SIMULATION OF OSCILLATIONS IN A NETWORK OF NEURONS USING INTEGRATE AND FIRE NEURON MODEL

DANNY NG WEE KIAT

MASTER OF ENGINEERING SCIENCE

FACULTY OF ENGINEERING AND SCIENCE
UNIVERSITI TUNKU ABDUL RAHMAN
JANUARY 2012

SIMULATION OF OSCILLATIONS IN A NETWORK OF NEURONS USING INTEGRATE AND FIRE NEURON MODEL

By

DANNY NG WEE KIAT

A dissertation submitted to the Department of Mechatronics and Biomedical Engineering,
Faculty of Engineering and Science,
Universiti Tunku Abdul Rahman,
in partial fulfilment of the requirements for the degree of
Master of Engineering Science
January 2012

ABSTRACT

SIMULATION OF OSCILLATIONS IN A NETWORK OF NEURONS USING INTEGRATE AND FIRE NEURON MODEL

Danny Ng Wee Kiat

Human brains display oscillatory patterns at characteristic frequency bands during various behavioural states. Phenomena of oscillations and synchronous firings of neurons are particularly important from the functional point of view, and have generated many interesting hypotheses concerning neural signal processing in the central nervous system. This study on the simulation of global oscillations of a network of neurons seeks to provide a better understanding of the nature of these phenomena. Simulations were conducted using the Integrate and Fire neuron model. C++ is used for the development of the simulation algorithm. OpenMP is adopted in the algorithm to enable parallel processing on a multicore CPU. The effects of the network size, connection probability, synaptic weight and synaptic delay on the global oscillations of a network of identical inhibitory neurons were investigated. Using an extension of the model by Latham et al., simulations of ultra-slow spontaneous oscillations comparable to those observed in cortical cultures of rat neurons were achieved.

ACKNOWLEDGEMENT

This dissertation would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study. I am heartily thankful to my supervisor, Prof. Dato' Dr Goh Sing Yau, whose encouragement, supervision and support from the preliminary to the concluding level enabled me to develop an understanding of the subject. His guidance’s helped me in all the time of research and writing of this thesis. I thank my fellow colleague in UTAR: Mok Siew Ying and Chan Siow Cheng for the stimulating discussion and insights on the subject.

APPROVAL SHEET

This dissertation/thesis entitled "**<u>Simulation of oscillations in a network of neurons using integrate and fire neuron model</u>**" was prepared by DANNY NG WEE KIAT and submitted as partial fulfilment of the requirements for the degree of Master of Engineering Science at Universiti Tunku Abdul Rahman.

Approved by:

____

(Prof. Dato' Dr Goh Sing Yau) Date:…………………..
Professor/Supervisor
Department of Mechanical and Material Engineering
Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

FACULTY OF ENGINEERING AND SCIENCE
UNIVERSITI TUNKU ABDUL RAHMAN

Date: ___________________

SUBMISSION OF THESIS

It is hereby certified that **DANNY NG WEE KIAT**(ID No: **09UEM03406**) has completed this thesis entitled "Simulation of Network oscillation using integrate and fire model" under the supervision of Prof. Dato' Dr Goh Sing Yau (Supervisor) from the Department of Department of Mechanical and Material Engineering, Faculty of Engineering and Science.

I understand that the University will upload softcopy of my thesis in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

____
(DANNY NG WEE KIAT)

DECLARATION

I Danny Ng Wee Kiat hereby declare that the dissertation/thesis is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTAR or other institutions.

____

(Danny Ng Wee kiat)

Date: _____________________

TABLE OF CONTENT

	Page
ABSTRACT	**II**
ACKNOWLEDGEMENT	**III**
APPROVAL SHEET	**IV**
DECLARATION	**VI**
LIST OF FIGURES	**IX**
LIST OF TABLES	**XI**
LIST OF ABBREAVATION	**XII**
CHAPTERS	
INTRODUCTION	1
LITERATURE REVIEW	2
2.1 Experimental Studies	2
2.1.1 In Vivo Studies	2
2.1.2 In Vitro Studies	2
2.2 Theoretical And Modelling Studies.	4
2.2.1 Integrate And Fire (IF) Model	4
2.2.2 Other More Detailed Neuron Models	4
2.2.3 Network Oscillations	5
2.2.4 Spontaneous Bursting	6
2.2.5 The Present Study	7
METHODOLOGY	9
3.1 Effects of Parameters of The IF Model on Global Oscillations In a Network of Homogenous Neurons	9
3.2 Simulation Algorithm	11
3.2.1 Simulation Algorithm Development	13
3.2.2 Implementation of Synaptic Delay	16
3.2.3 Implementation of The After Hyperpolarization Current	17
3.2.4 Simulation In Networks With Excitatory Neurons	18
3.3 Simulation of Ultra-Slow Spontaneous Oscillation	19
RESULTS AND DISCUSSION	25
4.1 Simulation Algorithm	25
4.1.1 Comparison of Single Neuron Simulations With Exact Solutions	25
4.1.2 Network Simulation	29
4.1.3 Effects of Parallel Processing on Simulation Speed	31

4.1.4 Simulations of A Network of Inhibitory And Excitatory Neurons 33
4.2 Effect of The Parameters of The IF Model on The Global Oscillations In A Network of Homogenous Neurons 36
4.2.1 Network Size 37
4.2.2 External Noise 38
4.2.3 Connection Probability 39
4.2.4 Synaptic Delay 40
4.2.5 Synaptic Weight 41
4.3 Simulation of Spontaneous Activity 42
4.3.1 Spontaneous Activity 42
4.3.2 Simulation of Ultra-Slow Spontaneous Oscillations 47

