6CC3EEP
Development and Implementation of Neuromorphic Algorithms for Autonomous Spatial Navigation

Final Project Report

Author: Mohamed Aamir Faaiz

Supervisor: Dr Bipin Rajendran

Student ID: 1824375

April 29, 2021

Abstract

The focus of this dissertation is around the thermotaxis behaviour of C. elegans worm. It investigated the use of the Leaky Integrate and Fire (LIF) model for the neurons. The model was inspired by 10 neuron network believed to control the thermotaxis behavior in the nematode Caenorhabditis Elegans (C. Elegance). The results suggests that it is possible to model a neural comparator and gradient detectory circuit for the thermotaxis behaviour of the C. Elegans nematode based on the LIF spiking model, perhaps in a simpler and more efficient way. The main application of this circuit would be autonomous robots that can operate without human interaction and operation, a potential area for bio-inspired circuits to expand to.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Mohamed Aamir Faaiz

April 29, 2021

Acknowledgements

I would like to begin by thanking my Professor, Dr Bipin Rajendran, for his gracious support and encouragement throughout my thesis project, and without whom I would have struggled. It has truly been a pleasurable, educational experience to work under him.
I now move to another group: family and friends. I wish to first thank my Parents, for instilling in me the importance of education and hard work. Their many years of love, care and warmth were vital in moulding me into the person I am.
I would also like to thank undergraduate Electronic Engineering students Nikolay Gospodinov and Xorsheed Zanjani for working together to implement a full prototype.

Contents

Chapter 1

Introduction

With the advent of Neuromorphic Computing, and its subsequent growth in popularity and utility, the demand for computational solutions that extend beyond the Von-Neumann bottleneck is on the rise; we explore biologically inspired solution for spatial navigation inspired by Caenorhabditis elegans thermotaxis or C.Elegans for short. C.Elegans is considered as the fundamental organism in development biology and due to the simplicity of its mapped out genome, it is widely used as a foundation in experimental biology. It comprises of 302 neurons and approximately 5000 chemical synapses. The worm is capable of complex behaviours such as responding to external stimuli like heat(thermotaxis)[4] and chemicals(chemotaxis)[9]. In this paper we develop a spiking neural network(SNN) inspired by the thermotaxis behaviour of the worm to navigate physical environments based on tracking variables like heat or light intensity, which could find applicabiliy in real world robotic applications.

1.1 Project Aims and Objectives

The aim of this project is to explore and improve upon different techniques for mimicking the behaviour of C.Elegans in-particular Mimicking the worm — An adaptive spiking neural circuit for contour tracking inspired by C. Elegans thermotaxis[4].

1.1.1 Objectives

Our objective is to reduce the computational cost of An adaptive spiking neural circuit for contour tracking inspired by C. Elegans thermotaxis [4] through the usage of simpler mathematical models for the neuron behaviour, as well as implementing the solution in both hardware and software.

1.2 Approach

The project was completed in three main stages:

- Literature Review : we performed a survey of some related literature to gain an understanding of the problem.
- Implementation: we implemented and explored the changes we thought would aid us in achieving the aims of the project.
- Analysis of results: we collated and analysed the results to understand the impact our changes had.

1.3 Report Structure

In Chapter 2 we begin by exploring the background behind the fundamentals of Neuromorphic Computing and Spiking Neural Networks and a background behind C.Elegans thermotaxis behaviour. Whereas in Chapter 3 we tackle the problem of mimicking the behaviour of the worm, explain our proposed improvements, their implementations and their results. Finally in Chapter 5 we summarise the project, its results and our achievements, we also talk about the potential future direction this work can be taken in.

Chapter 2

Background

2.1 Spiking Neural Networks

Spiking Neural Networks(SNNs) are a third generation of Artificial Neural Networks(ANNs) that model the behaviour of neuron spiking inspired by biology. Compared to formal neural networks, spiking neural networks (SNNs) have some remarkable advantages, such as the ability to model dynamical modes of network operations and computing in continuous real time values, significantly reduced energy consumption of SNNs realized in novel hardware known as Neuromorphic Hardware. There exists various mathematical models to describe the spiking behaviour of biological neurons such as the Leaky Integrate and Fire, Izhikevich and the Adaptive Exponential Integrate and Fire (AEIF) model used in the paper [4], this project will be based on. Our main goal was to simplify this implementation and to use neuron models that aren't computationally intensive and that allow for a streamlined implementation in hardware such as FPGA or Neuromorphic chips. Therefore, it was decided to avoid the usage of neuron models which incorporate a 2^{nd} order non-linearity. For this project, we employ the the Leaky Integrate and Fire(LIF) model of neurons for computational efficiency and simplicity purposes. The dynamics of LIF neurons is described by:

$$C\frac{dV_m(t)}{dt} = -g_L(V_m(t) - E_L) + I_{app}(t) + I_{syn}(t) \tag{2.1}$$

$$When\, V(t) \geq V_T, V(t) \leftarrow E_L$$

Here $V_m(t)$ represents membrane potential, E_L is the resting potential($-70mV$), and C($300pF$) and $g_L(30nS)$ model the membrane capacitance and leak conductance. When $V_m(t)$ exceeds the threshold voltage V_T , it is reset to E_L and remains at that value till the neuron comes out of its refractory period ($t_{ref} = 3ms$).

Refractory period is implemented by storing the latest spike issue time, s_k^{last} of each neuron in a vector; the membrane potential of a neuron is updated only when the current time step $n > s_k^{last} + (t_{ref}/\Delta t)$ [7].

Figure 2.1 displays the LIF neuron response when excited with a constant input current. The figure shows the spike frequency of the neuron converging to a value of 300Hz, this is because the maximum frequency of the neuron is $1/t_{ref}$

The synapse, with weight $w_{k,j}$ connecting neuron k to neuron j, transforms the incoming spikes (arriving at time instants $s_k^1, s_k^2, s_k^3, ...s_k^n$) into a post synaptic current into the j^{th} neuron from the k^{th} at time $t > s_k^n$ is described below.

$$I_{syn,j}(t) = I_o w_{jk} \sum_{m=1}^{n} [e^{-(t-s_k^m)/\tau 1} - e^{-(t-s_k^m)/\tau_2}] \tag{2.2}$$

Hence, the post-synaptic current is generated in response to the spikes and has to be calculated at every time instant based on all the previous instants of time when the pre-synaptic neuron spiked. One way to efficiently determine this spike-driven synaptic current is to maintain a spike kernal matrix $K(t)$.

$$k_{syn,j}(t) = I_o \sum_{m=1}^{n} [e^{-(t-s_k^m)/\tau_1} - e^{-(t-s_k^m)/\tau_2}] \tag{2.3}$$

$$I_{syn}(t) = W \times K(t) \tag{2.4}$$

The double decaying exponentials represent the synaptic kernal with $\tau_1 = 15ms$ and $\tau_2 = 3.75ms$

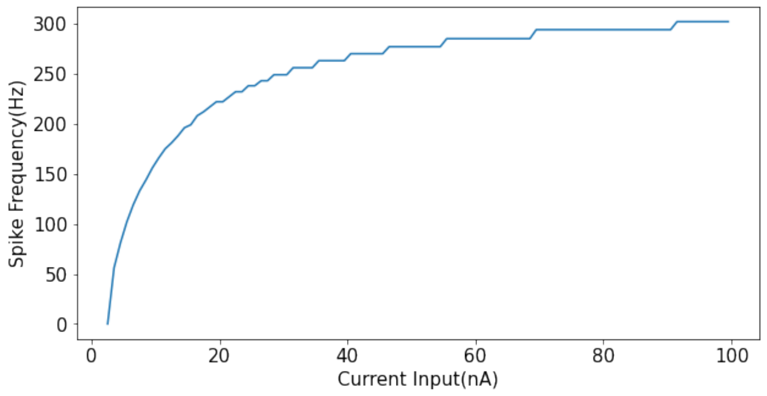

Figure 2.1: LIF neuron response when excited with a constant input current

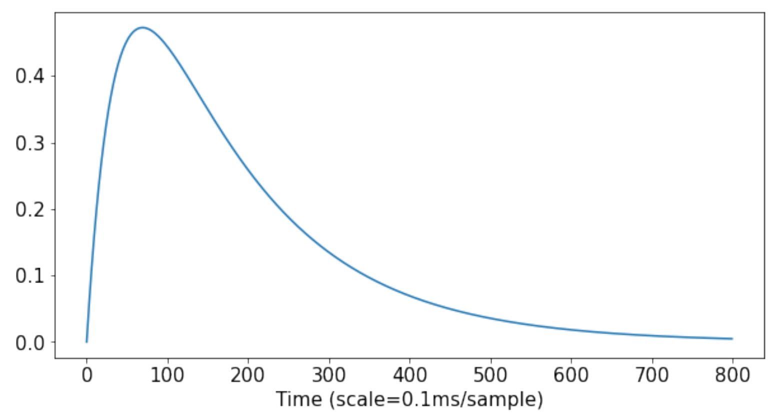

Figure 2.2: Synaptic Current Kernal

2.1.1 Numerical Methods for solving ODEs

Modelling changes in membrane potential under conditions where the stimulus current differs in a time-dependent manner is necessary for this project. As a result, rather than obtaining the solution to V(t) analytically, Eq. 2.1 is solved numerically. The Euler method is employed for this requirement as it is a straightforward numerical method for ODEs. The Euler is method for solving Eq. 2.1 is represented as follows :

$$\frac{dV}{dt} = f(V, t) \tag{2.5}$$

$$V(t + \Delta t) = V(t) + \Delta t(\frac{dV}{dt}) \tag{2.6}$$

$$V(t + \Delta t) = V(t) + \frac{\Delta t}{C}[-g_L(V(t) - E_L) + I_{app}(t) + I_{syn}(t)] \tag{2.7}$$

The numerical approximation for Eq. 2.1 is represented by Eq. 2.7, where Δt represents the euler time step.

2.1.2 Neural Network with Non-Plastic Synapses

To demonstrate the dynamics of the Leaky Integrate and Fire model, a simple simulation comprising of 2 neurons connected through an excitatory synapse of weight $w_{1,2} = 10,000$ and a constant input current of magnitude $2800pA$ is demonstrated in Figure 2.3. It is possible to modulate the spike frequency of the output neuron through customisation of the synaptic strength $w_{1,2}$. A higher value of $w_{1,2}$ will result in a greater spike frequency response, due to the second neuron being stimulated with a higher post synaptic current. Additionally, $w_{1,2}$ can be an inhibitory synapse, this can be achieved by lowering the magnitude of $w_{1,2}$ to reduce the probability of the output neuron from spiking or assigning a negative value to $w_{1,2}$, which will result in a negative post-synaptic current into N_2. Figure 2.4, represents the neuron dynamics when simulated with an inhibitory synapse of strength -500.

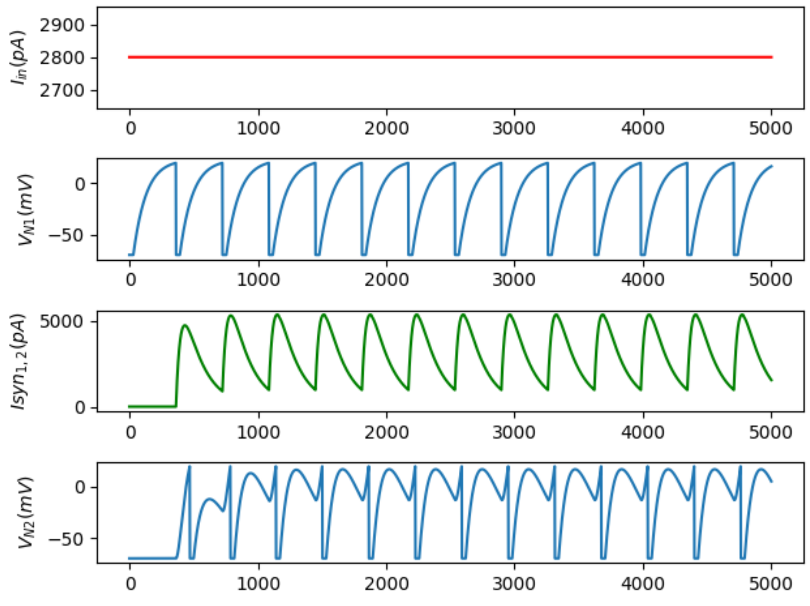

Figure 2.3: 2 Neuron Simulation $w_{1,2} = 10000$

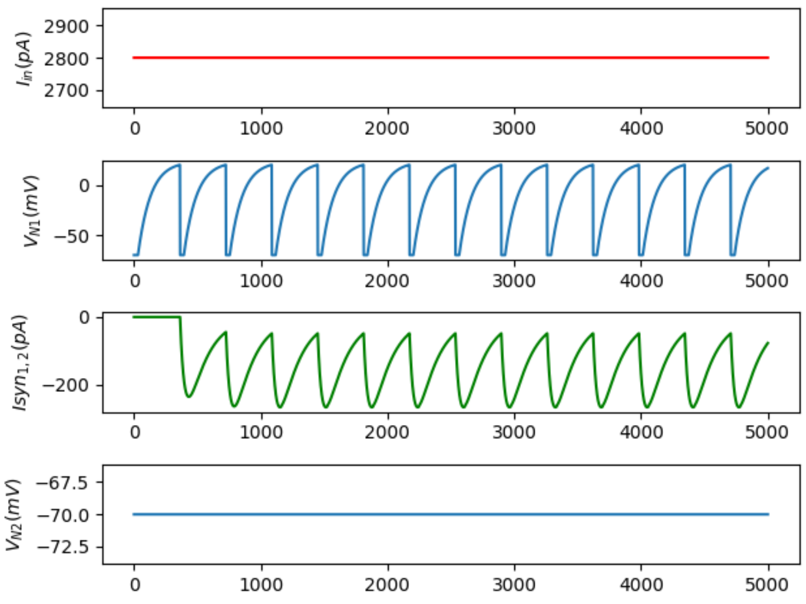

Figure 2.4: 2 Neuron Simulation $w_{1,2} = -500$

2.2 Low Precision Weight Encoding

In the field of Neuromorphic Hardware, synaptic connections between neurons are implemented using low precision weight encoding rather than the typical 32bit floating point weights used in ANNs. This is achieved by going beyond traditional CMOS technologies and utilising a new generation of nanoscale devices, particularly memristors, where the synaptic strength is encoded using resistance or the conductance of the device. One such category of memristive devices is known Phase Change Memory(PCM). Neuromorphic chips such as Intel's Loihi[1] and IBM's TrueNorth[3] employ such nanoscale devices in their architecture. If this project were to be taken to the next level of implementation in neuromorphic hardware, it will be essential to understand how the dynamics of the neural circuit will be impacted during this conversion. Therefore, a simple model was designed to convert the 32bit synaptic strengths to low precision(i.e 4/3 bit) encoding whilst also incorporating an element of noise in the reading of these weights. Figure 2.5 displays how the weights are encoded with 4 bit precision and read noise of standard deviation $\sigma = 0.1$.

