributes

            obj.value = 0;
            obj.time_from_spike = 0;
            obj.spike_time = 0;

        end

    end
end
```

5.4.6 GeneticAlgorithm.m

```
classdef GeneticAlgorithm

    properties

        m_mutation_rate;
        m_population_size;
        m_chromosone_length;
```

```
    m_population;
    m_structure;
    m_connections;
    m_children;
    m_network;
end

methods

    function obj = GeneticAlgorithm(mutation_rate, population_size,
    neurons, connections)

        obj.m_mutation_rate = mutation_rate;
        obj.m_population_size = population_size;
        n = sum(neurons);
        %node count multiply rule parameter encoding and tau minus
        %count of output node + fitness
        obj.m_chromosone_length = n * (1 + 10 + 5) -
        neurons(end) * (1 + 10 + 5) + 1;
        obj.m_structure = neurons;
        obj.m_connections = connections;
        len =  obj.m_population_size;
        chromosone = obj.m_chromosone_length;
        obj.m_population = zeros(len, chromosone);
        obj.m_children = zeros(len, chromosone);

    end

    function obj = Initialization(obj, init)

        for i = init : obj.m_population_size
            for j = 1 : obj.m_chromosone_length
               obj.m_population(i, j) = randi([0 1]);
            end
        end
    end

    function net = Decoding(obj, index, population)
```

```
    binary2num([1 1], 10);
    a = load('net');
    net = a.net;
    len = length(obj.m_structure);
    struct = obj.m_structure;
    l = 0;

    for i = 1 : len - 1
        for j = 1 : obj.m_structure(i)
            if i > 1
                l = i - 1;
            else
                l = i;
            end

            ind = ((i > 1)*struct(l) + j) * 16 - 16 + 1;
            net.neural{i}{1}{j}.rule = population(index, ind);
            net.neural{i}{1}{j}.t = binary2num(population(index,
            ind + 11 : ind + 16), 100);
            A = binary2num(population(index, ind + 1 : ind + 10), 1000);
            net.neural{i}{1}{j}.A1 = A;
            net.neural{i}{1}{j}.A2 = A;
        end
    end
end

function obj = Fitness(obj, populate)

    if populate == true
        population = obj.m_population;
    else
        population = obj.m_children;
    end

    len = size(obj.m_population);
    len = len(1);
    xRef = 10.1;
```

```
    yRef = 10.1;

    for i = 1 : len

        net = Decoding(obj, i, population);
        for k = 1:3
            [x, y, wR, wL, net] = RunSim(net, 10.1, 10.1, 5);
        end

        sumPath = 0;

        for k = 1:length(x)
            sumPath = sumPath + sqrt((x(k) - xRef)^2 + (y(k) - yRef)^2);
        end

       population(i, end) = 100 / sumPath;
    end

    if populate == true
        obj.m_population = population;
    else
        obj.m_children = population;
    end

end

function index = Selection(obj, population)

    choose1 = randi([1 obj.m_population_size]);
    choose2 = randi([1 obj.m_population_size]);

    while choose1 == choose2
        choose2 = randi([1 obj.m_population_size]);
    end

    if population(choose1, end) > population(choose2, end)
        index = choose1;
    else
```

```
        index = choose2;
    end

end

function obj = Crossover(obj)

    len = obj.m_population_size;

    for i = 1 : len
        divide_point = randi([1 (sum(obj.m_structure) -
        obj.m_structure(end) - 1)]);
        parent1 = Selection(obj, obj.m_population);
        parent2 = Selection(obj, obj.m_population);
        obj.m_children(i, 1 : divide_point * 16) =
        obj.m_population(parent1, 1 : divide_point * 16);
        obj.m_children(i, divide_point * 16 + 1 : end - 1) =
        obj.m_population(parent2, divide_point * 16 + 1 : end - 1);
    end

end

function obj = Mutation(obj)

    len = obj.m_population_size;
    len1 = obj.m_chromosone_length - 1;
    mutation_probability =  obj.m_mutation_rate;

    for i = 1 : len
        for j = 1 : len1
            if rand([1 1]) <= mutation_probability
                if obj.m_children(i, j) == 1
                    obj.m_children(i, j) = 0;
                else
                    obj.m_children(i, j) = 1;
                end
            end
        end
```

```
    end

end

function index = GetBest(obj, population)

    maximum = population(1, end);
    len = obj.m_population_size;
    index = 1;

    for i = 2 : len
       if population(i, end) > maximum
           maximum = population(i, end);
           index = i;
       end
    end

end

function obj = Replace(obj)

    obj.m_population(1, :) =
    obj.m_population(GetBest(obj, obj.m_population), :);
    obj.m_population(2, :) =
    obj.m_children(GetBest(obj, obj.m_children), :);

    len = obj.m_population_size;

    for i = 3 : len
        parent = Selection(obj, obj.m_population);
        child = Selection(obj, obj.m_children);

        if obj.m_population(parent, end) > obj.m_children(child, end)
            obj.m_population(i, :) = obj.m_population(parent, :);
        else
            obj.m_population(i, :) = obj.m_children(child, :);
        end
    end
```