CONCLUSIONS 52

REFRENCES 53
APPENDICES 63

LIST OF FIGURES

		Page
Figure 1	Experimental results showing spiking activities from 5 different cultures with spontaneous activity. Periodical fluctuations of activity over time at a very slow rate (0.001Hz – 0.005 Hz) can be observed in the cultures. The activity time bin is 10ms (Mok et al. [19])	3
Figure 2	OpenMP of parallel execution block	12
Figure 3	Flowchart for simulation algorithm	15
Figure 4	Implementation of synaptic delay	17
Figure 5	Experiment data showing spontaneous bursting activity for 10s in a culture of cortical neurons. a) Activity at troughs of ultra-slow spontaneous oscillations b) Activity at peaks of ultra-slow spontaneous oscillations. The activity time bin is 10ms (Mok et al. [19])	20
Figure 6	Experimental result showing ultra-slow changes in spontaneous activity. The activity time bin is 10ms (Mok et al. [19])	20
Figure 7	Spike sorting result on one of the electrode a) neuron fires throughout the whole recording. b) Neuron fires only at the peaks c, d) neuron with low level of activity. The activity time bin is 10ms (Mok et al. [19])	21
Figure 8	Comparison of simulated membrane voltage of single IF neuron model with exact solution	27
Figure 9	Comparison of simulated membrane voltage of single QIF neuron with the exact analytical solution	28
Figure 10	Comparison of results from simulation of a network of IF neurons	30
Figure 11	Results from simulation of QIF neurons network using different method	31
Figure 12	Speed of Simulation for a network of QIF Neurons	32
Figure 13	Speed of Simulation for a network of QIF with AHP Neurons	32
Figure 14	Synchronous Regular State (SR), g = 3, External noise =2 mV. The activity time bin is 10ms	34
Figure 15	Synchronous Irregular State (SI), Fast Oscillation, g = 6; External noise = 3 mV. The activity time bin is 10ms	35
Figure 16	Asynchronous Irregular State (AI), g = 5; External noise = 2 mV. The activity time bin is 10ms	35
Figure 17	Synchronous Irregular State (SI), Slow Oscillation, g = 4.5; External noise = 1.3 mV. The activity time bin is 10ms	36
Figure 18	Frequency of network oscillation for different network size	37
Figure 19	Frequency of network oscillation for different external noise	38
Figure 20	Frequency of network oscillation for different connection probability	39
Figure 21	Frequency of network oscillation for different synaptic delay	40
Figure 22	Frequency of network oscillation for different synaptic weight	41

Figure 23	Simulated activity for different Be and Bi values. Imax of 4 is used for the simulation. The activity time bin is 10ms	45
Figure 24	Experiment data showing spontaneous bursting activity for 10s in a culture of cortical neurons. The activity time bin is 10ms(Mok et al. [19])	46
Figure 25	Activity for network with Bi = 0.8, Be = 1.2 Imax = 5 to 3.8. The activity time bin is 10ms	47
Figure 26	Simulation results showing 5 peaks in 800s. Parameters for the simulation of activity: α = 0.07, upper threshold = 3.8.Subnetwork 1: number of neuron = 2000, τ_{φ} = 30000. Subnetwork 2: number of neurons = 8000, τ_{φ} = 1000. The activity time bin is 10ms	49
Figure 27	Changes of inhibitory properties, φ in subnetwork 1 and subnetwork 2 with parameters for the simulation of activity: α = 0.07, upper threshold = 3.8. Sub network 1: number of neuron = 2000, τ_{φ} = 30000. Subnetwork 2: number of neurons = 8000, $\tau\varphi$ = 1000	50
Figure 28	Simulation results showing 3 peaks in 800s. Parameters for the simulation of activity: α = 0.07, upper threshold = 3.8. Subnetwork 1: number of neuron = 3000, τ_{φ} = 30000. Subnetwork 2: number of neuron = 7000, τ_{φ} = 1000. The activity time bin is 10ms	51

LIST OF TABLES

		Page
Table 1	Parameters for simulation of homogenous network of inhibitory neuron	10
Table 2	Network Parameters for IF and QIF networks	29
Table 3	Network parameters for comparison of time taken for simulation (QIF)	31
Table 4	Network parameter for simulation of network of inhibitory and excitatory neurons	34
Table 5	Neuron parameters for simulation of spontaneous activity	43
Table 6	Network parameters for simulation of spontaneous activity	43
Table 7	Parameters for the inhibiting properties of a subnetwork	47

LIST OF ABBREAVATION

AHP	After Hyperpolarization
CSV	Comma Separated Values
EEG	Electroencephalography
IF	Integrate and Fire
MEA	Multi Electrode Array
ODE	order differential equation
OpenMP	Open Multi-Processing
QIF	Quadratic Integrate and Fire

CHAPTER 1

INTRODUCTION

Activities in the brain can be detected non-invasively using appropriate equipment and displayed as electroencephalography (EEG) signals. These EEG signals in the human brain display oscillation frequencies in the range of up to 100 Hz during various mental and physical activities. EEG oscillations can be detected in the frontal lobe of the brain during certain mental activities such as solving a mathematical problem, in the sensorimotor regions of the cortex during motor activities such as hand and foot movements, and in the occipital regions of the brain as evoked potentials when the eye is focused on an object. These EEG signals have been used to activate various devices in Brain-Computer Interfaces [1, 2]. The brain is a very complex organ consisting of many parts and many researchers are currently trying to unravel its mysteries as evidenced by the large number of articles in journals dealing with neuroscience [3, 4]. Numerous experiments have been carried out in vitro and in vivo to study the mechanisms underlying the generation of network oscillations [6-17]. It is hoped that the current study to simulate global oscillations in a network of neurons and the use of an extension of the model by Latham et al.[5] to simulate ultra-slow spontaneous oscillations will be an additional contribution to existing knowledge of oscillations in neuronal networks.