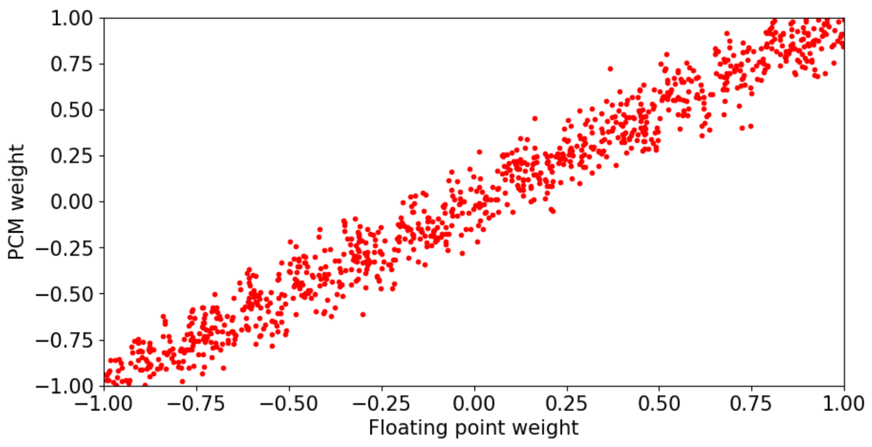

Figure 2.5: PCM Synaptic Weight Encoding

2.3 C.Elegans Thermotaxis

The C. Elegans worm interacts with the environment through it's 'brain' functionality. In order to make the worm do contour tracking and simulate the arising thermotaxis behaviour, a set temperature T_s, which is desired and considered 'ideal' is assumed. If away from T_s, random exploration of the environment should occur. This movement should favour areas where the temperature is more desirable. To do this, there is a comparator which checks if $T > T_s$ or

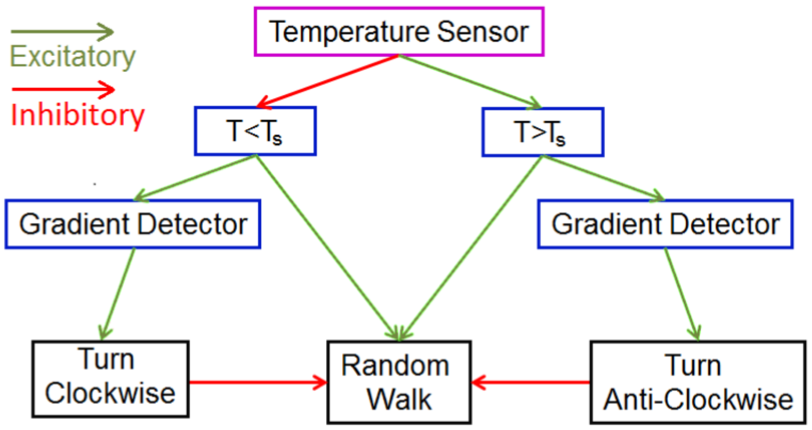

Figure 2.6: Block diagram for the bio-inspired neural circuit for contour tracking. The model steers the worm to regions close to the set point T_s by controlling the probability for random walk and deterministic turns. Adopted from [4]

$T < T_s$. If a temperature difference is detected, the worm will move in a random direction between -30^o and 30^o. However, before conducting that move, there is an additional check if the worm is moving towards a more favourable region or not. This happens through a gradient check $dT/dt > 0$ or $dT/dt < 0$. If a gradient is detected the worm will have a rapid movement at a degree of 25^o anticlockwise if the gradient is positive and to the clockwise direction if the gradient is negative. Furthermore, in order to explore larger area quicker, the worm will incrementally increase it's speed the longer it stays in a region of unfavourable temperature. On the contrary, when the worm is near the favourable temperature it will exponentially decay it's speed in order to prevent overshooting the target or high degree rotations. Finally, the movement inside the isotherm ($T = T_s$) will still be a random exploration within a smaller degree amplitude.

Chapter 3

Design & Implementation

3.1 Project Structure

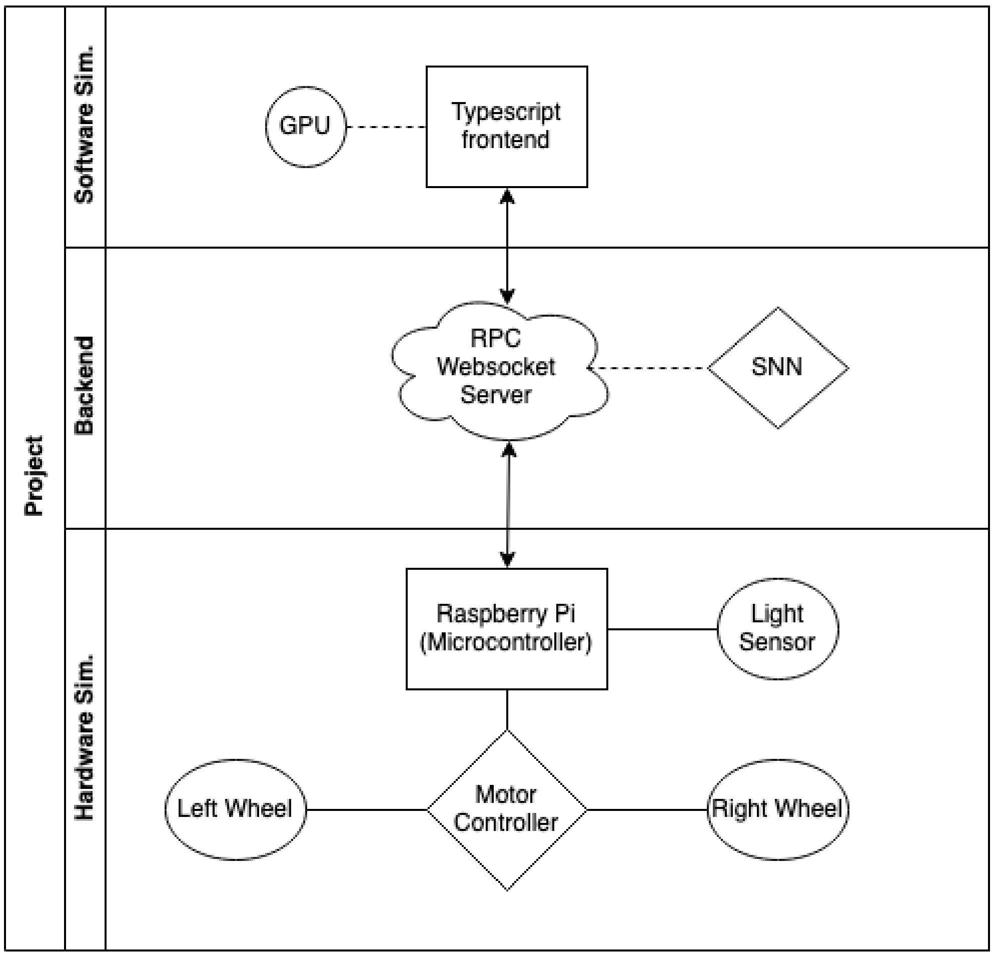

Figure 3.1: Project Structure

Figure 3.1, represents how the project was structured to implement an end-to-end software and hardware based solution to simulate the thermotaxis behavior of the worm. For the software based solution, the worm engages in tracking thermal isotherms and in hardware this is simulated with light intensity tracking instead of temperature.
This report will highlight the design and implementation of the Spiking Neural Network architecture component of the project.

3.2 C.Elegans Spiking Neural Network

The C.Elegans thermotaxis functionality can be simplified to four basic building blocks to handle the temperature contour tracking of the worm:

1. Comparators to verify if $T > T_s$ or $T < T_s$.
2. Gradient Detectors to detect if $dT/dt > 0$ or $dT/dt < 0$
3. Worm speed control mechanism
4. Random Walk

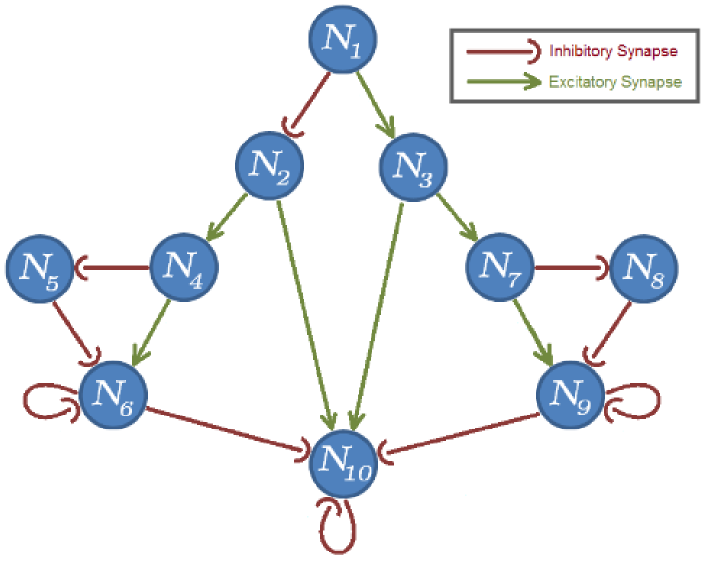

Figure 3.2: Schematic of the neural circuit for contour tracking. N_1 is a temperature sensitive input neuron. N_1-N_3 is a positive comparator, and N_7-N_8-N_9 is a gradient detector. Similarly, N_1-$N_2$is a complementary comparator, and N_4-N_5-N_6 is a gradient detector. N_6 and N_9 control the deterministic turns, while N_{10} controls the random exploration. Adopted from [4]

As mentioned in the Introduction section of this report, the behaviour to be replicated has largely been inspired by and described in the 'Mimicking the worm – an adaptive spiking neural circuit for contour tracking inspired by C. Elegans thermotaxis' paper by A. Bora, A. Rao, and B. Rajendran [4]. To achieve the thermotaxis and contour tracking functionality, we are using a 10-neuron neural circuit consisting of the starting temperature sensitive input neuron N_1, the two temperature comparator neurons N_2 and N_3, the positive and negative gradient detectors represented in the two micro-circuits N_4-N_5-N_6 and N_7-N_8-N_9, and finally the random walk neuron N_{10}.

In this report, we design and analyse the results of the comparator and gradient detector functionality of the thermotaxis.

3.3 Temperature Comparator

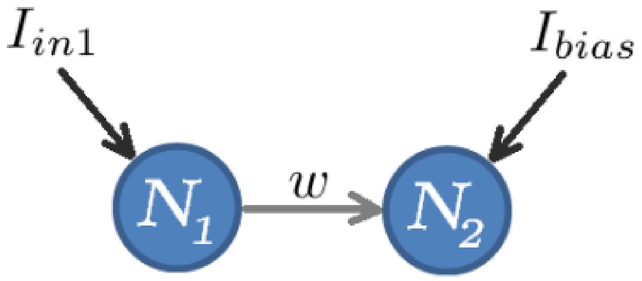

Figure 3.3: Comparator Microcircuit, where the output spike frequency at N2 exhibits a linear dependence on ambient temperature T . N1 is the sensory neuron, whose input current I_{in1} is linearly proportional to T . Parameters w and I_{bias} can be used to tune the response curve of the circuit. Adopted from [4]

It must be noted that this section is building on top of the work done by my colleague Xorsheed Zanjani. My contribution to this section is mainly integrating the comparator circuit in the back-end server and improving the existing neuron model by implementing a refractory period component in the spiking behavior, this incorporates more resilience in the neuron model by preventing it from saturating during the course of longer simulations and additional checking has been implemented to eliminate instances where the membrane potential doesn't go below it's resting potential (E_L) during simulations. These new additions to the neuron model, are necessary to prevent undesired spiking behaviour for practical and long term usage of the circuit. Furthermore, the optimal comparator model was extracted from the tests, and was re-run with low precision weight encoding.

Figure 3.3, shows the comparator circuit used in this project. I_{in1} has a linear dependence on the instantaneous temperature T of the environment. This relationship is represented in Eq.

3.1. Additionally, the worm could track any physical variable i.e light as long as the sensory neuron has a linear dependence on the variable.

$$I_{in1} = \alpha_T + \beta_T \times (T - T_s) \tag{3.1}$$

where $\alpha_T = 3750pA$ and $\beta_T = 350pA/K$

Approach

Our approach to testing comparator circuit:

1. Simulate comparator circuit with a temperature range from 18C - 23C and measure spike frequency of response of N_2 and N_3.