```
    end

    function obj = Evolve(obj, generations)

        obj = Initialization(obj, true);
        obj = Fitness(obj, true);

        for i = 1 : generations
            obj = Crossover(obj);
            obj = Mutation(obj);
            obj = Fitness(obj, false);
            obj = Replace(obj);

            top = GetBest(obj, obj.m_population);
            net = Decoding(obj, top, obj.m_population);

            for k = 1:3
                [x, y, wR, wL, net] = RunSim(net, 10.1, 10.1, 5);
            end
            obj.m_network = net;
            figure();
            plot(x, y)
            hold all
            plot(10.1, 10.1, 'r+');
            grid;

            if mod(i, 20) == 0
                half = ceil(obj.m_population_size / 2);
                obj = Initialization(obj, half);
                obj = Fitness(obj, true);
            end
        end

    end

end
```

```
end
```

5.4.7 RunSim.m

```
function [ xVec, yVec, outOmegaR, outOmegaL, net ] = RunSim( net, xRef, yRef,
time )

% defining a function RunSim for undergoing the simulating an
% entity which initially has the knowledge of just going in a
% horizontal path i.e. through a straight line from (10, 10) with zero
% slope

% Our goal is to use SNN to train the entity to take a path or
% trajectory such that it reaches a predefined destination assigned by the
% user beforehand

% position of the entity after one training/epoch and then those
% trained parameters being passed through the controller
% designed in Simulink titled mobileSim to get the resultant x (xVec)
% and y (yVec) coordinates of the entity as well as the orientation of the
% entity (fiVec) after being trained for that
% epoch

xVec = [];
yVec = [];
fiVec = [];

% defining global variables x_1, y_1 and fi_1 describing the initial
% coordinates of the entity and also the initial orientation of the entity
% global variables omegaR, omegaL trained parameters/constants from SNN
% which are passed through the Simulink Controller to get the (X,Y)
% coordinates of the entity as well the orientation of the entity fi

global x_1
global y_1
global fi_1
global omegaR
global omegaL
```

```
global fi

% for our simulation the initial coordinates of the entity is (10,10) and
% Zero degree orientation

x_1 = 10;
y_1 = 10;
fi_1 = 0;
omegaR = 0;
omegaL = 0;

outOmegaR = [];
outOmegaL = [];

for i = 0:0.05:time
    wR = [];
    wL = [];

    % calculating the closeness of the trained solution to the destination
    % point by calculating changes in the current trained coordinates of
    % the entity with that of the destination point coordinates

    deltaX = abs(xRef - x_1);
    deltaY = abs(yRef - y_1);
    rho = sqrt((xRef - x_1)^2 + (yRef - y_1)^2);
%     Phi = atan((yRef - y_1)/(xRef - x_1)) - fi_1;
%     n = Phi/pi;
%     deltaPhi = ((Phi/n) + pi)/(2*pi);
    Phi = atan2((yRef - y_1),(xRef - x_1)) - fi_1;
    n = atan2(sin(Phi),cos(Phi));
    deltaPhi = (n + pi)/(2*pi);
    omegaR1 = omegaR;
    omegaL1 = omegaL;

    if deltaX > 1
        deltaX = 1;
    end
    if deltaY > 1
```

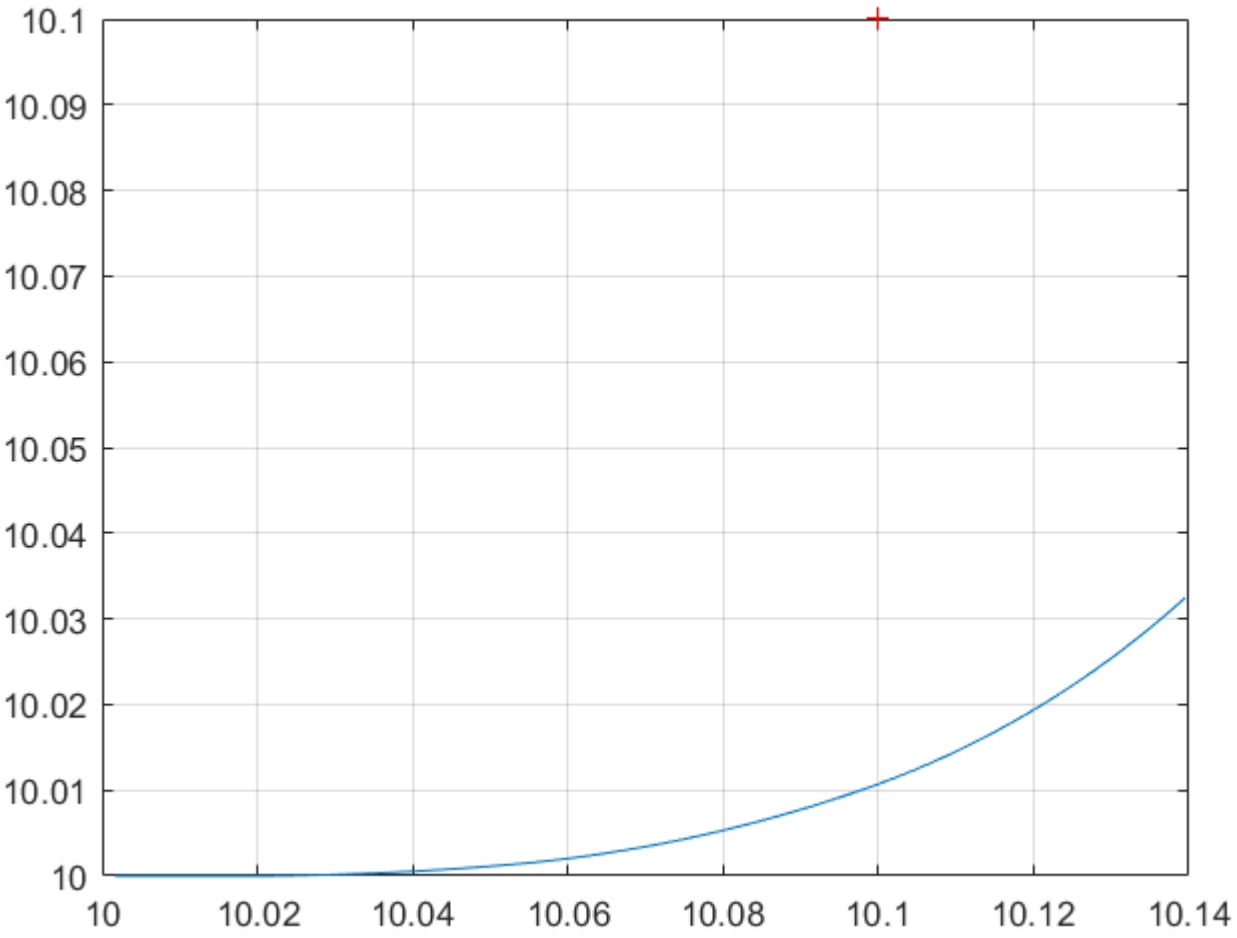
```
    deltaY = 1;
end
if rho > 1
    rho = 1;
end

