Master Thesis

Learning enhancement with high firing irregularity produced by a two compartment neuron

Pericleous Pericles

University of Cyprus

Computer Science department

December, 2011

Abstract

This thesis introduced a different approach in modeling and producing firing irregularity at high rates in order to investigate in an alternative way the claim of Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] that "high firing irregularity enhances learning". More specifically, this thesis introduces a neural network consisting of two compartment leaky integrate-and-fire model as a neuron to investigate firstly whether this model at the neuron level can produce high firing irregularity at high rates and secondly at the network level whether it can enhance learning.

To achieve the above, the two compartment model suggested by Lansky and Rodriguez [3] and Bressloff [4] is implemented and tested for producing high firing irregularity at high rates. In addition the leaky integrate-and-fire model with partial somatic reset is implemented as part of this thesis for the purposes of comparison with the two compartment leaky integrate-and-fire model in producing high firing irregularity. The results showed that the two-compartment leaky integrate and fire model can produce firing irregularity in high rates.

The current model (i.e., two compartment leaky integrate-and-fire) is applied to a neural network trained with reward-modulated spike-timing-depended plasticity with eligibility trace introduced by Florian [5]. For the purposes of comparison, two other networks were implemented by this thesis. One consisted of leaky integrate-and-fire model with total reset (same with the one used by Florian, [5]) and one which consisted of leaky integrate-and-fire model with somatic partial reset (same with the

one used by Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2]). All three networks are forced to fire at high rates in order to test whether the high firing irregularity at high rates that can be produced by the two of the three networks (i.e., the one with the leaky integrate-and-fire nodes with somatic partial reset and the one with the two compartment leaky integrate-and-fire models) can achieve enhancement in learning as Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] claim. The results showed that the two networks that can fire irregularly at high rates performed better in terms of learning than the one that fires regularly. This is possible as high firing irregularity leads to more accurate correlation between pre-synaptic and post-synaptic spike timing and reinforcement signals.

Furthermore it was observed that the network that consisted of two compartment LIF nodes had better result than the network that consisted of LIF model with somatic partial reset as nodes. This cannot be easily explained because the different type of modeling sets limits in terms of comparison. Therefore, further investigation is needed in order to explore the reasons for the better performance by networks which consisted of two compartment LIF neurons.

Besides verifying the claim by Christodoulou's and Cleanthous [1] and Cleanthous and Christodoulou [2] that high firing irregularity enhances learning, this thesis also introduces o different way of neuron modeling that can achieve high firing irregularity at high rates.

Learning enhancement with high firing irregularity produced by a two compartment neuron

Pericleous Pericles

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Cyprus

Recommended for Acceptance

By the Department of Computer Science

December, 2011

APPROVAL PAGE

Master Degree

LEARNING ENHANCEMENT WITH HIGH FIRING IRREGULARITY PRODUCED BY A TWO COMPARTMENT NEURON

Presented by

Pericleous Pericles

Research Supervisor

_____

Ass.Prof. Chris Christodoulou

Committee Member

_____

Prof. Christos N. Schizas

Committee Member

_____

Prof. Constantinos Pattihis

University of Cyprus

December, 2011

Contents

Chapter 1 (Introduction) 1

1.1 Incentive 1

1.2 Related work 2

1.3 Motivation 3

1.4 Thesis outline 5

Chapter 2 (Background) 6

2.1 Computational neuroscience 6

2.2 Learning 8

2.3 Learning Algorithms 8

2.4 Reinforcement Learning 9

2.5 Spiking Neural Networks using Reinforcement Learning 11

2.6 Reinforcement Learning through STDP 12

2.7 Florian's modulated STDP by a global reward signal 13

2.8 High Firing Irregularity of Cortical Cells at high rates 14

2.9 Leaky Integrator Neuron Model with Partial Reset 16

2.10 High Firing irregularity enhances learning 17

Chapter 3 (Design) 20

3.1 Overview 20

3.2 Two point model 21

3.3 The network 25

3.4 Learning approach 26

3.5 Implementations 31

Chapter 4 (Results & Discussion) 32

4.1 Two compartment model: Can it produce high firing irregularity at high rates? 33

4.1.1 Parameters of the models 34

4.1.2 Model comparison and Discussion 35

4.1.2.1 Potentials of the models 36

4.1.2.2 Output Spike trains 38

4.1.2.3 ISI distribution histograms and Autocorrelograms 39

4.1.2.3.1 ISI distribution Histogram 40

4.1.2.3.2 Autocorrelation 41

4.1.2.3.3 Poisson-type firing 43

4.2 Does high firing irregularity enhance learning produced by a two compartment model? 44

4.2.1 Training of the network 44

4.2.1.1 Training Parameters 46

4.2.2 Results 47

4.2.3 Understanding the reasons of better performance with models that can produce high firing irregularity 50

4.2.4 Why the two compartment LIF performs better? 52

5.1 Conclusion 53

5.2 Future work 54

APPENDIX I 61

APPENDIX II 89

LIST OF FIGURES

Figure 1 Standard reinforcement-learning model from [31]..... 10

Figure 2 Two compartment model of two interconnected LIF compartments. CD:dendrite capacitor, RD: dendrite resistance, Rc: junctional resistance, CM: membrane capacitor and RM): is membrane resistance. 22

Figure 3 Membrane and Dendritic Potential produced by a two point model (equations 2,3) 24

Figure 4 The network architecture for sovling the XOR problem 25

Figure 5 [taken from [8]] Illustration of the dynamics of the variables used by MSTDP and MSTDPED and the effects of those rules and of STDP on the synaptic strength for sample spike trains and reward..... 28

Figure 6 (taken from [8]) Part of figure 5. 29

Figure 7 (taken from [8]) Part of figure 5. 29

Figure 8 (taken from [8]) Part of figure 5. 30

Figure 9 Simulation of the model potential (membrane) for LIF model with partial somatic reset. See section 4.1.1 for the parameters that are being used for this simulation..... 36

Figure 10 Simulation of the model potential (dendritic-membrane) for two compartment LIF model. See section 4.1.1 for the parameters that are being used for this simulation 37

Figure 11 Output spike train of each model (First: LIF with partial somatic reset. Second: Two compartment LIF). See section 4.1.1 for the parameters used for this simulation. The current spike trains were taken when both simulations fired around 80-90Hz rate. 38

Figure 12 ISI distribution histogram for both models (1st : LIF with somatic partial reset at 100Hz firing rate, mean ISI at 9.4ms and CV=0.7. 2nd : Two compartment LIF at 100Hz firing rate, mean ISI at 10.2ms and CV=0.72). For the parameters see 4.1.1. 40

Figure 13 Autocorrelogram example. 41

Figure 14 Autocorrelogram for both model (1st : LIF with somatic partial reset at 100Hz firing rate, mean ISI at 9.4ms and CV=0.7. 2nd : Two compartment LIF at 100Hz firing rate, mean ISI at 10.2ms and CV=0.72). For the parameters see 4.1.1. 42

Figure 15 Average firing rate of the output neuron after learning, for the four different XOR input patterns (A: LIF with total reset model. B: LIF with somatic partial reset model. C: Two compartment LIF) 48

Figure 16 [taken from [29]] Effect of regularity in the value of the synaptic strength This figure is a modified version of the one presented in the original paper for the learning algorithm Florian [8] 51

LIST OF TABLES

Table 1 The model parameters that were used for the produce of the results in section 4.1.2..... 34
Table 2 Learning parameters for the experiments presented in section 4.2.2..... 46

LIST OF ABBREVIATIONS

LIF: Leaky-integrate-and fire

STDP: Spike-timing-dependent plasticity

EPSP: Excitatory Postsynaptic Potential

MSTDP: Reward modulated Spike-timing-dependent plasticity

MSTDPET: Reward modulated Spike-timing-dependent plasticity with eligibility trace

CV: Coefficient of Variation

LIST OF SYMNOLS

Symbols used by standard reinforcement learning (figure 1)

I,i : input to the agent

s: current state

a: agent action as output

T: Task

r: reinforcement signal

B: agent Behavior

Symbols used by LIF equation

R_m : membrane resistance

C_m : membrane capacitor

$I(t)$: input at time (t)

V_m : membrane voltage

Symbols used by Two compartment model (figure 2)

C_D: dendrite capacitor

R_D: dendrite resistance

Rc: junctional resistance

C_M: membrane capacitor

R_M: membrane resistance

Symbols used by two compartment model equations

τ_r : junctional time constant

$\mu(t)$: external input

τ_1 : dendrite time constant

τ_2: membrane time constant

$X_1(t)$: dendritic potential

$X_2(t)$: membrane potential

Symbols used by the learning algorithm of reward-modulated spike-timing-dependent plasticity with eligibility trace

W_{ij}: efficacy of the synapse from neuron j to i

γ: leering rate

dt: duration of a time step

r: global reward signal

z: eligibility trace

β: discount factor between 0 and 1

τ_z : is the time constant for the exponential decay of z

ζ: change of z resulting from the activity in the last time step

P^+_{ij} : tracks the influences of presynaptic spikes

P^-_{ij} : tracks the influence of postsynaptic spikes

τ_+, τ_- : ranges of interspike intervals over which synaptic changes occur and according to the standard ant symmetric STDP model

A_+**:** positive constant parameter

A_-**:** negative constant parameters.

$f_j(t)$ **:** signifies firing of neuron i

Symbols used by table of parameters (table 1):

Vth : Threshold

Vreset: Resting potential

Trefr: Refractory period

Tm : Membrane time constant = **Cm(Membrane capacitor)*Rm (Membrane resistance)**

Td: Dendrite time constant = **Cm(Dendrite capacitor)*Rm (Dendrite resistance)**

a: reset parameter

rC: Junctional time constant

Chapter 1 (Introduction)

1.1 Incentive

1.2 Related work

1.3 Motivation

1.4 Thesis Outline

1.1 Incentive

Learning is the process of transforming information and experience into knowledge and skills. The ability to learn differs between and within species. Since learning is a cognitive process (brain function), the existence of different brains is the reason for differences in learning between species. This, though, cannot explain the phenomenon of differences in learning within the same species. For example, human beings do not all learn something with the same rate or in equal amounts of time. These differences can also be observed even in the case of the same person since the learning rate depends on a variety of parameters such as time, mood, age, prior knowledge etc. Is there something common in those parameters? What are the changes in the brain function that are dependent on these parameters?

Neuroscience and Neuroinformatics are two of the many disciplines which investigate the phenomenon of learning. Others are Philosophy, Education,

Psychology, Mathematics, Informatics etc. In their investigations about neuron behaviors, neuroscientists are experimenting on real neurons in order to record and to analyze their behavior. The rationale of this thesis is based upon one of those experiments and the investigations conducted by an Informatics scientist about a neuron behavior discovered by such experiments.

1.2 Related work

During the analysis of spike trains recorded from the cortical neuron, Softky and Koch [6,7] found that these cells in vivo fire irregularly at high rates. After testing different models in order to investigate which of them can reproduce this phenomenon, Softky and Koch [6,7] found that the simple leaky integrate-and-fire (LIF) model [8,9] failed to reproduce the experimentally observed high firing irregularity because the model predicted very low firing variability ($Cv << 1$ where the CV is the Coefficient of Variation which is a measure of spike train irregularity defined as the standard deviation divided by the mean interspike interval) for realistic depolarizations of Excitatory Postsynaptic Potential (EPSP) and membrane time constants.

Based on Softky's and Koch's [6,7] findings, Bugmann, Christodoulou & Taylor [10] and Christodoulou & Bugmann [11] attempted to find a way to make the LIF model able to reproduce the high firing irregularity at high rates. Through testing, they found that LIF model with partial somatic reset can produce the high firing irregularity observed by Softky and Koch [6,7].

Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] attempted to see whether the high firing irregularity at high rates of cortical cells in vivo has any

functional significance. Based on the assumption that this phenomenon is not coincidental, they assumed that the high firing irregularity, observed by Softky and Koch [6,7], can enhance learning. In order to prove this, they used a network of LIF neurons with partial somatic reset (with the LIF neurons being the same as the model presented by Bugmann, Christodoulou & Taylor [10] and Christodoulou & Bugmann [11]) and trained it with reward-modulated spike-timing-dependent plasticity (STDP) with eligibility trace proposed by Florian [5]. During training, the network was forced to fire irregularly and was tested on XOR benchmark problem and the Prisoner Dilemma game [12]. The results showed that the problems were solved by the network that was forced to fire irregularly during the training with a better performance (enhanced learning) than by the normal network. According to Christodoulou and Cleanthous [1] "this happened due to more accurate correlations between presynaptic and postsynaptic spike timings and reinforcement signals". The verification of Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] hypothesis is a very important one and highlights the necessity for a more detailed analysis of the high firing irregularity phenomenon.

1.3 Motivation

Further research in this area is imperative. For example it would be interesting to investigate whether the same result could be produced by a network based on the theory of two compartment neuron models. More specifically, the basic idea is to test whether high firing irregularity can be produced by a network of two compartment LIF neurons (Lansky and Rodrigues [3] and Bressloff [4]) where the first compartment is the dendritic compartment and the second compartment is the somatic

compartment. Furthermore, in case this network passes this test, it could be tested whether it can produce the High Firing Irregularity Enhanced Learning as in the case of the studies Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2].

Furthermore, in Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] the network consists of single compartment LIF neurons where dendritic and somatic points are a single point. In this node the dendritic potential is assumed to return to the resting level following the membrane potential after a spike. With the application of the two compartment model where the dendritic potential does not return to the resting level and the reset is applied only to the membrane potential, the model becomes slightly more realistic.

In this thesis a network based on two-compartment LIF neurons was introduced and tested in order to investigate first if the two-compartment model as a single neuron is able to produce firing irregularity at high rates and furthermore if the network, that consists of neurons based on this model that fire irregularly at high rate, enhances learning as Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] claim for a network of the single compartment LIF neuron. In addition, for comparison purposes the single compartment models with total somatic reset and partial somatic reset and their corresponding networks, same as with the ones used in Florian [5] and Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] are implemented. The comparison of the results of all networks are presented and discussed in order to prove that the network and the model that is tested by this thesis are able to a) fire irregularly in high rates and b) by firing irregularly at high rates the network of two-compartment neurons can enhance learning.

By re-verifying the results of Christodoulou and Cleanthous [1] and Cleanthous

and Christodoulou [2] with a network of two-compartment LIF neurons, this thesis can not only verify the assumption that high firing irregularity enhances learning, but also introduces a different way of testing this phenomenon.

1.4 Thesis outline

Chapter 2 reviews the exist knowledge upon which this thesis is based. A review of the literature related to machine learning, learning algorithms and the reasons of selecting the current machine learning algorithm for training a spiking neural network are presented. Moreover a review of studies related to this thesis are presented in this chapter. Finally, a review of the basics of two compartment models as is being used in this thesis and also presented by Lansky and Rodriguez [3] and Bressloff [4] is presented in this chapter.

Chapter 3 describes the design of the computational system with respect to its architecture and the implementation. The two compartment model and the network that used by this thesis are presented and described.

Chapter 4 presents and discusses the the results of testing the two compartment LIF model in producing high firing irregularity in comparison with the LIF with partial somatic reset model and also the results of testing the networks used in this thesis in the XOR problem solving.

Chapter 5 includes the conclusions of this thesis and provides suggestions further investigation.

Chapter 2 (Background)

2.1 Computational neuroscience

2.2 Learning

2.3 Learning Algorithms

2.4 Reinforcement Learning

2.5 Spiking Neural Network using Reinforcement Learning

2.6 Reinforcement Learning through STDP

2.7 Florian's modulated STDP by a global reward signal

2.8 High Firing Irregularity of Cortical Cells at high rates

2.9 Leaky Integrator Neuron Model with Partial Reset

2.10 High Firing irregularity enhances learning

2.11 Two Compartment model

2.1 Computational neuroscience

The aim of Computational neuroscience is to investigate and understand the computational properties of the brain (from single neuron to whole neural network systems). This is done by using methods from computer science and mathematics combined with the experience gained from the area of experimental neuroscience. Computer science, electronic engineering, mathematics and physics form the computational and theoretical approaches to neuroscience and are concerned with the theoretical aspects of information processing by neural systems.

In more detail, in Computational neuroscience mathematical methods and computer science are used in order to investigate and understand the nervous system behaviors. There are two main approaches for these investigations. In the first approach mathematical and computational models are used in order to simulate in detail a neuron behavior. In the second approach the brain is analyzed as an abstract computing device, therefore researchers can describe neural function within the theoretical frameworks. This allows alternative analytical approaches. This thesis belongs to the first category of approaches where models are used to investigate brain (neuron) behavior.

Computational neuroscience may be viewed as the discipline which relates to artificial intelligence and more specifically the computational intelligence since both disciplines are trying to understand the information processing capabilities of the neural system (brain). For general neuroscience and for computational neuroscience the understanding of the functional properties of the brain is very important and led the ultimate goal of this field. In computational intelligence the understanding of any aspect of biological information provides new approaches in building of intelligent systems.

One of the main research areas in artificial intelligence and therefore in the area of computational neuroscience is machine learning which is described in the next section.

2.2 Learning

In order to define a learning problem, we need to be aware of a) the class of tasks, b) the measure of performance improvement and c) the source of experience. **"**A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T, as measured by P, improves with experience E" [13]. For example a computer that learns to solve the XOR problem can improve its performance as measured by its ability to give the correct answers (task) through experience obtained from training. In general, machine learning aims to answer the question of how to build a computer program that improves its performance at some task through experience. Machine learning includes ideas from a great variety of disciplines such as artificial intelligence, probability and statistics, computational complexity, information theory, psychology and neurobiology, control theory, and philosophy. Well defined learning problems require a well-specified task, a performance metric, and a source of training experience. In order to design a machine learning approach to solve a problem one will face, many question the type of training experience that should be used, the target function to be learned and the representation that should be used for this target function. The basic question, though, is about the algorithm that should be used. In other words, which algorithm is more suitable in each case.

2.3 Learning Algorithms

Several algorithms exist for machine learning and each of them has its advantages and disadvantages. The three main categories of machine learning algorithms are a)

supervised, b) reinforcement and c) unsupervised. In supervised learning the training data consists of a set of training examples that include the input object and the desired output value. In this case, the algorithms take the input and by applying the corresponding algorithms function produce an output for each valid input. The network calculates the error between the desired output (supervised signal) and the actual output and follows the algorithm rules to make changes in order to minimize the error. In reinforcement learning which is discussed in detail in (2.3), a reward or penalty is given to the network based on its output in order to force the network to make changes which will increase the reward. In unsupervised learning the network tries to find relations and hidden structures in unlabeled data. The difference between unsupervised learning and the other two categories (supervised learning and reinforcement learning), is that there is no error or reward signal to evaluate the output. For the purposes of this thesis reinforcement learning has been used. The rationale for using reinforcement learning and a more detailed explanation of it is provided in the following section.

2.4 Reinforcement Learning

According to Florian [5]: “Reinforcement Learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal action to achieve its goals”. Some agents learn to play board games such as chess, learn to solve problems such as XOR or learn to optimize operation in factories. The basic idea in reinforcement learning is that for each time the agent performs an action in its environment, a trainer provides a reward or penalty to help the agent to know if its action has a positive or negative effect on the environment according to its

goal. For example for an agent that learns to recognize letters, the trainer gives a reward to the agent each time the agent predicts the letter correctly, a penalty for each time the agent predicts the letter wrong and zero reward on all other states. The goal is to train the agent to choose the sequence of actions that will give it the greatest reward. Two main strategies exist for solving reinforcement learning problems. The first is to search a variety of behaviors in order to find one that performs well in the environment. This approach is used in genetic algorithms and genetic programming. The second strategy is to use statistical techniques and dynamic programming methods to estimate the usefulness of taking an action in states of the world [14].

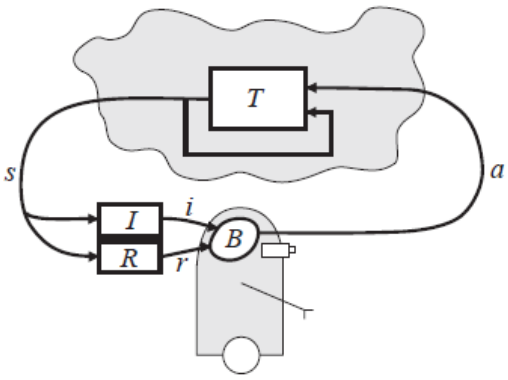


Figure 1 Standard reinforcement-learning model from [14]

In standard reinforcement-learning model as is shown in figure 1, on each interaction step the agent receives as input (i), some clue from the current state (s) of the environment. Then the agent selects an action (a) to give as output. Based on the selected action the state of the environment changes. This change to the environment based on it feedback (good or bad) is returned to the agent as a reinforcement signal (r). The behavior of the agent (B) should choose actions that will increase the long-turn reword and not the penalty reinforcement signal. The agent learns to do this over time by trial and error [14].

Reinforcement learning has several differences to the supervised learning. The main one is that in reinforcement learning there in no presentation of input object and corresponding desired output. In reinforcement learning after each action the agent is told the immediate reward but not the action that will give it the long-term biggest reward. It is the agent itself that by collecting experience about the state, actions, transitions and rewards will find the way to gain the maximum long term reward [14]. Reinforcement learning is related to issues of search and planning algorithms in artificial intelligence. In artificial intelligence, the search algorithms use the graph of states that they generate their selves and use them for a satisfactory performance. Planning algorithms operate in a similar way, albeit within a more complex construct, in which states are represented by compositions of logical expressions. Based on the fact that those artificial intelligence algorithms require a predefined model of state the reinforcement learning algorithms are more general since reinforcement learning assumes that the entire state space can be enumerated and stored in memory [14].

2.5 Spiking Neural Networks using Reinforcement Learning

Reinforcement learning has been successfully applied to spiking neural networks recently. Some reinforcement learning method achieve learning by utilizing various biological properties of neurons such us neurotransmitter release used by Seung [15], spike timing used by Florian [5], Izhkevich [16], Farries and Fairhall [17] and Legenstein et al. [18] or firing irregularity used by Xie and Seung [19]. All of the above methods are biologically plausible and this is the reason for being applied successfully in biological realistic neural models. The fact that those methods are

biologically plausible makes them ideal methods to be used in investigations of neuron behaviors. For this reason this thesis used reinforcement learning through MSTDP, this is described below.

2.6 Reinforcement Learning through STDP

As Florian [5] said: “The persistent modification of synaptic efficacy as a function of the relative timing of pre-and postsynaptic spikes is a phenomenon known as spike-timing-dependent plasticity (STDP)”.

The discovery that synaptic changes are depended on the relative timing of pre- and postsynaptic action potentials is owed to the work of Markram et al [20]. As showed by their experiments, there is a potentiation of a synapse when the post-synaptic spike follows the pre-synaptic spike within a time window of a few tens of milliseconds and a depression of the synapse where the order of spikes is reversed. The current type STDP is also called Hebbian from Donald Olding Hebb [21] who discovered the change in synapse when the pre-synaptic neuron forces the postsynaptic neuron to fire.

In more detail in Hebbian STDP, the plasticity mechanism strengthens the synapse when a pre-synaptic neuron contributes to the firing of the post-synaptic neuron making the pre-synaptic neuron more effectively in causing the postsynaptic neuron to fire. In other experiments, done by Dan & Poo [22], Bell et al [23], Egger et al [24] and Roberts & Bell [25], an anti-Hebbian STDP synapse was found when the sign of changes was opposite to the Hebbian STDP.

Florian [5] claims that modulation of STDP by a global reward signal can lead to reinforcement learning for a spiking neural network. According to Florian [5],

Hebbian spike-timing-dependent plasticity can enable a network to associate a stable output to a particular input. Furthermore, he claims that it is possible to control whether to reinforce the casual relationships or not based on whether this will lead to something positive. More specifically, the causal relationships are reinforced only when this action leads to something positive and weakened otherwise. In this case the synapse should feature Hebbian STDP when the reward is positive and anti-Hebbian STDP when the reward is negative. This will lead the neural network to learn to associate a particular input to a desirable output, as determined by the reward and not to an arbitrary output, determined by the initial state of the network

2.7 Florian's modulated STDP by a global reward signal

Florian [5] showed that modulation of STDP by a global reward signal leads to reinforcement learning. A neural spiking network that used modulated STDP as learning was used to prove this claim. Moreover a network with an eligibility trace that kept a decaying memory of the effects of recent spike pairing to allow learning in the case that reward is delayed was proposed.