CHAPTER 2

LITERATURE REVIEW

2.1 Experimental studies

2.1.1 In Vivo Studies

Oscillations are a prevalent phenomenon in biological neural networks. Oscillations in the form of electroencephalograms (EEG) are present in different brain structures, with frequencies ranging from 0.5 Hz (δ rhythm) to 40-80 Hz (γ rhythm), and even up to 200 Hz [6]. Many studies focus on the mechanism for the generation of γ rhythm [7] in the brain as the γ rhythm is related to numerous cognitive and sensory functions [8]. Slow oscillations of brain waves less than 15 Hz are usually related to sleep or to the relaxed state of the brain. Slow oscillations can be detected in vivo during various sleep states [8]. The occurrence of slow oscillatory activity below 1Hz can be observed at the cortical, thalamocortical and hippocampal regions in vivo in animals under anaesthesia [10, 11, 12].

2.1.2 In Vitro studies

Cunningham et al.[13] showed that it is possible to induce gamma oscillations in vitro in slices of cortical neurons. Sanchez et al.[14] showed that brain slices immersed in bathing medium that mimic extracellular ionic composition in situ can generate slow oscillations.

Chen et al.[15] showed that spontaneous activation of neurons can be detected in cultures of neurons. Zhu et al.[16] also detected slow spontaneous oscillations of activities around 0.005Hz in cultured of hippocampal neurons. Latham et al.[17] conducted experiments to study the link between the fraction of endogenously active cells and network firing pattern. Neurotransmitter blockers are applied in the cultures to examine the presence of endogenously active cells.

Mok et al.[18] recently reported on ultra-slow spontaneous activities in MEA cultures of rat cortical neurons. Figure 1 shows an extract of the experimental results obtained from these cultures. Ultra-slow spontaneous oscillations lower than 0.005Hz are be observed in the spiking activity of neurons. These activity patterns emerged spontaneously in certain cultures after forth week in vitro.

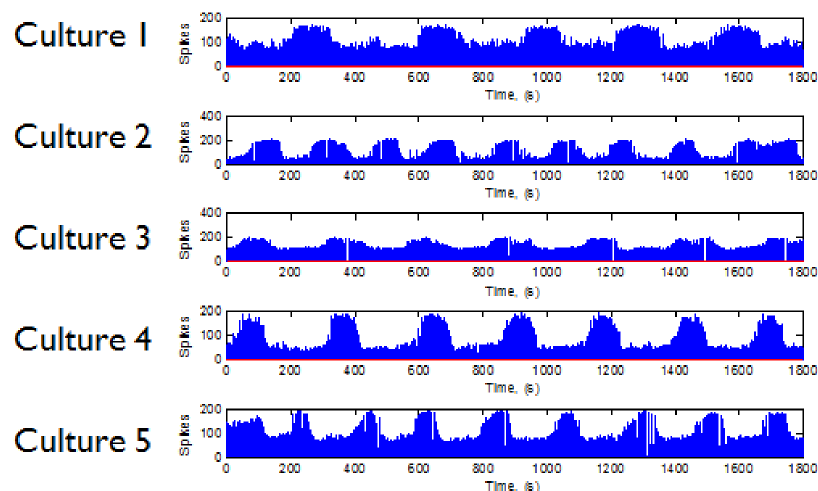

Figure 1 Experimental results showing spiking activities from 5 different cultures with spontaneous activity. Periodical fluctuations of activity over time at a very slow rate (0.001Hz – 0.005 Hz) can be observed in the cultures. The activity time bin is 10ms(Mok et al.[19])

2.2 Theoretical and modelling studies.

2.2.1 Integrate and fire (IF) model

Bruce Knight [20] introduced the term “Integrate and Fire” (IF) neuron in his studies on the encoding for a population of neurons. The basic IF neuron is characterized as a capacitor in parallel with a resistor [4]. The membrane voltage of a neuron is represented by the voltage drop across the capacitor in the IF model. The generalization of a basic IF model allows better representation of a neuron. The Quadratic IF (QIF) model is the simplest of a large number of more realistic neuron models [21, 22].

The basic IF, QIF and its variants are usually used to study the dynamics of networks of spiking neurons. Simulations for a network that has a large number of neurons can be conducted efficiently using the basic IF model. For certain limiting cases, the equations describing the behaviour of a network of IF neurons can be solved exactly.

2.2.2 Other More Detailed Neuron Models

The model introduced by Hodgkin and Huxley [23] describes a neuron by three different ionic currents across the membrane. Activity of the neuron depends on the current components from each of the ionic channels. Gating variables in the equation describe the probability that an ionic channel is open. Morris et al.[24] introduced the Morris–Lecar model which has a two dimensional description of neuronal spike dynamics. The model contains 2

equations, the first describes the membrane potential and the second describes the slow recovery variable.

Izhikevich [25] showed that the above more detailed models take a large number of floating point calculations per iteration. Lob et al.[26] conducted parallel event-driven neural network simulations using the Hodgkin-Huxley neuron model. On a single CPU computer, it took an average of 11 thousand seconds to complete a simulation of 15 seconds of activity in a network of 100 Hodgkin-Huxley neurons. Large clusters of computational units are required to efficiently simulate a network of neurons using the more detailed neuron models [27, 28, 29].