% scaling down by 10 which will be scaled up by 10 afterwards

omegaR1 = omegaR1 / 10;
omegaL1 = omegaL1 / 10;

% assigning inputs to the first layer of SNN
% as there are 6 neurons in the first layer of our SNN, there are
% correspondingly 6 inputs

in(1) = 1 / deltaX;
in(1) = round(50 / in(1));
in(2) = 1 / deltaY;
in(2) = round(50 / in(2));
in(3) = 1 / rho;
in(3) = round(50 / in(3));
in(4) = 1 / deltaPhi;
in(4) = round(50 / in(4));
in(5) = 1 /omegaR1;
in(5) = round(50 / in(5));
in(6) = 1 /omegaL1;
in(6) = round(50 / in(6));

net.neural{1}{1}{1}.count = in(1);
net.neural{1}{1}{2}.count = in(2);
net.neural{1}{1}{3}.count = in(3);
net.neural{1}{1}{4}.count = in(4);
net.neural{1}{1}{5}.count = in(5);
net.neural{1}{1}{6}.count = in(6);

% Running the network 50 times and then taking the average of the
% outputs as the actual output
```

```
    for j = 1:50
        net = Run(net);
        wR = [wR net.net_output(1)];
        wL = [wL net.net_output(2)];
    end

    % as we had scaled down omegaR and omegaL by 10 earlier now after
    % taking average of 50 outputs we scale up omegaR and omegaL by 10 and
    % hence division by 5 and not 50

omegaR = sum(wR)/5;
omegaL = sum(wL)/5;

outOmegaR = [outOmegaR omegaR];
outOmegaL = [outOmegaL omegaL];

% passing the outputs omegaR and omegaL from the SNN to the mobileSim
% Simulink Controller in order to get the new (X,Y) coordinates and the
% orientation of the entity
% These are then updated as the new x_1, y_1 and fi_1 which are again
% passed through the SNN network until the destination point is reached

sim('mobileSim');

xVec = [xVec x(2)];
yVec = [yVec y(2)];
fiVec = [fiVec fi(2)];

x_1 = x(2);
y_1 = y(2);
fi_1 = fi(2);