Generally, both networks simulated the STDP by increasing the weights between two neurons that fired in pre-post order between the window and decrease the weights between neurons that fired in post-pre order between the window. In addition, the network's used a global reward that was given to the network for each correct fire and penalty otherwise. Both networks achieve their goal which was to calculate the XOR function (XOR Benchmark).

Florian claimed that the causal nature of the STDP window seems to be an important factor for the learning performance of the proposed learning rules. The

proposed reinforcement learning mechanism retains the continuity between itself and the experimentally observed STDP, and this makes it biologically plausible. He also claimed that the introduction of the eligibility trace does not contradict what is currently known about biological STDP, as it simply implies that synaptic changes are not instantaneous but are implemented through the generation of a set of biochemical substance that decay exponentially after generation. The new feature (i.e., modulatory effect of the reward signal) may be implemented in the brain by a neuromodulator.

The fact that the proposed spiking neural network learning was biologically plausible makes it an ideal learning algorithm to been used in spiking neural networks for investigating behaviors of real neurons such as firing irregularity at high rates that Softky and Koch [6,7] found on cortical cells of the brain.

2.8 High Firing Irregularity of Cortical Cells at high rates

Based on the theory about spiking in cells according to which when a typical nerve cell is injected with enough current, it will fire a regular sequence of action potentials, Sofkty and Koch [6,7] showed that cortical cells in vivo usually fire irregularly at high rates, reflecting synaptic input from presynaptic cells as well as intrinsic biological properties. The experiments were conducted on awake macaque monkey. The trains were recorded from V1 (Knierim and Van Essen 1992) and MT (Newsome et al. 1989). Traces were chosen from well-isolated, fast firing, non-bursting neurons.

They also tried to simulate this high firing irregularity at high rates, using a simple integrate-and-fire model, but the model failed to reproduce this high firing irregularity. The next step was to use the more realistic Hodgkin-Huxley model [26] whose firing currents are continuous functions of voltage. This is not the case in the

integrate-and-fire model which has a discontinuous firing threshold and no such sensitive voltage regime. They mention however, that this would make a significant difference only in neurons that spend a lot of integration time resting just below the threshold and this is not the case in cortical cells which have high firing rates and hence no stationary resting potential during periods of peak activation. Therefore a simulation of Hodgkin Huxley like neuron in the presence of random synaptic input was needed for a persuasive test.

Softky and Koch [6,7] simulated this biophysically very detailed compartmental model but the results at the end were in agreement with the simple integrator models and not with what was recorded in vivo monkey cells . Sofkty and Koch [6, 7] suggested that the problem was the present knowledge of pyramidal cell biophysics and dynamics.

Softky and Koch [7] also mention that for the traditional view of cortical firing variability where the information of neural code is carried in the average spike rate (frequency code), a neuron with irregular firing is the worst case of carrying the information in its average rate. On the other hand an alternative view is that each spike arrival's time signifies an independent message of some sort (an asynchronous binary pulse code). In this case a neuron with irregular firing would be the most appropriate one for carrying information in its individual spike times. This makes the phenomenon of high firing irregularity a very important area for research. The fact that highly irregular firing of cortical neurons at high rates cannot be reproduced by a single neuron performing the temporal integration of Excitatory Post-Synaptic Potential generated by independent stochastic input spike trains that Softky and Koch [6,7] observed triggered investigations into alternative ways of producing irregular spike trains.

While many methods were proposed in order to reproduce Sofkty and Koch's findings, Bugmann , Christodoulou, & Taylor [10] have shown that a LIF neuron model with partial somatic reset was a very promising candidate for reproducing the observed highly firing irregularity at high rates.

2.9 Leaky Integrator Neuron Model with Partial Reset

In their work Bugmann, Christodoulou, & Taylor [10] investigated the mechanism by which partial reset affects the firing pattern and, by this, proved that partial reset is a simple and powerful tool for controlling the irregularity of spike train fired by a leaky integrator neuron model with random inputs. They also showed that this mechanism enables a single neuron with a realistic membrane time constant to reproduce the highly irregular firing of cortical neurons at high rates.

Partial reset as presented by Shigematsu et al. [27] is a mechanism where an output spike does not completely reset the membrane potential of the neuron model. In Bugmann , Christodoulou, & Taylor [10] model when a LIF neuron fires it resets the potential of the capacitor to $V=\beta*Vth$, where Vth is the threshold of the model and β is the a reset parameter between 0-1. By using partial reset, the temporal integration of random input spikes is exploited for maintaining the average potential of the neuron at a small distance from the threshold during the whole integration time, allowing input current fluctuations to cause firing at random times.

By comparing the results from Rospars and Lansky [28] and those of Christodoulou et al [11] who showed that $CV>1$ when no resetting was used and with the results of Sofkty and Koch [6] who showed that $CV<1$ for $\beta=0$, Bugmann ,

Christodoulou, & Taylor [10] proved that partial reset may allow a fine control of the irregularity of the spike trains.

2.10 High Firing irregularity enhances learning

Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] investigated whether high firing irregularity is utilized by the brain for the proposes of learning optimization. In other words, they tried to answer the question whether high firing irregularity enhances learning According to Florian [5] a biological realistic implementation of reinforcement learning on a spiking neural network is achieved by modulating STDP with a reward signal. Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] in their work used this implementation in combination with the LIF with partial somatic reset model that Bugmann , Christodoulou, & Taylor [10] suggested as well as different approach of getting firing irregularity at high rates, namely the use of the temporally correlated inputs (for more information see [2]) , in order to investigate whether the high firing irregularity enhances MSTDP.

In order to test this assumption, the XOR benchmark problem and the Prisoner's Dilemma game [12] were used. The first step was to achieve the high firing irregularity of the LIF neuron at high rates, using the partial somatic reset mechanism [10]. The next step was the implementation of the testing networks. It has to be noted that in the case of the prisoner's dilemma, two networks are needed to represent the two prisoners. In order to test whether high firing irregularity enhances the efficiency

of MSTDP with eligibility trace to perform the chosen learning task, three different simulations were carried out for each task: one where the network's units had the standard LIF model with total reset, one where the network's units had the standard LIF with the partial somatic reset mechanism and a final one where the network units had the standard LIF, but the inputs to the system were temporally correlated. In the last two networks the LIF neurons are able to produce irregularity in contrast with the first network. The output firing rate of all networks was targeted to be equal, so in case of any difference in leaning efficiency of the networks this would be due to the firing irregularity and not to an increasing firing rate. The high output firing rate which was targeted for all systems was within the high rate bound in which cortical cells in vivo fire irregularly shown by Sofkty and Koch [6,7].

As is shown by the results of [1,2], in the case of the XOR problem, even though all three systems learned the XOR function, the network with the partial somatic reset mechanism and the one which received temporally correlated inputs preformed much better in the task. The measure of efficiency was the difference between the output firing rates for input patterns {1,1} and input patterns {0,1} and {1,0}. Moreover the results of the simulations in the prisoner's dilemma game have shown that all three systems learn to cooperate, but when the system comprises of LIF neurons with partial somatic reset and when the system receives temporally correlated inputs, the accumulated payoff is much higher than when there is total reset after each firing spike.

In general the findings from the simulations in [1,2] showed that high firing irregularity at high rates enhances reward-modulated STDP with eligibility trace. Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] claim that this is due to more accurate correlations between pre-synaptic and post-synaptic spike

timings and reinforcement signals. More spercifically in the case of regular firing, two matching spike pairs are possible to be associated with opposite in sign reinforcement signals. This will confuse the directions of the plasticity for the current synapse. In case of high firing irregularity this situation is prevented by weakening this possibility [1,2].

What Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] found suggests that the high firing irregularity is utilized by brain for learning optimization. In other words high firing irregularity enhances learning.

Chapter 3 (Design)

3.1 Overview

3.2 Two point model

3.3 The network

3.4 Learning approach

3.5 Implementations

3.1 Overview

For the purposes of this thesis we have developed a neural network model based on two compartments. The main difference of this approach is that unlike single compartment LIF models, this model follows the assumption that the dendrite's potential is never reset.

Before the creation of a network of two compartment neurons that would fire irregularly, and since the model has never been tested in the past in reproducing high firing irregularity at high rates this single neuron model that consists of two compartments (dendrite - membrane) had to be tested in order to investigate whether the high firing irregularity at high rates shown by Softky and Koch [6,7], can be produced by such model (as it happened with the case of the partial somatic reset model shown by Bugmann, Christodoulou & Taylor [10] and Christodoulou & Bugmann [11]).

The next step was to design a neural network where the learning approach proposed by Florial [5] is tested in order to investigate whether a more realistic neural network, such as this one, would produce the result that Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] showed (i.e, the high firing irregularity enhances learning).

3.2 Two point model

As mentioned in the previous section, for the purposes of this thesis first of all a single (two point) neuron model needed to be tested in order to investigate whether it could reproduce the results reported Bugmann, Christodoulou & Taylor [10] and Christodoulou & Bugmann [11]. The chosen model was the one presented by Lansky and Rodriguez [3] and Bressloff [4].

There is a variety of approaches for modeling the neuronal activity with deterministic biophysical concepts which are very powerful in explaining the generation of the various types of membrane potential like the Hodgkin-Huxley model [26]. The needs of this thesis, however, led us to choose a lightest model. The LIF model is the best for this investigation since the evolution of the neuron membrane potential is described by the following simple stochastic differential equation.

$$I(t) - \frac{V_m(t)}{R_m} = C_m \frac{dV_m(t)}{dt} \quad (1)$$

where R_m is he membrane resistance, C_m is the membrane capacitor, $I(t)$ is the input at time (t) and V_m is the membrane voltage.

Of course this model is not focused on the geometrical architecture of the cell. There are many complex models that are focused on the geometrical architecture of the cell, albeit those models are not focused on the direct presentation of the input output transfer. As in the case of Lansky and Rodriguez [3] and Bressloff [4] for this thesis we need a simplified model that will focus on the description of the input-output transfer. This led us to the use of LIF model that follows the same assumption as with Lansky and Rodriguez [3] and Bressloff [4]:

1. The neuron is assumed to be a mode of two interconnected compartments (dendritic and membrane zone)
2. The input is presented only at the dendritic compartment
3. The reset mechanism is used only at membrane zone.

The above assumptions led us to architecture similar to the architecture shown in figure 2.

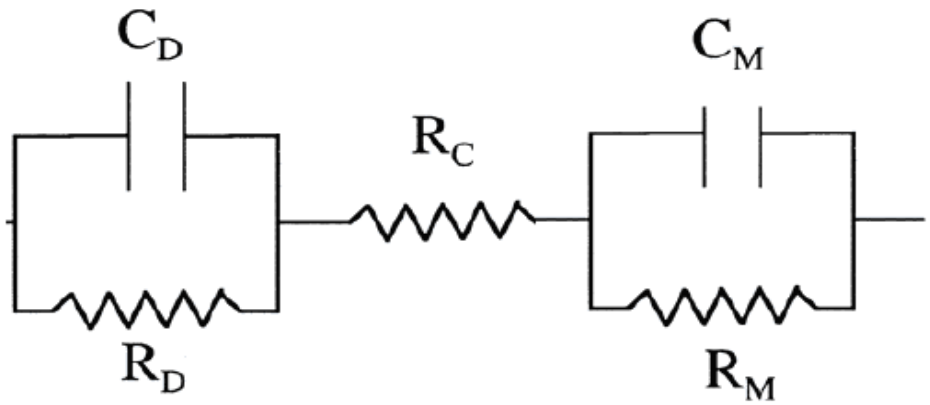


Figure 2 Two compartment model of two interconnected LIF compartments. C_D:dendrite capacitor, R_D: dendrite resistance, Rc: junctional resistance, C_M: membrane capacitor and R_M): is membrane resistance**.**

In the corresponding model based on the Bressloff [4] and Lansky and Rodriguez [3] the simplifying assumption that the dendritic compartment depolarization is not influenced by the voltage at the trigger zone is removed and therefore when the membrane potential resets, there is a feedback in the dendritic system due to the coupling between membrane and dendrites. It is a simple example of an excitable or active system (soma) coupled to a non-excitable or passive system (dendrite). The equations which describe the potential in both zone (dendrite and membrane) are the following:

$$dX_1(t) = \left(\frac{X_1(t)}{\tau_1} + \frac{X_2(t) - X_1(t)}{\tau_r} + \mu(t)\right) dt \qquad (2)$$

$$dX_2(t) = \left(\frac{X_2(t)}{\tau_2} + \frac{X_1(t) - X_2(t)}{\tau_r}\right) dt \qquad (3)$$

Where τ_r is the junctional time constant, $\mu(t)$ represents the external input, τ_1 is the dendrite time constant and τ_2 is the membrane time constant , $X_1(t)$ is the dendritic potential at time t and $X_2(t)$ is the membrane potential at time t. As shown in the equations, the potential in both dendrite and membrane at time t are depended on each other as is shown in figure 3.

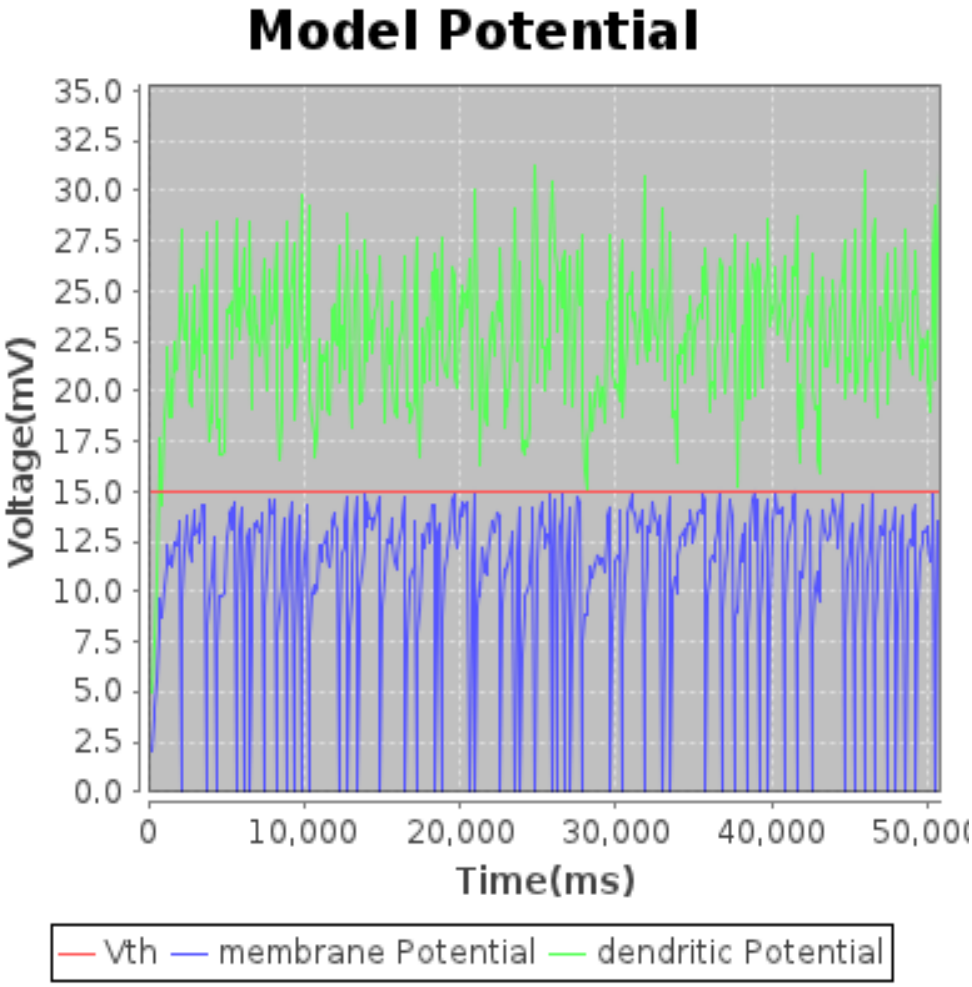


Figure 3 Membrane and Dendritic Potential produced by a two point model (equations 2,3)

This model was tested in producing high firing irregularity. Later on this model was used for the creation of the neural network as a node on it. The network is described in the following subsection.

3.3 The network

As we mentioned in the introduction, for the purposes of this thesis a neural network based on two compartment neurons that will fire irregularly had to be tested and in order to investigate whether it would produce the same results with the ones reported by Christodoulou and Cleanthous [1] Cleanthous and Cleanthous [2]. The proposed neural network architecture in this thesis is based on the architecture used by Florian [5] for solving the XOR problem. This architecture which was used by Florian [5] and also in Christodoulou and Cleanthous [1] Cleanthous and Cleanthous [2] network is presented in figure 4 and described in the following paragraph.

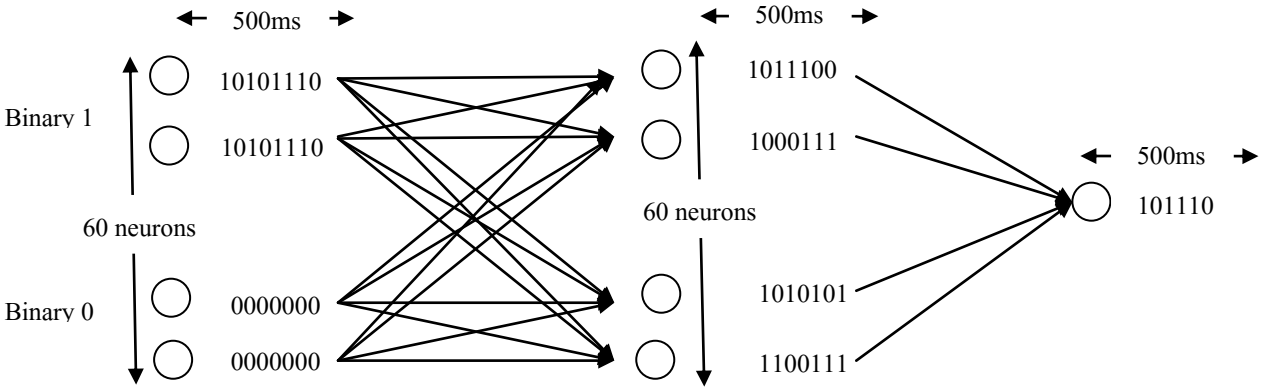


Figure 4 The network architecture for sovling the XOR problem

This is a feed forward network which consists of 60 input neurons , 60 hidden neurons and 1 output neuron. Each layer has full feed-forward connectivity to the next one. The first 30 input neurons represent the first binary input and the next 30 represent the second binary input. Binary input “1” was encoded by a poison spike

train of 40hz firing rate and the binary input “0” was encoded by the absence of spiking. Each input patter presentation lasted 500ms.

3.4 Learning approach

As mentioned in the background chapter and introduction, the current work uses modulated STDP [5] by a global reward signal as a learning algorithm for our network. According to Florial [5], in reward-modulated STDP with eligibility trace the efficacy of the synapse from neuron j to i is changed according to equation 3:

$$W_{ij}(t - dt) = W_{ij}(t) + \gamma\ dt\ r(t + dt) Z_{ij}(t + dt) \quad \textbf{(3)}$$

where γ is the learning rate, dt is the duration of a time step, r is the global reward signal and z is the eligibility trace modified according to equation (4):

$$Z_{ij}(t + dt) = \beta Z_{ij}(t) + \zeta_{ij}(t)/\tau_z \quad \textbf{(4)}$$

where β is discount factor between 0 and 1, ζ is a notation for the change of z resulting from the activity in the last time step and τ_z in the time constant for the exponential decay of z. At time t, ζ is computed by the following set of equations (5, 6, and 7):

$$\zeta_{ij}(t) = P^{+}{}_{ij}\ f_i(t) + P^{-}{}_{ij}\ f_j(t) \quad \textbf{(5)}$$

$$P^{+}{}_{ij}(t) = P^{+}{}_{ij}(t - dt)exp(-dt/\tau_{+}) + A_{+}f_{j}(t) \quad \textbf{(6)}$$

$$P^{-}{}_{ij}(t) = P^{-}{}_{ij}(t - dt)exp(-dt/\tau_{-}) + A_{-}f_{i}(t) \quad \textbf{(7)}$$

Where the variable $P^{+}{}_{ij}$ tracks the influences of presynaptic spikes and the variable $P^{-}{}_{ij}$ tracks the influences of the postsynaptic spikes. The time constants τ_{+} and τ_{-} determine the ranges of interspike intervals over which synaptic changes occur and according to the standard antisymmetric STDP model. A_{+} and A_{-} are positive and negative constant parameters respectively. The parameter $f_{j}(t)$ is 1 if neuron i has fired at time step t and 0 if the neuron i does not fire at time step t. The evolution and dynamics of the above parameters through time are being shown in figure 4 and described in the following paragraph.

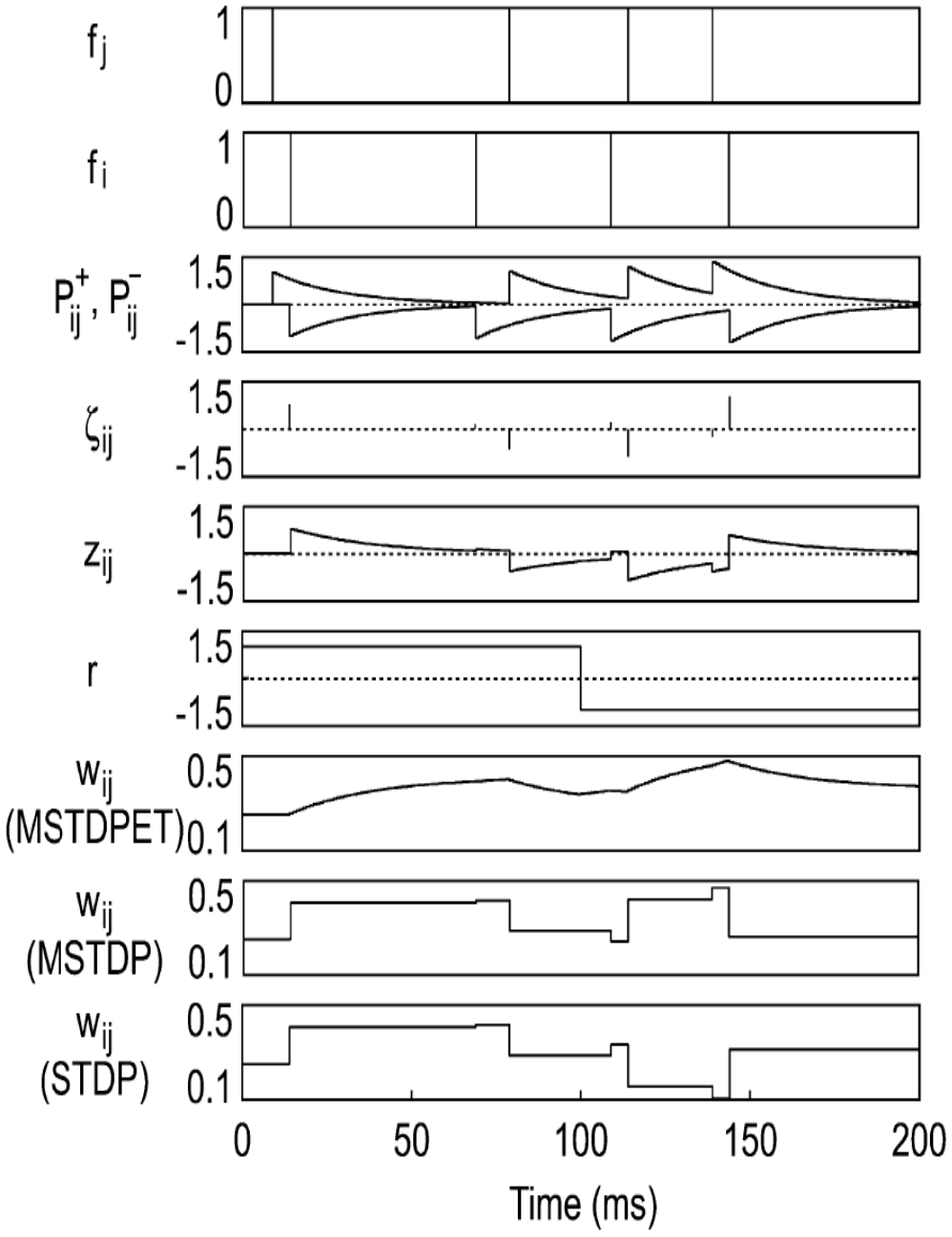


Figure 5 (taken from [5]) Illustration of the dynamics of the variables used by MSTDP and MSTDPED and the effects of those rules and of STDP on the synaptic strength for sample spike trains and reward.