2. Estimate spike frequency response for the same temperature range but with different synaptic strength and I_{bias}.

3. Gather the output neuron response data for varying parameters and visulise.

4. Evaluate results against the data presented in [1]

5. Extract optimal parameter combination and re-run the simulation with low precission weight encoding.

3.3.1 Positive Comparator

For the positive comparator, when the instantaneous temperature T of the worm environment exceeds the set point threshold temperature T_s, a ramp response is what should be expected between temperature and spike frequency of the output neuron of the comparator circuit. This is displayed in Figure 3.4. Moreover, varying the the parameters I_{bias} and $w_{1,3}$, it is possible to adjust the gradient of the response of the circuit. Additionally, we see the circuit has a maximum output frequency in the tested temperature range of $\approx 80Hz$, the analysis shows that the system output spikes sparsely and spike frequency of the system doesn't exceed $150Hz$, which means the model will be energy efficient in a hardware implementation and the model is biologically plausible. Analysis of the data presented in Figure 3.4, shows the optimal parameter combination for a set temperature threshold T_s of 20C is $I_{bias} = -2500pA$ and $w_{1,3} = 7000$.

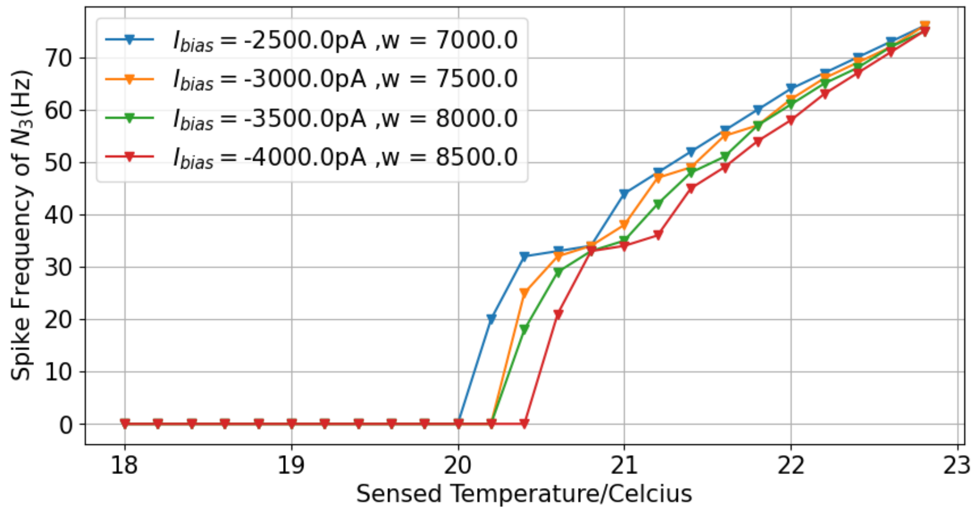

Figure 3.4: Positive Comparator Response, $T_s = 20C$

3.3.2 Positive Comparator with Low Precission Weights

Using the optimal parameter combination from the previous experiment, the simulation was re-run with 4bit low precission weight encoding and a read noise of standard deviation $\sigma = 0.1$. The results were gathered like previously.

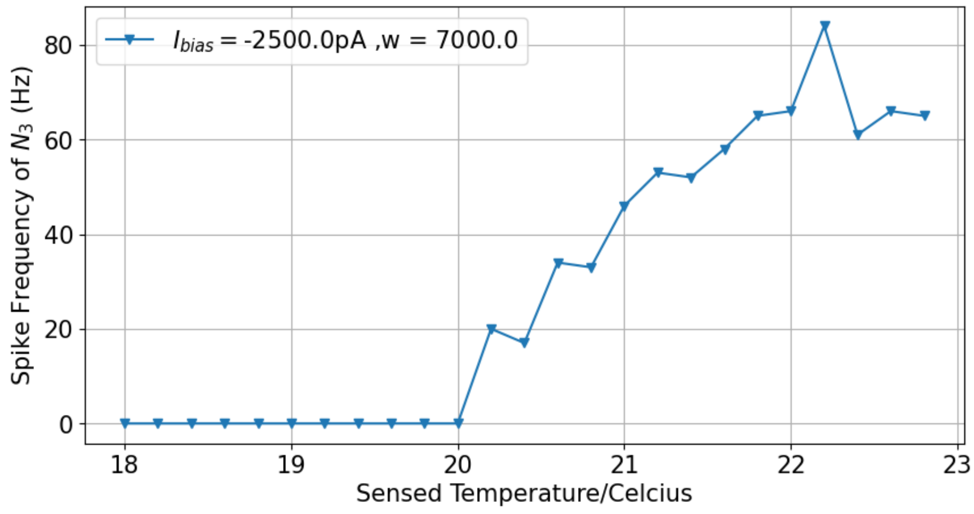

Figure 3.5: Positive Comparator Response with 4bit PCM Weights, $T_s = 20C$

Figure 3.5, shows the response of the circuit with the low precission weight encoding implementation. The response shown, indicates that the desired network behavior for the positive comparator circuit with PCM weights and the model has been achieved is plausible for neuromorphic hardware implementation.

3.3.3 Negative Comparator

The negative comparator, should provide a complementary response to that of the positive comparator such that Neuron 2 will spike only when the instantaneous temperature T of the worm environment is below the set temperature threshold T_s. This response can be achieved by using parameter combinations complementary to that of positive comparator i.e positive I_{bias} and negative $w_{1,2}$. Figure 3.6, shows we have achieved the desired complementary response to that of the positive comparator. Analysis of the data presented in Figure 3.6, shows the

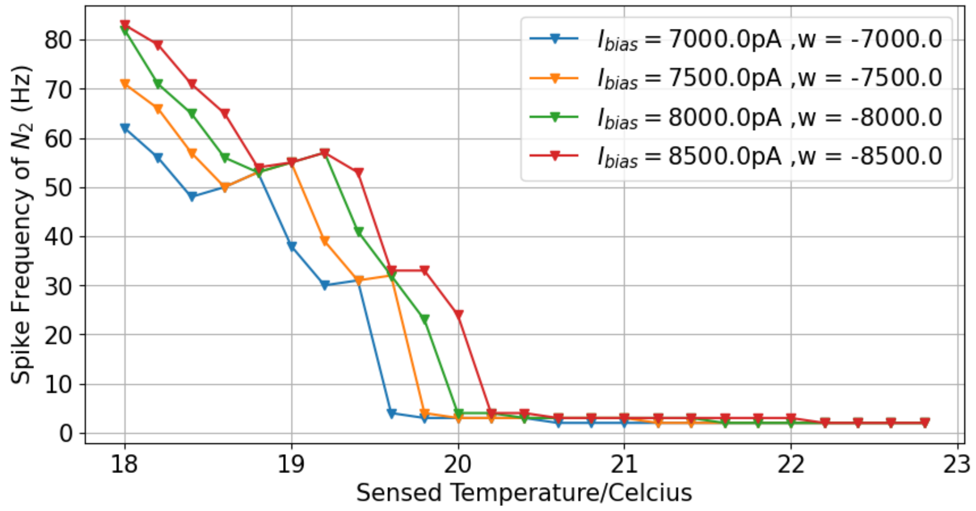

Figure 3.6: Negative Comparator Response, $T_s = 20C$

optimal parameter combination for a set temperature threshold T_s of 20C is $I_{bias} = 8000pA$ and $w_{1,3} = -8000$.

3.3.4 Negative Comparator with Low Precision Weights

Figure 3.7, shows the response of the circuit with the low precision weight encoding implementation. The response shown, indicates that the desired network behavior for the negative comparator circuit with PCM weights has been achieved and that the model is plausible for neuromorphic hardware implementation.

3.3.5 Comparator Circuit Testing & Evaluation

From the analysis of results presented and the optimal parameters extracted, a simulation environment was set up with a fluctuating temperature for the comparator network $N_1 - N_2 - N_3$ and a visual response from the circuit was extracted as shown in Figure 3.8

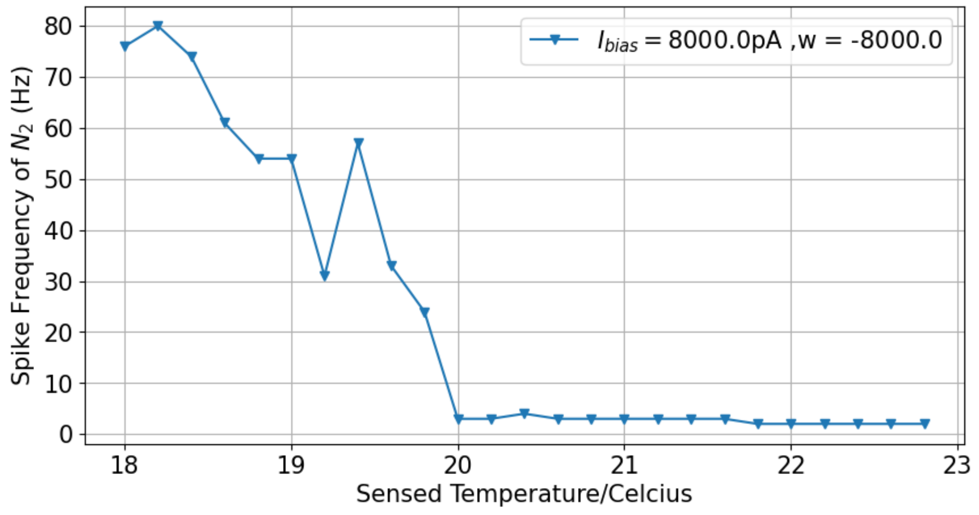

Figure 3.7: Negative Comparator Response with 4bit PCM Weights, $T_s = 20C$

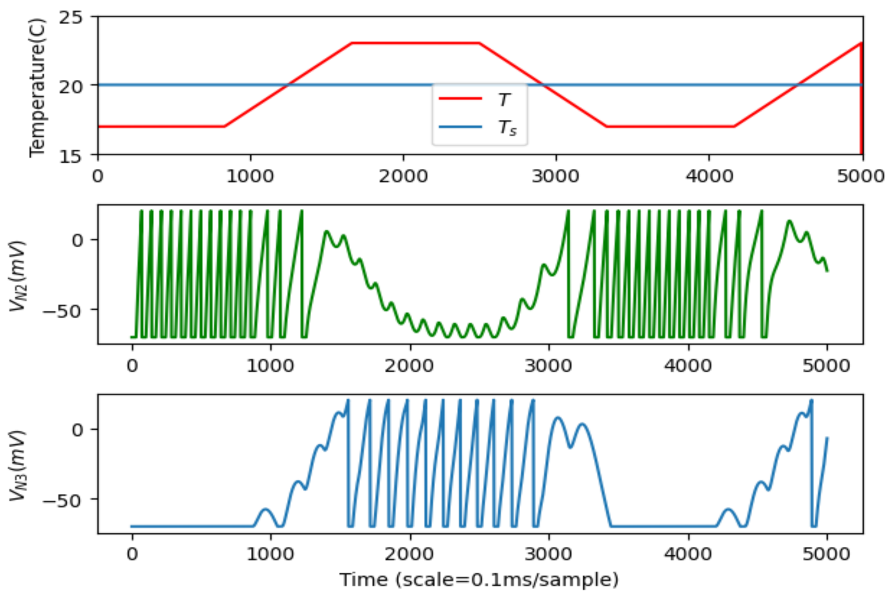

Figure 3.8: Comparator Simulation with fluctuating instantaneous temperature

Figure 3.8 displays the membrane response for N_2 and N_3 and what we observe is a complementary response from each neuron i.e when N_2 is spiking N_3 will not be spiking and vice versa. Therefore, the desired response has been achieved. Additionally Figure 3.9, shows the membrane response for when the system is simulated with low precission weight encoding and a similar complementary response from both neurons is observed, whilst adhering to the T_s threshold set point temperature.

In both test cases, the desired response from the circuit has been achieved and this component of the neural network can be concluded as successful.

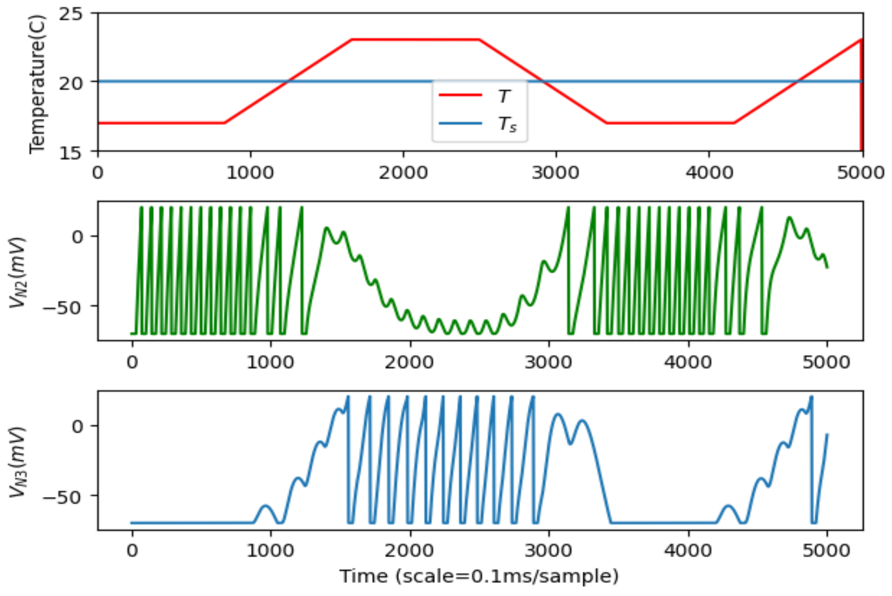

Figure 3.9: Comparator Simulation with PCM weights

3.4 Temporal Gradient Detector

The basic operation of temperature T gradient detection from first principles is resembled as follows:

$$\frac{\delta T}{\delta t} = \frac{f(T + \delta t) - f(T)}{\delta t} \tag{3.2}$$

To mimic this algorithmic operation, a 3 neuron network is implemented as shown in Figure 3.10. The excitatory path way $N_4 - N_6$ essentially computes the $\frac{f(T+\delta t)}{\delta t}$ operation and the inhibitory pathway $N_4 - N_5 - N_6$ essentially will compute the $\frac{-f(T)}{\delta t}$ operation.