2.2.3 Network Oscillations

Oscillations are important as they are required in the process of information coding and transmission [30, 31, 32]. Synchrony of spiking activities in the network causes the formation of network wide oscillations. The mechanism for the generation of oscillations and synchronization of a network is often the focus of many theoretical studies [33, 34, 35, 36, 37, 38].

A simple network of inhibitory neurons driven by external excitatory inputs under certain condition can exhibit oscillatory events. Brunel et al.[36] showed that it is possible to generate global fast oscillations in a network of inhibitory neurons. Inhibitory coupling in the network can act to synchronize the oscillatory activity in the network [39]. Heterogeneous networks consisting of

inhibitory and excitatory neurons can exhibit a wide range of behaviour depending on the parameters and inputs given to the network [37, 40].

In an analysis to investigate the time structure of activity in neuronal network models, Gerstner [4] showed that a noiseless system was always unstable. The instability may lead to collective oscillations of the entire neuron population or to higher harmonics where all neurons split into several subnetworks. Noise added to the network suppressed fast oscillations and stabilized the system. The period of network oscillations was shown to increase with randomized inputs.

Brunel et al.[36] showed that the period of global oscillations is dependent on the characteristics of the external input. He reported that external noise applied to the networks can produce a phase diffusion of the global oscillations. Likewise, increasing the noise level was also found to strongly damp and decrease the amplitude of the oscillatory activity. Traub et al.[42] constructed a model of a single column thalamocortical network model exhibiting gamma oscillations, sleep spindles, and epileptogenic bursts. Activities were generated from random ectopic axonal action potentials occurring at glutamatergic cells within the column.

2.2.4 Spontaneous Bursting

Modelling studies are conducted by various researchers to explain the phenomena of spontaneous bursting [5, 43, 44, 45]. Based on the studies conducted by Latham et al.[5], one of the parameter that controls the firing

pattern for a neuronal network depends on the percentage of endogenously active neurons. If the percentage of endogenously active neurons falls below a threshold, the network will become silent. When the percentage is above the threshold, activity can be observed from the simulated network. Kudela et al.[44] showed that the balance between the excitation and inhibition in the network is an important factor that modulates the bursting activity of a network. Synaptic properties in the network also play an important role in the generation of a synchronous bursting event. Tsodyks et al.[43] simulated a network capable of generating population burst at particular time intervals. The synaptic characteristics such as the connection strength and synaptic depression between neurons in the model can influence the activity pattern of a network. Volman et al.[45] studied the effect of network structure on the bursting activity in a cultured network. The underlying architecture of a network can influence the pattern of the network activity.

2.2.5 The Present Study

The present study investigates the computer simulation of global oscillations in a network consisting a large number of neurons using the IF neuron model. A simulation algorithm for solving the IF model was created using C++. Parallel solvers were implemented in the algorithm to take advantage of multicore processors available on the market. Using this algorithm, simulations were carried out to study the effect of the different parameters of the IF model on the global oscillations of a network of neurons. An extension of the model by Latham et al.[5]was used to simulate ultra-low

oscillations observed by Mok et al.[18] in dissociated cortical cultures of rat neurons.

CHAPTER 3

METHODOLOGY

3.1 Effect of Parameters of the IF Model on Global Oscillations in a Network of Homogenous Neurons

The differential equation governing the depolarization of the membrane potential $V_i(t)$ for a sparsely connected network composed of N number of identical inhibitory IF neurons is given by

$$\frac{dV_i(t)}{dt} = \frac{1}{\tau}[-(V_i(t) - V_r) + RI_i(t) + J_{ext}r] \tag{1}$$

whereV_r is the resting potential, $RI_i(t)$ is the synaptic input, J_{ext} is the external jump amplitude and r is the external input rate. The network is constructed based on the total number of neurons and the connection probability between the neurons. The network receives internal inputs from a predetermined number of connections from other neurons in the network and external excitatory inputs.

In our study, each neuron within the network receives an external excitation input. External inputs are statistically independent and can be well approximated by a Poisson distribution. The average part, μ_{ext} and the fluctuating part, σ_{ext} of the external synaptic input are explicitly given by.

$$\mu_{ext} = J_{ext} C_{ext} \nu_{ext} \tau \quad (2a)$$
$$\sigma_{ext} = J_{ext} \sqrt[2]{C_{ext} \nu_{ext} \tau} \quad (2b)$$

where τ is the membrane time constant, J_{ext} is the external jump amplitude and$C_{ext}\nu_{ext}$ is the mean firing rate.

A Poisson random process is used to generate the external input rates, r. The rates are multiplied with the external input voltage jumpJ_{ext} to simulate external excitation of the membrane. Internal activities are conveyed through the term $RI(t)$. The representation of the internal input is given by

$$RI_i(t) = \sum_j w_{ij} J_j \sum_k \delta(t - t_j^k - D) \quad (3)$$

where w_{ij} is the synaptic weight, J_j is the internal voltage jump, t_j^k is the emission time of k-the spike at neuron j and D is the synaptic delay.

The network parameters are varied to study the behaviour of network. The range of variation of the parameters is shown in Table 1.