As shown in figure 5, when a spike arrives in presynaptic area, P^{+}_{ij} gets a value of 1 and through the passage of time it follows a decay based on the time constant τ_{+} (equation 5). In the same way when a spike arrives on postsynaptic area, P^{-}_{ij} gets value the value of -1 and through the passage of time it follows a decay based on the time constant τ_{-} (equation 6). This is shown in figure 5

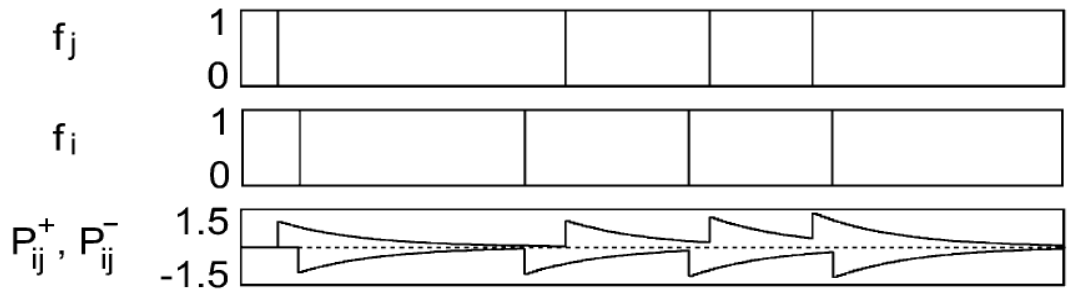


Figure 6 (taken from [5]) Part of figure 5.

Based on equation 4, ζ_{ij} takes positive value from the addition of P^{+}_{ij} and P^{-}_{ij} based on the pre-post firing activity (if the firing activity was only in presynaptic area then ζ_{ij} takes the value of P^{+}_{ij}, if firing activity was only in postsynaptic area then ζ_{ij} takes the value of P^{-}_{ij} only, if both areas had firing activity then it takes the sum of P^{+}_{ij} and P^{-}_{ij} and if neither area has firing activity then it takes the value of 0).

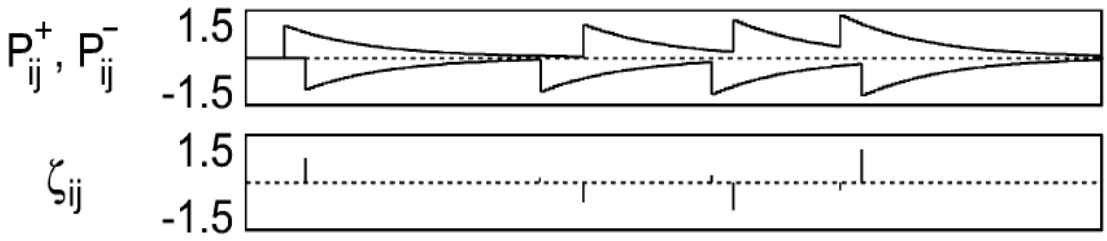


Figure 7 (taken from [5]) Part of figure 5.

Eligibility trace (Z_{ij}) keeps the trend of the learning (equation 3) to provide the system with the force of learning in case that the learning has delay, as shown in the figure, it gets the value 1 or -1 (depends on the pre-post firing activity) and through the pass of time it follows a decay based on the time constant τ_z (see figure 4).

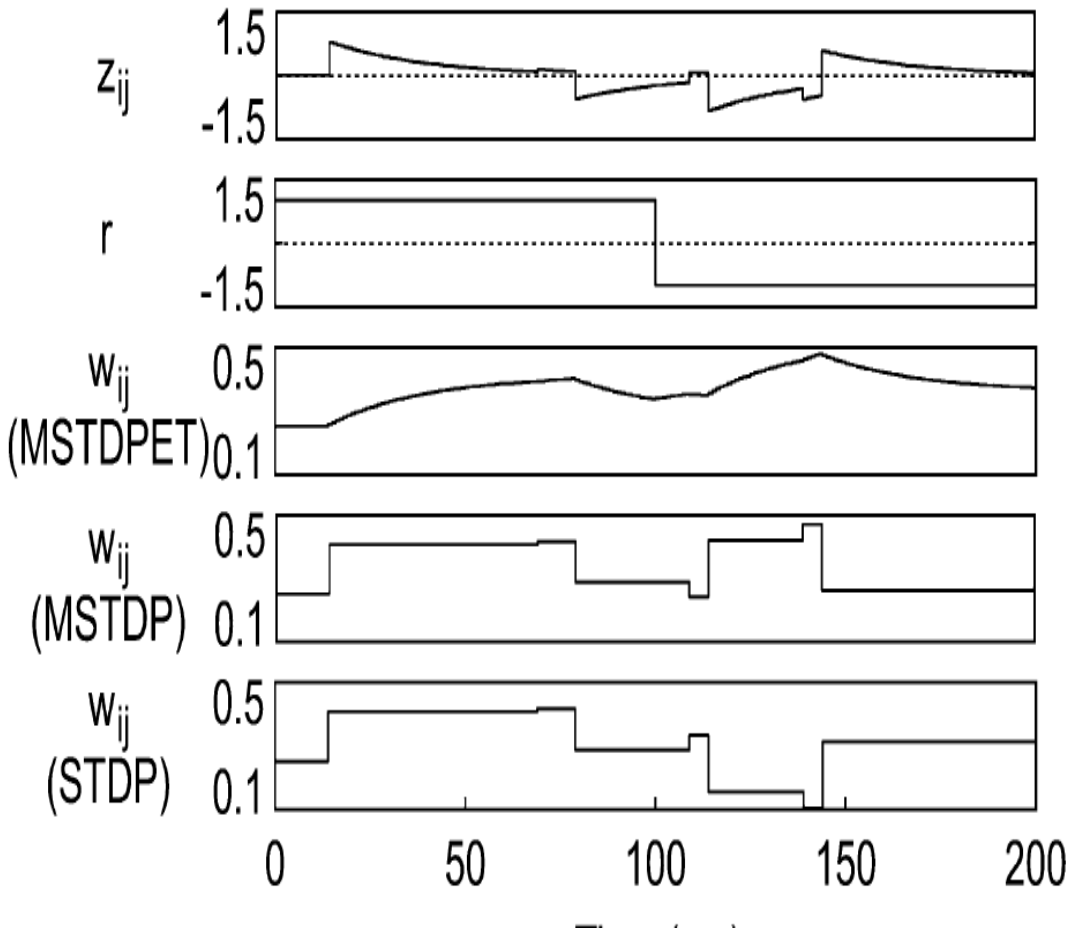


Figure 8 (taken from [5]) Part of figure 5.

The weights (equation 2) go stronger when the reward (r) is +1 or when there is presynaptic fire activity before the postsynaptic and on the other hand weights go weaker when the reward is – 1 or when there is postsynaptic fire activity before the presynaptic one. In case of the MSTDPET as mentioned earlier the eligibility trace keeps the learning (see figure 4).

In general the synapses go stronger every time a presynaptic spike comes before the postsynaptic one and weaker on the other way. Furthermore, in the case of the MSTDP, with the introduction of reward the network changes the synapses (makes them stronger or weaker) in order to maximize the global reward. Finally, in the case of MSTDPET, the eligibility parameter keeps the network to have the previous trend until the learning comes and this provides the better performance in case of learning delay.

3.5 Implementations

All the implementations of the above sections in chapter 3 are presented in the appendix. The implementation includes the implementation of the single compartment LIF with total partial reset, the implementation of single compartment LIF with partial somatic reset and the implementation of the two compartments LIF. In addition, the implementation includes the implementation of the three networks used for comparison in this thesis (one with single compartment LIF with total somatic reset as node, one with single compartment LIF with partial somatic reset as node and one with two compartment LIF as node) and the learning approach that is used for the experiments.

Chapter 4 (Results & Discussion)

4.1. Two compartment model: Can it produce high firing irregularity at high rates?

4.1.1. Parameters of the models

4.1.2. Model comparison and Discussion

4.1.2.1 Potentials of the models

4.1.2.2. Output Spike trains.

4.1.2.3 ISI distribution histograms and Autocorrelograms

4.1.2.3.1 ISI distribution Histogram

4.1.2.3.2. Autocorrelation

4.1.2.3.3 Poisson-type firing

4.2. Does high firing irregularity enhance learning produced by a two compartment model?

4.2.1. Training of the network

4.2.1.1 Training Parameters.

4.2.2. Results

4.2.3 Understanding the reasons of better performance with models that can produce high firing irregularity.

4.2.4 Why the two compartment LIF performs better?

4.1 Two compartment model: Can it produce high firing irregularity at high rates?

As mentioned in the introduction, for the purposes of this thesis the first step was to test whether a two compartment model of an LIF neuron can fire irregularly at high rates as the model suggested by Bugmann, Christodoulou & Taylor [10] and Christodoulou & Bugmann [11].

As mentioned in the previous chapter the two point model was based on the model used by Lansky and Rodrigues [3] and Bressloff [4]. Two interconnected compartments (dendritic and somatic) where the input is present only in the dendritic compartment and the reset mechanism is used only at the membrane zone. For comparison purposes the model used by Bugmann, Christodoulou & Taylor [10] and Christodoulou & Bugmann [11] (LIF with partial somatic reset) was also implemented by this thesis.

4.1.1 Parameters of the models

The table below shows the parameters of the model that was used for the results presented in section 4.1.2.

Parameter's	**LIF with partial somatic reset**	**Two compartment LIF**
Vth (Threshold)	15 mV	15 mV
Vreset (Resting potential)	0 mV	0 mV
Trefr (Refractory period)	2 ms	2 ms
Tm (Membrane time constant) = **Cm(Membrane capacitor)*Rm (Membrane resistance)**	20 ms	2 ms
Td (Dendrite time constant) = **Cm(Dendrite capacitor)*Rm (Dendrite resistance)**	-	15 ms
a (reset parameter)	0.91	-
rC (Junctional time constant)	-	2.5 ms

Table 1 The model parameters that were used for the produce of the results in section 4.1.2.

It must be noted that the difference in the time constant parameters of the two compartment model is due to the difference in the way it is modelled and the dependency between the two compartments. The relatively small membrane time constant, is due to the fact that the membrane potential is dependent on the dendritic potential. More specifically, when a spike arrives at the input, both the dendritic and the membrane potentials are increased. However, during the decay period towards rest, the membrane continues to be affected by the depolarisation of the dendritic compartment, making its leak rate much slower than expected by the small membrane leak time constant. This leads to a much larger effective membrane leak time constant.

4.1.2 Model comparison and Discussion

In this section, the comparison of the two models (i.e, the two compartment LIF and LIF with partial reset) will be presented in order to prove that the two compartment LIF model can produce the same high firing irregularity at high rates, as in the case of the LIF with somatic partial reset used by Bugmann, Christodoulou & Taylor [10] and Christodoulou & Bugmann [11].

The comparison will be based on the results produced by the models as demonstrated in the following figures (9, 10,11,12,14):

1. Two graphs where the evolution of the potential of each model (for the LIF with somatic reset the membrane potential and for the two compartment LIF, the dendritic and the membrane potential) is demonstrated (Figures 9 and 10).
2. Two graphs which demonstrate the output spike trains for each model on 100Hz firing rate (Figure 11).

3. Two interspike interval histograms (one for each model - Figure 12).
4. Two autocorellograms (one for each model- Figure 14).

4.1.2.1 Potentials of the models

In this section, the two model potentials will be presented for comparison in order to understand the difference in the evolution of their potentials and the behavior of each model.

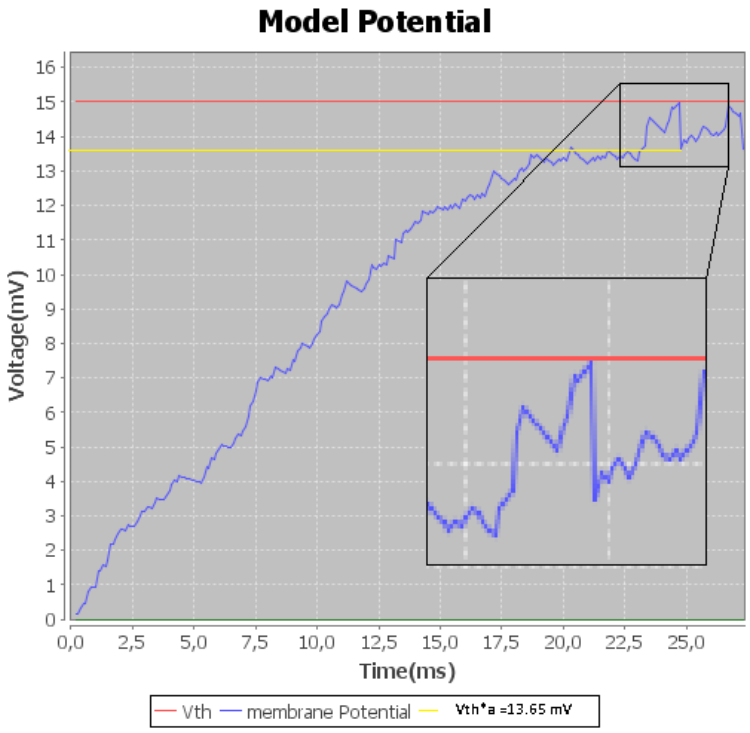


Figure 9 Simulation of the model potential (membrane) for LIF model with partial somatic reset. See section 4.1.1 for the parameters that are being used for this simulation

As shown in figure 9 the potential of the model is increasing ,as soon as there is incoming input to the system, until it reaches the threshold. At this point the model after firing is not reset to the reset value of the model 0mV but at 13.65mV (Vth * a, where a in the reset parameter) [yellow line].

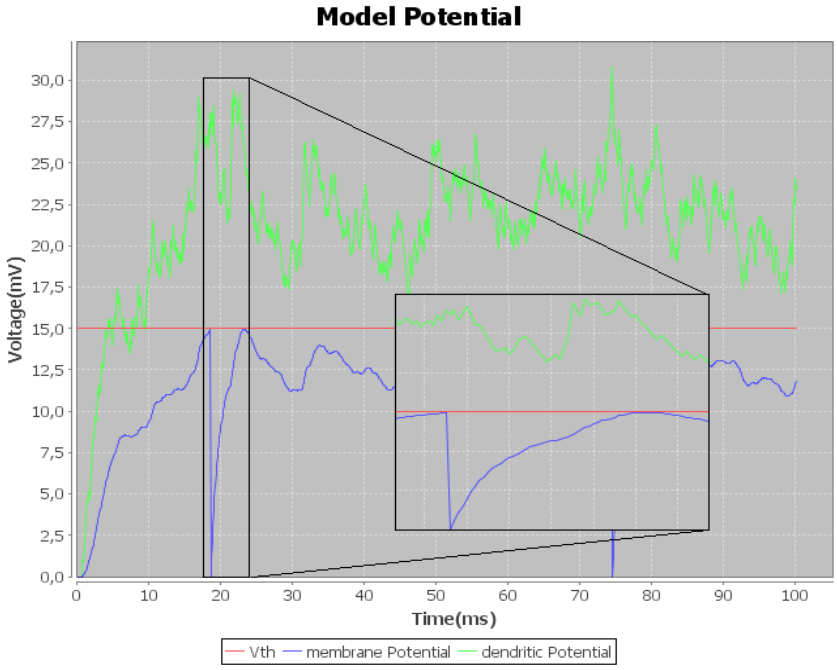


Figure 10 Simulation of the model potential (dendritic-membrane) for two compartment LIF model. See section 4.1.1 for the parameters that are being used for this simulation

As shown in figure 10 the model potentials (dendritic-membrane) are increasing as soon as there is incoming input to the system. There is dependence between them (dendritic potential and membrane potential) due to the coupling between them as mentioned in section 3.2. The figure shows that the dependence is quite noticeable when the membrane potential reaches the threshold where the reset

mechanism is applied only in the membrane potential (as mentioned in section 3.2) and the dendritic potential is not resetting. On the following millisecond the dendritic potential pulls the membrane potential up and the membrane potentials pulls down the dendritic potential at the same time.

4.1.2.2 Output Spike trains

In this section, the output spike trains of each model will be compared in order to test whether the two compartment LIF model has the similar behavior with the LIF with somatic partial reset.

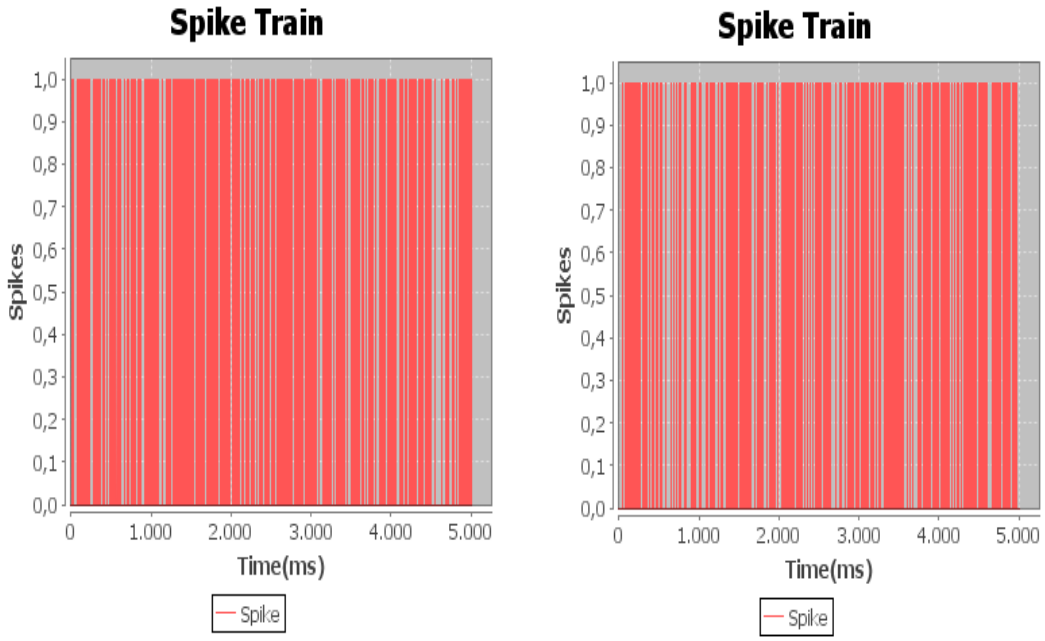


Figure 11 Output spike train of each model (First: LIF with partial somatic reset. Second: Two compartment LIF). See section 4.1.1 for the parameters used for this simulation. The current spike trains were taken when both simulations fired around 80-90Hz rate**.**

As shown in figure 11, both models have the similar behavior. More specifically, although both models fire at high rates they have no regular firing behavior and this can be observed from the figure since the spikes in both models have variable interspike intervals between them.

4.1.2.3 ISI distribution histograms and Autocorrelograms

Analysis of experimental data has been performed by Shadlen and Newsome [29] who plotted the experimental ISI histogram distribution (recorded from the area MT of an alert monkey, see Figures 1C in [29]) which can be fitted to an exponential probability density function, pointing to an underlying generating process of Poisson type. In this chapter an ISI distribution histogram and an Autocorrelogram for each model will be presented in order to prove that both models have Poisson-type firing based on Tuckwell [30] where poisson-type firing is verified if the interspike intervals are both expotentialy distributed (shown by ISI distribution histogram) and independent (shown by autocorrelogram).

In addition in both graphs (ISI distribution histogram and Autocorrelogram), the coefficient of variation is mentioned. The coefficient of variation (CV) is defined as the ratio of the standard deviation σ to the mean μ (mean inter spike interval). The standard deviation of an exponential distribution is equal to its mean, therefore its coefficient of variation is equal to 1. Therefore, distributions with CV << 1 are considered to be of low variance, while those with CV >> 1 are considered to be of high variance. As already mentioned the coefficient of variation of an exponential distribution is equal to 1, therefore the CV can be considered as a measure of spike

train irregularity defined as the standard deviation divided by the mean interspike interval.

4.1.2.3.1 ISI distribution Histogram

The distribution ISI histogram demonstrates the distribution of the observed times between the spikes collected in 'bins' of fixed width. Immediately after a spike a neuron has an absolute refractory period in which it is unable to fire another spike, so the first few bins of the histogram (in our models the corresponding bins for 2 ms- see refractory period in section 4.1.1) will be empty. The distribution of inter-event times for a wholly random process fits a negative exponential distribution on ISI histogram.

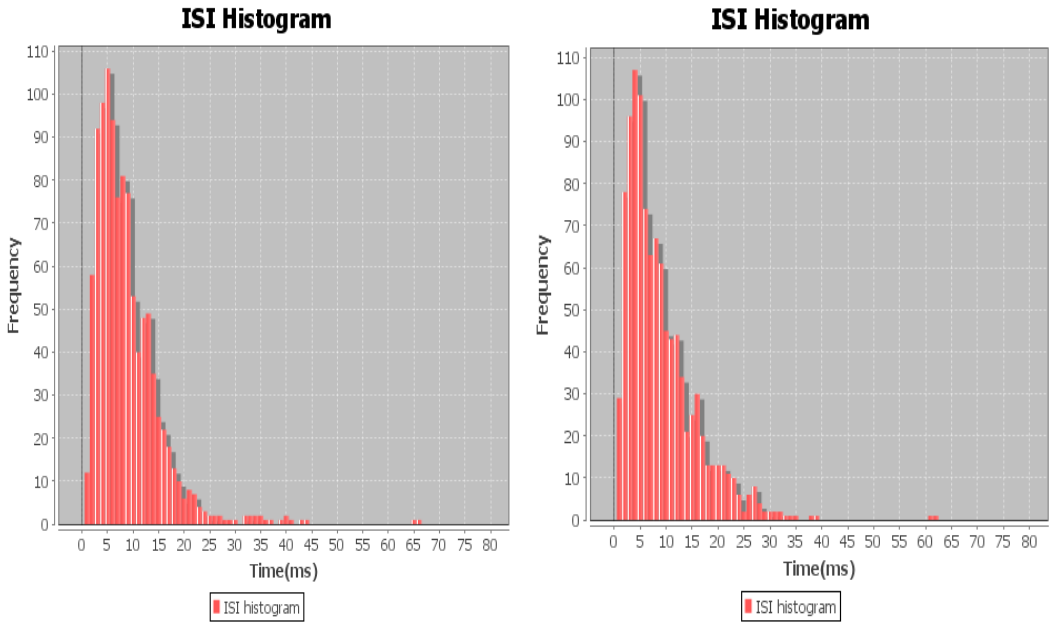


Figure 12 ISI distribution histogram for both models (1st : LIF with somatic partial reset at 100Hz firing rate, mean ISI at 9.4ms and CV=0.7. 2nd : Two compartment LIF at 100Hz firing rate, mean ISI at 10.2ms and CV=0.72). For the parameters see 4.1.1.

As we can observe in Figure 13, in both models the distribution of interspike intervals fits a negative exponential distribution same with the distribution of a wholly random process. At this point both models prove that at high rates (100Hz) they can exhibit an exponential distribution in ISI histogram same with the one shown by Shadlen and Newsome [29].

4.1.2.3.2 Autocorrelation

The mathematical representation of the degree of similarity between a given data set and a time delayed version of itself over sequential time intervals is called Autocorrelation. The difference between autocorrelation and normal correlation is that in the case of the first the two different time series are the same time series that are being used twice (once in its original form and once lagged one or more time period) [31].