Current transmission takes longer through $N_4 - N_5 - N_6$ than it does through $N_4 - N_6$. The successful spike frequency of N_6 can resemble a derivative action since the currents in these pathways have opposite signs.

The network should only send out a spike when there is a gradient detected. Therefore, there has to be some sort of competition between the current that is comming in both pathways, and they should be trying to cancel each other out, such that only when there is a change in N4-N6, a spike response is created.

w_{56} should adapt in such a way that the net current that is comming into N_6 should be zero.

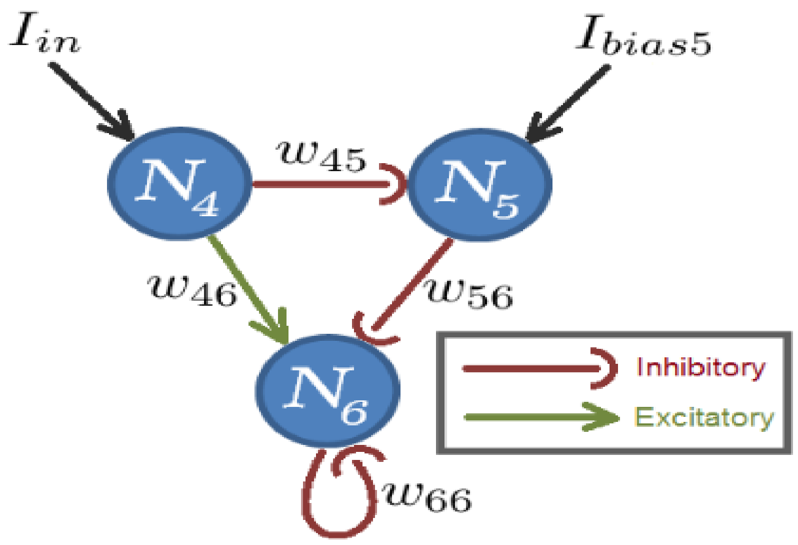

Figure 3.10: The neural micro-circuit for gradient detection. N_6 spikes in response to positive gradients in the input current I_{in} at N_4. Synapses w_{45} and w_{56} are inhibitory, while w_{46} is excitatory. The strength of w_{56} depends on the time of arrival of spikes at N_5. Adopted from [4]

3.4.1 Memoryless Weight Adaptation Rule

In the paper 'Mimicking the worm – an adaptive spiking neural circuit for contour tracking inspired by C. Elegans thermotaxis' paper by A. Bora, A. Rao, and B. Rajendran [4], a memoryless weight adaptation rule for the synaptic strength between N_5 and N_6 was derived and is described by Eq. 3.3. The rule is considered memoryless as it does not require prior data of the spiking behaviour of the neurons in the circuit to optimise the synaptic strength $w_{5,6}$. This synaptic strength optimisation was implemented to run synchronously with the Leaky Integrate & Fire function.

$$\Delta w_{56} = \frac{c}{\tau a}\delta(t - t^{N_5}) + \frac{d - w_{56}}{\tau a}\Delta t \tag{3.3}$$

(Adopted from [4] ,eq.9)

where t^{N_5} indicates the time instants where Neuron 5 has spiked.

Equation 3.3 describes a spike triggered weight adaptation rule for w_{56}. $\frac{c}{\tau a}\delta(t - t^{N_5})$ describes the increment in w_{56} for when Neuron N_5 is fired. The synapse's natural propensity to shift its weight towards the value d is represented by $\frac{d-w_{56}}{\tau a}\Delta t$.

Impact of tuning parameters

The memoryless weight adaptation rule comprises of 3 parameters that we could tune. However, the basis behind tuning these parameters is given below.

1. τa: This variable controls the pace of the optimisation of w_{56}, which has a knock-on effect on the spiking behaviour of N_6 and sensitivity of the gradient detector. If τa is too small, this would increase the optimisation pace of w_{56} and cause w_{56} to reach a stable point before N_6 spikes for a sufficient period of time. Additionally, if τa is too large, the optimisation pace of w_{56} will be too small, resulting in unwanted spiking behaviour in N_6 even after there is no derivative in the input current comming in and thus increasing the response time of the circuit.

2. w_{46}: This variable controls the amplification of the difference between inputs from N_4 and N_5 into N_6. This is because the converging point of the weight adaptation rule increases with w_{46}. A decrease in w_{46} would lead to a decrease in sensitivity and will increase response time. The impact of choosing a large value would result in a quick response time of the circuit resulting in unwanted spiking behaviour in N_6 even after there is no derivative in the input current comming in.

3. w_{66}: This is a self-inhibitory synapse for the output neuron N_6 to incorporate stability in the response of the circuit.

3.4.2 Genetic Algorithm Parameter Optimisation

The memory-less weight adaptation rule, along with the circuit parameters such as I_{bias} and the synaptic connections between neurons incorporate a high degree of complexity when trying to find the optimal parameter combination for an effective solution to the problem and doing this optimisation by hand is certainly not the way around. Moreover, optimising the parameters to achieve the desired response in this case can be considered as a NP-Hard problem and in such a scenario genetic algorithms prove to be useful in providing close to optimal solutions in a reasonable period of time. In Genetic Algorithms, we have a pool or a population of plausible solutions to the given problem. These solutions then undergo recombination and mutation (natural selection), producing new children, and the process is repeated over various generations. Each individual is assigned a fitness value (based on a scoring function) and the fitter individuals are given a higher probability to mate and yield more “fitter” individuals. This is in line with the Charles Darwin’s theory of biological evolution by natural selection.

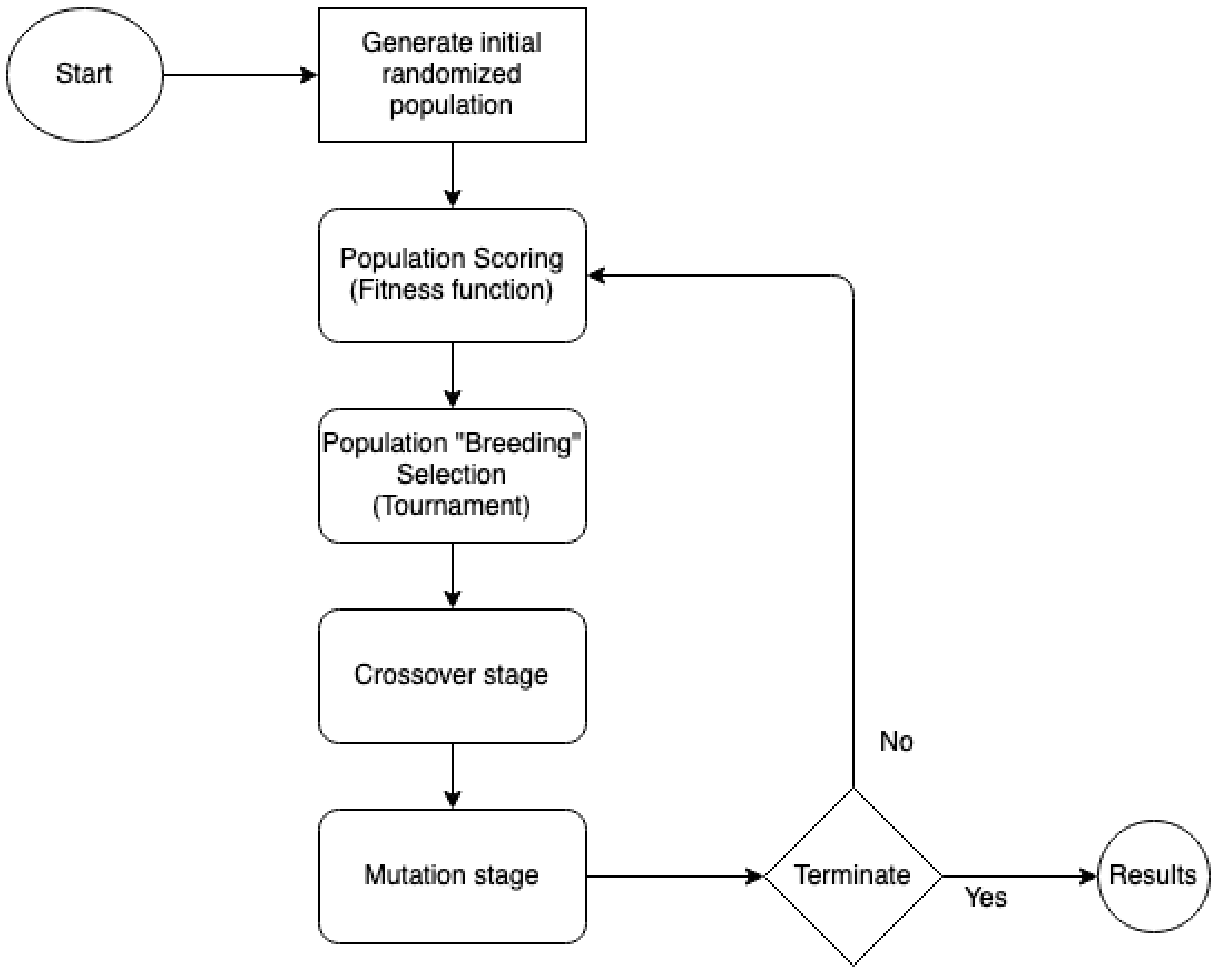
Start
Generate initial randomized population
Population Scoring (Fitness function)
Population "Breeding" Selection (Tournament)
Crossover stage
Mutation stage
Terminate
No
Yes
Results

Figure 3.11: Genetic Algorithm for Weight Adaptation Rule Optimisation

In this way we keep "evolving" better individuals or solutions over generations, till we reach a stopping criterion [10][6][6].

Approach

We begin by creating a vector of current values ranging from $2600pA$ to $3000pA$ for the input of the circuit which comprises of 6 gradient changes which includes positive, negative and zero gradient signals within a period of 30 seconds. This current vector will be the basis of evaluating when Neuron 6 should spike and estimating the fitness of the parameter combination for the genetic algorithm. The visualisation of the input current in Figure 3.12, displays the time periods where we would want Neuron 6 to spike. From this figure we conclude that it's undesirable for N_6 to spike between time periods 0-5seconds and 15-25seconds as there is no positive gradient in the input current. If N_6 were to fire in these time periods, the parameter combination will be penalised or assigned a lower fitness value and vice versa. The fitness

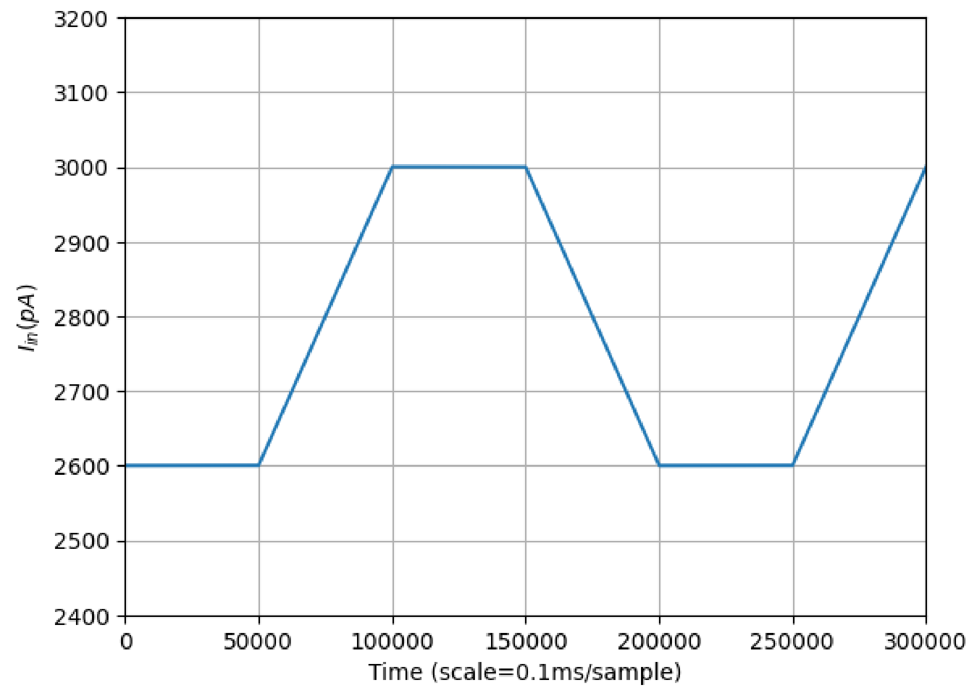

Figure 3.12: Input Current Generation

function is described in detail, later on in this section.

Parameter Ranges

We optimise the parameters $w56_{initial}$,w_{45},w_{46}, w_{66}, I_{bias}, c, τa, d. Additionally, we assigned each parameter a plausible range of values as shown below.