Parameters	Range	
	Min	Max
Network Size	1000	250000
External Noise	1mV	12mv
Connection probability	0.05	1
Synaptic Delay	1.5ms	4ms
Synaptic Weight	0.2	1

Table 1 Parameters for simulation of homogenous network of inhibitory neuron

3.2 Simulation Algorithm

The simulation algorithm is developed using C++ to take advantage of the multicore processor architecture. Many of the calculations in the simulation of a neuronal network can be performed simultaneously. The differential equation governing the evolution of the membrane voltage and synaptic current can be solved concurrently by means of parallel computing. The speed of computation can be increase with the implementation of parallel computing [46, 47].

The Open Multi-Processing (OpenMP) [48] application programming interface (API) is used to facilitate the implementation of parallel computing. OpenMP allows multiprocessing on a shared memory system. A master thread can be parallelised using an OpenMP compiler directive, creating a numbers of slave threads to work on a task. After all the slave threads have executed the task, the slave treads will join back to the master thread to allow continuation of program execution. Figure 2 illustrates a master thread that forks off to multiple slave treads during an execution of an OpenMP PARALLEL directive. OpenMP is commonly used to implement numerical simulation which requires a large number of calculations [49, 50, 51, 52].

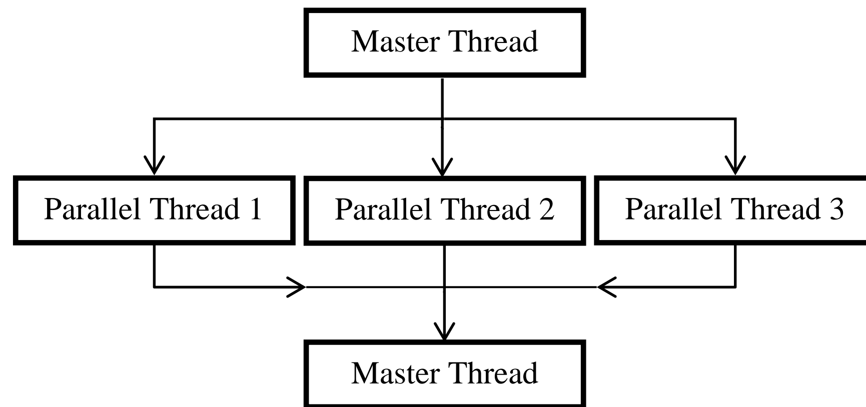

Figure 2 OpenMP of parallel execution block

Another advantage of using C++ is the ability to use the Object-oriented Programming (OOP) method for the implementation of neurons [53]. An object can be represented as a class in a program. A class can contain both data and function. By treating a neuron as a functional unit, a neuron class can be created to represent the parameters and features of the particular neuron. OOP is widely used in the creation of simulation software for neurons and neural networks [54, 55, 56, 57].

A numerical procedure is implemented to solve the IF first-order differential equation (ODE). Small time steps are taken to solve the IF equation to get the evolution of the membrane potential. There are a number of numerical methods that can be applied to solve the ODE [58]. For the current simulation, the numerical method used for solving of the ODE is the Fourth-Order Runge-Kutta Method [59].

3.2.1 Simulation Algorithm Development

The equation describing the dynamics of the membrane is represented as a class. The class will hold the equation in its operator. The function and variable describing the equation are implemented as a member of the class. The code below shows an example for the implementation of an equation class.

```
1    class LinIF{
2        private:
3            double tau, Er, v_thres;
4        public:
5            double v, J;
6            bool type
7
8            LinIF(double e):
9                v(e),
10               Er(-65),
11               tau(20),
12               J(0),
13               v_thres(-55){}
14
15           ~LinIF(){}
16
17           double operator()(double y, double t){
18               return (1/tau) * (-(y-Er)) + J;
19           }
20
21           bool update(){
22               if (v >= v_thres){
23                   v = -65;
24                   return true;
25               }
26               else
27                   return false;
28           }
29   };
```

Class in Line 1 of the code is the keyword for the declaration of an expended data structure in C++. The variable describing the equation will be

stored in the data members declared from Line 2 to Line 5 of the code. The equation is implemented in the default operator with 2 inputs y and t. The last part of the class, line 20 to 27 contain a function to check the reset condition of the IF model. When the voltage crosses the threshold, the membrane potential is reset back to the resting potential.

The next step in the algorithm is the implementation of the numerical solution. A template function is created to solve the equation class. The function will run the equation through all the necessary steps in solving the differential equation.

```
1   template <typename function>double runge_kutta_4th(function equation,
    double initial, double t, double dt){
2           double k1 = equation(initial, t);
3           double k2 = equation(initial + 0.5 * k1 * dt, t + 0.5 *dt);
4           double k3 = equation(initial + 0.5 * k2 * dt, t + 0.5 *dt);
5           double k4 = equation(initial + k3 * dt, t + dt);
6           return initial + (k1 + 2*k2 + 2*k3 + k4)/6 * dt;
7   }
```

The code above shows an example of implementation of Runge-Kutta method in solving the differential equation. The beginning, midpoint and endpoint estimates are calculated and stored in variables k1 to k4. Knowing the function, the initial value, the time and the time step size, the average slopes from k1 to k4 are calculated and added to the initial values to get the results at time $t = t_{n+1}$.

The main simulation algorithm will create objects based on the neuron class. Information describing the connectivity, type and others properties of the

network for each individual neuron will be stored into the vectors. Network information is generated based on the required parameters for every particular simulation. Data in the vector will be recalled during the simulation steps to determine the network characteristics of a neuron.