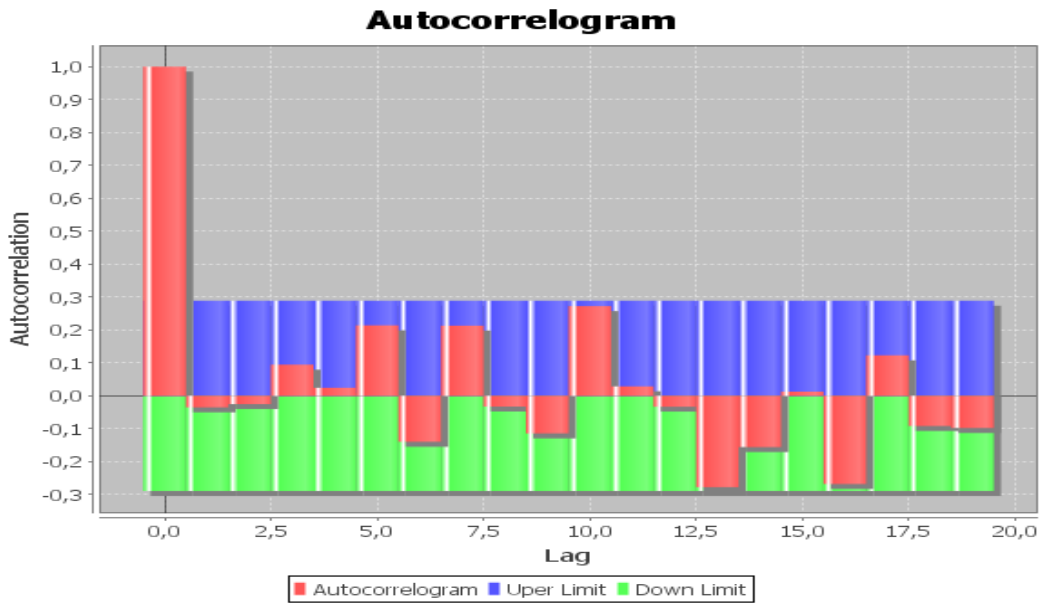


Figure 13 Autocorrelogram example**.**

Autocorrelogram is a commonly used tool for checking independence in a dataset. This independence is ascertained by computing autocorrelations for data values at varying time lags. If the dataset is independent, such autocorrelations should be near zero for any and all time-lag separations. If the dataset is dependent, then one or more of the autocorrelations will be significantly non-zero. The limits where an autocorrelation will be near to zero are specified from the limit lines in Figure 13).

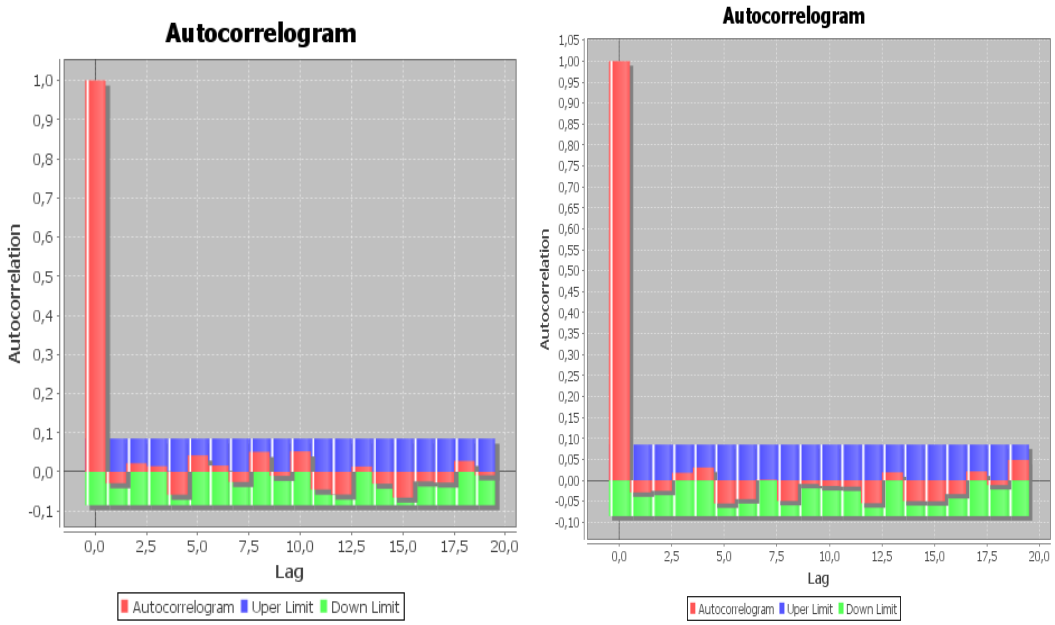


Figure 14 Autocorrelogram for both model (1st : LIF with somatic partial reset at 100Hz firing rate, mean ISI at 9.4ms and CV=0.7. 2nd : Two compartment LIF at 100Hz firing rate, mean ISI at 10.2ms and CV=0.72). For the parameters see 4.1.1.

Upper and Down limits on figure 13 define the confidence limits (95%) which are the acceptable limits for ISI independence of the curves. From Tuckwell [30] this is given by the equation $\pm 1.96/\sqrt{n}$ where n is the number of the interspike intervals (n=500) for each autocorrelogram. See section 4.1.1 for the parameters that are being used for this simulation.

As shown in figure 14, both models achieve the indepentance in firing activity since in both graphs all autocorrelations are between the limits specified to be the confidence limits for ISI independence as is shown by Tuckwell [30].

4.1.2.3.3 Poisson-type firing

With the results presented in the previous two sections (4.1.2.3.1-4.1.2.3.2) and based on [30] we can claim that the two compartment LIF model can have a poisson type firing in high rates same as the LIF model with partial somatic reset. This is proven by the fact that the interspike intervals are exponentialy distributed (see section 4.1.2.3.1) and, at the same time, independent (see section 4.1.2.3.2). In addition the captures presented in the previous section show a mean ISI at around 9-10 ms and CV around 0.7-0.75 which are values that together with the above facts indicate that we are closed to spike train irregularity based on what is mentioned in section 4.1.2.3 about the CV.

4.2 Does high firing irregularity enhance learning produced by a two compartment model?

As mentioned in the introduction, for the purposes of this thesis, the second step was to test whether the two compartment LIF model can he produce the results of Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] in reward MSTDPET learning [5].

In more detail, the model (two compartment LIF) has been tested in producing firing irregularity at high rates as the LIF with somatic partial reset model [10,11]. The comparison of the results of two models show that the two compartment LIF model is able to produce high firing irregularity at high rates.

This model (two compartment LIF) was used as node in a neural network. This neural network was trained with reward MSTDPED [5] and during the training the network was forced to fire irregularly in order to test whether similar results would be produced as in the case of Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] .

4.2.1 Training of the network

The network achieves learning through a process of rewarding and penalising according to the the output that it produces responding to a specific input. As in the case of Florian [5], here in order to train the network for XOR solving, the four input patterns were all presented in a random order in each learning epoch for 500ms. During the presentation, when the correct output was 1, the network received a reward r=1 for each output spike that occured and 0 in all other cases. If the correct

output was 0, the network received a negative reward r=-1 (penalty) for each output spike and 0 in all other cases. The reward was awarded to the network during the time step immediately after the output spike. 50% of the input-hidden synapse were randomly selected to be inhibitory while the rest of them were excitatory. The synaptic weights were hard bounded between 0 and 5 mV (for excitatory synapses) and between -5 mV and 0 (for inhibitory synapses). In the case of the network of two compartment LIF neurons the bounds were hard bounded between 0 and 1 mV (for excitatory synapses) and between -1 mV and 0 (for inhibitory synapses). The initial weights for the synapse were generated randomly within the specified bounds. The experiment took 200 learning epochs. If the network at the end of an experiment presented output firing rate for pattern {1,1} smaller that the firing rate for input patterns {0,1} and {1,0}, the network was considered to have learned the XOR function. The output firing rate of input pattern {0,0} was always 0 since as we mention in section 3.2 binary input "0" was encoded by the absence of spiking.

4.2.1.1 Training Parameters

The training parameters used in producing the results, shown in the next section (4.2.2,) are the same with the parameters that Florian used in [5] shown in table 2, with the only one that was different being the learning rate which was found empirically. Also the bounds in weights in the two compartment model are set to smaller values. The reason is that through testing the network showed that works better with these bounds. Although we chose to use the same parameters in order to achieve comparability between the networks of LIF with partial/total somatic reset and the network with the two compartment LIF, we should note that these were not identical since different weights have been used in each network. The difference in weights could not be avoided since weights in each network functioned differently due to modelling differences.

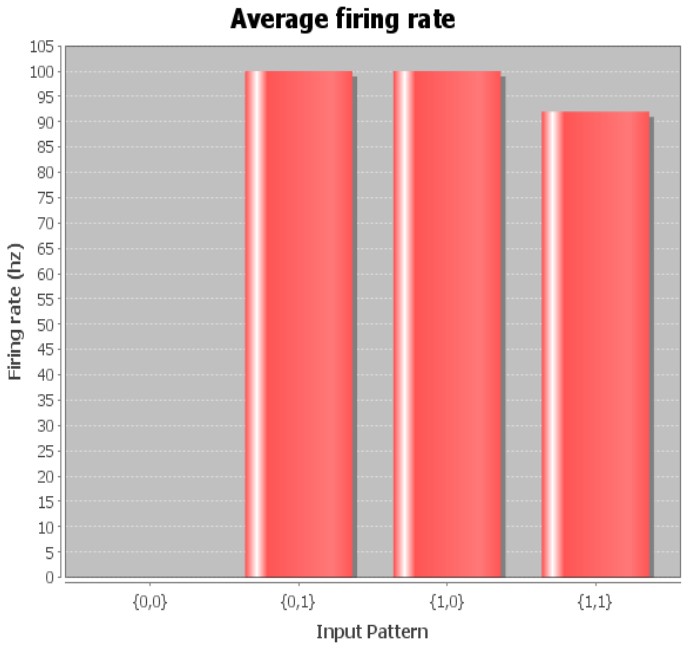
Parameters (see section 3.4 for explanation of the parameter below)	**Single compartment LIF with total/partial somatic reset**	**Two compartment LIF**
τ_+	20 ms	20 ms
τ_-	20 ms	20 ms
A_+	1	1
A_-	-1	-1
τ_x	25ms	25ms
γ	0.01	0.01
β	0.5	0.5
Weight bounds	-5mV to 0mV (for inhibitory synapses) ,0mV to 5mV (for excitatory synapses)	-1mV to 0mV (for inhibitory synapses) ,0mV to 1mV (for excitatory synapses)

Table 2 Learning parameters for the experiments presented in section 4.2.2

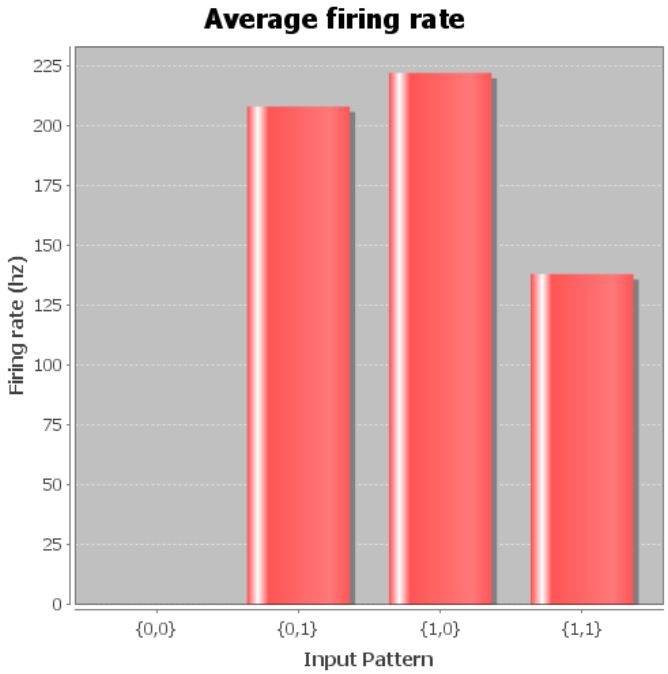
4.2.2 Results

In this section the results of the training of the three networks (LIF,LIF with somatic partial reset and two compartment LIF) with reward MSTDPET will be presented and compared in order to prove that a network which has nodes that fire irregularly can learn better than a network which has nodes that fire regularly.

A)



B)



C)

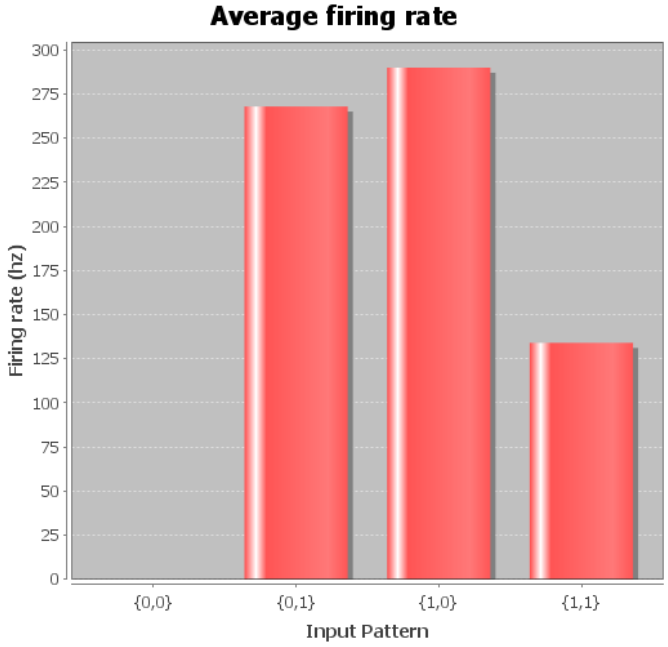


Figure 15 Average firing rate of the output neuron after learning, for the four different XOR input patterns (A: LIF with total reset model. B: LIF with somatic partial reset model. C: Two compartment LIF)

As shown in Figure 17 all three networks achieved learning but the results in LIF with partial somatic reset and two compartment LIF are better than the results of LIF with total reset. In more detail the network that consists of single compartment LIF with total partial reset suppressed the output firing rate for input patterns {1,1} 8% of the average output firing rate for input patterns {0,1} and {1,0}. In case of the network that consists of single compartment LIF with partial somatic reset the network suppressed the output firing rate for input patterns {1,1} 40% of the average output firing rate for input patterns {0,1} and {1,0}. At last in case of network that consists of two compartment LIF the network suppressed the output firing rate for input patterns {1,1} 52.72% of the average output firing rate for input patterns {0,1} and {1,0}. This is due to the high irregular firing that the LIF with partial somatic reset and two compartment LIF can produce which enabled the algorithm to perform more accurate correlations between pre-synaptic and postsynaptic spike timings and reinforcement signals.

4.2.3 Understanding the reasons of better performance with models that can produce high firing irregularity

As shown in the previous section the models that can fire irregularly at high rates (LIF with somatic partial reset and two compartment LIF) exhibit better performance in terms of learning. We believe as supported by Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] that this is due to the fact that high firing irregularity leads to more accurate correlation between pre-synaptic and postsynaptic spike timings and reinforcement signals.

In more detail in the case of regular firing, two matching spike pairs are possible to be associated with opposite in sign reinforcement signals. This will confuse the directions of the plasticity for the current synapse.. In case of high firing irregularity this situation is prevented by weakening this possibility [1,2]

The illustration of the dynamics of the variables used by reward-modulated STDP with eligibility trace showing the effects on the synaptic strength when spike trains are regular that is used in Cleanthous and Christodoulou [2] is also shown here (Fig 18) for better understanding how the regularity may destroy learning.

In Figure 18 (taken by Cleanthous and Christodoulou [2]) that shows the synaptic strength changes with time for two regular presynaptic and postsynaptic spike trains is shown. The problem is noticeable in this case if we see the synaptic strength which wavers around a given value until the reinforcement signal changes sign where it keeps waving around another value. The effect of any pre-post spike pair is cancelled by the next one during the time period where the constant reward/penalty is given to the network and the value of the synaptic strength remains up normally

constant. This destroys the learning since it causes the destruction of learning for this period of time [2].

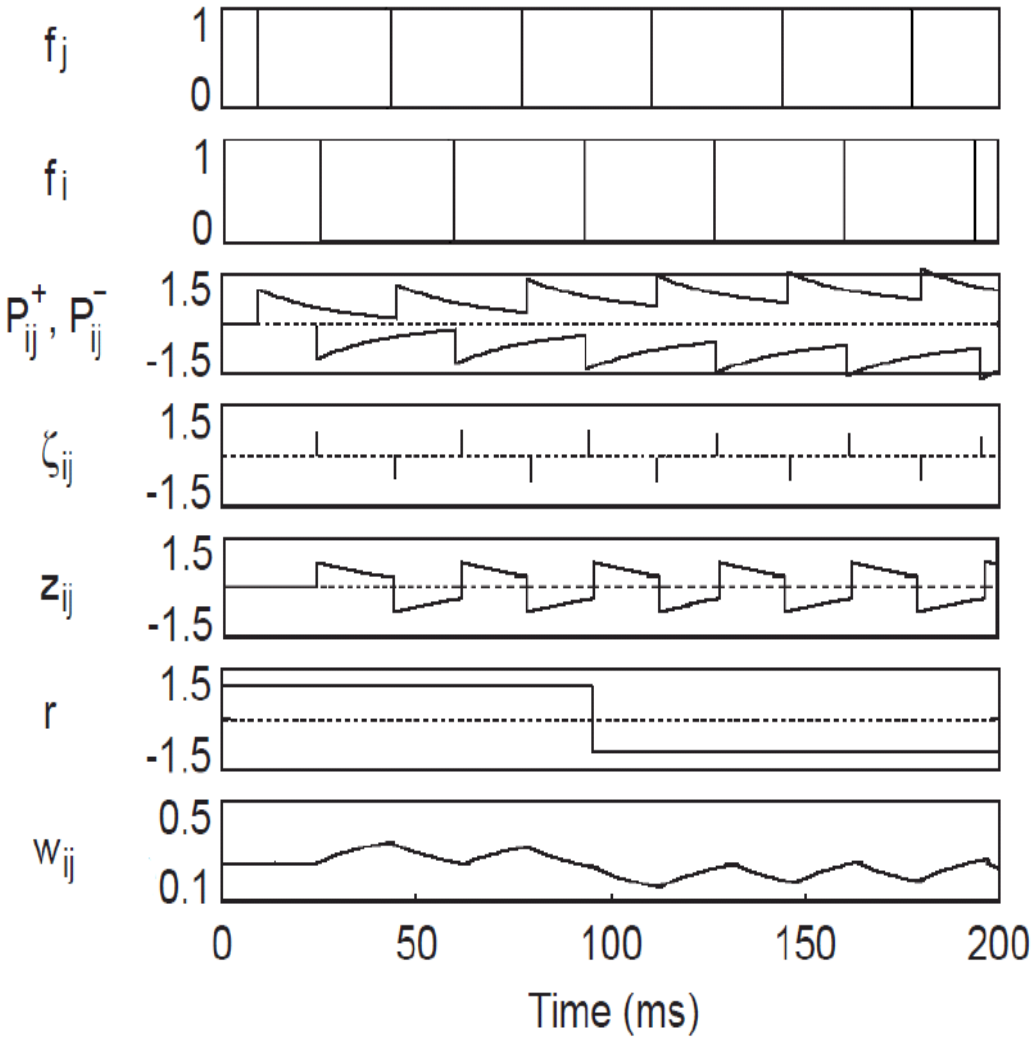


Figure 16 [taken from [2]] Effect of regularity in the value of the synaptic strength This figure is a modified version of the one presented in the original paper for the learning algorithm Florian [5]

Furthermore, if we consider the whole period of learning, we can detect another wavering. The average change made in the strength of the synapse by the reward is cancelled when the penalty signal comes causing the average synaptic value to become equal to its starting value. In this case there is no learning. Although on the above case the reward scale is equal to the penalty scale which leads the synaptic strength to waver around its starting value, if the reward scale is not equal to the penalty scale the synaptic strength will wavers around a value different from its starting value and by having this wavering the learning will be degraded. In general, regularity impairs learning because it causes the value of the synaptic strength to have this wavering behavior [2].

4.2.4 Why the two compartment LIF performs better?

As shown from the figures (in section 4.2.2) the result produced by the network which consisted of two compartment LIF nodes are better than the results produced be the network which consisted of LIF with partial somatic reset nodes. Having in mind that both models are firing irregularly in high rates, what is the parameter that makes the difference?

The better performance of the network of the two compartment model could be due to a variety of reasons. One can claim that the difference in weight bounds does not allow comparison due to the difference in experiment parameters. As mentioned in section 4.2.1.1, though, this was unavoidable because of the different modelling type that caused the weights to work in differently in each network.

More experiments and tests are needed in order to investigate the phenomenon of better performance by the two compartment LIF model.

Chapter 5 (Conclusion and Future work)

5.1 Conclusion

5.2 Future work

5.1 Conclusion

This thesis, by introducing a neural network consisting of two compartment leaky integrate-and-fire model as a neuron, investigated the claim of Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] that "high firing irregularity enhances learning". This was achieved by using two compartment LIF neuron modeling similar to the models used by Lansky and Rodriguez [3] and Bressloff [4].

After it was implemented, the model (two compartment LIF) was first tested in terms of producing high firing irregularity in high rates. For purposes of comparison in terms of producing high firing irregularity, the LIF model with partial somatic reset is also implemented as part of this thesis . As shown in the results the two compartment LIF model is able to produce high firing irregularity at high rates.

The two compartment LIF was applied to a network as a node and the network was forced to fire in high rates. The comparison with the other two networks (the one with single compartment LIF with total somatic reset as a node and the one with the single compartment LIF with partial somatic reset as a node) showed that the networks that can fire irregularly in high rates (single compartment LIF with partial

somatic reset and two compartment LIF) perform better in solving the XOR problem. This verifies the claim of Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] with a different method. According to Christodoulou and Cleanthous [1] and Cleanthous and Christodoulou [2] the better performance is due to the fact that high firing irregularity leads to more accurate correlation between pre-synaptic and post-synaptic spike timing and reinforcement signals

Furthermore, it was observed that the network which consisted of two compartment LIF nodes had better results that the network which consisted of LIF model with somatic partial reset as nodes. This cannot be easily explained because the different type of modeling sets limits in terms of comparison. Therefore, further investigation is needed in order to explore the reasons for the better performance by networks which consisted of two compartment LIF neurons.

5.2 Future work

A substantial part of this this thesis was dedicated to the implementation of all three models (single compartment LIF with total somatic reset, single compartment LIF with partial somatic reset and two compartment LIF) and to their corresponding networks in order to increase comparability and better inform the discussion for the result presented in chapter 4. The implementation of all three models leads to a variety of possibilities in terms of further investigation.

In this thesis the verification of Christodoulous and Cleanthous' [1] and Cleanthous and Christodoulou's [2] claim was achieved by using reward modulated STDP with eligibility trace (MSTDPET) (Florian [5]) as learning. A possibility for further investigation could be the testing of the already implemented networks in a

different learning methods like reinforcement learning of Stochastic Synaptic Transmission used by Seung [15]. A different learning approach, will allow further testing of the claim that high firing irregularity enhances learning.

Another possibility could be the testing of the MSTDPET (Florian [5]) applied to the already implemented networks in solving different problems like character recognition problem where the network is required to recognize the letters of the alphabet according to their declared of nature traits. This, of course, will demand modifications of the current networks architecture in order to fit the chosen problem.

References

[1] Christodoulou, C. and Cleanthous, A., **Does high firing irregularity enhance learning?** 2011, Neural Computation 23, 656-663.

[2] Cleanthous, A. and Christodoulou, C., **Learning optimization by high firing irregularity**. 2011, Brain Research, doi: 10.106/j.brainres.2011.07.025.

[3]Lansky, P and Rodriguez, R., **Two-compartment stochastic model of a neuron**. 1999, Physica D, 267–286.

[4]Bressloff P.C., **Dynamics of compartmental model integrate-and-fire neuron with somatic potential reset**. 1995, Physica D, 399–412.

[5] Florian, R.V., **Reinforcement learning through modulation of spike-timing dependent synaptic plasticity.** 2007, Neural Computation 19, 1468-1502.

[6] Softky, W. and Koch, C., **Cortical cells should fire regularly, but do not.** 1992, Neural Computation 4, 643–646.

[7] Softky, W. and Koch, C., **The highly irregular firing of cortical cells is inconsistent with temporal integration of random EPSPs.** 1993, Journal of Neuroscience 13, 334–350.

[8] Lapicque, L., **Recherches quantitatives sur lexcitation electrique des nerfs traitee comme une polarisation.** 1907, Journal de Physiologie et Pathologie Generale 9, 620–635.

[9] Stein, R.B., **Some models of neuronal variability.** 1967, Biophysical Journal 7, 37–68.