Parameter	Range	Precision
$w56_{initial}$	-10000 to -500	0dp
w_{45}	-5000 to -500	0dp
w_{46}	5000 to 15000	0dp
w_{66}	-6000 to -2000	0dp
I_{bias}	2000pA to 8000pA	0dp
τa	5s to 30s	2dp
c	$5Hz^-1$ to $20Hz^-1$	2dp
d	-10000 to -500	2dp

Table 3.1: Parameters

Fitness Function

From Figure 3.12, we define time periods where N_6 spiking is desirable(right_spikes) and non-desirable(wrong_spikes). Using this data, we score the population of parameters on each generation. To penalise and reward in our scoring, we take out 12 points for each correct spike and add 12 points for each wrong spike. Furthermore, when there are no wrong spikes we take out 50 points and subsequently add 50 points whenever there are no correct spikes. It must be noted that a lesser score is better. The equation is represented below.

$$x = len(wrong\_spikes)$$

$$y = len(right\_spikes)$$

$$score = 100 + 12x - 12y - \begin{cases} 50 & x < 1 \\ 0 & x \geq 1 \end{cases} + \begin{cases} 50 & y < 1 \\ 0 & y \geq 1 \end{cases}$$

GPU Implementation

Initial testing of the genetic algorithm optimisation was done on CPU and this was a time consuming task. It was then decided to translate the existing Python code to achieve GPU acceleration using NVIDIA's CUDA and NUMBA, these packages tremendously simplified the process of achieving that. Numba supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels and device functions conforming to the CUDA execution model. Numba virtually makes it seem that the kernel has direct access to NumPy arrays, which streamlines the process of writing GPU kernels. NumPy arrays passed to the kernel as arguments are implicitly moved between the CPU and the GPU [5].
GPU acceleration of the genetic algorithm along with the number of parameters and their ranges for the optimisation improved in speed by 10×. Note, the GPU used for this test was the NVIDIA GeForce MX330.

Optimal Parameters

The final results of the optimisation problem is shown below, where the fitness function is evaluated for 10 generations/iterations.

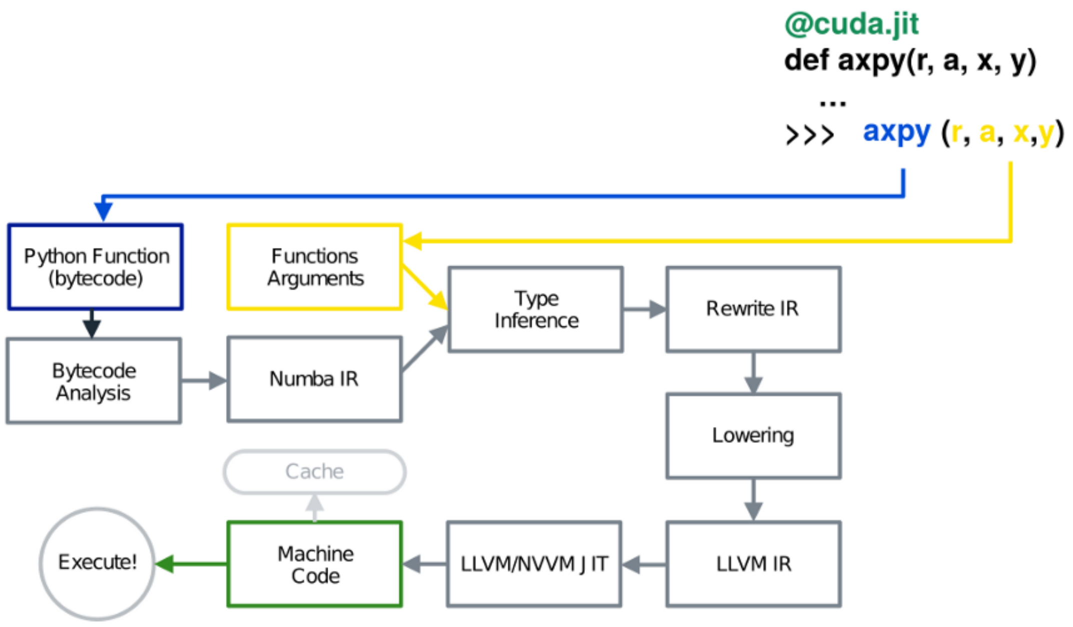

Figure 3.13: A high-level view of Numba's compilation and execution pipeline. Adopted from [8]

Parameter	Value
$w_{56_{initial}}$	-2742
w_{45}	-1,869
w_{46}	6,840
w_{66}	-4,202
I_{bias}	3811pA
τa	23s
c	8.37Hz^-1
d	-5720

Table 3.2: Optimal parameters for gradient detector circuit

3.4.3 Gradient Detector Testing & Evaluation

Using the optimal parameter configuration described in table 3.2, a 10 seconds simulation was created, stimulating N_4 with a fluctuating input current I_{in} as shown in figure 3.14. Like previously, the input current has 6 gradients in the simulation time. Looking at the response of N_6, it is observed that the circuit is functioning as required, such that N_6 is spiking when there is a positive gradient in the input current. Moreover, it also observed that the memory-less weight adaptation rule in running synchronously with the simulation. Additionally, a graph $\delta(t - t^{N_6})$ is also plotted below; this graph displays a clearer interpretation of where exactly

N_6 is spiking during the simulation.

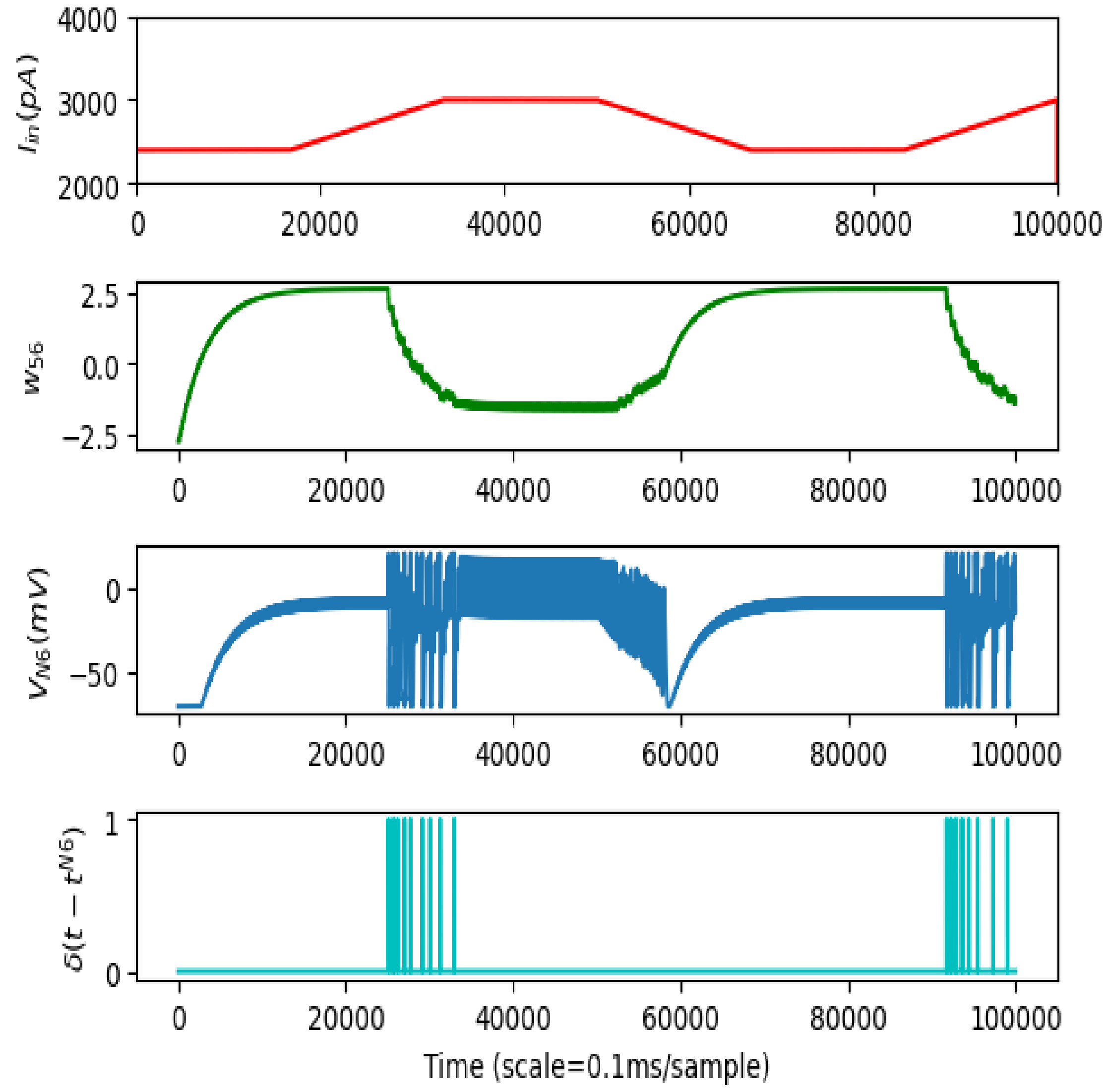

Figure 3.14: Simulation of Gradient Detector

We have achieved the desired response for the gradient detector circuit and this component of the neural network can be concluded as successful. However, there could be more conditions set in the genetic algorithm scoring component to ensure that N_6 is at its resting potential when there is no derivative in the input current comming in. Looking at Figure 3.14, we see that we could further optimise this response from N_6 which will result in a smoother output with minimal noise.

Taking this further, the gradient detector was simulated with 4 bit precision weight encoding and with a read noise of standard deviation $\sigma = 0.1$. The simulations were run with the same fluctuating input current vector as previously to objectively compare the results. Moreover, due to the random noise present in the PCM synaptic strengths; the simulation was run for 10 iterations to understand the robustness of the model when utilising this technology. In all test cases, there was significantly more noise in the response of the circuit. Moreover, in certain test cases N_6 was spiking even in the presence of no positive gradient in the input current. However, this spiking was relatively sparse when compared to the spike rate in regions where there was a positive gradient in the input current.

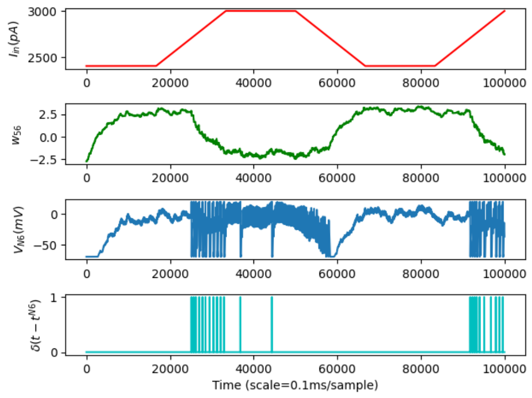

Figure 3.15: PCM Gradient Detector Simulation 1

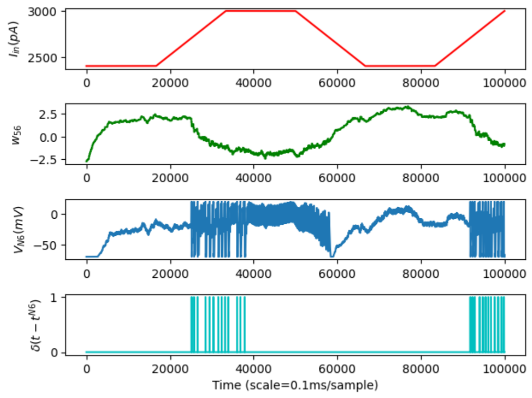

Figure 3.16: PCM Gradient Detector Simulation 2

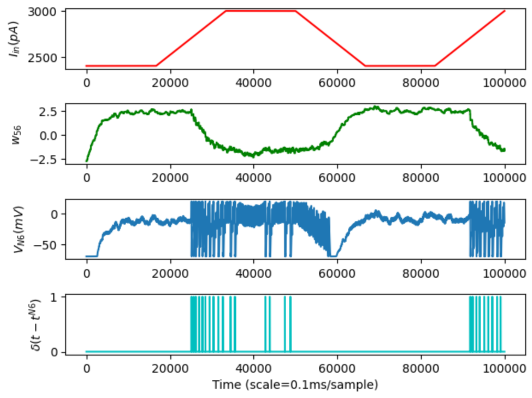

Figure 3.17: PCM Gradient Detector Simulation 3

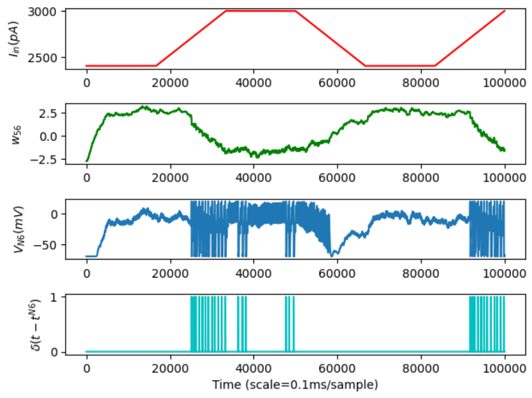

Figure 3.18: PCM Gradient Detector Simulation 4

The Figures above display the responses taken from four of the simulations carried out. Figure 3.16. displays the ideal response in these test cases as there is no spiking behaviour in the presence of non-positive gradients.

Due to the random noise present when using PCM weights the solution is not the most ideal for applications which require a high degree of precision. However, optimising the parameters using these low precision weights in the genetic algorithm was not tested and this could potentially be an area to look further.

Chapter 4

Results/Evaluation

The main target for this project was to design and evaluate the SNN architecture component of the project. To evaluate the models covered in this paper and the thermal tracking capability of the worm, the SNN was integrated on the back-end and utilised in the software and hardware based simulation environments created by Nikolay Gospodinov.