end

end
```

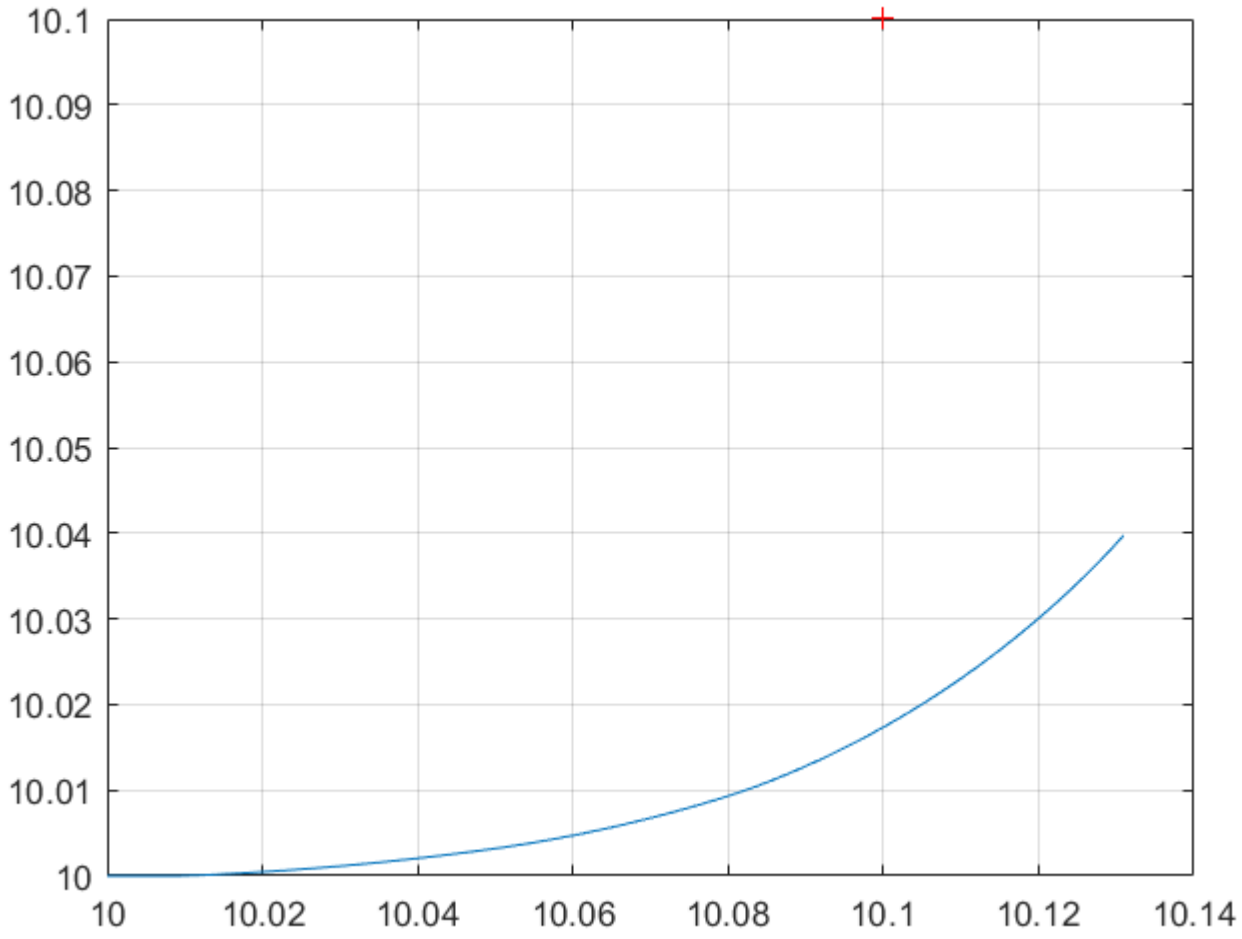
5.5 Results of Simulation

The simulation was executed for 45 Generations. For Training SNN, it took almost 5 days to simulate 45 generations without using GPU. In the above simulation, (10,10) is the predefined starting point for the entity and (10.1,10.1) is the predefined destination point.
Below are the results of some of the important generations:

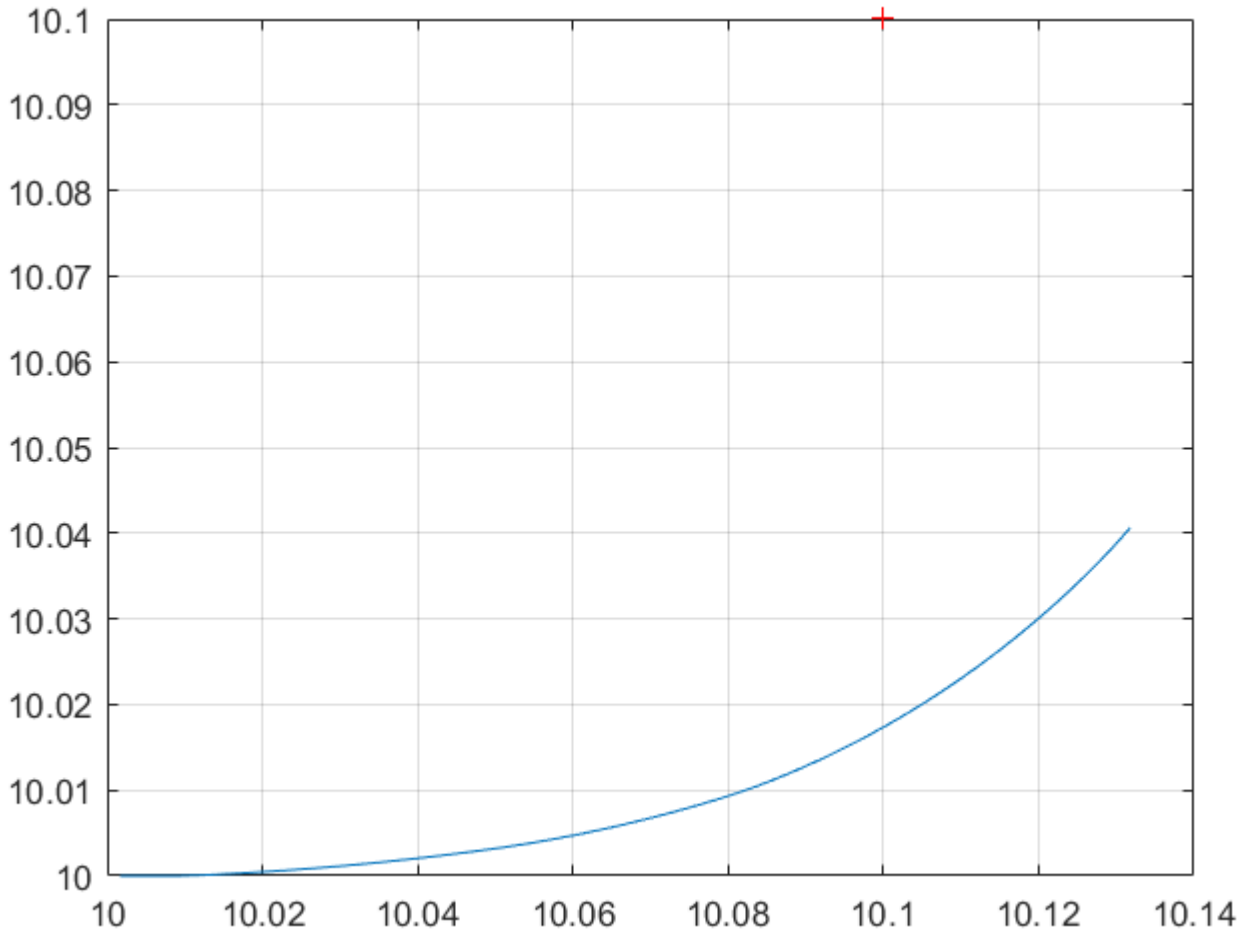
5.5.1 Generation 1



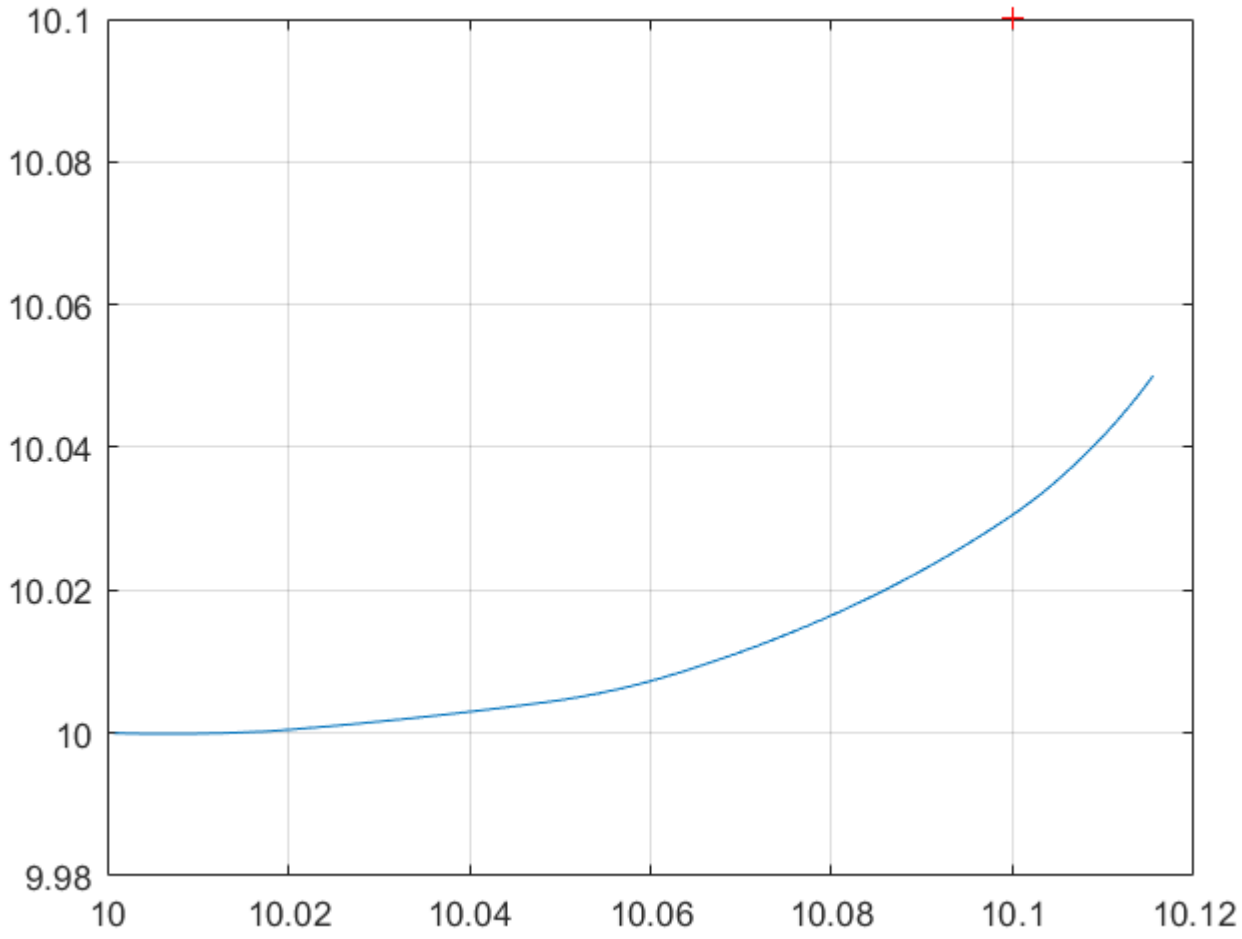
5.5.2 Generation 3



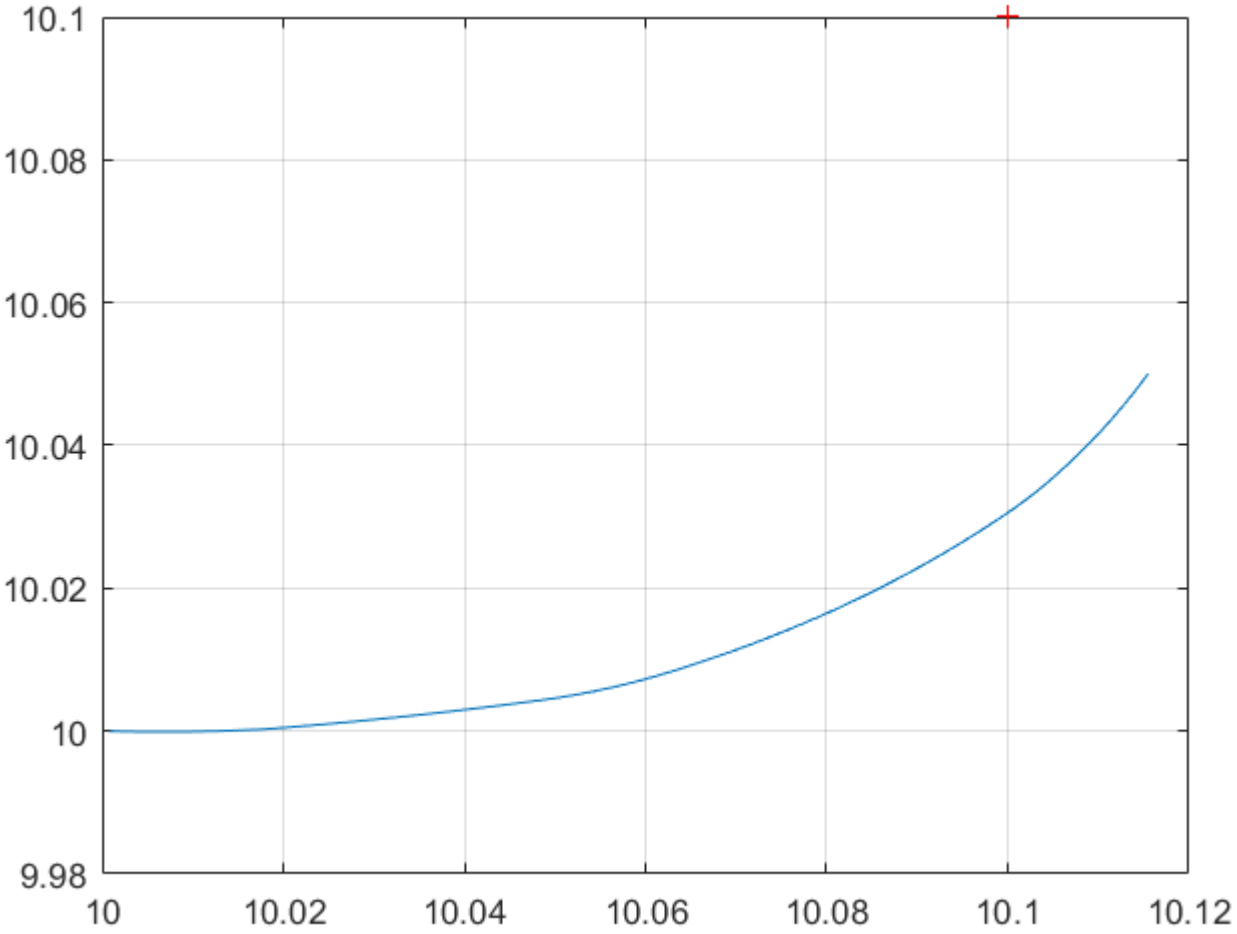
5.5.3 Generation 5



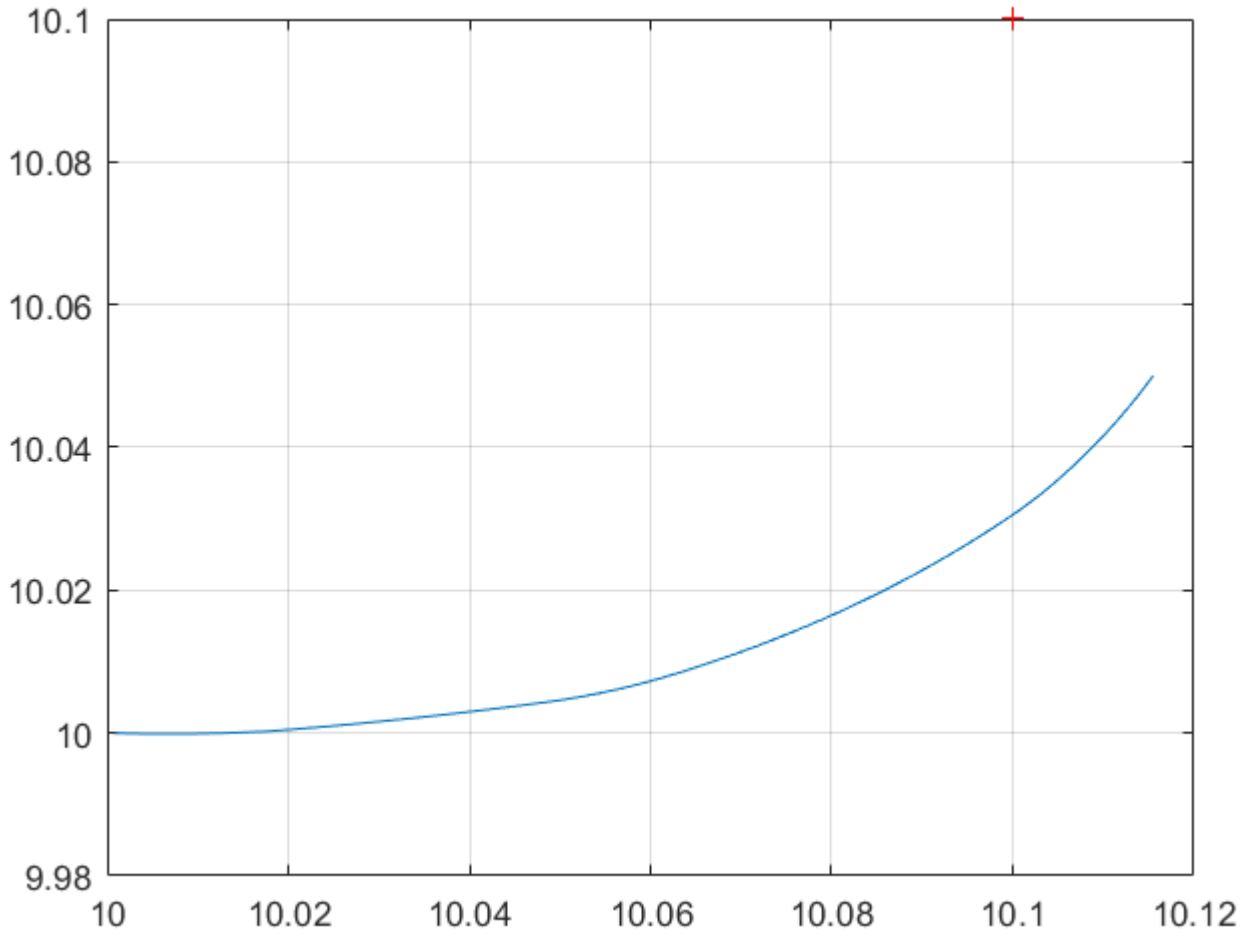
5.5.4 Generation 6



5.5.5 Generation 25



5.5.6 Generation 45



5.6 Conclusions from Simulation of a Complex Control System

- While training SNN was showing good reinforcements/improvements until Generation 6.
- After Generation 6, SNN showed no improvements even though it was undergoing training.

- We think that the reason for the above might be that the initial hyper-parameters that we had defined for SNN needs to be optimized more.
- Also there needs to be some changes in the Spike Time Dependent Plasticity Paradigm Parameters.
- We used Genetic Algorithm as an optimizer of STDP/SNN parameters which needs to be changed by some other optimizer for checking whether the optimizer is the problem or not.
- If optimizer is the problem then we need to either change the optimizer or change the initial hyper-parameters of the Genetic Algorithm optimizer.
- We can also change the structure of the hidden layer of SNN to make the results more accurate.