[10] Bugmann, G., Christodoulou, C. and Taylor, J., **Role of temporal integration and fluctuation detection in the highly irregular firing of a leaky integrator neuron model with partial reset.** 1997, Neural Computation 9, 985–1000.

[11] Christodoulou, C. and Bugmann, G., **Coefficient of variation (CV) vs mean interspike interval (ISI) curves: what do they tell us about the brain.** 2001, Neurocomputing 38- 40, 1141–1149.

[12] Rappoport, A. and Chammah, A. M.. **Prisoner's dilemma**. 1965, University of Michigan Press, Ann Arbor, MI.

[13] Mitchell, T. M., **Machine Learning**. 1997, McGraw-Hill, Boston, MA.

[14] Michael L. L. and Andrew W. M **Reinforcement Learning: A Survey**. 1996, Journal of Artificial Intelligence Research 4, 237-285.

[15] Seung, H. S., **Learning in spiking neural networks by reinforcement of synaptic transmission.** 2003, Neuron 40, 1063-1073.

[16] Izhkevich, E.M., **Solving the distal reward problem through linkage of STDP and dopamine signaling.** 2007, Cereb. Cortex 17, 2443-2452

[17] Farries, M.A. and Fairhall, A.L, **Reinforcment learning with modulated spike-timing-dependent synaptic plasticity.** 2007, J. Neurophysiol. 98, 3648,3665

[18] Legenstein, R.,Pecevski, D., Maass,W., **A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback.** 2008, PLoS Comput. Biol. 4 (10), e10000180.

[19] Xie, X. and Seung, H.S., . **Learning in neural networks by reinforcement of irregular spiking.** 2004, Psychological Review E69, 41909

[20] Markram, H., Lübke, J., Frotscher, M. and Sakmann, B., **Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs.** 1997, Science, 275(5297), 213–215

[21] Hebb, D.O., **The organization of behavior.** 1949,. New York: Wiley & Sons

[22] Dan, Y. and Poo, M.-M., **Hebbian depression of isolated neuromuscular synapses in vitro.** 1992,. Science, 256(5063), 1570–1573.

[23] Bell, C. C., Han, V. Z., Sugawara, Y. and Grant, K., **Synaptic plasticity in a cerebellum-like structure depends on temporal order.** 1997, Nature, 387(6630), 278–281.

[24] Egger,V., Feldmeyer,D. and Sakmann, B., **Coincidence detection and changes of synaptic efficacy in spiny stellate neurons in rat barrel cortex.** 1999, Nature Neuroscience, 2(12), 1098–1105

[25] Roberts, P. D. and Bell, C. C., **Spike timing dependent synaptic plasticity in biological systems.** 2002, Biological Cybernetics, 87(5–6), 392–403.

[26] Hodgkin, A. L. and Huxley, A. F., **A Quantitative Description of Membrane Current and its Application to Conduction and Excitation in Nerve.** 1952, Journal of Physiology, 117: 500-544

[27] Shigematsu Y., Akiyama S. and Matsumoto G. **A spike-firing neural cell (SAM)**, 1992, Extended Abstracts, 4th Int. Symp. on Bioelectronic and Molecular Electronic Devices, Miyazaki (edited by R & D Association for Future Electron Devices, Japan), 13-14.

[28] Rospars J.P. and Lansky P., **Stochastic neuron model without resetting of dendritic potential: application to the olfactory system**. 1993, Biological Cybernetics, 69, 283-294.

[29] Shadlen M. N. and Newsome W. T., **The variable discharge of cortical neurons: Implications for connectivity, computation, and information coding**, 1998, J of Neurosci. 18 3870-3896.

[30] H. C. Tuckwell, **Introduction to theoretical neurobiology: Volume 2: nonlinear and stochastic theories** 1988, (Cambridge University Press, New York), 217-220

[31] http://www.investopedia.com

APPENDIX I

Appendix I includes the source code for the single neuron investigation. This contains the modeling of leaky-integrated-and-fire neuron with total and partial somatic reset. It,also contains the modeling of two compartment leaky-integrated-and- fire model .

Classes description:

forGraph.class contains the methods used for drawing the graphs needed for investigation of a single neuron (i.e. , membrane potential, dendrite potential, output spiketrains, interspike interval distribution, autocorrelogram e.t.c)

generalMethods.class contains the general methods used in the simulation (i.e., method for generate poison spiketrains, method for calculating the C.V or the interspike intervals, method calculate the autocorrelation of an insterspike interval e.t.c)

readFromFile.class contains the method that read the parameters for a .txt file (i.e., input current, membrane time constant, modeling type e.t.c)

leakyIntegrateAndFire.class contains the simulation processing.

Parameter.txt

simulationTime: The simulation duration in ms

Vrest: Rested potential in mV

Vth: Threshold in mV

Vreset: Reset value in mV

Trefr: Refractory period in ms

Tm: Membrane time constant in ms

Rm: Membrane resistance in ms

Ie: Input current to the system in mV

dt: time step in ms

t_interval: time interval in ms

a: Reset parameter for partial reset

reset: the reset mechanism that you want to use(total/partia)

modelL: the modeling type (single-point/two-point)

dTm: dendrite time constant in ms

dRm: dendrite resistance in ms

rC: junctional time constanct in ms

inputSpikeTrainSampleNum: sample number for input spike train

inputSpikeTrainRateStart : starting rate in hz

inputSpikeTrainRateEnd: ending rate in hz

showGeneralGraphs: choose if you want to see the general graphs (yes/no)

showInputOutputGraph: choose if you want to see the input output function graph (yes/no)

showCvGraph: choose if you want to see the C.V graph (yes/no)

showInputSpikeTrain: choose if you want to see the input spiketrain graph (yes/no)

showModelPotential: choose if you want to see the model potential graph (yes/no**)**

ShowSpikeTrain: choose if you want to see the output spiketrain graph (yes/no)

ShowIsi: choose if you want to see the isi distribution graph (yes/no)

ShowMembranPotential: choose if you want to see the membrane potential graph (yes/no)

ShowDedriticPotential: choose if you want to see the dendrite potentila graph (yes/no)

ShowAutocorrelationGraph: choose if you want to see the autocorrelation graph (yes/no)

IsiHistogramBinSize: choose the bin size for interspike interval distribution.

Executing the simulation:

In order to run the simulation, java is needed. With java the only thing to be done is to enter the folder of the coding in command prompt and execute the following command :

Javac *.java

After this set the parameters to be used in the simulation in parameter.txt file and then execute the following command:

Java leakyIntegrateAndFire

It must be mentioned that for the creation of the general graphs needed for this simulation the jFreeChart library is needed. The corresponding library is included in the folder of the corresponding coding.

The source code of the classes described is being shown in the rest of the appendix I.

forgraph.java

```
import java.awt.Image;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartFrame;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;


public class forGraph {

      // This method used to create the general charts for the
simulation
      public  void createGeneralChart (XYSeries data,XYSeries
dendrite,XYSeries thresh,XYSeries spiketrain,XYSeries isi,XYSeries
autocorrelogram,XYSeries autoUp,XYSeries autoDown){


            //Create instance of generalMethods class
            generalMethods meth =new generalMethods();

            //Create instance of leakyIntegrateAndFire class
            leakyIntegrateAndFire lif=new leakyIntegrateAndFire();

            Image image = null;
            Image imageBack = null;

            //Create dataset collection for graph
            XYSeriesCollection dataset = new XYSeriesCollection();
            //Add the threshold data to the dataset collection
            dataset.addSeries(thresh);
            //Add the membrane potential data to the dataset
collection
            dataset.addSeries(data);
            //Add the dendrite potential data to the dataset
collection
            dataset.addSeries(dendrite);

            //Create dataset 2 collection for graph
            XYSeriesCollection dataset2 = new XYSeriesCollection();
            //Add spiketrain data to dataset 2 collection
            dataset2.addSeries(spiketrain);

            //Create dataset 3 collection for graph
            XYSeriesCollection dataset3 = new XYSeriesCollection();
            //Add interspike interval dataset to dataset 3 collection
            dataset3.addSeries(isi);
            //Set width automatically
            dataset3.setAutoWidth(true);

            //Create dataset 4 collection for graph
            XYSeriesCollection dataset4 = new XYSeriesCollection();
```

```
            //Add membrane potential to dataset 4 collection
            dataset4.addSeries(data);

            //Create dataset 5 collection for graph
            XYSeriesCollection dataset5 = new XYSeriesCollection();
            //Add the dendrite potential to dataset 5 collection
            dataset5.addSeries(dendrite);

            //Create dataset 6 collection for graph
            XYSeriesCollection dataset6 = new XYSeriesCollection();
            //Add autocorrelogram data to dataset 6 collection
            dataset6.addSeries(autocorrelogram);
            //Add the Up limit data to dataset 6 collection
            dataset6.addSeries(autoUp);
            //Add the Down limit data to dataset 6 collection
            dataset6.addSeries(autoDown);
            //Set width automatically
            dataset6.setAutoWidth(true);

            //Create frame 1-2-3-4-5-6
            ChartFrame frame1 = null;
            ChartFrame frame2= null;
            ChartFrame frame3= null;
            ChartFrame frame4= null;
            ChartFrame frame5= null;
            ChartFrame frame6= null;

            //In case that the user choose to see the model potential
graph in parameter.txt file
            if(lif.showModelPotential.equalsIgnoreCase("yes")){
                  //Create the chart
                  JFreeChart chart =
ChartFactory.createXYLineChart("Model
Potential","Time(ms)","Voltage(mV)", dataset,
PlotOrientation.VERTICAL,true, true,false);
                  //Set background image
                  chart.setBackgroundImage(imageBack);
                  //set the chart to frame 1
                  frame1=new ChartFrame("Mem Potential",chart);

                  //Create x,y plot
                  XYPlot  plot = chart.getXYPlot();
                  //Set the background image
                  plot.setBackgroundImage(image);
                  //Set the location of frame 1
                  frame1.setLocation(0,0);
                  //Set frame 1 visible
                  frame1.setVisible(true);
                  //Set the size of frame 1
                  frame1.setSize(400,400);
            }

            //In case that the user choose to see the output
spiketrain graph in parameter.txt file
            if(lif.ShowSpikeTrain.equalsIgnoreCase("yes")){
                  //Create the chart
                  JFreeChart chart2 =
ChartFactory.createXYLineChart("Spike Train","Time(ms)","Spikes",
dataset2, PlotOrientation.VERTICAL,true, true,false);
                  //Set background image
                  chart2.setBackgroundImage(imageBack);
```

```
                    //set the chart to frame 2
                    frame2=new ChartFrame("Spike Train",chart2);

                    //Create x,y plot
                    XYPlot  plot2 = chart2.getXYPlot();
                    //Set the background image
                    plot2.setBackgroundImage(image);
                    //Set the location of frame 2
                    frame2.setLocation(400,0);
                    //Set frame 2 visible
                    frame2.setVisible(true);
                    //Set the size of frame 2
                    frame2.setSize(400,400);
              }
              //In case that the user choose to see the interspike
interval distribution graph in parameter.txt file
              if(lif.ShowIsi.equalsIgnoreCase("yes")){
                    //Create the chart
                    JFreeChart chart3 =
ChartFactory.createHistogram("ISI Histogram","Time(ms)","Frequency",
dataset3, PlotOrientation.VERTICAL,true, true,false);
                    //Set background image
                    chart3.setBackgroundImage(imageBack);
                    //set the chart to frame 3
                    frame3=new ChartFrame("isi",chart3);

                    //Create x,y plot
                    XYPlot  plot3 = chart3.getXYPlot();
                    //Set the background image
                    plot3.setBackgroundImage(image);
                    //Set the location of frame 3
                    frame3.setLocation(800,0);
                    //Set frame 3 visible
                    frame3.setVisible(true);
                    //Set the size of frame 3
                    frame3.setSize(400,400);
              }
              //In case that the user choose to see the membrane
potential graph in parameter.txt file
              if(lif.ShowMembranPotential.equalsIgnoreCase("yes")){
                    //Create the chart
                    JFreeChart chart4 =
ChartFactory.createXYLineChart("Membrane
Potential","Time(ms)","Voltage(mV)", dataset4,
PlotOrientation.VERTICAL,true, true,false);
                    //Set background image
                    chart4.setBackgroundImage(imageBack);
                    //set the chart to frame 4
                    frame4=new ChartFrame("Membrane Potential",chart4);

                    //Create x,y plot
                    XYPlot  plot4 = chart4.getXYPlot();
                    //Set the background image
                    plot4.setBackgroundImage(image);
                    //Set the location of frame 4
                    frame4.setLocation(0,400);
                    //Set frame 4 visible
                    frame4.setVisible(true);
                    //Set the size of frame 4
                    frame4.setSize(400,400);
              }
```

```
            //In case that the user choose to see the dedritic
potential graph in parameter.txt file
            if(lif.ShowDedriticPotential.equalsIgnoreCase("yes")){

                  //Create the chart
                  JFreeChart chart5 =
ChartFactory.createXYLineChart("Dendritic
Potential","Time(ms)","Voltage(mV)", dataset5,
PlotOrientation.VERTICAL,true, true,false);
                  //Set background image
                  chart5.setBackgroundImage(imageBack);
                  //set the chart to frame 4
                  frame5=new ChartFrame("Dendritic
Potential",chart5);

                  //Create x,y plot
                  XYPlot  plot5 = chart5.getXYPlot();
                  //Set the background image
                  plot5.setBackgroundImage(image);
                  //Set the location of frame 5
                  frame5.setLocation(400,400);
                  //Set frame 5 visible
                  frame5.setVisible(true);
                  //Set the size of frame 5
                  frame5.setSize(400,400);
            }

            //In case that the user choose to see the Autocorrelagram
in parameter.txt file
            if(lif.ShowAutocorrelationGraph.equalsIgnoreCase("yes")){
                  //Create the chart
                  JFreeChart chart6 =
ChartFactory.createHistogram("Autocorrelogram","Lag","Autocorrelation
", dataset6, PlotOrientation.VERTICAL,true, true,false);
                  //Set background image
                  chart6.setBackgroundImage(imageBack);
                  //set the chart to frame 6
                  frame6=new ChartFrame("Autocorrelogram",chart6);

                  //Create x,y plot
                  XYPlot  plot6 = chart6.getXYPlot();
                  //Set the background image
                  plot6.setBackgroundImage(image);
                  //Set the location of frame 6
                  frame6.setLocation(800,400);
                  //Set frame 6 visible
                  frame6.setVisible(true);
                  //Set the size of frame 6
                  frame6.setSize(400,400);
            }

            //wait for anykey to continue
            meth.getCh();
            //Dispose frames 1-2-3-4-5-6
            if(frame1!=null)
                  frame1.dispose();
            if(frame2!=null)
                  frame2.dispose();
            if(frame3!=null)
                  frame3.dispose();
            if(frame4!=null)
```

```
                frame4.dispose();
            if(frame5!=null)
                frame5.dispose();
            if(frame6!=null)
                frame6.dispose();

            //Clear the dataset 1-2-3-4-5-6
            dataset.removeAllSeries();
            dataset2.removeAllSeries();
            dataset3.removeAllSeries();
            dataset4.removeAllSeries();
            dataset5.removeAllSeries();
            dataset6.removeAllSeries();
            }

      //This method is used for the combine charts like C.V chart
      public  void createCombineChart (XYSeries data,XYSeries
data2,String title,String xAxis,String yAxis,int positionX,int
positionY,String Type){
            //Create instance of generalMethods class
            generalMethods meth =new generalMethods();

            Image image = null;
            Image imageBack = null;

            //Create dataset collection for graph
            XYSeriesCollection dataset = new XYSeriesCollection();
            //Add data to dataset collection
            dataset.addSeries(data);
            //Add data2 to dataset collection
            dataset.addSeries(data2);
            //Set width automatical
            dataset.setAutoWidth(true);
            //In case that the type parameter was "train"
            if(Type.equalsIgnoreCase("train")){
                  //Create the chart
                  JFreeChart chart =
ChartFactory.createXYLineChart(title,xAxis,yAxis, dataset,
PlotOrientation.VERTICAL,true, true,false);
                  //Set background image
                  chart.setBackgroundImage(image);
                  //set the chart to frame
                  ChartFrame frame=new ChartFrame(title,chart);

                  //Create x,y plot
                  XYPlot  plot = chart.getXYPlot();
                  //Set the background image
                  plot.setBackgroundImage(image);
                  //Set the location of frame
                  frame.setLocation(positionX,positionY);
                  //Show frame
                  frame.show();
                  //Set the size of frame
                  frame.setSize(400,400);

                  //wait for anykey to continue
                  meth.getCh();
                  //Dispose frame
                  frame.dispose();
            }//In case that the type parameter was "scat"
            else if(Type.equalsIgnoreCase("scat")){
```

```
                    //Create the chart
                    JFreeChart chart =
ChartFactory.createScatterPlot(title,xAxis,yAxis, dataset,
PlotOrientation.VERTICAL,true, true,false);
                    //Set background image
                    chart.setBackgroundImage(image);
                    //set the chart to frame
                    ChartFrame frame=new ChartFrame(title,chart);

                    //Create x,y plot
                    XYPlot  plot = chart.getXYPlot();
                    //Set the background image
                    plot.setBackgroundImage(image);
                    //Set the location of frame
                    frame.setLocation(positionX,positionY);
                    //Show frame
                    frame.show();
                    //Set the size of frame
                    frame.setSize(400,400);

                    //wait for anykey to continue
                    meth.getCh();
                    //Dispose frame
                    frame.dispose();
              }
        }

        //This method is used for the single charts like input-output
chart
        public void createChart (boolean close,XYSeries data,String
title,String xAxis,String yAxis,int positionX,int positionY,String
Type){

              //Create instance of generalMethods class
              generalMethods meth =new generalMethods();

              Image image = null;
              Image imageBack = null;

              //Create dataset collection for graph
              XYSeriesCollection dataset = new XYSeriesCollection();
              //Add data to dataset collection
              dataset.addSeries(data);
              //Set width automatically
              dataset.setAutoWidth(true);

              //In case that the type parameter was "train"
              if(Type.equalsIgnoreCase("train")){
                    //Create the chart
                    JFreeChart chart =
ChartFactory.createXYLineChart(title,xAxis,yAxis, dataset,
PlotOrientation.VERTICAL,true, true,false);
                    //Set background image
                    chart.setBackgroundImage(image);
                    //set the chart to frame
                    ChartFrame frame=new ChartFrame(title,chart);

                    //Create x,y plot
                    XYPlot  plot = chart.getXYPlot();
                    //Set the background image
                    plot.setBackgroundImage(image);
```

```
                    //Set the location of frame
                    frame.setLocation(positionX,positionY);
                    //Show frame
                    frame.show();
                    //Set the size of frame
                    frame.setSize(400,400);

                    //wait for anykey to continue
                    meth.getCh();
                    //Dispose frame
                    frame.dispose();

              }//In case that the type parameter was "scat"
              else if(Type.equalsIgnoreCase("scat")){
                    //Create the chart
                    JFreeChart chart =
ChartFactory.createScatterPlot(title,xAxis,yAxis, dataset,
PlotOrientation.VERTICAL,true, true,false);
                    //Set background image
                    chart.setBackgroundImage(image);
                    //set the chart to frame
                    ChartFrame frame=new ChartFrame(title,chart);

                    //Create x,y plot
                    XYPlot  plot = chart.getXYPlot();
                    //Set the background image
                    plot.setBackgroundImage(image);
                    //Set the location of frame
                    frame.setLocation(positionX,positionY);
                    //Show frame
                    frame.show();
                    //Set the size of frame
                    frame.setSize(400,400);

                    //wait for anykey to continue
                    meth.getCh();
                    //Dispose frame
                    frame.dispose();
              }
        }

        }
```

generalMethods.java

```
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.ArrayList;
import java.util.Arrays;
import javax.swing.JFrame;
import javax.swing.JRootPane;



public class generalMethods {

	//Generate poison spike train
	public  int generate(double rate,double timestep) {
		double t = -Math.log(Math.random()) / rate; //new ISI in
seconds
		t = t * 1000; //ISI in ms
		t = t / timestep; //ISI in simulation steps
		return (int) Math.round(t); //round to nearest int
	}

	//For poison spike train
	public  double[] generateSeries(int duration, double
rate,double timestep) {
		int timesteps = (int) (duration / timestep);
		double[] series = new double[timesteps];
		Arrays.fill(series, 0);
		int soFar = generate(rate,timestep);
		while (soFar < timesteps) {
			series[soFar] = 1;
			soFar += generate(rate,timestep);
		}
		return series;

	}

	//Method for press anykey to continue
	public  void getCh() {
	  final JFrame frame = new JFrame();
	  synchronized (frame) {
		frame.setUndecorated(true);

frame.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
		frame.addKeyListener(new KeyListener() {
		    public void keyPressed(KeyEvent e) {
			synchronized (frame) {
			    frame.setVisible(false);
			    frame.dispose();
			    frame.notify();
			}
		    }

		    public void keyReleased(KeyEvent e) {
		    }

		    public void keyTyped(KeyEvent e) {
		    }
		});
```

```
            frame.setVisible(true);
            try {
                frame.wait();
            } catch (InterruptedException e1) {
            }
        }
    }

      //Method for calculate the Coefficient of variation
      public  double coefficientOfVariation(ArrayList<Double> isi){
            double cv=0;
            double standardDeviacion=0;
            double sum=0;
            double sum2=0;
            double average=0;

            for(int i=0; i<isi.size(); i++){
                  sum=sum+isi.get(i);
            }

            average=sum/isi.size();

            for(int i=0; i<isi.size(); i++){
                  sum2=sum2+Math.pow((isi.get(i)-average),2);
            }

            standardDeviacion=Math.sqrt(sum2/(isi.size()));

            cv=standardDeviacion/average;



            return cv;
      }

      //Method for creation of interspike interval dataset
      public  ArrayList<Double> interspikeInterval(int[]
spiketrain,double timestep){
            ArrayList<Double> isi = new ArrayList<Double>();
            double countmSec=0;
            for (int i=0; i<spiketrain.length; i++){
                  if (spiketrain[i]==1){
                        isi.add(countmSec);
                        countmSec=0;
                  }
                  countmSec=countmSec+timestep*1;
            }
            return isi;
      }

      //Method for creation of interspike interval distribution
      public  int[] interspikeIntervalHistogram(ArrayList<Double>
isi,double binSize){
            int[] count = new int[80];
            double range=0;
            double lastposition=0;

            for(int j=0; j<isi.size(); j++){
                  range=0;
                  lastposition=0;
                  for(int i=0; i<80; i++){
```

```
                        range=lastposition;
                        if(range<=isi.get(j)&&
isi.get(j)<=range+binSize)
                              count[i]=count[i]+1;
                        lastposition=range+binSize;
                  }
            }
            return count;
      }

      //Method for creation of input spiketrain
      public double[] createInputSpikeTrain(int count,double rate,
int duration,double timestep){
      double[] inputSpike = new double[(int)
Math.rint(duration/timestep)];
      double[][] spike = new double[count][];

      for(int i=0; i<count; i++){
            spike[i]=generateSeries(duration, rate,timestep);
      }

      for (int i=0; i<(int) Math.rint(duration/timestep); i++){
            for(int j=0; j<count; j++){
                  inputSpike[i]+=spike[j][i];
            }
      }


      return inputSpike;
      }

      //Method for creation autocorrelogram
      public double[] autoCorrelation(int size,int lag,double[] isi){
          double[] R = new double [size+1];
          float sum=0;
          double average;

          for(int i=0; i<isi.length; i++){
                  sum=(float) (sum+isi[i]);
            }

            average=sum/isi.length;



          for (int i=0;i<lag;i++) {

            sum=0;
              for (int j=0;j<size-i;j++) {

                  sum+=(isi[j]-average)*(isi[j+i]-average);
              }
              R[i]=sum/(size-i);

          }
          return R;
      }
      }
```

readFromFile.java

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class readFromFile {

	//Method for read the parameters of the simulation from
parameter.txt file
	public void readFromFile(String filename) {
		BufferedReader in = null;

		try {
			in = new BufferedReader(new FileReader(filename));
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		}
		String line = "";
		String[] temp;


		try {
			line = in.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
		temp = line.split(" ");

	leakyIntegrateAndFire.simulationTime=Double.parseDouble(temp[1]
);//Read simulationTime

		try {
			line = in.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
		temp = line.split(" ");

	leakyIntegrateAndFire.Vrest=Integer.parseInt(temp[1]);//Read
Vrest

		try {
			line = in.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
		temp = line.split(" ");

	leakyIntegrateAndFire.Vth=Integer.parseInt(temp[1]);//Read Vth

		try {
			line = in.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
```

```
            temp = line.split(" ");

      leakyIntegrateAndFire.Vreset=Integer.parseInt(temp[1]);//Read
Vreset

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.Trefr=Double.parseDouble(temp[1]);//Read
Trefr