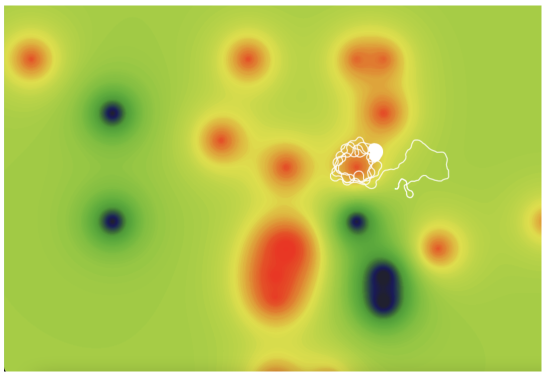
Figure 4.1: Simulation 1: Randomised Initial Position, $T_s = 20C$

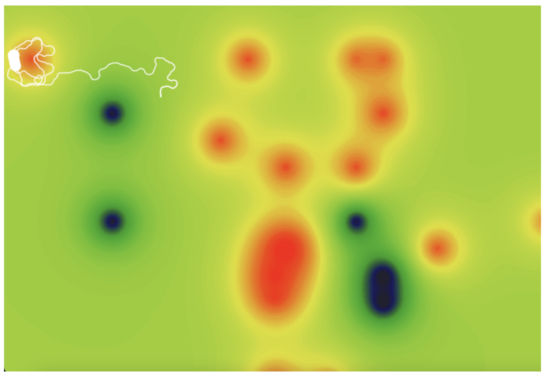
Figure 4.2: Simulation 2: Manually Selected Initial Position, $T_s = 20C$

Figures 4.1 and 4.2 represents the software based thermal environment(heat map) with temperatures ranging from 17-23C. The white line is essentially the trajectory that the worm follows based on the spatial temperature and what we see is that the worm is capable of contour tracking and navigating to the environment of the set point temperature $T_s = 20C$. Figure 4.1 displays the first test that was done where the initial position of the worm on the 2-D heat-map was randomised whereas Figure 4.2 displays the second test where the initial position of the worm was manually set. In both test cases the worm is capable replicating the same contour tracking behaviour.

Chapter 5

Conclusion and Future Work

5.1 Summary of Work

We have researched the thermotaxis behaviour of C.Elegans and implemented spiking neural networks using the leaky integrate and fire neuron model to replicate the behaviour of the worm.
Furthermore, this architecture was later used to implement an end to end software-hardware based solution to simulate the behaviour of the thermotaxis.

5.2 Achievements

The results of this dissertation suggests that it is possible to model the thermotaxis behaviour of C.Elegans using the Leaky Integrate & Fire neuron model. Moreover, the results from the simulations suggest that the neurons spike sparsely and are within ranges for biological plausibility.
Additionally, the implementation of the genetic algorithm to optimise the parameters for the neural network in a reasonable amount of time will be tremendously helpful for future usage and extend-ability to other practical applications using networks similar to this and beyond.

5.3 Future Work

The next step for this project would be to implement the solution using FPGA in the back-end component of the project. This would bring about a tremendous speed up in terms of performance of the network.

The further next step would be to utilise the Nengo framework [2] for SNNs to replicate the neural circuit in this paper and perform the simulation in neuromorphic hardware such as Intel Loihi [1]. Chips like this this are designed solely for the purpose of SNNs and will be the optimal solution in terms of performance and energy efficiency for this project.

Finally, this model could be extended to navigate 3-dimensional environments such that would be in the case of an autonomous quadcopter navigating environments using solely ultrasonic sensors. This will require more sensory neurons and defining new mapping models to convert data from the sensors to an input current for the circuit.

References

[1] Intel. 2021. Neuromorphic computing - next generation of ai. Available at `https://www.intel.com/content/www/us/en/research/neuromorphic-computing.html`.

[2] Nengo.ai. 2021. Nengo. Available at `https://www.nengo.ai`.

[3] Research.ibm.com. 2021. Ibm research:brain-inspired chip. Available at `https://www.research.ibm.com/articles/brain-chip.shtml`.

[4] Ashish Bora, Arjun Rao, and Bipin Rajendran. Mimicking the worm — an adaptive spiking neural circuit for contour tracking inspired by c. elegans thermotaxis. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 2079–2086, 2014.

[5] NYU Center for Data Science. Introduction to numba: Cuda programming. Available at `https://nyu-cds.github.io/python-numba/05-cuda/`.

[6] Farshad Kiyoumarsi. Mathematics programming based on genetic algorithms education. *Procedia - Social and Behavioral Sciences*, 192:70–76, 2015.

[7] Shruti R. Kulkarni, John M. Alexiades, and Bipin Rajendran. Learning and real-time classification of hand-written digits with spiking neural networks. In *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 128–131, 2017.

[8] Graham Markall. The life of a numba kernel: A compilation pipeline taking user defined functions in python to cuda kernels. Available at ://tinyurl.com/3uu4pejc.

[9] Shibani Santurkar and Bipin Rajendran. C. elegans chemotaxis inspired neuromorphic circuit for contour tracking and obstacle avoidance. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2015.

[10] TutorialsPoint. Genetic algorithms - introduction. Available at https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm.

Appendix A

Extra Information

A.1 Parameters Used In Simulation

Parameter	Value	Parameter	Value
C	300pF	Δt	0.1ms
g_L	30nS	τ_1	15ms
E_L	-70mV	τ_2	3.75ms
V_T	20mV	c	$8.37Hz^-1$
τ_a	23ms	d	-5720

A.2 Synaptic Strengths & Bias Currents

Parameter	Value	Parameter	Value
w_{12}	-8000	w_{13}	7000
I_{bias_2}	8000pA	I_{bias_3}	-2500pA
w_{45}	-1869	w_{78}	-1869
w_{46}	6840	w_{89}	6840
w_{66}	-4202	w_{99}	-4202
I_{bias_5}	3811pA	I_{bias_8}	3811pA

Appendix B

User Guide

B.1 Instructions

This source code presents the code used to implement and design the SNN.

B.1.1 SNN Implementation & Experimentation

Prerequisites:

- Python 3.7
- NumPy
- Matplotlib
- Pip3
- Jupyter Notebooks

B.1.2 Backend/Networking layer

Prerequisites:

- Python 3.7
- NumPy
- Pip3

Appendix C

Source Code

C.1 Instructions

Different parts of the code are accessible, as described below:

- SNN Design & Experimentation: In Paper
- Networking & SNN: `https://github.com/ngkcl/fyp-networking-snn`

Final Year Project - King's College London

April 29, 2021

```
[ ]: """
     King's College London
     Department of Engineering
     6CCS3EEP

     Implementation of Caenorhabditis Elegans Brain/Neuronal Dynamics using Spiking␣
      ↪Neural Networks

     Author: Aamir Faaiz <mohamed.faaiz@kcl.ac.uk/aamirfaaiz@outlook.com>
     Date: April 2021

     Instructions: Run this code block first
     Note: Kernal will have to be resarted to clear all variables stored in memory
     """
     import numpy as np
     import matplotlib.pyplot as plt
     import time
     from scipy.stats import norm
     from scipy.misc import derivative
     from scipy import signal
     from numba import jit
     from numba import cuda, vectorize
     import random
     from copy import deepcopy
     plt.style.use('default')
```

0.0.1 Dynamics Model & Neuron Circuit for Contour Tracking

Schematic of the neural circuit for contour tracking. N1 is a temperature sensitive input neuron. N1 − N3 is a positive comparator, and N7−N8−N9 is a gradient detector. Similarly, N1−N2 is a complementary comparator, and N4 − N5 − N6 is a gradient detector. N6 and N9 control the deterministic turns, while N10 controls the random exploration.

0.0.2 Leaky Integrate and Fire

The dynamics of the membrane potential V (t), in the LIF neuron model is given by the equation:

$C\frac{dV(t)}{dt} = -g_L(V(t) - E_L) + I_{app}(t)$ [1]

Using Eulers method, we can approximate numerical solutions to the differential equation:

$V(t + \Delta t) = V(t) + \Delta t(\frac{dV}{dt})$

$V(t + \Delta t) = V(t) + \frac{\Delta t}{C}[-g_L(V(t) - E_L) + I_{app}(t)]$

[ ]:
```
#Functions
def simulateLIFRefrac(dt,I,C,gL,EL,VT,W,ker,Nk,I_bias,tref=0):
    '''Approximate LIF Neuron dynamics by Euler's method.

    Parameters
    ----------
    dt :number
       Euler time-step (ms)

    C,gL,EL,VT:number
        neuronal parameters

    I:1D NumPy array
        Input current (pA)

    W:1D NumPy array
        Synaptic Weight Strengths  - Connectivity Matrix

    ker:1D NumPy Array
        Current Kernal

    Ibias: NumPy Array
        Current Bias

    tref: number
        Refractory Period (ms)

    Returns
    -------
    V,Isyn :  NumPy array (mV), NumPy array (pA)
        Approximation for membrane potential computed by Euler's method.
    '''

    V=EL*np.ones((np.shape(I)[0],2*np.shape(I)[1])) #initialising V
    stim=np.zeros((np.shape(I)[0],2*np.shape(I)[1])) #initialising synaptic␣
 ↪stimulus
    Isyn=np.zeros((np.shape(I)[0],2*np.shape(I)[1])) #initialising synaptic␣
 ↪current
```

```
    spike_instants = np.zeros(np.shape(I)[0]) #spike instants for neurons
    for n in range(0,np.shape(I)[1]-1):
        neuron_refractory = np.where(n>spike_instants+(tref/dt))
        if(np.size(neuron_refractory)>0):
            V[neuron_refractory,n+1] = V[neuron_refractory,n] + (dt/
↪C)*(I[neuron_refractory,n]
                                                    +Isyn[neuron_refractory,n]
                                                    +I_bias[neuron_refractory,n]
                                                    -gL*(V[neuron_refractory,n]-EL))
        else:
            V[:,n+1] = EL
        error_check=np.where(V[:,n+1]<EL)[0]
        if np.size(error_check)>0:V[error_check,n+1]=EL #Set to EL when V goes␣
↪below EL
        check=np.where(V[:,n+1]>VT)[0]
        if np.size(check)>0:
            V[check,n+1]=EL
            V[check,n]=VT #Artificial Spike
            spike_instants[check] = n
            stim[check,n+1:n+1+Nk] += ker
            Isyn[:,n+1:] = 1000*np.matmul(W,stim[:,n+1:])

    return V,Isyn


def␣
↪gradientDetect(dt,I,C,gL,EL,VT,ker,Nk,I_bias,tref,c,tau_a,d,w45,w46,w56_initial,w66,pcm=Fal
↪
    '''Gradient Detection

    Parameters
    ----------
    dx :number
       Euler time-step (ms)

    C,gL,EL,VT:number
        neuronal parameters

    I:1D NumPy array
        Input current (pA)

    ker:1D NumPy Array
        Current Kernal

    Ibias: Number
        Neuron 5 Current Bias(pA)
    tref: number
        Refractory Period (ms)

    c,tau_a,d: number
        Weight Adaptation Parameters

    w45,w46,w56_initial,w66: number
        Synaptic Strengths
    pcm: Boolean
        Low Precission Weight Encoding

    Returns
    -------
    V,Isyn,w56_weights :  NumPy array (mV), NumPy array (pA), NumPy array

    '''
    W_gdetect = np.zeros((3,3))
    W_gdetect[1,0] = w45
    W_gdetect[2,0] = w46
    W_gdetect[2,1] = w56_initial
    W_gdetect[2,2] = w66
    w56_weights=[]
    V=EL*np.ones((np.shape(I)[0],2*np.shape(I)[1])) #initialising V
    stim=np.zeros((np.shape(I)[0],2*np.shape(I)[1])) #initialising synaptic␣
↪stimulus
    Isyn=np.zeros((np.shape(I)[0],2*np.shape(I)[1])) #initialising synaptic␣
↪current
    spike_instants = np.zeros(np.shape(I)[0]) #spike instants for neurons
    Ibias = np.zeros((np.shape(I)[0],2*np.shape(I)[1])) #initialising bias␣
↪current
    Ibias[1] = I_bias #Setting Neuron 5 bias current
    spike_instants = np.zeros(np.shape(I)[0]) #spike instants for neurons
    for n in range(0,np.shape(I)[1]-1):
        neuron_refractory = np.where(n>spike_instants+(tref/dt))
        if(np.size(neuron_refractory)>0):
            V[neuron_refractory,n+1] = V[neuron_refractory,n] +(dt/
↪C)*(I[neuron_refractory,n]
                                                 +Isyn[neuron_refractory,n]
                                                 + Ibias[neuron_refractory,n]
                                                 -gL*(V[neuron_refractory,n]-EL))
        else:
            V[:,n+1] = EL
        error_check=np.where(V[:,n+1]<EL)[0]
        if np.size(error_check)>0:V[error_check,n+1]=EL #Set to EL when V goes␣
↪below EL
        check=np.where(V[:,n+1]>VT)[0]
        if np.size(check)>0:
            V[check,n+1]=EL
            V[check,n]=VT #Artificial Spike
            spike_instants[check] = n
            stim[check,n+1:n+1+Nk] += ker
            Isyn[:,n+1:] = 1000*np.matmul(W_gdetect,stim[:,n+1:])
            #Checking if N5 has spiked
            if(V[1][n]==VT):
               if(pcm): W_gdetect[2,1] = pcm_weights(W_gdetect[2,1] + ((c/
 ↪tau_a)+ ((d-W_gdetect[2,1])/tau_a)))
               else: W_gdetect[2,1] = W_gdetect[2,1] + ((c/tau_a)+␣
 ↪((d-W_gdetect[2,1])/tau_a))
            else:
                if(pcm): W_gdetect[2,1]+= pcm_weights(((d-W_gdetect[2,1])/
 ↪tau_a))
                else: W_gdetect[2,1]+= ((d-W_gdetect[2,1])/tau_a)
        w56_weights.append(W_gdetect[2,1])
#         print(W_gdetect[2,1])
    return V,Isyn,w56_weights


def currentKernal(dT,tau1,tau2,kerlen):
    ''' Synaptic Current Kernel Generation
    Parameters
    ----------
    dT: number
        Time-step (ms)
    tau1, tau2: number
        Time Constants
    kerlen: number
        Time Span of Kernal (ms)
    Returns
    -------
    ker,Nk: Current Kernal, Number of Kernals
    '''
    # Obtaining kernel
    Nk = np.int_(kerlen/dT)
    xker = np.linspace(0, kerlen,Nk , endpoint=True)
    ker =  np.array(np.exp(-xker/tau1)-np.exp(-xker/tau2))
    return ker,Nk


def pcm_weights(w, precision=4, write_noise_stdv=0.1):
    ''' PCM Weights Generation
    Parameters
    ----------
    w:  NumPy Matrix
        Neuron Connectivity Matrix
    precision: number
        Number of precission bits for weights
    write_noise_stdv: number
        Standard deviation of weights
    Returns
    -------
    w_renorm: PCM Weights
    '''
    quant_max = 2**(precision-1) -1
    w_max = np.amax(np.abs(w))
    w_norm = w/w_max
    w_scaled = w_norm * quant_max
    w_round = np.round(w_scaled)
    w_clip = np.clip(w_round,-quant_max,quant_max)
    w_renorm = w_clip * w_max / quant_max

    if write_noise_stdv > 0.0:
        w_renorm = w_renorm + np.random.normal(0,1,np.shape(w)) *␣
 ↪write_noise_stdv

    return w_renorm


def currentInputGen(T,Ts,alpha,beta):
    '''Temperature Input to Current Generation for Neuron 1

    Parameters
    ----------
    T: Temperature Input
    Ts: Set Temperature Point
    alpha,beta: constants

    returns: Current Input for Neuron 1(pA)
    '''
    return (alpha + (beta*(T-Ts)))

def currentGen(scale,num_pulses):
    ''''Current Generation with a varying pulse width

    Parameters
    ----------
    scale: number
        Factor to increase pulse width by
    num_pulses: number
        Number of pulses to generate
    Returns
    -------
    Iinp: 1D NumPy Array Current Matrix
    '''
    Iinp=np.zeros(1)
    ton = scale*(np.arange(num_pulses)+1)
    for n in range(0,len(ton)):
        Iinp = np.hstack([Iinp, 2700*np.ones(ton[n])])
        Iinp = np.hstack([Iinp, np.zeros(400)])
    return Iinp


def␣
 ↪plotter(x,y1,title,yaxlabel,y1label=None,y2=None,y2label=None,usedark=False):
    '''Graph Plotting

    Parameters
    ----------
    x:1D NumPy array
       NumPy array to be used as the x-axis data

   y1,y2:1D NumPy array
        NumPy array to be used as the y-axis data


    yaxlabel:String
        Y axis label

    y1label,y2label:String
        Legend Labels

     title:String
         Graph title

    '''
    plt.style.use('dark_background') if usedark else plt.style.use('default')
    plt.figure(figsize=(10,5)) #Customisable size
    plt.rcParams.update({'font.size': 15})
    plt.grid(True)
    plt.plot(x,y1,'g-', label = y1label)
    if(y2 is not None):plt.plot(x,y2,'r-', label = y2label),plt.legend()
    plt.title(title)
    plt.xlabel("Time/ms")
    plt.ylabel(yaxlabel)
    plt.show()
```