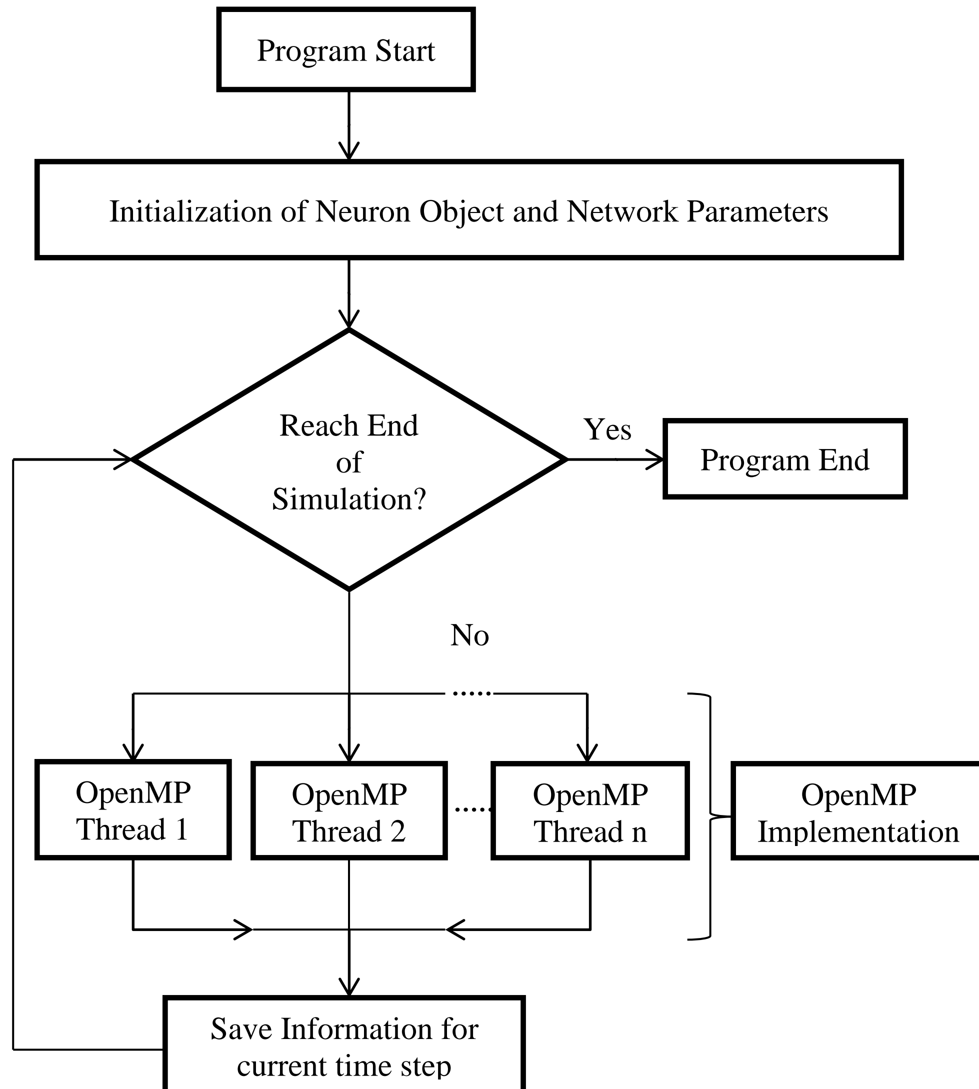

Figure 3 Flowchart for simulation algorithm

Figure 3 shows a simplified flowchart describing the overall simulation algorithm. All the required objects and parameters are initialized at the start of the program. The simulation is carried out for every time step, Δt until a pre-set

time limit. Individual neuron equations are solved in the iteration loop using OpenMP to distribute the processing load. With a multicore processor, the threads created can be run simultaneously to shorten the time required to complete the simulation. The output for every time steps will be recorded and saved to the hard disk as CSV files. The summation of spikes and the membrane voltage are some of the outputs that can be saved in this step.

The completed code is compiled using g++ with flags, -O2 and -fopenmp under Ubuntu. Level 2 optimization of the code will increase the speed of the program and it is done automatically by the compiler. The fopenmp flag is needed to enable the use of OpenMP directive in the coding.

3.2.2 Implementation of Synaptic Delay

Synaptic delays are implemented in the code using double ended queue (deque). Data in the deque can be added and removed from the head and the end of the list. Spike information from the presynaptic neuron is stored at the end of the deque. The length of the deque, n is obtained by dividing the desired time of delay by the time step. Spikes that arrive at the postsynaptic neuron is removed from the start of deque and used for processing at that particular simulation time step.

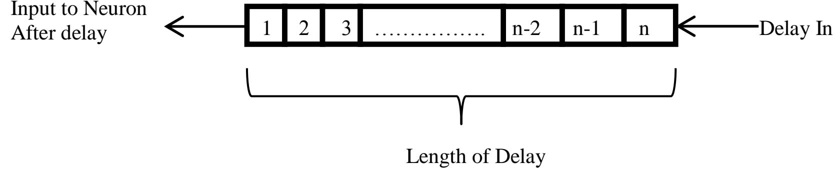

Figure 4 Implementation of synaptic delay

Figure 4 shows the flow of a deque implemented for a synapse. Each synapse will have a unique deque assigned to it to store the information on its synaptic delay.

3.2.3 Implementation of the After Hyperpolarization current

The hyperpolarization of a neuron occurs after spike generation due to open potassium ion channel and the influx of calcium ion channel. It causes the membrane potential of a neuron to fall below the resting potential. The phenomena of After Hyperpolarization (I_{AHP}) current can bemodelled using the equation,

$$I_{AHP} = \overline{g}_{AHP}(u - \varepsilon_K) \tag{4a}$$

$$\frac{dg_{AHP}}{dt} = -\frac{g_{AHP}}{\tau_{AHP}} + \delta_{AHP} g \sum_{\mu} \delta(t - t^{\mu}) \tag{4b}$$

where, g_{AHP} determines the amplitude of conductance, δ_{AHP} determines the change in conductance and t^{μ} is the time a spiking event occurs.