- As a result various parameters and objects need to be changed and played with to get the desired result for the given application. Once we get the desired configuration of SNN as well as Genetic Algorithm optimizer, we can train SNN such that it can direct the entity to go from a predefined starting point to a predefined destination point.
- Even if we succeed in making the entity go from a predefined starting point to a predefined destination point, one major drawback is that if we change the starting and destination point, SNN needs to be trained again and hence the entity which is expected as the trained constants ω_L and ω_R would change for different starting and destination points.
- With regards to the Simulink Controller Paradigm, it was working as expected and was showing no oscillations from the desired output which concludes that the steady state error was infinitesimally small.
- In order to make the approach efficient enough, we need a GPU which can train SNN in run time. By training SNN in run time, SNN can direct the entity instantaneously using the trained constants ω_L and ω_R which are passed through the Simulink Controller so that the entity reaches the predefined destination point and stops.

6 References

- Simple model of spiking neurons - IEEE Journals & Magazine
  https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1257420&tag=1
- Modeling a Spiking Neuron I - Integrate-and-Fire Model
  http://fourier.eng.hmc.edu/e180/lectures/signal2/node4.html
- NeuralNetworksResearch - aHuman Wiki
  http://wiki.ahuman.org/index.php/NeuralNetworksResearch
- Hodgkin-Huxley Model
  http://icwww.epfl.ch/ gerstner/SPNM/node14.html
- Hodgkin and Huxley: Superheroes