0.0.3 Saturating Current of LIF Neuron

Stimulating LIF Neuron with input current ranging from 2.4nA to 100nA and measuring spike frequency response

[ ]:
```
#Saturation Point of LIF Neuron

#Neuron parameters for simulation
C = 300 #Membrane Capacitance(pF)
gL = 30 #Membrane Leak Conductance(nS)
EL = -70 #Resting Potential(mV)
VT = 20  #Resting Potential(mV)

def LIF(dT,I,C,gL,EL,VT,tref):
    V = EL*np.ones((np.shape(I)))
    #iterating in the temporal dimension
    spike_instant = 0
    for n in (range((np.shape(I))[1]-1)):
        if(n>spike_instant+(tref/dT)):
            V[:,n+1] = V[:,n] + (dT/C) * (-gL*(V[:,n]-EL)+I[:,n])
        else:
            V[:,n+1] = EL
        check=np.where(V[:,n+1]>VT)[0]
        if np.size(check)>0:
            V[check,n+1]=EL
            spike_instant = n
            V[check,n]= VT #Artificial Spike
    return V

currents = np.arange(2500,100000,1000) #Simulating input current ranging from␣
 ↪2500pA to 100000pA
T = 1000 #Simulation Time (ms)
dt = 0.1 #Euler Time Step(ms)
tref = 3 #Refractory Period(ms)
iter = int(T/dt) #Total number of time-steps
frequencies = []
for i in range(len(currents)):
    Iinp = currents[i]*np.ones(iter)
    Iinp = np.reshape(Iinp,(1,iter))
    Vmem = LIF(dt,Iinp,C,gL,EL,VT,tref)
    # Estimating Spike Frequency Response
    spike_count = np.shape(np.where(Vmem[0] == VT))[1]
    frequencies.append(spike_count)

#Plotting
plt.figure(figsize=(10,5)) #Customisable size
plt.rcParams.update({'font.size': 18})
plt.plot(0.001*currents,frequencies)
plt.title("Spike Frequency Response of LIF Neuron")
plt.xlabel("Current Input(nA)")
plt.ylabel("Spike Frequency(Hz)")
plt.grid()
plt.show()
```

0.0.4 PCM Weights Demonstration

[ ]:
```
a=-1+2*np.random.random_sample((1000, 1))
b=pcm_weights(a)
plt.style.use('default')
f=plt.figure(figsize=(10,5))
plt.rcParams.update({'font.size': 15})
plt.grid(True)
plt.plot(a ,b, 'r.',)

plt.xlabel('Floating point weight')
plt.ylabel('PCM weight')
plt.axis([-1,1,-1,1])
plt.show(block = False)
```

[ ]:
```
#Neuron Parameters
C = 300 #Membrane Capacitance(pF)
gL = 30 #Membrane Leak Conductance(nS)
EL = -70 #Resting Potential(mV)
VT = 20  #Resting Potential(mV)
dt = 0.1 #Euler Time Step(ms)
# Obtaining kernel
kerlen = 80
tau1 = 15 #Time Constant for Kernal Generation(ms)
tau2 = tau1/4 #Time Constant for Kernal Generation(ms)
ker,Nk = currentKernal(dt,tau1,tau2,kerlen)
plt.rcParams.update({'font.size': 10})
plt.plot(ker)
plt.title("Synaptic Current Kernal")
plt.xlabel("Time(ms)")
plt.grid()
```

0.0.5 Experimenting with Temperature Comparators

We need two comparators: 1) Negative Comparator: $T < T_s$ Detection
2) Positive Comparator: $T > T_s$ Detection

A complementary comparator (i.e., a network that produces a proportional spike response for deviations of local temperature below the set-point) can be realized by making w negative and I_{bias} positive.

```
[ ]: def␣
     ↪comparatorTests(threshold_temp,temperatures,alpha,beta,I_bias,comparator_weights,pcm=False)
     ↪
         ''' Spike frequency response from comparator with varying parameters
          Parameters
         ----------
         threshold_temp: Number
             Threshold temperature

         temperatures: 1D NumPy Array
             Temperature range for simulation

         alpha,beta: Number
             Constants for Input Current Generation

         bias_currents: NumPy Matrix
             Bias Currents for simulation

         comparator_weights: NumPy Matrix
             Comparator Weights for simulation

         pcm: Boolean
             Clone to PCM
         '''
         T = 1000 #Total time of simulation(ms)
         dt = 0.1 #Euler Time-Step(ms)
         iter = int(T/dt)
         Numneurons = 2
         tref = 3 #Neuron Refractory Period(ms)
         frequencies = []
         for i in range(len(comparator_weights)):
             spike_frequency = []
             for t in range(len(temperatures)):
                 Temp = temperatures[t]*np.ones(iter)
                 Iinp = currentInputGen(Temp,threshold_temp,alpha,beta) #Set␣
     ↪Threshold Temperature to 20C
                 I = np.vstack((Iinp,np.zeros((Numneurons-1,np.shape(Iinp)[0]))))
                 t = dt*(np.arange(len(I[1])))
                 if(pcm): Vmem1,Isyn1 =␣
     ↪simulateLIFRefrac(dt,I,C,gL,EL,VT,pcm_weights(comparator_weights[i]),ker,Nk,I_bias[i],tref)
                 else: Vmem1,Isyn1 =␣
     ↪simulateLIFRefrac(dt,I,C,gL,EL,VT,comparator_weights[i],ker,Nk,I_bias[i],tref)#Simulate
                 # Estimating Spike Frequency
                 spike_count = np.shape(np.where(Vmem1[1,0:np.shape(I)[1]] == VT))[1]
                 spike_frequency.append(spike_count)
             frequencies.append(spike_frequency)
         return frequencies
```

0.0.6 Positive Comparator

```
[ ]: #Setting up bias currents for simulations
T=1000
dT=.1
iter=int(T/dT)
I_bias = np.zeros((4,2,iter))
I_bias[0][1] = -2500
I_bias[1][1] = -3000
I_bias[2][1] = -3500
I_bias[3][1] = -4000

#Setting up weights for simulations
comparator_weights = np.ones((4,2,2))
comparator_weights[0] = [[0,0],
                         [7.0,0]]
comparator_weights[1] = [[0,0],
                         [7.50,0]]
comparator_weights[2] = [[0,0],
                         [8.0,0]]
comparator_weights[3] = [[0,0],
                         [8.5,0]]
#Temperature range for simulations
temperatures = np.arange(18,23,0.2)
threshold_temp = 20 #Celcius
alpha = 3750 #pA
beta  = 350  #pA/K
positive_frequency_response =␣
 ↪comparatorTests(threshold_temp,temperatures,alpha,beta,I_bias,comparator_weights)
#Plotting
plt.style.use('default')
plt.figure(figsize=(10,5)) #Customisable size
plt.rcParams.update({'font.size': 15})
for i in range(len(positive_frequency_response)):
    plt.plot(temperatures,positive_frequency_response[i], label = "$I_{bias} =␣
 ↪$" + str(I_bias[i][1][0]) + "pA"
             + " ,w = " + str(1000*comparator_weights[i][1][0]), marker =␣
 ↪"v")
plt.legend()
# plt.title("Spike Frequency Response of Positive Comparator")
plt.xlabel("Sensed Temperature/Celcius")
plt.ylabel("Spike Frequency of $N_3$(Hz)")
plt.grid(True)
plt.show()
```

0.0.7 Positive Comparator with Low Precission Weights

```
[ ]: #Using Optimal Parameter Solution and simulation with PCM Weights
     #Setting up bias currents for simulations
     positive_comparator_bias = np.zeros((1,2,iter))
     positive_comparator_bias[0][1] = -2500

     #Setting up weights for simulations
     positive_comparator_weight = np.ones((1,2,2))
     positive_comparator_weight[0] = [[0,0],
                                       [7.0,0]]

     positive_pcm_frequency_response =␣
      ↪comparatorTests(20,temperatures,alpha,beta,positive_comparator_bias,positive_comparator_wei

     #Plotting
     plt.style.use('default')
     plt.figure(figsize=(10,5)) #Customisable size
     plt.rcParams.update({'font.size': 15})
     plt.plot(temperatures,positive_pcm_frequency_response[0], label = "$I_{bias} =␣
      ↪$" + str(positive_comparator_bias[0][1][0]) + "pA"
               + " ,w = " + str(1000*positive_comparator_weight[0][1][0]), marker␣
      ↪= "v")


     plt.legend()
     # plt.title("Spike Frequency Response of Positive Comparator with 4bit PCM␣
      ↪Weights")
     plt.xlabel("Sensed Temperature/Celcius")
     plt.ylabel("Spike Frequency of $N_3$ (Hz)")
     plt.grid(True)
     plt.show()
```

0.0.8 Negative Comparator

```
[ ]: #Setting up bias currents for simulations
     I_bias = np.zeros((4,2,iter))
     I_bias[0][1] = 9000
     I_bias[1][1] = 9500
     I_bias[2][1] = 10000
     I_bias[3][1] = 10500

     I_bias[0][1] = 7000
     I_bias[1][1] = 7500
     I_bias[2][1] = 8000
     I_bias[3][1] = 8500
```

```
#Setting up weights for simulations
negative_comparator_weights = np.ones((4,2,2))
negative_comparator_weights[0] = [[0,0],[-7.0,0]]
negative_comparator_weights[1] = [[0,0],[-7.50,0]]
negative_comparator_weights[2] = [[0,0],[-8.0,0]]
negative_comparator_weights[3] = [[0,0],[-8.5,0]]
#Temperature range for simulations
temperatures = np.arange(18,23,0.2)
alpha = 3750 #pA
beta = 350 #pA/K
threshold_temp = 20 #Celcius
negative_frequency_response =␣
 ↪comparatorTests(threshold_temp,temperatures,alpha,beta,I_bias,negative_comparator_weights)

#Plotting
plt.style.use('default')
plt.figure(figsize=(10,5)) #Customisable size
plt.rcParams.update({'font.size': 15})
for i in range(len(positive_frequency_response)):
    plt.plot(temperatures,negative_frequency_response[i], label = "$I_{bias} =␣
 ↪$" + str(I_bias[i][1][0]) + "pA"
                 + " ,w = " + str(1000*negative_comparator_weights[i][1][0]),␣
 ↪marker = "v")
plt.legend()
# plt.title("Spike Frequency Response of Negative Comparator")
plt.xlabel("Sensed Temperature/Celcius")
plt.ylabel("Spike Frequency of $N_2$ (Hz)")
plt.grid(True)
plt.show()
```