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.Tm=Double.parseDouble(temp[1]);//Read Tm

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.Rm=Double.parseDouble(temp[1]);
//Read Rm

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.Ie=Double.parseDouble(temp[1]);//Read Ie

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.dt=Double.parseDouble(temp[1]);//Read dt

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.t_interval=Double.parseDouble(temp[1]);//
Read t_interval
```

```
            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.a=Double.parseDouble(temp[1]);//Read a

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.restetType=temp[1];//Read
restetType

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.modelType=temp[1];//Read modelType


            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.dTm=Double.parseDouble(temp[1]);//Read
dTm

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.dRm=Double.parseDouble(temp[1]);//Read
dRm

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.rC=Double.parseDouble(temp[1]);//Read rC


            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
```

```
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.inputSpikeTrainSampleNum=Integer.parseInt
(temp[1]);//Read inputSpikeTrainSampleNum

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.inputSpikeTrainRateStart=Double.parseDoub
le(temp[1]);//Read inputSpikeTrainRateStart

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.inputSpikeTrainRateEnd=Double.parseDouble
(temp[1]);//Read inputSpikeTrainRateEnd

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.showGeneralGraphs=temp[1];//Read
showGeneralGraphs

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.showInputOutputGraph=temp[1];//Read
showInputOutputGraph

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.showCvGraph=temp[1];//Read
showCvGraph

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.showInputSpikeTrain=temp[1];//Read
showInputSpikeTrain
```

```
            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.showModelPotential=temp[1];//Read
showModelPotential

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.ShowSpikeTrain=temp[1];//Read
ShowSpikeTrain

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.ShowIsi=temp[1];//Read ShowIsi

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            leakyIntegrateAndFire.ShowMembranPotential=temp[1];//Read
ShowMembranPotential

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.ShowDedriticPotential=temp[1];//Read
ShowDedriticPotential

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      leakyIntegrateAndFire.ShowAutocorrelationGraph=temp[1];//Read
ShowAutocorrelationGraph
            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
```

```
		leakyIntegrateAndFire.binSize=Double.parseDouble(temp[1]);//Rea
d binSize
		}
}
```

leakyIntegrateAndFire.java

```
import org.jfree.data.xy.XYSeries;
import java.util.ArrayList;


public class leakyIntegrateAndFire {

	//Parameters for the simulation
	static double simulationTime = 0; //Simulation time
	static double Vrest = 0;// Resting potential
	static double Vth = 0; // Threshold
	static double Vreset = 0; //reset potential
	static double Trefr = 0;// refractory periot
	static double Tm = 0; // Membrane time constant
	static double Rm = 0; // Mebrane resistance
	static double dTm = 0; //Dendrite time constant
	static double dRm = 0; //Dendrite resistance
	static double rC=0; //Junctional time constant
	static double Ie = 0; // Input current
	static double dt = 0; // time step
	static double a=0; // reset parameter
	static double t_interval = 0; // time interval
	static double binSize=0; //bin size for interspike interval
distribution
	static String restetType=""; // Reset type
	static String modelType=""; // model type (two point or singe
point)
	static int inputSpikeTrainSampleNum=0; //input spiketrain
sample number
	static double inputSpikeTrainRateStart=0; //starting spiketrain
rate
	static double inputSpikeTrainRateEnd=0; //ending spiketrain
rate
	static String showGeneralGraphs=""; // Yes or No for showing
general graphs
	static String showInputOutputGraph="";// Yes or No for showing
input output transfer
	static String showCvGraph="";// Yes or No for showing CV graph
	static String showInputSpikeTrain="";// Yes or No for showing
input spiketrain
	static String showModelPotential="";// Yes or No for showing
model potential
	static String ShowSpikeTrain="";// Yes or No for showing output
spiketrain
	static String ShowIsi="";// Yes or No for showing interspike
interval distribution
	static String ShowMembranPotential="";// Yes or No for showing
mebrane potential
	static String ShowDedriticPotential="";// Yes or No for showing
dedrite potential
	static String ShowAutocorrelationGraph="";// Yes or No for
showing autocorellogram

	public static void main(String[] args){

		//instance to generalMethod class
```

```
            generalMethods meth =new generalMethods();
            //instance to forGraph class
            forGraph graph = new forGraph();
            //instance to readFromFile class
            readFromFile rf = new readFromFile();

            //give the filename for parameters and read them
            rf.readFromFile("parameters.txt");

            double t=t_interval;
            double refractory=0;
            double cv=0;
            double sumForAverage=0.0;//For find the average intespike
interval
            double average=0.0;//For find the average intespike
interval
            double LIFcount = 0;//Global time for count the intervals
of the whole simulation
            int count=0; //For count the number of interval in order
to puts spikes in spiketrain array.
            double input;//For count the number of interval in order
to check if there is a spike or not based on the input spike train
            int Vcount=1;//For count the number of interval in order
to record the membrane potential
            double[] spikes = new double[(int)
(Math.rint(simulationTime/t_interval))];//Array for input spike train
            int[] spikeTrain= new int[(int)
((Math.rint(simulationTime/t_interval))+1)];//Array for spike train
            double[] V = new double[(int)
((Math.rint(simulationTime/t_interval))+1)];//Array for membrane
potential
            double[] dV = new double[(int)
((Math.rint(simulationTime/t_interval))+1)];//Array for dendrite
potential
            V[0]=0;//Initialize membrance potential to "0"
            dV[0]=0;//Initialize dendrite potential to "0"

            //Create the arraylist for graphs
            ArrayList<Double> Vmem = new ArrayList<Double>();
            ArrayList<Double> Vdend = new ArrayList<Double>();
            ArrayList<Double> Time = new ArrayList<Double>();
            ArrayList<Double> Vthres = new ArrayList<Double>();
            ArrayList<Double> Spikes = new ArrayList<Double>();
            ArrayList<Double> interspikeInterval = new
ArrayList<Double>();
            ArrayList<Integer> isiHistogramIntervals = new
ArrayList<Integer>();
            ArrayList<Double> isiHistogramTime = new
ArrayList<Double>();
            ArrayList<Double> coefficientOfVariation = new
ArrayList<Double>();
            ArrayList<Double> meanISI = new ArrayList<Double>();
            ArrayList<Double> inputSpike = new ArrayList<Double>();
            ArrayList<Double> UpLimit = new ArrayList<Double>();
            ArrayList<Double> DownLimit = new ArrayList<Double>();
            ArrayList<Double> inputRate = new ArrayList<Double>();
            ArrayList<Double> outputRate = new ArrayList<Double>();


            //Create the series for graphs
            XYSeries potential = new XYSeries("membrane Potential");
```

```
            XYSeries dendritePotential = new XYSeries("dendritic
Potential");
            XYSeries Threshold = new XYSeries("Vth");
            XYSeries spikeTrains = new XYSeries("Spike");
            XYSeries isiGraph = new XYSeries("ISI histogram");
            XYSeries CV = new XYSeries("CV over Mean ISI");
            XYSeries theoreticalCV=new XYSeries("Theoretical Line");
            XYSeries inputSpikeTrain = new XYSeries("Input Spike
Train");
            XYSeries autocorrelogram = new
XYSeries("Autocorrelogram");
            XYSeries autoUpLimit= new XYSeries("Uper Limit");
            XYSeries autoDownLimit= new XYSeries("Down Limit");
            XYSeries inputOutputFun=new XYSeries("Input - Output
Rate");

            //Initial the output spiketrian to 0's
            for(int i=1; i<(int) ((simulationTime/t_interval)); i++){
                  spikeTrain[i]=0;
            }

            //End spike train

            //Set starting rate
            double rate=inputSpikeTrainRateStart;

            //Start Simulation
            while(LIFcount<(inputSpikeTrainRateEnd-
inputSpikeTrainRateStart)){
                  //For spike count
                  int spikeCount=0;

                  //output rate of the simulation
                  double outputRateCal=0;

                  //Create the inputspiketrain
                  spikes =
meth.createInputSpikeTrain(inputSpikeTrainSampleNum,rate,(int)simulat
ionTime,dt);

                  //Add input spiketrain to arraylist for input
spiketrain graph
                  for(int i=0; i<spikes.length; i++){
                        inputSpike.add(spikes[i]);

                  }

                  //Create the x,y series for input spiketrain graph
                  for (int x=0; x<inputSpike.size(); x++){
                        inputSpikeTrain.add(x,inputSpike.get(x));

                  }

                  //Check if the user wants to see the input
spiketrain graph
                  if(showInputSpikeTrain.equalsIgnoreCase("yes"))
                        //Show input spiketrain graph
                        graph.createChart(true,inputSpikeTrain,
"Input Spike Train","Sim(Time)","Spike", 0, 0,"train");

                  //initial the refractory time
```

```
                  refractory=2;

                  //start simulation
                  while(t<(simulationTime)){


                        //Creae the input to the system
                        if(spikes[count]>0){
                              input=(Ie/dt)*spikes[count];
                        }else{
                              input=0;
                        }


                        //add time interval for refractory periot
                        refractory+=t_interval;


                        //If the model type is single point
                        if(modelType.equalsIgnoreCase("single-
point")){
                              //Calculate the potential of membrane
                              V[Vcount]=V[Vcount-1] + ((-V[Vcount-
1]+Rm*input)/Tm)*dt;//membrane

                        }
                        //In the model type is two point
                        else if(modelType.equalsIgnoreCase("two-
point")){
                              //Calculate the dedrite potential
                              dV[Vcount]=dV[Vcount-1]+((-dV[Vcount-
1]/dTm)+((V[Vcount-1]-dV[Vcount-1])/rC)+input)*dt;
                              //Calculate the membrane potential
                              V[Vcount]=V[Vcount-1]+((-V[Vcount-
1]/Tm)+((dV[Vcount]-V[Vcount-1])/rC))*dt;

                        }

                        //Check if the membrane potential pass the
threshold
                        if(V[Vcount]>=Vth && refractory>=Trefr){
                              spikeTrain[Vcount]=1;//Add spike to the
spiketrain
                              V[Vcount]=0;//Simulate the spike for
next if
                              refractory=0;//set refractory to 0
                              spikeCount++;//count spike
                        }

                        //in case of spike
                        if(V[Vcount]==0)
                              //if the reset used is total reset

      if(restetType.equalsIgnoreCase("total"))
                                    V[Vcount]=Vreset;//Reset the
potential to "0" after the spike
                              //if the reset used is partia reset
                              else
if(restetType.equalsIgnoreCase("partial"))
                                    V[Vcount]=Vth*a;//Partial reset
```

```
                        t+=t_interval;//count interval
                        Vmem.add(V[Vcount]);//Add the current
potential for the graph
                        Vdend.add(dV[Vcount]);//Add the current
potential of dendrite for graph
                        Time.add(t);//Add the current global
simulation time for graph


                        count++;//count for input spike train
                        Vcount++;//count interval for membrane
potential

                  }
                  //get the interspike intervals for the output
spiketrain
                  interspikeInterval =
meth.interspikeInterval(spikeTrain,t_interval);//get the interspike
interval

                  //Calculate the C.V
                  if(interspikeInterval.size()==0){
                        cv=0;
                        average=0;
                  }
                  else{

      cv=meth.coefficientOfVariation(interspikeInterval);


                        sumForAverage=0;

                        for(int i=0; i<interspikeInterval.size();
i++){

      sumForAverage=sumForAverage+interspikeInterval.get(i);
                        }


      average=sumForAverage/interspikeInterval.size();
                  }


                  double[] isiForACF=new
double[interspikeInterval.size()];//array for interspike intervals
for autocorrelation

                  //Create the autocorrelation graph
                  for(int i=0; i<interspikeInterval.size(); i++){

      isiForACF[i]=Float.parseFloat(interspikeInterval.get(i).toStrin
g());
                  }

                  if(isiForACF.length>=20){

      isiForACF=meth.autoCorrelation(isiForACF.length,20, isiForACF);
                  }
                  else{
```

```
      isiForACF=meth.autoCorrelation(isiForACF.length,isiForACF.lengt
h, isiForACF);
                  }

                  double max=isiForACF[0];
                  for(int i=0; i<isiForACF.length; i++){
                        if(max<isiForACF[i])
                              max=isiForACF[i];
                  }
                  for(int i=0; i<isiForACF.length; i++){
                        isiForACF[i]=isiForACF[i]/max;
                  }

                  double
Uplimit=(1.96/Math.sqrt(interspikeInterval.size()));
                  double Downlimit=(-
1.96/Math.sqrt(interspikeInterval.size()));


                  if(isiForACF.length>=20){
                        for(int i=0; i<20; i++){
                              UpLimit.add(Uplimit);
                              DownLimit.add(Downlimit);
                        }

                        for(int i=0; i<20; i++)
                        {
                              autocorrelogram.add(i,isiForACF[i]);
                              autoUpLimit.add(i,UpLimit.get(i));
                              autoDownLimit.add(i,DownLimit.get(i));
                        }
                  }
                  else
                  {
                        for(int i=0; i<isiForACF.length; i++){
                              UpLimit.add(Uplimit);
                              DownLimit.add(Downlimit);
                        }

                        for(int i=0; i<isiForACF.length; i++)
                        {
                              autocorrelogram.add(i,isiForACF[i]);
                              autoUpLimit.add(i,UpLimit.get(i));
                              autoDownLimit.add(i,DownLimit.get(i));
                        }
                  }
                  //End autocorrelation graph creation

                  //Calculate the output firing rate
            outputRateCal=(spikeCount/simulationTime)*1000;


            //Print the C.V
                  System.out.print("\n " +"CV:"+ cv +" ");

                  if(cv!=0){
                        //Add C.V to array for graph
                        coefficientOfVariation.add(cv);
                        //Pring the mean ISIS
```

```
                        System.out.print("\n " +"ISI mean:"+
average+" ");
                        System.out.print("\n ");
                        //Add the mean ISI to array for graph
                        meanISI.add(average);
                  }
                  //Print the input and output firing rates
                  System.out.print(" Input rate : "+rate+ " Output
rate: " + outputRateCal);
                  LIFcount=LIFcount+1;
                  Vcount=1;
                  t=t_interval;
                  count=0;
                  rate+=1;
                  //Add input rate to array for graph
                  inputRate.add(rate);
                  //Add output rate to array for graph
                  outputRate.add(outputRateCal);
                  //Add input and output rate to x,y plot for graph
                  inputOutputFun.add(rate, outputRateCal);

                  //Create isisHistogram graph
                  int[] isiHist = new int[80];

      isiHist=meth.interspikeIntervalHistogram(interspikeInterval,bin
Size);
                  double time = 0;
                  double lastOne=0.002;
                  for(int i=0; i<isiHist.length; i++){

                        time=lastOne;
                        isiHistogramIntervals.add(isiHist[i]);
                        isiHistogramTime.add(time);
                        lastOne=lastOne+binSize;

                  }
                  //end isisHistogram graph


                  //Set threshold for graph
                  for(int x=0; x<Time.size(); x++){
                        Vthres.add(Vth);
                  }

                  //create spike train for graph
                  for(int x=0; x<Time.size(); x++){
                        if(spikeTrain[x]==1)
                              Spikes.add((double) 1);
                        else
                              Spikes.add((double) 0);
                  }
                  //End



                  //Set datasets  for general graph's
                  for (int x=0; x<Time.size(); x++){

      dendritePotential.add(Time.get(x),Vdend.get(x));//Dedrite
potential
```

```
      potential.add(Time.get(x),Vmem.get(x));//membrane potential

      Threshold.add(Time.get(x),Vthres.get(x));//threshold line

      spikeTrains.add(Time.get(x),Spikes.get(x));//output spiketrains
                  }

                  //Set dataset for general graphs
                  for(int x=0; x<isiHistogramTime.size(); x++){

      isiGraph.add(isiHistogramTime.get(x),isiHistogramIntervals.get(
x));//interspike interval distribution
                  }


                  //show the general graphs in case that the
corresponing parameter in parameter.txt file is yes
                  if(showGeneralGraphs.equalsIgnoreCase("yes")){
                        meth.getCh();

      graph.createGeneralChart(potential,dendritePotential,Threshold,
spikeTrains,isiGraph,autocorrelogram,autoUpLimit,autoDownLimit);
                  }

                  //Clear the arrays
                  potential.clear();
                  Threshold.clear();
                  dendritePotential.clear();
                  spikeTrains.clear();
                  isiHistogramTime.clear();
                  isiHistogramIntervals.clear();
                  autocorrelogram.clear();
                  autoUpLimit.clear();
                  autoDownLimit.clear();
                  inputSpikeTrain.clear();
                  inputSpike.clear();
                  interspikeInterval.clear();

                  //Clear the graphs datasets
                  isiGraph.clear();
                  Spikes.clear();
                  Vthres.clear();
                  Vmem.clear();
                  Vdend.clear();
                  Time.clear();
                  UpLimit.clear();
                  DownLimit.clear();


                  //Set the input spike train to 0's again
                  for(int i=1; i<spikeTrain.length; i++){
                        spikeTrain[i]=0;
                  }
                  //End
            }

            //create cv over meanIsi  graph
            double theoC=0;
            for (int x=0; x<meanISI.size(); x++){
```

```
      CV.add(meanISI.get(x),coefficientOfVariation.get(x));
            }
            for(float mISI=2; mISI<=25; mISI+=0.1){
                  theoC=Math.sqrt((mISI-Trefr)/mISI);
                  theoreticalCV.add(mISI,theoC);
            }
            //show the input output function graph in case that the
corresponing parameter in parameter.txt file is yes
        if(showInputOutputGraph.equalsIgnoreCase("yes"))
            graph.createChart(true,inputOutputFun, "Input-Output",
"Input", "Output", 0, 0, "scat");
        //show the C.V graph in case that the corresponing parameter
in parameter.txt file is yes
        if(showCvGraph.equalsIgnoreCase("yes"))
                  graph.createCombineChart(CV,theoreticalCV,"CV over
mean ISI","mean ISI (ms) ","CV(T)",0,0,"scat");
            System.out.print("\n Simulation End");
      }

}
```

APPENDIX II

Appendix II includes the source code for neural network investigation. This contains the neural network of single point leaky integrate and fire with total somatic reset model, the neural network consisted of single point leaky integrate and fire with partial somatic reset model and the neural network consisted of two point leaky integrate and fire model.

Classes description:

forGraph.class contains the methods that are being used for drawing the graphs needed

usefullMethods.class contains the general methods that are being used in the simulation

getParameters.class contains the method that read the parameters for a .txt file (i.e., input current, membrane time constant, modeling type e.t.c)

inputNeuron.class I the object class for the inputNeurons

hiddenNeuron.class I the object class for the hiddenNeurons

outputNeuron.class I the object class for the outputNeurons

Simulation.class contains the simulation processing.

Parameter.txt

durationOfSimulation: Simulation duration in ms

timeStep : time step in ms

inputLayerNumberOfNeurons: number of input layer neurons

hiddenLayerNumberOfNeurons : number of hidden layer neurons

outputLayerNumberOfNeurons : number of output layer neurons

rate: rate in hz for input spiketrains

dRm: dendrite resistance in ms

dTm: dendrite time constant in ms

Rm: membrane resistant in ms

Tm: membrane time constant in ms

Vth: threshold in mV

a: reset parameter

Trefr : refractory period in ms

Vreset: reset value in mV

tPlus: P+ time constant in ms for STDP

tMinus: P- time constant in ms fot STDP

aPlus: constant A+ for STDP

aMinus: constant A- for STDP

Tz: eligibility trace time constant for STDP

b: discount factor of eligibility trace for STDP

rC: junctional time constant

gama: learning rate

percentOfInhibitory : % for inhibitory synapse

model: modeling type(single-point/two-point)

reset : reset mechanism (partial/total)

bounds: weight bounds for STDP

Executing the simulation:

In order to run the simulation, java is needed. With java the only thing to be done is to enter the folder of the coding in command prompt and execute the following command :

Javac *.java

After this set the parameters to be used in the simulation in parameter.txt file and then execute the following command:

Java Simulation

It must be mentioned that for the creation of the general graphs needed for this simulation the jFreeChart library is needed. The corresponding library is included in the folder of the corresponding coding.

The source code of the classes described is being shown in the rest of the appendix II.

forGraph.java

```
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartFrame;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;

public class forGraph {


      //For creation of average firing rate graphs
      public void createChart (double value00,double value01,double
value10,double value11,String title,String xAxis,String yAxis,int
positionX,int positionY){
            //Create the category dataset
            DefaultCategoryDataset  dataset = new
DefaultCategoryDataset ();

            //Pass the values to the dataset
            dataset.setValue(value00, "rate", "{0,0}");
            dataset.setValue(value01, "rate", "{0,1}");
            dataset.setValue(value10, "rate", "{1,0}");
            dataset.setValue(value11, "rate", "{1,1}");

            //Create the frame
            ChartFrame frame1 = null;
            //Create the chart
            JFreeChart chart =
ChartFactory.createBarChart(title,xAxis, yAxis,dataset,
PlotOrientation.VERTICAL, false, true, false);

            //pass the chart to frame
            frame1=new ChartFrame("Average firing rate",chart);