  https://www.swarthmore.edu/NatSci/echeeve1/Ref/HH/HHmain.htm
- The Hodgkin-Huxley Model
  $https://www.st-andrews.ac.uk/\ wjh/hh_model_intro$
- $http://www.math.pitt.edu/\ bdoiron/assets/ermentrout-and-terman-ch-1.pdf$
- $https://www.izhikevich.org/publications/spikes.pdf$
- Hodgkin-Huxley Model — Neuronal Dynamics online book Wulfram Gerstner http://neuronaldynamics.epfl.ch/online/Ch2.S2.html
- $https://thesai.org/Downloads/IJARAI/Volume4No7/Paper_1-A_Minimal_Spiking_Neural$
- GitHub - https://github.com/Shikhargupta/Spiking-Neural-Network
- Neural networks and deep learning
  Nielsen - Michael A.
  http://neuralnetworksanddeeplearning.com/chap2.html
- $www.researchgate.net/figure/Biological-STDP-from-16-and-simplified-STDP-used-in-the-proposed-PCM-implementation_fig2_230727604$
- https://www.izhikevich.org/publications/spikes.pdf

- Eye, Brain, and Vision
  http://hubel.med.harvard.edu/book/b10.htm
- Hien, Dang Ha The. "A Guide to Receptive Field Arithmetic for Convolutional Neural Networks." Medium, ML Review, 5 Apr. 2017, medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807
- Acta neurobiologiae experimentalis.
  F Ponulak - A Kasinski
  https://www.ncbi.nlm.nih.gov/pubmed/22237491
- I. Single Neuron Models
  http://icwww.epfl.ch/ gerstner/SPNM/node11.html
- Viceriel. "Viceriel/Spiking-Neural-Network-Controler." GitHub, github.com/Viceriel
  /Spiking-neural-network-controler.
- "Build Software Better, Together." GitHub, github.com/topics/spiking-neural-networks.
- Tavanaei, Amirhossein and Anthony S. Maida. "A Minimal Spiking Neural Network to Rapidly Train and Classify Handwritten Digits in Binary and 10-Digit Tasks." (2015).