The spiking of a neuron at timet^{μ} will trigger an increase in g_{AHP}. This causes the generation of the AHP current on the membrane potential at the time

of spike. This current causes the membrane potential to drop to a lower point when the membrane recovers after spiking.

The AHP current is modelled as an object in the simulation algorithm. The code for the differential equation is shown below. The differential part of the equation is described in the operator of the class. The Runge-Kutta4th order numerical scheme will be used to solve the differential equation object.

```
1   class neuron_internal_dynamics{
2       private:
3           double tau, wi, del, z;
4
5       neuron_internal_dynamics(double tau, double del):
6           tau(tau),
7           wi(0),
8           del(del),
9           z(0){}
10
11      ~neuron_internal_dynamics(){}
12
13      double operator()(double y, double t){
14          return -(y/tau) + del * wi;
15      }
16  };
```

3.2.4 Simulation in Networks with Excitatory Neurons

The simulation code is modified further to facilitate the simulation of networks with excitatory neurons. The neuron class is modified to accommodate a parameter for excitatory or inhibitory neurons. A Boolean variable is used to determine the type of neuron. During simulations, the neuron type is checked to ensure that the correct routine for inhibitory or excitatory neurons is called. The accuracy of the current simulation algorithm

is checked against the results provided by Brunel et al.[37] for a network of sparsely connected excitatory and inhibitory neurons.

3.3 Simulation of Ultra-Slow Spontaneous Oscillation

Spontaneous activities in a neuronal network have already been modelled by Latham et al.[5]. I will extend this model to simulate ultra-slow spontaneous oscillations observed by Mok et al.[18] in cortical cultures of rat neurons. Ultra-slow changes to the firing rate were observed in dissociated cortical cultures grown on an 8x8 grid MEA. Global activity is obtained by summing up all the activity present on each of the individual electrodes.

Ultra-slow spontaneous oscillatory patterns are observed as a result of fluctuations in the number of neurons activated at the peaks and troughs. Figure shows the differences in the number of spikes between the time ranges from 200s to 210s and from 400s to 410s for culture 4. Fewer neurons are active during bursting events in Figure 5 (a) compare to Figure 5 (b).

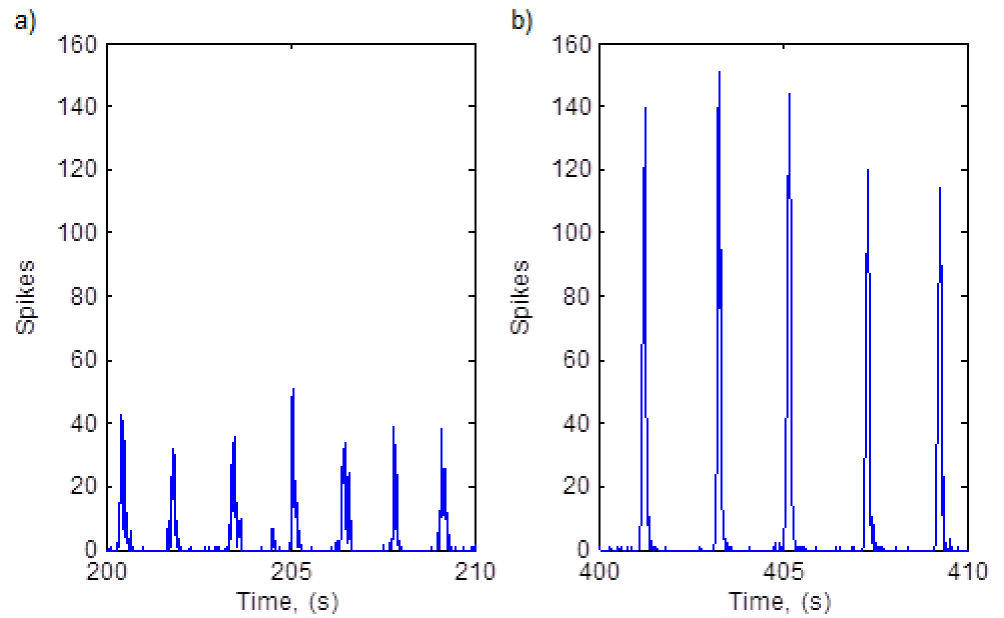

Figure 5 Experimental data showing spontaneous bursting activity for 10s in a culture of cortical neurons. a) Activity at troughs of ultra-slow spontaneous oscillations b) Activity at peaks of ultra-slow spontaneous oscillations. The activity time bin is 10ms(Mok et al. [19])