            //Set frame location
            frame1.setLocation(0,0);
            //Set visible
            frame1.setVisible(true);
            //Set frame size
            frame1.setSize(400,400);
      }
}
```

getParameters.java

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class getParameters {

	//Method to read the parameters
	public void readFromFile(String filename) {

		BufferedReader in = null;

		try {
			in = new BufferedReader(new FileReader(filename));
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		}
		String line = "";
		String[] temp;


		try {
			line = in.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
		temp = line.split(" ");

	Simulation.durationOfSimulation=Integer.parseInt(temp[1]);//Rea
d simulation time

		try {
			line = in.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
		temp = line.split(" ");
		Simulation.timeStep=Double.parseDouble(temp[1]);//Read
time step

		try {
			line = in.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
		temp = line.split(" ");

	Simulation.inputLayerNumberOfNeurons=Integer.parseInt(temp[1]);
//Read number of input layer neurons

		try {
			line = in.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
```

```
            temp = line.split(" ");

      Simulation.hiddenLayerNumberOfNeurons=Integer.parseInt(temp[1])
;//Read number of hidden layer neurons

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      Simulation.outputLayerNumberOfNeurons=Integer.parseInt(temp[1])
;//Read number of output layer neurons

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.rate=Integer.parseInt(temp[1]);//Read rate for
the input spiketrains generation

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.dRm=Double.parseDouble(temp[1]);//Read dedrite
resistance

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.dTm=Double.parseDouble(temp[1]);//Read dedrite
time constant

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.Rm=Double.parseDouble(temp[1]);//Read membrane
resistance

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.Tm=Double.parseDouble(temp[1]);//Read
membrane time constant
```

```
            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.Vth=Double.parseDouble(temp[1]);//Read
threshold

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.a=Double.parseDouble(temp[1]);//Read reset
parameter


            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.Trefr=Double.parseDouble(temp[1]);//Read
refractory

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.Vreset=Double.parseDouble(temp[1]);//Read
reset value

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.tPlus=Double.parseDouble(temp[1]);//Read time
constant for P+ (STDP)

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.tMinus=Double.parseDouble(temp[1]);//Read time
constant for P- (STDP)

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
```

```
            Simulation.aPlus=Integer.parseInt(temp[1]);//Read
constant number (A+ STDP)

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.aMinus=Integer.parseInt(temp[1]);//Read
constant number (A- STDP)

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.Tz=Integer.parseInt(temp[1]);//Read time
constant for eligibility trace (STDP)


            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.b=Double.parseDouble(temp[1]);//Read discount
factor (STDP)

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.rC=Double.parseDouble(temp[1]);//Read
junctional time constant

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.gama=Double.parseDouble(temp[1]);//Read
learning rate

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");

      Simulation.percentOfInhibitory=Integer.parseInt(temp[1]);//Read
the % of inhibitory neurons

            try {
                  line = in.readLine();
```

```
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.model=temp[1];//Read model type (single point
or two point)

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.reset=temp[1];//Read reset type

            try {
                  line = in.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            temp = line.split(" ");
            Simulation.bound=Double.parseDouble(temp[1]);//Read weith
bounds (STDP)


      }
}
```

hiddenNeuron.java

```
public class hiddenNeuron {
	public double[] weights; // the weight of the neuron for each
timeStep
	public int spikeTrain; // The output Spike train of the neuron
in each timeStep
	public double dV; // The Dendrite potential  of the neuron in
each timeStep
	public double V; // The Membrane potential  of the neuron in
each timeStep
	public double input; // The input that the neuron takes for
each timeStep
	public double refractory; // For calculating the refractory
period of the neuron
	public double[] PijPlus;  //STDP Pij+
	public double[]PijMinus; //STDP Pij-
	public double[] zita;//STDP ζij;
	public double[] z;//STDP z

	//Constractor
	public hiddenNeuron(int fromNum,int toNum,int numberOfNeurons){
		this.weights=new double[toNum];
		this.spikeTrain=0;
		this.dV=0;
		this.V=0;
		this.input=0;
		this.refractory=0;
		this.PijPlus=new double[numberOfNeurons];
		this.PijMinus=new double[numberOfNeurons];
		this.zita=new double[numberOfNeurons];
		this.z=new double[numberOfNeurons];
	}
}
```

inputNeuron.java

```
public class inputNeuron {
	public double[] weights; // the weight of the neuron for each
timeStep
	public int spikeTrain; // The output Spike train of the neuron
in each timeStep
	public double[] PijPlus;  //STDP Pij+
	public double[] PijMinus; //STDP Pij-
	public double[]zita;//STDP ζij;
	public double[] z;//STDP z

	//Constractor
	public inputNeuron(int fromNum,int toNum,int numberOfNeurons){
		this.weights=new double[toNum];
		this.spikeTrain=0;
		this.PijPlus=new double[numberOfNeurons];
		this.PijMinus=new double[numberOfNeurons];
		this.zita=new double[numberOfNeurons];
		this.z=new double[numberOfNeurons];
	}
}
```

outputNeuron.java

```
public class outputNeuron {
	public int spikeTrain; // The output Spike train of the neuron
in each timeStep
	public double dV; // The Dendrite potential  of the neuron in
each timeStep
	public double V; // The Membrane potential  of the neuron in
each timeStep
	public double input; // The input that the neuron takes for
each timeStep
	public double refractory; // For calculating the refractory
period of the neuron

	//Constractor
	public outputNeuron(int fromNum,int toNum,int numberOfNeurons){
		this.spikeTrain=0;
		this.dV=0;
		this.V=0;
		this.input=0;
		this.refractory=0;
	}
}
```

usefullMethods.java

```
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.Arrays;
import java.util.Random;
import javax.swing.JFrame;
import javax.swing.JRootPane;


public class usefullMethods {

	//For poison spiketrain generations
	public  int generate(double rate,double timestep) {
		double t = -Math.log(Math.random()) / rate; //new ISI in
seconds
		t = t * 1000; //ISI in ms
		t = t / timestep; //ISI in simulation steps
		return (int) Math.round(t); //round to nearest int
	}

	//For poison spiketrain generation
	public  int[] generateSeries(int duration, double rate,double
timestep) {
		int timesteps = (int) (duration / timestep);
		int[] series = new int[timesteps];
		Arrays.fill(series, 0);
		int soFar = generate(rate,timestep);
		while (soFar < timesteps) {
			series[soFar] = 1;
			soFar += generate(rate,timestep);
		}
		return series;

	}
	//For generate 0's spike train (no spikes)
	public  int[] generateZeroSeries(int duration,double timestep)
{
		int timesteps = (int) (duration / timestep);
		int[] series = new int[timesteps];
		Arrays.fill(series, 0);
		return series;

	}

	//Create the input spiketrain
	public int[][] createInputSpikeTrain(int count,int rate, int
duration,double timestep,int firstBinary,int secondBinary){
		int[][] spike = new int[count][];

		for(int i=0; i<count; i++){
			spike[i]=generateZeroSeries(duration,timestep);
		}

		if(firstBinary==1 && secondBinary==1){
			for(int i=0; i<count; i++){
```

```
                        spike[i]=generateSeries(duration,
rate,timestep);
                  }
            }
            else if(firstBinary==0 && secondBinary==0){
                  for(int i=0; i<count; i++){

      spike[i]=generateZeroSeries(duration,timestep);
                  }

            }
            else if(firstBinary==1 && secondBinary==0){
                  for(int i=0; i<count/2; i++){
                        spike[i]=generateSeries(duration,
rate,timestep);
                  }
                  for(int i=count/2; i<count; i++){

      spike[i]=generateZeroSeries(duration,timestep);
                  }
            }
            else if(firstBinary==0 && secondBinary==1){
                  for(int i=0; i<count/2; i++){

      spike[i]=generateZeroSeries(duration,timestep);
                  }
                  for(int i=count/2; i<count; i++){
                        spike[i]=generateSeries(duration,
rate,timestep);
                  }

            }
            return spike;
      }

      //Wait for any key to continue
      public  void getCh() {
        final JFrame frame = new JFrame();
        synchronized (frame) {
            frame.setUndecorated(true);

frame.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
            frame.addKeyListener(new KeyListener() {
                public void keyPressed(KeyEvent e) {
                    synchronized (frame) {
                        frame.setVisible(false);
                        frame.dispose();
                        frame.notify();
                    }
                }

                public void keyReleased(KeyEvent e) {
                }

                public void keyTyped(KeyEvent e) {
                }
            });
            frame.setVisible(true);
            try {
                frame.wait();
            } catch (InterruptedException e1) {
```

```
            }
        }
    }

    //For getting random numbers
    public double nextDouble(Random r, int lower, int higher) {
        int ran = r.nextInt();
        double x = (double)ran/Integer.MAX_VALUE * higher;
        return x + lower;
     }

    //swap
    private static void swap(int[] a, int i, int change) {
          int helper = a[i];
          a[i] = a[change];
          a[change] = helper;
    }

    //Shuffle array method
    public static void shuffleArray(int[] a) {
          int n = a.length;
          Random random = new Random();
          random.nextInt();
          for (int i = 0; i < n; i++) {
                int change = i + random.nextInt(n - i);
                swap(a, i, change);
          }
    }

    //Swap neurons
    private static void swapInputNeurons(inputNeuron[] a, int i,
int change) {
          inputNeuron helper = a[i];
          a[i] = a[change];
          a[change] = helper;
    }

    //Shuffle neurons
    public static void shuffleInputNeurons(inputNeuron[] a) {
          int n = a.length;
          Random random = new Random();
          random.nextInt();
          for (int i = 0; i < n; i++) {
                int change = i + random.nextInt(n - i);
                swapInputNeurons(a, i, change);
          }
    }

}
```