- https://homepages.cwi.nl/ sbohte/publication/phdthesis.pdf
- Designing Spiking Neural Networks - IEEE Conference Publication, ieeexplore.ieee.org/document/7451989/.
- "Spiking Neural Network Conversion Toolbox." Spiking Neural Network Conversion Toolbox - SNN Toolbox 0.1.0 Documentation, snntoolbox.readthedocs.io/en/latest/guide/intro.html.
- Soni, Devin. "Spiking Neural Networks, the Next Generation of Machine Learning." Towards Data Science, Towards Data Science, 11 Jan. 2018, towardsdatascience.com/spiking-neural-networks-the-next-generation-of-machine-learning-84e167f4eb2b.
- Nichols, Eric et al. "Biologically Inspired SNN for Robot Control." IEEE Transactions on Cybernetics 43 (2013): 115-128.

- Alnajjar, F. S. and K. Murase. "Use-dependent Synaptic Connection Modification in SNN Generating Autonomous Behavior in a Khepera Mobile Robot." 2006 IEEE Conference on Robotics, Automation and Mechatronics (2006): 1-6.

- Allen, J. N. et al. "A Low-Power Haar-Wavelet Preprocessing Approach for a SNN Olfactory System." 2007 2nd International Design and Test Workshop (2007): 222-225.

- Moreira, Guilherme et al. "Understanding the SNN Input Parameters and How They Affect the Clustering Results." IJDWM 11 (2015): 26-48.

- Maass, Wolfgang. "Networks of spiking neurons: The third generation of neural network models." Electronic Colloquium on Computational Complexity (ECCC) 3 (1996): n. pag.

- Bagheri, Alireza et al. "Adversarial Training for Probabilistic Spiking Neural Networks." CoRR abs/1802.08567 (2018): n. pag.

- Bohte, Marcel. "Spiking Neural Networks Spiking Neural Networks." (2003).

- Maass, Wolfgang. "Fast Sigmoidal Networks via Spiking Neurons." Neural computation 9 2 (1997): 279-304.