0.0.9 Negative Comparator with Low Precission Weights

[ ]:
```
#Using Optimal Parameter Solution and simulation with PCM Weights
#Setting up bias currents for simulations
negative_comparator_bias = np.zeros((1,2,iter))
negative_comparator_bias[0][1] = 8000

#Setting up weights for simulations
negative_comparator_weight = np.ones((1,2,2))
negative_comparator_weight[0] = [[0,0],
                                 [-8.0,0]]

negative_pcm_frequency_response =␣
 ↪comparatorTests(20,temperatures,alpha,beta,negative_comparator_bias,negative_comparator_wei
#Plotting
plt.style.use('default')
plt.figure(figsize=(10,5)) #Customisable size
plt.rcParams.update({'font.size': 15})
plt.plot(temperatures,negative_pcm_frequency_response[0], label = "$I_{bias} =␣
↪$" + str(negative_comparator_bias[0][1][0]) + "pA"
            + " ,w = " + str(1000*negative_comparator_weight[0][1][0]), marker␣
↪= "v")
plt.legend()
# plt.title("Spike Frequency Response of Comparator with 4bit PCM Weights")
plt.xlabel("Sensed Temperature/Celcius")
plt.ylabel("Spike Frequency of $N_2$ (Hz)")
plt.grid(True)
plt.show()
```

0.0.10 Gradient Detector

Operation required to detect temperature gradients:

$f(t+dt)-f(t)$

The Neuron Ciruit below is the design choice for the gradient detector operation. The excitatory synapse between N4-N6 essentially computes $f(t+dt)$ and the connectivity $N_4-N_5-N_6$ essentially computes the $-f(t)$ operation.

The network should only send out a spike when there is a gradient detected. Therefore, there has to be some sort of competition between the current that is comming in both pathways, and they should be trying to cancel each other out, such that only when there is a change in N4-N6, a spike response is created.

w_{56} should adapt in such a way that the net current that is comming into N_6 should be zero.

0.0.11 Memoryless Synaptic Strength Adaptation

Modifying the strength of each synapse using the following rule:

$\Delta w_{56}=\frac{c}{\tau_a}\delta(t-t^{N_5})+\frac{d-w_{56}}{\tau_a}\Delta t$

t^{N5} denotes the instants where neuron N_5 has spiked

0.0.12 30 Second Simulation (Gradient Detector Test)

```
[ ]: #Generate Current Input for gradient detection simulation
T = 30000 #ms
dT = 0.1
iter=int(T/dT) #Number of timesteps
gradients = 6 #Number of gradients for simulation
gradient_period = int(iter/gradients) #dt for gradient switching
x_stop = gradient_period*dT

# 6 gradients for simulation:
```

```
#     1. Constant for t1
#     2. Positive for t2
#     3. Constant for t3
#     4. Negative for t4
#     5. Constant for t5
#     6. Positive for t6

#Initialising Currents
lower_bound = 2400 #pA
upper_bound = 3000 #pA
positive_gradient = ((upper_bound-lower_bound)/(dT*gradient_period))
negative_gradient = -positive_gradient

Iinp = np.zeros(iter)
Iinp[:gradient_period] = lower_bound
Iinp[gradient_period:2*gradient_period] = (positive_gradient*np.
 ↪linspace(0,x_stop,gradient_period))+lower_bound
Iinp[2*gradient_period:3*gradient_period] = upper_bound
Iinp[3*gradient_period:4*gradient_period] = (negative_gradient*np.
 ↪linspace(0,x_stop,gradient_period))+upper_bound
Iinp[4*gradient_period:5*gradient_period] = lower_bound
Iinp[5*gradient_period:6*gradient_period] = (positive_gradient*np.
 ↪linspace(0,x_stop,gradient_period))+lower_bound
I1= np.vstack((Iinp,np.zeros((3-1,np.shape(Iinp)[0]))))
plt.plot(Iinp)
plt.ylabel("$I_{in}(pA)$")
plt.xlabel("Time (scale="+str(dT)+"ms/sample)")
plt.axis([0, iter,2000,4000])
plt.grid()
```

0.0.13 Genetic Algorithm for weight adaptation tuning

[ ]:
```
# Genetic algorithm
'''
Ranges:

c:  5 to  15 ,
d: -0.5 to  -10,
tau_a: 5 to 30

#Weights!
w45: -0.5 to -10
w46: 5 to 15
w56_initial: -0.5 to -10
w66 = -2 to -6 #N6-N6 weight for stability

I_bias: 20000 to 8000
```

```
'''
POPULATION_SIZE = 100
N_ITERS = 10

def get_rand(min, max):
 return round(random.uniform(min, max),1)


def get_spikes_n6(vo):
  # vn6 = vo[:-1]
  return np.where(vo[2]>=VT)[0]


def score(c, d, tau_a, w45, w46, w56_initial, w66,I_bias):
  Vmem1,Isyn1,w56_weights =␣
 ↪gradientDetect(dT,I1,C,gL,EL,VT,ker,Nk,I_bias,tref,c,tau_a,d,w45,w46,w56_initial,w66)
  sp = get_spikes_n6(Vmem1)

  # wrong_spikes = [x for x in sp if (x < 2000 or x > 4000)]
  # right_spikes = [x for x in sp if (x >= 2000 and x <= 4000)]

  wrong_spikes = [x for x in sp if (x < 50000 or (x > 100000 and x < 250000))]
  right_spikes = [x for x in sp if (x >= 50000 and x <= 100000) or (x >= 250000␣
 ↪and x<=300000)]

  score = 100 + 12 * len(wrong_spikes)  - 12 * len(right_spikes)
  score += -(50 if len(wrong_spikes) < 1 else 0) + (50 if len(right_spikes) < 1␣
 ↪else 0)

  if score < 0:
    score = 0

  return score
#   return 100 + 9 * len(wrong_spikes)  - 12 * len(right_spikes) - (50 if␣
 ↪len(wrong_spikes) < 1 else 0) + (50 if len(right_spikes) < 1 else 0)


def selection(pop, scores, k=3):
  # first random selection
        selection_ix = np.random.randint(len(pop))
        for ix in np.random.randint(0, len(pop), k-1):
                # check if better (e.g. perform a tournament)
                if scores[ix] < scores[selection_ix]:
                        selection_ix = ix
        return pop[selection_ix]
```

```python
def crossover(p1, p2, r_cross):
  child1, child2 = deepcopy(p1), deepcopy(p2)
  if get_rand(0,1) < r_cross:
    print('[] Crossover happening with: ' + str(child1) + ' and ' + str(child2))
    c_pt = get_rand(0, abs(p1[0] - p2[0]))
    d_pt = get_rand(0, abs(p1[1] - p2[1]))
    tau_pt = get_rand(0, abs(p1[2] - p2[2]))
    w45_pt = get_rand(0, abs(p1[3] - p2[3]))
    w46_pt = get_rand(0, abs(p1[4] - p2[4]))
    w56_initial_pt = get_rand(0, abs(p1[5] - p2[5]))
    w66_pt = get_rand(0, abs(p1[6] - p2[6]))
    I_bias_pt = get_rand(0, abs(p1[7] - p2[7]))

    c1 = min(p1[0], p2[0]) + c_pt
    c2 = max(p1[0], p2[0]) - c_pt

    d1 = min(p1[1], p2[1]) + d_pt
    d2 = max(p1[1], p2[1]) - d_pt

    tau1 = min(p1[2], p2[2]) + tau_pt
    tau2 = max(p1[2], p2[2]) - tau_pt

    w45_1 = min(p1[3], p2[3]) + w45_pt
    w45_2 = max(p1[3], p2[3]) - w45_pt

    w46_1 = min(p1[4], p2[4]) + w46_pt
    w46_2 = max(p1[4], p2[4]) - w46_pt

    w56_initial_1 = min(p1[5], p2[5]) + w56_initial_pt
    w56_initial_2 = max(p1[5], p2[5]) - w56_initial_pt

    w66_1 = min(p1[6], p2[6]) + w66_pt
    w66_2 = max(p1[6], p2[6]) - w66_pt

    I_bias_1 = p1[7] + I_bias_pt
    I_bias_2 = p2[7] - I_bias_pt

    child1 = (c1, d1, tau1, w45_1, w46_1, w56_initial_1, w66_1,I_bias_1)
    child2 = (c2, d2, tau2, w45_2, w46_2, w56_initial_2, w66_2,I_bias_2)
  return [child1, child2]

# mutation operato
def mutation(body, r_mut):
  for i in range(len(body)):
    # check for a mutation
    if get_rand(0, 1) < r_mut:
      print('[] Mutation occuring for: ' + str(body))
      # flip the bit
      new_val = body[i]
      if i == 0:
        new_val = get_rand(5, 15)
      elif i == 1:
        new_val = get_rand(-0.5, -10)
      elif i == 2:
        new_val = get_rand(5, 30)
      elif i == 3:
        new_val = get_rand(-5, -0.5)
      elif i == 4:
        new_val = get_rand(5, 15)
      elif i == 5:
        new_val = get_rand(-10, -0.5)
      elif i == 6:
        new_val = get_rand(-6, -2)
      elif i == 7:
        new_val = get_rand(2000,8000)
      body = list(body)
      body[i] = new_val
      body = tuple(body)
      print('[] New value is: ' + str(body))

# @vectorize(target="parallel")
def genetic_algo():
  r_cross = 0.35
  r_mut = 0.05

  # Returns population of tuples of the form:
  #(c, d, tau_a,w45,w46,w56_initial,w66,Ibias)
  # population = [(get_rand(5, 20), get_rand(-10, -0.5), get_rand(5, 30),␣
 ↪get_rand(-10, -0.5), get_rand(5, 15), get_rand(-10, -0.5), get_rand(-6,␣
 ↪-2),get_rand(2000,6000)) for _ in range(POPULATION_SIZE)]
  population = [(get_rand(5, 15), get_rand(-10, -0.5), get_rand(5, 30),␣
 ↪get_rand(-5, -0.5), get_rand(5, 15), get_rand(-10, -0.5), get_rand(-6,␣
 ↪-2),get_rand(2000,8000)) for _ in range(POPULATION_SIZE)]
  n_pop = len(population)

  best, best_eval = 0, score(*population[0])

  for gen in range(N_ITERS):
    print('Populations:')
    print(population)

    # Get scores
    scores = [score(*c) for c in population]
    print('Scores:')
    print(scores)

    # Check for new best scores
    for i in range(n_pop):
      if scores[i] < best_eval:
        best, best_eval = population[i], scores[i]
        print(">%d, new best f(%s) = %.3f" % (gen,  population[i], scores[i]))

    # Perform tournament
    selected = [selection(population, scores) for _ in range(POPULATION_SIZE)]

    children = list()
    for i in range(0, n_pop, 2):
      # get selected parents in pairs
      try:
        p1, p2 = selected[i], selected[i+1]
      except IndexError:
        if p1 is None:
          break
        children.append(p1)

      # crossover and mutation
      for c in crossover(p1, p2, r_cross):
        # mutation
        mutation(c, r_mut)
        # store for next generation
        children.append(c)
    population = children
  print('Best is:')
  print(best)

genetic_algo()
```

[ ]:
```
#Simulation
Numneurons = 3 #N4-N5-N6
I_bias = 2800
tref = 3 #Refractory Period(ms)
#Gradient Detector Parameters
c,d,tau_a,w45,w46,w56_initial,w66,I_bias = (8.365406866817512, -5.
↪7172469083885336, 22.993129497338796, -1.868965162763052, 6.
↪8399624018718255, -2.7420209289913564, -4.202035141656892, 3811.
↪0644999147507)
I1= np.vstack((Iinp,np.zeros((Numneurons-1,np.shape(Iinp)[0]))))
Vmem1,Isyn1,w56_weights =␣
↪gradientDetect(dT,I1,C,gL,EL,VT,ker,Nk,I_bias,tref,c,tau_a,d,w45,w46,w56_initial,w66)#Simul
↪with weight adaptation
```

[ ]:
```
#6s Simulation
#Plotting
plt.figure(1)
plt.subplot(411)
# plt.title("Gradient Detetor Simulation")
plt.plot(Iinp,'r')
plt.axis([0, iter,2000,4000])
plt.ylabel("$I_{in}(pA)$")

plt.subplot(412)
plt.plot(w56_weights,'g')
plt.ylabel("$w_{56}$")

plt.subplot(413)
plt.plot(Vmem1[2][:iter])
plt.ylabel("$V_{N6}(mV)$")

spike_times = dT*np.where(Vmem1[2,0:np.shape(I1)[1]]==VT)[0] #Spike times of N6
stim = np.zeros(iter)
#Creating delta function for N6 spike times
for i in range(len(spike_times)):stim[(int(spike_times[i]/dT))] = 1

#Plotting delta function for N6 spikes for clarity
plt.subplot(414)
plt.plot(stim,'c')
plt.ylabel("$\delta(t-t^{N6})$")
plt.xlabel("Time (scale="+str(dT)+"ms/sample)")

plt.tight_layout()
plt.show()
```