Simulation.java

```
public class Simulation {

	static int durationOfSimulation=0; //Simulation time
	static double timeStep=0; //Time step
	static int inputLayerNumberOfNeurons=0; // number of input
layer neurons
	static int hiddenLayerNumberOfNeurons=0; // number of hidden
layer neurons
	static int outputLayerNumberOfNeurons=0; // number of output
layer neurons
	static int rate=0; // rate
	static double Ie=0; // Input to the system
	static double dRm=0; // dedrite resistance
	static double dTm=0; // dedrite timeconstant
	static double Rm=0; // membrane resistance
	static double Tm=0; // membrane timeconstant
	static double Vth=0; // Threshold
	static double Trefr=0; // refractory period
	static double Vreset=0; // reset value
	static double tPlus=0; // time constant of P+ (STDP)
	static double tMinus=0; // time constant of P- (STDP)
	static int aPlus=0; // constant value (A+ STDP)
	static int aMinus=0; // constant value (A- STDP)
	static int Tz=0;// eligibility trace time constant
	static double b=0; // discount variable
	static double gama=0; // learning rate
	static int r=0; // reward signal
	static double a=0; // reset parameters
	static double rC=0; // junctional time constant
	static int foundSpike=0; //Count spikes
	static int percentOfInhibitory=0; //% of inhibitory neurons
	static String model=""; // Model type (singe or two point)
	static String reset=""; // reset type (partial or total)
	static double bound=0; // weight bounds (STDP)

	public static void main(String[] args){

		//Use get parameters to pass the values to simulation
parameters
		getParameters params = new getParameters();
		//instance of usefullmethod class
		usefullMethods meth=new usefullMethods();
		//instance of forGraph class
		forGraph graph=new forGraph();

		//read the parameters form parameter file name
		params.readFromFile("parameters.txt");


		//spike train for each input pattern
		int[][] inputSpikeTrain=new
int[inputLayerNumberOfNeurons][(int)
Math.rint(durationOfSimulation/timeStep)];
```

```
            int[][] inputSpikeTrain00=new
int[inputLayerNumberOfNeurons][(int)
Math.rint(durationOfSimulation/timeStep)];
            int[][] inputSpikeTrain01=new
int[inputLayerNumberOfNeurons][(int)
Math.rint(durationOfSimulation/timeStep)];
            int[][] inputSpikeTrain10=new
int[inputLayerNumberOfNeurons][(int)
Math.rint(durationOfSimulation/timeStep)];
            int[][] inputSpikeTrain11=new
int[inputLayerNumberOfNeurons][(int)
Math.rint(durationOfSimulation/timeStep)];


            //THE NETWORK

            //input Layer of the netWork
            inputNeuron[] inputNeurons = new
inputNeuron[inputLayerNumberOfNeurons];
            for(int i=0; i<inputLayerNumberOfNeurons; i++){
                  inputNeurons[i]=new
inputNeuron(inputLayerNumberOfNeurons,hiddenLayerNumberOfNeurons,inpu
tLayerNumberOfNeurons);
            }
            //hidden Layer of the netWork
            hiddenNeuron[] hiddenNeurons = new
hiddenNeuron[hiddenLayerNumberOfNeurons];
            for(int i=0; i<hiddenLayerNumberOfNeurons; i++){
                  hiddenNeurons[i]=new
hiddenNeuron(hiddenLayerNumberOfNeurons,outputLayerNumberOfNeurons,hi
ddenLayerNumberOfNeurons);
            }
            //output Layer of the netWork
            outputNeuron[] outputNeurons = new
outputNeuron[outputLayerNumberOfNeurons];
            for(int i=0; i<outputLayerNumberOfNeurons; i++){
                  outputNeurons[i]=new
outputNeuron(outputLayerNumberOfNeurons,2,outputLayerNumberOfNeurons)
;
            }

            //Dt
            double dt=timeStep;
            //Input current
            double input=0;
            //firing rate for patter 0,0
            double rate00=0;
            //firing rate for patter 0,1
            double rate01=0;
            //firing rate for patter 1,0
            double rate10=0;
            //firing rate for patter 1,1
            double rate11=0;
            //global reward for each epoch
            int reward=0;


            //create input set
            int[][] inputData=new int[4][2];
            inputData[0][0]=0;
            inputData[0][1]=0;
```

```
            inputData[1][0]=0;
            inputData[1][1]=1;
            inputData[2][0]=1;
            inputData[2][1]=0;
            inputData[3][0]=1;
            inputData[3][1]=1;
            //End

            double t=timeStep;

            int[] list=new int[4];
            list[0]=0;
            list[1]=1;
            list[2]=2;
            list[3]=3;


            //Initialize the staring weight in random number between
the bound
            for(int i=0; i<inputLayerNumberOfNeurons-
(percentOfInhibitory*inputLayerNumberOfNeurons)/100; i++){
                  for(int j=0; j<hiddenLayerNumberOfNeurons; j++){
                        inputNeurons[i].weights[j]=(0 +
(double)(Math.random()*bound));
                  }
            }
            for(int i=inputLayerNumberOfNeurons-
(percentOfInhibitory*inputLayerNumberOfNeurons)/100;
i<inputLayerNumberOfNeurons; i++){
                  for(int j=0; j<hiddenLayerNumberOfNeurons; j++){
                        inputNeurons[i].weights[j]=((0 +
(double)(Math.random()*bound))*-1);
                  }
            }

            for(int i=0; i<hiddenLayerNumberOfNeurons; i++){
                  for(int j=0; j<outputLayerNumberOfNeurons; j++){
                        hiddenNeurons[i].weights[j]=(0 +
(double)(Math.random()*bound));
                  }
            }
            //End initialization of staring weights

            //shuffle the input neurons
            meth.shuffleInputNeurons(inputNeurons);

            //create the spiketrains for each input pattern

      inputSpikeTrain00=meth.createInputSpikeTrain(inputLayerNumberOf
Neurons, rate, durationOfSimulation, timeStep, 0,0);

      inputSpikeTrain01=meth.createInputSpikeTrain(inputLayerNumberOf
Neurons, rate, durationOfSimulation, timeStep, 0,1);

      inputSpikeTrain10=meth.createInputSpikeTrain(inputLayerNumberOf
Neurons, rate, durationOfSimulation, timeStep, 1,0);

      inputSpikeTrain11=meth.createInputSpikeTrain(inputLayerNumberOf
Neurons, rate, durationOfSimulation, timeStep, 1,1);
```

```
            //Fix the system starting output firing rate
            double Ie01=1;
            //Set input spiketrain to be the pattern 0,1
            inputSpikeTrain=inputSpikeTrain01;

            //Found an input current to the system that force the
network to start with firing rates within a values set
            while(foundSpike>55 || foundSpike<45){

                  //Check if the found spike is under 45 spikes for
current input pattern
                  if(foundSpike<45)
                        //increase the input current
                        Ie01=Ie01+0.01;
                  //Check if the found spike is more than 55
                  else if (foundSpike>55)
                        //decrease  the input current
                        Ie01=Ie01-0.01;
                  foundSpike=0;

                  while(t<=durationOfSimulation){

                        //Prepare the input multiply with the weights
of the inputneurons
                        for(int i=0; i<hiddenLayerNumberOfNeurons;
i++){
                              hiddenNeurons[i].input=0;
                              for(int j=0;
j<inputLayerNumberOfNeurons; j++){

      hiddenNeurons[i].input+=inputSpikeTrain[j][(int)
Math.rint(t/timeStep)-1]*inputNeurons[j].weights[i];
                              }
                        }

                        //Calculate the potentials at current
timestep for hidden neurons
                        for(int i=0; i<hiddenLayerNumberOfNeurons;
i++){

                              //Set Starting potentials to "0"
                              if((int) Math.rint(t/timeStep)==1){
                                    hiddenNeurons[i].dV=0;
                                    hiddenNeurons[i].V=0;
                              }
                              //End

                              hiddenNeurons[i].refractory+=timeStep;

                              //Check if there is a spike or not
                              if(hiddenNeurons[i].input>0 ||
hiddenNeurons[i].input<0 ){

      input=(hiddenNeurons[i].input*Ie01)/dt;
                              }else{
                                    input=0;
                              }
                              //End
                              //In case the user choose the single
point modeling
```

```
                                if(model.equalsIgnoreCase("single-
point"))
                                {     //Calculate the membrane
potential of the current hidden neuron

      hiddenNeurons[i].V=hiddenNeurons[i].V+((-
hiddenNeurons[i].V+Rm*input)/Tm)*dt;

                                }
                                else if (model.equalsIgnoreCase("two-
point")){
                                      //Calculate the dedrite potential
for the current hidden neuron

      hiddenNeurons[i].dV=hiddenNeurons[i].dV+((-
hiddenNeurons[i].dV/dTm)+((hiddenNeurons[i].V-
hiddenNeurons[i].dV)/rC)+input)*dt;
                                      //Calculate the membrane
potential fo the current hidden neuron

      hiddenNeurons[i].V=hiddenNeurons[i].V+((-
hiddenNeurons[i].V/Tm)+((hiddenNeurons[i].dV-
hiddenNeurons[i].V)/rC))*dt;

                                }//if the membrane potential pass the
threshold
                                if(hiddenNeurons[i].V>=Vth &&
hiddenNeurons[i].refractory>Trefr){
                                      hiddenNeurons[i].spikeTrain=1;
                                      //In case the user choose total
reset

      if(reset.equalsIgnoreCase("total")){

      hiddenNeurons[i].V=Vreset;//Reset the potential to "0" after
the spike
                                      }//In case the user choose
parital reset
                                      else if
(reset.equalsIgnoreCase("partial")){
                                            hiddenNeurons[i].V=Vth*a;
                                      }//Set the refractory time to 0
                                      hiddenNeurons[i].refractory=0;

                                }else{hiddenNeurons[i].spikeTrain=0;}

                          }
                          //End

                          //Create the input to the output neuron
include the multiplication with the weights
                          for(int i=0; i<outputLayerNumberOfNeurons;
i++){
                                outputNeurons[i].input=0;
                                for(int j=0;
j<hiddenLayerNumberOfNeurons; j++){

      outputNeurons[i].input+=hiddenNeurons[j].spikeTrain*hiddenNeuro
ns[j].weights[i];
                                }
                          }
```

```
                        //End

                        //Calculate the potentials at current
timestep for output neurons
                        for(int i=0; i<outputLayerNumberOfNeurons;
i++){
                              //set the starting potential to 0
                              if((int) Math.rint(t/timeStep)==1){
                                    outputNeurons[i].dV=0;
                                    outputNeurons[i].V=0;
                              }
                              outputNeurons[i].refractory+=timeStep;

                              //Check if there is a spike or not
                              if(outputNeurons[i].input>0 ||
outputNeurons[i].input<0){
                                    //set the input to the system

      input=(outputNeurons[i].input*Ie01)/dt;
                              }else{//no input to the system
                                    input=0;
                              }
                              //End

                              //In case the user choose the single
point modeling
                              if(model.equalsIgnoreCase("single-
point"))
                              {     //Calculate the membrane
potential of the current output neuron

      outputNeurons[i].V=outputNeurons[i].V+((-
outputNeurons[i].V+Rm*input)/Tm)*dt;
                              }//In case the user choose the two
point modeling
                              else if (model.equalsIgnoreCase("two-
point")){
                                    //Calculate the dedrite potential
of the current output neuron

      outputNeurons[i].dV=outputNeurons[i].dV+((-
outputNeurons[i].dV/dTm)+((outputNeurons[i].V-
outputNeurons[i].dV)/rC)+input)*dt;
                                    //Calculate the membrane
potential of the current output neuron

      outputNeurons[i].V=outputNeurons[i].V+((-
outputNeurons[i].V/Tm)+((outputNeurons[i].dV-
outputNeurons[i].V)/rC))*dt;
                              }
                              //If the membrane potential pass the
threshold
                              if(outputNeurons[i].V>=Vth &&
outputNeurons[i].refractory>Trefr){
                                    //Count the spike
                                    foundSpike++;
                                    //In case the user choose total
reset

      if(reset.equalsIgnoreCase("total")){
```

```
      outputNeurons[i].V=Vreset;//Reset the potential to "0" after
the spike
                                        }
                                        //In case the user choose partial
reset
                                        else if
(reset.equalsIgnoreCase("partial")){
                                              outputNeurons[i].V=Vth*a;
                                        }
                                        //Set the refractory time to 0
                                        outputNeurons[i].refractory=0;
                                  }
                            }
                            //End
                            t+=timeStep;
                      }
                      t=timeStep;
                      //Print the output spikes number
                      System.out.println(foundSpike);
                }
                foundSpike=0;

                double Ie10=1;
                //Set input spiketrain to be the pattern 1,0
                inputSpikeTrain=inputSpikeTrain10;
                //Found an input current to the system that force the
network to start with firing rates within a values set
                while(foundSpike>55 || foundSpike<45){
                      //Check if the found spike is under 45 spikes for
current input pattern
                      if(foundSpike<45)
                            //increase the input current
                            Ie10=Ie10+0.01;
                      //Check if the found spike is more than 55
                      else if (foundSpike>55)
                            //decrease  the input current
                            Ie10=Ie10-0.01;
                      foundSpike=0;
                      while(t<=durationOfSimulation){
                            //Prepare the input multiply with the weights
of the inputneurons
                            for(int i=0; i<hiddenLayerNumberOfNeurons;
i++){
                                  hiddenNeurons[i].input=0;
                                  for(int j=0;
j<inputLayerNumberOfNeurons; j++){

      hiddenNeurons[i].input+=inputSpikeTrain[j][(int)
Math.rint(t/timeStep)-1]*inputNeurons[j].weights[i];
                                  }
                            }

                            //Calculate the potentials at current
timestep for hidden neurons
                            for(int i=0; i<hiddenLayerNumberOfNeurons;
i++){

                                  //Set Starting potentials to "0"
                                  if((int) Math.rint(t/timeStep)==1){
                                        hiddenNeurons[i].dV=0;
```

```
                                    hiddenNeurons[i].V=0;
                              }
                              //End

                              hiddenNeurons[i].refractory+=timeStep;

                              //Check if there is a spike or not
                              if(hiddenNeurons[i].input>0 ||
hiddenNeurons[i].input<0 ){

      input=(hiddenNeurons[i].input*Ie10)/dt;
                              }else{
                                    input=0;
                              }
                              //End
                              //In case the user choose the single
point modeling
                              if(model.equalsIgnoreCase("single-
point"))
                              {
                                    //Calculate the membrane
potential of the current hidden neuron

      hiddenNeurons[i].V=hiddenNeurons[i].V+((-
hiddenNeurons[i].V+Rm*input)/Tm)*dt;
                              }//In case the user choose the two
point modeling
                              else if (model.equalsIgnoreCase("two-
point")){
                                    //Calculate the dedrite potential
for the current hidden neuron

      hiddenNeurons[i].dV=hiddenNeurons[i].dV+((-
hiddenNeurons[i].dV/dTm)+((hiddenNeurons[i].V-
hiddenNeurons[i].dV)/rC)+input)*dt;
                                    //Calculate the membrane
potential for the current hidden neuron

      hiddenNeurons[i].V=hiddenNeurons[i].V+((-
hiddenNeurons[i].V/Tm)+((hiddenNeurons[i].dV-
hiddenNeurons[i].V)/rC))*dt;
                              }
                              //if the membrane potential pass the
threshold
                              if(hiddenNeurons[i].V>=Vth &&
hiddenNeurons[i].refractory>Trefr){

                                    hiddenNeurons[i].spikeTrain=1;
                                    //In case the user choose total
reset

      if(reset.equalsIgnoreCase("total")){

      hiddenNeurons[i].V=Vreset;//Reset the potential to "0" after
the spike
                                    }//In case the user choose
parital reset
                                    else if
(reset.equalsIgnoreCase("partial")){
                                          hiddenNeurons[i].V=Vth*a;
                                    }
```

```
                                    //Set the refractory time to 0
                                    hiddenNeurons[i].refractory=0;

                              }else{hiddenNeurons[i].spikeTrain=0;}

                        }
                        //End

                        //Create the input to the output neuron
include the multiplication with the weights
                        for(int i=0; i<outputLayerNumberOfNeurons;
i++){
                              outputNeurons[i].input=0;
                              for(int j=0;
j<hiddenLayerNumberOfNeurons; j++){

      outputNeurons[i].input+=hiddenNeurons[j].spikeTrain*hiddenNeuro
ns[j].weights[i];
                              }
                        }
                        //End

                        //Calculate the potentials at current
timestep for output neurons
                        for(int i=0; i<outputLayerNumberOfNeurons;
i++){

                              //set the starting potential to 0
                              if((int) Math.rint(t/timeStep)==1){
                                    outputNeurons[i].dV=0;
                                    outputNeurons[i].V=0;
                              }
                              outputNeurons[i].refractory+=timeStep;

                              //Check if there is a spike or not
                              if(outputNeurons[i].input>0 ||
outputNeurons[i].input<0){
                                    //set the input to the system

      input=(outputNeurons[i].input*Ie10)/dt;
                              }else{//no input to the system
                                    input=0;
                              }
                              //End
                              //In case the user choose the single
point modeling
                              if(model.equalsIgnoreCase("single-
point"))
                              {
                                    //Calculate the membrane
potential of the current output neuro

      outputNeurons[i].V=outputNeurons[i].V+((-
outputNeurons[i].V+Rm*input)/Tm)*dt;
                              }//In case the user choose the two
point modeling
                              else if (model.equalsIgnoreCase("two-
point")){
                                    //Calculate the dedrite potential
of the current output neuron
```

```
      outputNeurons[i].dV=outputNeurons[i].dV+((-
outputNeurons[i].dV/dTm)+((outputNeurons[i].V-
outputNeurons[i].dV)/rC)+input)*dt;
                                    //Calculate the membrane
potential of the current output neuron

      outputNeurons[i].V=outputNeurons[i].V+((-
outputNeurons[i].V/Tm)+((outputNeurons[i].dV-
outputNeurons[i].V)/rC))*dt;
                              }
                              //If the membrane potential pass the
threshold
                              if(outputNeurons[i].V>=Vth &&
outputNeurons[i].refractory>Trefr){
                                    //Count the spike
                                    foundSpike++;

                                    //In case the user choose total
reset

      if(reset.equalsIgnoreCase("total")){

      outputNeurons[i].V=Vreset;//Reset the potential to "0" after
the spike
                                    }//In case the user choose
partial reset
                                    else if
(reset.equalsIgnoreCase("partial")){
                                          outputNeurons[i].V=Vth*a;
                                    }
                                    //Set the refractory time to 0
                                    outputNeurons[i].refractory=0;
                              }
                        }
                        //End
                        t+=timeStep;
                        //System.out.println(t);
                  }
                  t=timeStep;
                  System.out.println(foundSpike);
            }
            foundSpike=0;
            double Ie11=1;
            //Set input spiketrain to be the pattern 1,1
            inputSpikeTrain=inputSpikeTrain11;
            //Found an input current to the system that force the
network to start with firing rates within a values set
            while(foundSpike>55 || foundSpike<45){
                  //Check if the found spike is under 45 spikes for
current input pattern
                  if(foundSpike<45)
                        //increase the input current
                        Ie11=Ie11+0.01;
                  //Check if the found spike is more than 55
                  else if (foundSpike>55)
                        //decrease  the input current
                        Ie11=Ie11-0.01;
                  foundSpike=0;
                  while(t<=durationOfSimulation){
```

```
                        //Prepare the input multiply with the weights
of the inputneurons
                        for(int i=0; i<hiddenLayerNumberOfNeurons;
i++){
                              hiddenNeurons[i].input=0;
                              for(int j=0;
j<inputLayerNumberOfNeurons; j++){

      hiddenNeurons[i].input+=inputSpikeTrain[j][(int)
Math.rint(t/timeStep)-1]*inputNeurons[j].weights[i];
                              }
                        }

                        //Calculate the potentials at current
timestep for hidden neurons
                        for(int i=0; i<hiddenLayerNumberOfNeurons;
i++){

                              //Set Starting potentials to "0"
                              if((int) Math.rint(t/timeStep)==1){
                                    hiddenNeurons[i].dV=0;
                                    hiddenNeurons[i].V=0;
                              }
                              //End

                              hiddenNeurons[i].refractory+=timeStep;

                              //Check if there is a spike or not
                              if(hiddenNeurons[i].input>0 ||
hiddenNeurons[i].input<0 ){

      input=(hiddenNeurons[i].input*Ie11)/dt;
                              }else{
                                    input=0;
                              }
                              //End
                              //In case the user choose the single
point modeling
                              if(model.equalsIgnoreCase("single-
point"))
                              {
                                    //Calculate the membrane
potential of the current hidden neuron

      hiddenNeurons[i].V=hiddenNeurons[i].V+((-
hiddenNeurons[i].V+Rm*input)/Tm)*dt;
                              }
                              //In case the user choose the two point
modeling
                              else if (model.equalsIgnoreCase("two-
point")){
                                    //Calculate the dedrite potential
for the current hidden neuron

      hiddenNeurons[i].dV=hiddenNeurons[i].dV+((-
hiddenNeurons[i].dV/dTm)+((hiddenNeurons[i].V-
hiddenNeurons[i].dV)/rC)+input)*dt;
                                    //Calculate the membrane
potential fo the current hidden neuron

      hiddenNeurons[i].V=hiddenNeurons[i].V+((-
```

```
hiddenNeurons[i].V/Tm)+((hiddenNeurons[i].dV-
hiddenNeurons[i].V)/rC))*dt;

                                    }
                                    //if the membrane potential pass the
threshold
                                    if(hiddenNeurons[i].V>=Vth &&
hiddenNeurons[i].refractory>Trefr){
                                          hiddenNeurons[i].spikeTrain=1;
                                          //In case the user choose total
reset

      if(reset.equalsIgnoreCase("total")){

      hiddenNeurons[i].V=Vreset;//Reset the potential to "0" after
the spike
                                          }//In case the user choose
partial reset
                                          else if
(reset.equalsIgnoreCase("partial")){
                                                hiddenNeurons[i].V=Vth*a;
                                          }
                                          hiddenNeurons[i].refractory=0;

                                    }else{hiddenNeurons[i].spikeTrain=0;}
                              }
                              //End

                              //Create the input to the output neuron
include the multiplication with the weights
                              for(int i=0; i<outputLayerNumberOfNeurons;
i++){
                                    outputNeurons[i].input=0;
                                    for(int j=0;
j<hiddenLayerNumberOfNeurons; j++){

      outputNeurons[i].input+=hiddenNeurons[j].spikeTrain*hiddenNeuro
ns[j].weights[i];
                                    }
                              }
                              //End

                              //Calculate the potentials at current
timestep for output neurons
                              for(int i=0; i<outputLayerNumberOfNeurons;
i++){

                                    //set the starting potential to 0
                                    if((int) Math.rint(t/timeStep)==1){
                                          outputNeurons[i].dV=0;
                                          outputNeurons[i].V=0;
                                    }
                                    outputNeurons[i].refractory+=timeStep;

                                    //Check if there is a spike or not
                                    if(outputNeurons[i].input>0 ||
outputNeurons[i].input<0){

      input=(outputNeurons[i].input*Ie11)/dt;
                                    }else{
                                          input=0;
```

```
                                }
                                //End
                                //In case the user choose the single
point modeling
                                if(model.equalsIgnoreCase("single-
point"))
                                {
                                      //Calculate the membrane
potential of the current output neuron

      outputNeurons[i].V=outputNeurons[i].V+((-
outputNeurons[i].V+Rm*input)/Tm)*dt;
                                }//In case the user choose the two
point modeling
                                else if (model.equalsIgnoreCase("two-
point")){
                                      // Calculate the dedrite
potential of the current output neuron

      outputNeurons[i].dV=outputNeurons[i].dV+((-
outputNeurons[i].dV/dTm)+((outputNeurons[i].V-
outputNeurons[i].dV)/rC)+input)*dt;
                                      // Calculate the membrane
potential of the current output neuron

      outputNeurons[i].V=outputNeurons[i].V+((-
outputNeurons[i].V/Tm)+((outputNeurons[i].dV-
outputNeurons[i].V)/rC))*dt;
                                }

                                //If the membrane potential pass the
threshold
                                if(outputNeurons[i].V>=Vth &&
outputNeurons[i].refractory>Trefr){
                                      //Count the spike
                                      foundSpike++;
                                      //Set the indicator to 1 to know
that the current output neuron at current time step has spike
                                      outputNeurons[i].spikeTrain=1;
                                      //In case the user choose total

      if(reset.equalsIgnoreCase("total")){

      outputNeurons[i].V=Vreset;//Reset the potential to "0" after
the spike
                                      }
                                      //In case the user choose partial
reset
                                      else if
(reset.equalsIgnoreCase("partial")){
                                            outputNeurons[i].V=Vth*a;
                                      }
                                      outputNeurons[i].refractory=0;
                                }
                          }
                          //End
                          t+=timeStep;
                          //System.out.println(t);
                    }
                    t=timeStep;
                    System.out.println(foundSpike);
```

```
            }
            //End Fixing starting rate

            foundSpike=0;
            //The simulation (training)
            for(int epoch=0; epoch<200; epoch++){

                  meth.shuffleArray(list);

                  //pass the patterns to the system for the current
epoch
                  for(int set=0; set<4; set++){

                        if(list[set]==0)
                              inputSpikeTrain=inputSpikeTrain00;

                        if(list[set]==1){
                              inputSpikeTrain=inputSpikeTrain01;
                              Ie=Ie01;
                        }
                        if(list[set]==2){
                              inputSpikeTrain=inputSpikeTrain10;
                              Ie=Ie10;
                        }
                        if(list[set]==3){
                              inputSpikeTrain=inputSpikeTrain11;
                              Ie=Ie11;
                        }

                        //initialize the refractory time to
refractory period
                        for(int i=0; i<hiddenLayerNumberOfNeurons;
i++){
                              hiddenNeurons[i].refractory=Trefr;
                        }
                        for(int i=0; i<outputLayerNumberOfNeurons;
i++){
                              outputNeurons[i].refractory=Trefr;

                        }


                        double inputWeight;
                        double hiddenWeight;
                        while(t<=durationOfSimulation){


                              //Prepare the input multiply with the
weights of the inputneurons
                              for(int i=0;
i<hiddenLayerNumberOfNeurons; i++){
                                    hiddenNeurons[i].input=0;
                                    for(int j=0;
j<inputLayerNumberOfNeurons; j++){

      hiddenNeurons[i].input+=inputSpikeTrain[j][(int)
Math.rint(t/timeStep)-1]*inputNeurons[j].weights[i];
                                    }
                              }
```

```
                                    //Calculate the potentials at current
timestep for hidden neurons
                                    for(int i=0;
i<hiddenLayerNumberOfNeurons; i++){

                                          //Set Starting potentials to "0"
                                          if((int)
Math.rint(t/timeStep)==1){

                                                hiddenNeurons[i].dV=0;
                                                hiddenNeurons[i].V=0;
                                          }
                                          //End


      hiddenNeurons[i].refractory+=timeStep;

                                          //Check if there is a spike or
not
                                          if(hiddenNeurons[i].input>0 ||
hiddenNeurons[i].input<0 ){

      input=(hiddenNeurons[i].input*Ie)/dt;
                                          }else{
                                                input=0;
                                          }
                                          //End

                                          //In case the user choose the
single point modeling

      if(model.equalsIgnoreCase("single-point"))
                                          {
                                                //Calculate the membrane
potential of the current hidden neuron

      hiddenNeurons[i].V=hiddenNeurons[i].V+((-
hiddenNeurons[i].V+Rm*input)/Tm)*dt;
                                          }
                                          //In case the user choose the two
point modeling
                                          else if
(model.equalsIgnoreCase("two-point")){
                                                //Calculate the dedrite
potential for the current hidden neuron

      hiddenNeurons[i].dV=hiddenNeurons[i].dV+((-
hiddenNeurons[i].dV/dTm)+((hiddenNeurons[i].V-
hiddenNeurons[i].dV)/rC)+input)*dt;
                                                //Calculate the membrane
potential fo the current hidden neuron

      hiddenNeurons[i].V=hiddenNeurons[i].V+((-
hiddenNeurons[i].V/Tm)+((hiddenNeurons[i].dV-
hiddenNeurons[i].V)/rC))*dt;
                                          }
                                          //if the membrane potential pass
the threshold
                                          if(hiddenNeurons[i].V>=Vth &&
hiddenNeurons[i].refractory>Trefr){
                                                //Set the indicator to 1 to
know that the current neuron at the current time step has spike
```

```
      hiddenNeurons[i].spikeTrain=1;
                                          //In case the user choose
total reset
      if(reset.equalsIgnoreCase("total")){
      hiddenNeurons[i].V=Vreset;//Reset the potential to "0" after
the spike
                                          }//In case the user choose
parital reset
                                          else if
(reset.equalsIgnoreCase("partial")){
      hiddenNeurons[i].V=Vth*a;
                                          }
                                          //Set the refractory time
to 0
      hiddenNeurons[i].refractory=0;
                                    }
                                    else{
                                          //Set the indicator to 0 to
know that the current neuron at the current time set has no spike
      hiddenNeurons[i].spikeTrain=0;
                                    }
                              }
                              //End

                              //Create the input to the output neuron
include the multiplication with the weights
                              for(int i=0;
i<outputLayerNumberOfNeurons; i++){
                                    outputNeurons[i].input=0;
                                    for(int j=0;
j<hiddenLayerNumberOfNeurons; j++){
      outputNeurons[i].input+=hiddenNeurons[j].spikeTrain*hiddenNeuro
ns[j].weights[i];
                                    }
                              }
                              //End

                              //Calculate the potentials at current
timestep for output neurons
                              for(int i=0;
i<outputLayerNumberOfNeurons; i++){
                                    if((int)
Math.rint(t/timeStep)==1){
                                          outputNeurons[i].dV=0;
                                          outputNeurons[i].V=0;
                                    }
      outputNeurons[i].refractory+=timeStep;

                                    //Check if there is a spike or
not
                                    if(outputNeurons[i].input>0 ||
outputNeurons[i].input<0){
```

```
      input=(outputNeurons[i].input*Ie)/dt;
                                        }else{
                                              input=0;
                                        }
                                        //End
                                        //In case the user choose the
single point modeling

      if(model.equalsIgnoreCase("single-point"))
                                        {
                                              //Calculate the membrane
potential of the current output neuron

      outputNeurons[i].V=outputNeurons[i].V+((-
outputNeurons[i].V+Rm*input)/Tm)*dt;
                                        }//In case the user choose the
two point modeling
                                        else if
(model.equalsIgnoreCase("two-point")){
                                              //Calculate the dedrite
potential of the current output neuron

      outputNeurons[i].dV=outputNeurons[i].dV+((-
outputNeurons[i].dV/dTm)+((outputNeurons[i].V-
outputNeurons[i].dV)/rC)+input)*dt;
                                              //Calculate the membrane
potential of the current output neuron

      outputNeurons[i].V=outputNeurons[i].V+((-
outputNeurons[i].V/Tm)+((outputNeurons[i].dV-
outputNeurons[i].V)/rC))*dt;
                                        }
                                        //If the membrane potential pass
the threshold
                                        if(outputNeurons[i].V>=Vth &&
outputNeurons[i].refractory>Trefr){
                                              //Count the spike
                                              foundSpike++;
                                              //Set the indicator to 1 to
know that the current output neuron at current time step has spike

      outputNeurons[i].spikeTrain=1;

      if(reset.equalsIgnoreCase("total")){

      outputNeurons[i].V=Vreset;//Reset the potential to "0" after
the spike
                                              }//In case the user choose
partial reset
                                              else if
(reset.equalsIgnoreCase("partial")){

      outputNeurons[i].V=Vth*a;
                                              }
                                              //Set the refractory time
to 0

      outputNeurons[i].refractory=0;

                                              //Check for reward
```

```
      if((inputData[list[set]][0]==0 &&
inputData[list[set]][1]==1)||(inputData[list[set]][0]==1 &&
inputData[list[set]][1]==0)){
                                                r=1;
                                                reward++;
                                          }else{
                                                r=-1;
                                                reward--;
                                          }
                                    }
                                    else{
                                          //Set the indicator to 0 to
know that the current neuron at the current time set has no spike

      outputNeurons[i].spikeTrain=0;
                                                r=0;
                                    }
                                    //Make the changes
                                    for(int pi=0;
pi<inputLayerNumberOfNeurons; pi++){
                                          for(int pj=0;
pj<hiddenLayerNumberOfNeurons; pj++){

                                                //Set the P+ for the
input neuron in case that the pre neuron has spike

      if(inputSpikeTrain[pi][(int) Math.rint(t/timeStep)-1]>0){

      inputNeurons[pi].PijPlus[pj]=inputNeurons[pi].PijPlus[pj]*Math.
exp(-dt/tPlus)+aPlus*1;
                                                }
                                                //Set the P+ for the
input neuron in case that the pre neuron has no spike
                                                else{

      if(inputNeurons[pi].PijPlus[pj]==0)

      inputNeurons[pi].PijPlus[pj]=0;
                                                      else

      inputNeurons[pi].PijPlus[pj]=inputNeurons[pi].PijPlus[pj]*Math.
exp(-dt/tPlus);
                                                }
                                                //Set the P- for the
input neuron in case that the post neuron has spike

      if(hiddenNeurons[pj].spikeTrain>0){

      inputNeurons[pi].PijMinus[pj]=inputNeurons[pi].PijMinus[pj]*Mat
h.exp(-dt/tMinus)+aMinus*1;
                                                }
                                                //Set the P+ for the
input neuron in case that the post neuron has no spike
                                                else{

      if(inputNeurons[pi].PijMinus[pj]==0)

      inputNeurons[pi].PijMinus[pj]=0;
                                                      else
```

```
      inputNeurons[pi].PijMinus[pj]=inputNeurons[pi].PijMinus[pj]*Mat
h.exp(-dt/tMinus);
                                                      }
                                                      //Set the zita in
case that both pre - post neuron has spike

      if(hiddenNeurons[pj].spikeTrain>0 && inputSpikeTrain[pi][(int)
Math.rint(t/timeStep)-1]>0)

      inputNeurons[pi].zita[pj]=inputNeurons[pi].PijPlus[pj]+inputNeu
rons[pi].PijMinus[pj];
                                                      //Set the zita in
case that post neuron has spike
                                                      else
if(hiddenNeurons[pj].spikeTrain>0 && inputSpikeTrain[pi][(int)
Math.rint(t/timeStep)-1]==0)

      inputNeurons[pi].zita[pj]=inputNeurons[pi].PijPlus[pj];
                                                      //Set the zita in
case that pre neuron has spike
                                                      else
if(hiddenNeurons[pj].spikeTrain==0 && inputSpikeTrain[pi][(int)
Math.rint(t/timeStep)-1]>0)

      inputNeurons[pi].zita[pj]=inputNeurons[pi].PijMinus[pj];
                                                      //either pre - post
has no spike
                                                      else

      inputNeurons[pi].zita[pj]=0;


                                                }
                                          }
                                          for(int pi=0;
pi<hiddenLayerNumberOfNeurons; pi++){
                                                for(int pj=0;
pj<outputLayerNumberOfNeurons; pj++){

                                                      //Set the P+ for the
hidden neuron in case that the pre neuron has spike

      if(hiddenNeurons[pi].spikeTrain>0){

      hiddenNeurons[pi].PijPlus[pj]=hiddenNeurons[pi].PijPlus[pj]*Mat
h.exp(-dt/tPlus)+aPlus*1;
                                                      }
                                                      //Set the P+ for the
hidden neuron in case that the pre neuron has no spike
                                                      else{

      if(hiddenNeurons[pi].PijPlus[pj]==0)

      hiddenNeurons[pi].PijPlus[pj]=0;
                                                            else

      hiddenNeurons[pi].PijPlus[pj]=hiddenNeurons[pi].PijPlus[pj]*Mat
h.exp(-dt/tPlus);
                                                      }
```

```
                                                    //Set the P- for the
hidden neuron in case that the post neuron has spike

      if(outputNeurons[pj].spikeTrain>0){

      hiddenNeurons[pi].PijMinus[pj]=hiddenNeurons[pi].PijMinus[pj]*M
ath.exp(-dt/tMinus)+aMinus*1;
                                                    }
                                                    //Set the P- for the
input neuron in case that the post neuron has no spike
                                                    else{

      if(hiddenNeurons[pi].PijMinus[pj]==0)

      hiddenNeurons[pi].PijMinus[pj]=0;
                                                          else

      hiddenNeurons[pi].PijMinus[pj]=hiddenNeurons[pi].PijMinus[pj]*M
ath.exp(-dt/tMinus);
                                                    }

                                                    //Set the zita in
case that both pre - post neuron has spike

      if(outputNeurons[pj].spikeTrain>0 &&
hiddenNeurons[pi].spikeTrain>0)

      hiddenNeurons[pi].zita[pj]=hiddenNeurons[pi].PijPlus[pj]+hidden
Neurons[pi].PijMinus[pj];
                                                    //Set the zita in
case that post neuron has spike
                                                    else
if(outputNeurons[pj].spikeTrain>0 && hiddenNeurons[pi].spikeTrain==0)

      hiddenNeurons[pi].zita[pj]=hiddenNeurons[pi].PijPlus[pj];
                                                    //Set the zita in
case that pre neuron has spike
                                                    else
if(outputNeurons[pj].spikeTrain==0 && hiddenNeurons[pi].spikeTrain>0)

      hiddenNeurons[pi].zita[pj]=hiddenNeurons[pi].PijMinus[pj];
                                                    //either pre - post
has no spike
                                                    else

      hiddenNeurons[pi].zita[pj]=0;

                                                          }
                                              }

                                              //End


                                              //Calculate Z and Update weights
for inputNeurons
                                              for(int pi=0;
pi<inputLayerNumberOfNeurons; pi++){
                                                    for(int pj=0;
pj<hiddenLayerNumberOfNeurons; pj++){
                                                    //Current weight
```

```
      inputWeight=inputNeurons[pi].weights[pj];
                                                  //Calculate z

      inputNeurons[pi].z[pj]=b*inputNeurons[pi].z[pj]+inputNeurons[pi
].zita[pj]/Tz;
                                                  //Change weight

      inputNeurons[pi].weights[pj]=inputNeurons[pi].weights[pj]+gama*
dt*r*inputNeurons[pi].z[pj];
                                                  //Apply the weight
bounds
                                                  //In case the synapse
is exhibitory  and the new weight goes under the 0
                                                  if(inputWeight>0 &&
inputNeurons[pi].weights[pj]<0)

      inputNeurons[pi].weights[pj]=0;
                                                  //In case the synapse
is inhibitory and the new weight goes bigger that 0
                                                  else if(inputWeight<0
&& inputNeurons[pi].weights[pj]>0)

      inputNeurons[pi].weights[pj]=0;
                                                  //In case the new
weight pass the upper bound

      if(inputNeurons[pi].weights[pj]>bound)

      inputNeurons[pi].weights[pj]=bound;
                                                  //In case the new
weight pass the lower bound
                                                  else
if(inputNeurons[pi].weights[pj]<-bound)

      inputNeurons[pi].weights[pj]=-bound;
                                                }
                                          }

                                          //Calculate Z and Update weights
for hiddenNeurons
                                          for(int pi=0;
pi<hiddenLayerNumberOfNeurons; pi++){
                                                for(int pj=0;
pj<outputLayerNumberOfNeurons; pj++){
                                                  //Current weight

      hiddenWeight=hiddenNeurons[pi].weights[pj];
                                                  //Calculate z

      hiddenNeurons[pi].z[pj]=b*hiddenNeurons[pi].z[pj]+hiddenNeurons
[pi].zita[pj]/Tz;
                                                  //Change weight

      hiddenNeurons[pi].weights[pj]=hiddenNeurons[pi].weights[pj]+gam
a*dt*r*hiddenNeurons[pi].z[pj];
                                                  //Apply the weight
bounds
                                                  //In case the synapse
is exhibitory  and the new weight goes under the 0
```

```
                                                if(hiddenWeight>0 &&
hiddenNeurons[pi].weights[pj]<0)

      hiddenNeurons[pi].weights[pj]=0;
                                                //In case the synapse
is inhibitory and the new weight goes bigger that 0
                                                else
if(hiddenWeight<0 &&hiddenNeurons[pi].weights[pj]>0)

      hiddenNeurons[pi].weights[pj]=0;
                                                //In case the new
weight pass the upper bound

      if(hiddenNeurons[pi].weights[pj]>bound)

      hiddenNeurons[pi].weights[pj]=bound;
                                                //In case the new
weight pass the lower bound
                                                else
if(hiddenNeurons[pi].weights[pj]<-bound)

      hiddenNeurons[pi].weights[pj]=-bound;
                                                      }
                                          }

                              }
                              //End
                              t+=timeStep;
                        }
                        t=timeStep;
                        //Print the input patter and the found output
spikes number
                        System.out.println("Input: " +
inputData[list[set]][0] + " " + inputData[list[set]][1] + " Found
spikes " + foundSpike);
                        System.out.println(" ");

                        if(list[set]==0)
                              rate00=foundSpike;

                        if(list[set]==1)
                              rate01=foundSpike;

                        if(list[set]==2)
                              rate10=foundSpike;

                        if(list[set]==3)
                              rate11=foundSpike;

                        foundSpike=0;

                        inputWeight=0;
                        hiddenWeight=0;

                  }
                  System.out.println("End of epoch: " + epoch + "
Global reward: " +reward);
                  System.out.println("--------------");
                  System.out.println(" ");
                  reward=0;
```

```
            }
            //Create the graph with the output firing rates for all
patterns

      graph.createChart(((rate00/500)*1000),((rate01/500)*1000),((rat
e10/500)*1000),((rate11/500)*1000), "Average firing rate", "Input
Pattern", "Firing rate (hz)", 500, 500);

      }
}
```