- Diesmann, Markus et al. "Stable propagation of synchronous spiking in cortical neural networks." Nature 402 6761 (1999): 529-33.

- Lee, Junhaeng et al. "Training Deep Spiking Neural Networks Using Backpropagation." Front. Neurosci. (2016).

- Memmesheimer, Raoul-Martin and Marc Timme. "Designing the dynamics of spiking neural networks." Physical review letters 97 18 (2006): 188101.

- Sporea, Ioana. "Supervised learning in multilayer spiking neural networks." Neural computation 25 2 (2012): 473-509.

7 Contributions by Team Members

- Anand yadav (15110015) - Literature Survey (Read various papers and articles), Understanding different SNN Models, Simulation of firing a Neuron, Understanding the working of STDP Algorithm
- Arik Pamnani (15110032) - Literature Survey (Read various papers and articles), Simulation of firing a Neuron, Understanding the working of STDP Algorithm, Understanding the implementation of Binary classification using SNN
- Gandhi Meet Bankim (15110049) - Simulation of Complex Control System consisting of training an SNN with Genetic Algorithm to direct an entity from a predefined starting point to a predefined destination point
- Ravi Shrimal (15110102) - Literature Survey (Read various papers and articles), Understanding the working of STDP Algorithm, Simulation of firing a Neuron, Understanding the implementation of Binary classification using SNN
- Shivdutt Sharma (15110125) - Literature Survey (Read various papers and articles), Simulation of firing a Neuron, Understanding the implementation of Binary classification using SNN, Understanding the working of STDP Algorithm