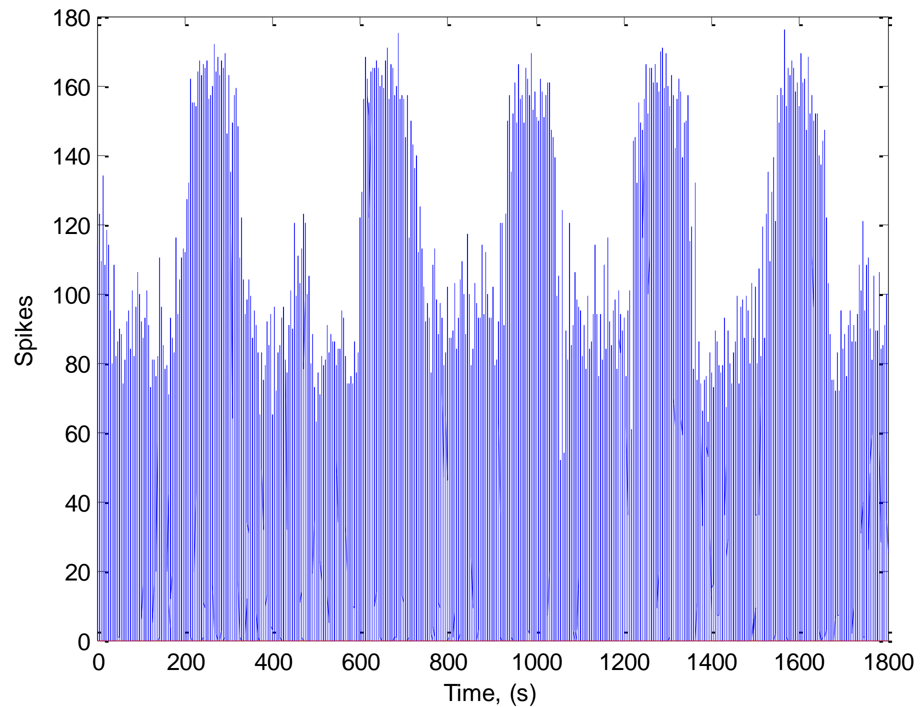

Figure 6 Experimental results showing ultra-slow changes in spontaneous activity. The activity time bin is 10ms (Mok et al. [19])

Figure 6 shows a sample data obtained from the experiment showing ultra-slow periodical changes in the spontaneous activity. Two distinct firing rates can be observed in the sample data above. About 90 spikes per time bin can be observed at periods with low firing rates and about 160 spikes per time bin at times with high firing rates.

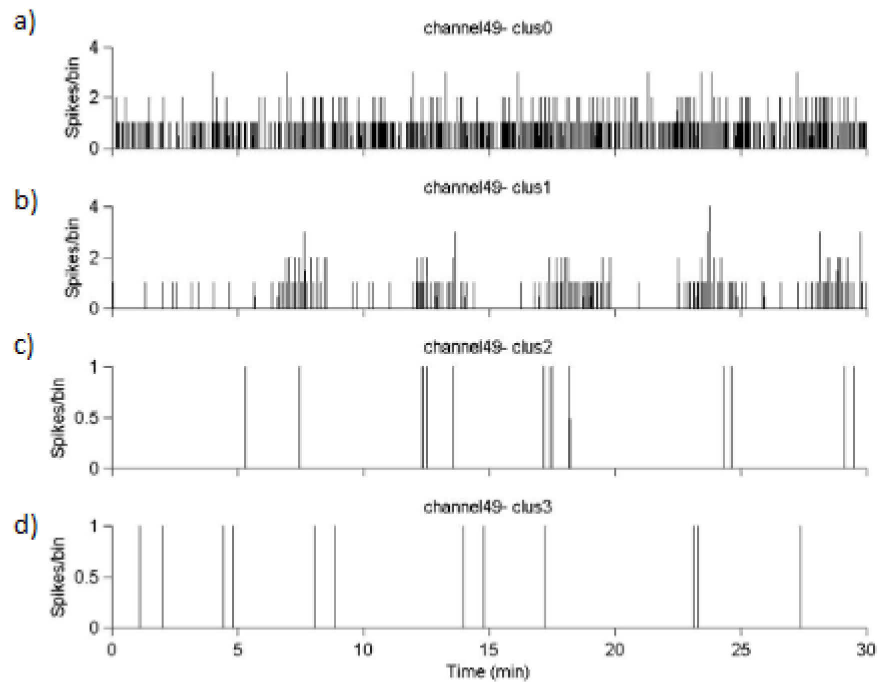

Figure 7 Spike sorting results on one of the electrodes a) neuron fires throughout the whole recording. b) neuron fires only at the peaks c, d) neuron with low level of activity. The activity time bin is 10ms (Mok et al. [19])

Why the neurons exhibit this ultra-slow spontaneous oscillating behaviour is not fully understood. However when spike sorting algorithms [60, 61] were used to separate the activities from different neurons, the experimental data showed that some neurons fire continuously while others fire only at the peaks and not at the troughs.

The time evolution equation [5] for the membrane potential of neuron $v_i(t)$ for modelling activity observed in the experiment is

$$T_m \frac{dv_i(t)}{dt} = \alpha(v_i(t) - v_r)(v_i(t) - v_t) + I_{a,i}(t) - \left(g_{k,i} + g_{k-ca,i}\right)(v_i(t) - \varepsilon_k) - \left(v_i(t)\tilde{I}_i - \tilde{I}_{\varepsilon,i}\right) \quad (5a)$$

$$\frac{d\tilde{I}_i}{dt} = -\frac{\tilde{I}_i}{\tau_s} + r_s \sum_{j,u} w_{ij}\delta(t - t_j^u) \quad (5b)$$

$$\frac{d\tilde{I}_{\varepsilon,i}}{dt} = -\frac{\tilde{I}_{\varepsilon,i}}{\tau_s} + r_s \sum_{j,u} w_{ij}\delta(t - t_j^u) \quad (5c)$$

$$\frac{dg_{k,i}}{dt} = -\frac{g_{k,i}}{\tau_{K,i}} + \delta_{k,i} \sum_{\mu} \delta(t - t_i^u) \quad (5d)$$

$$\frac{dg_{k-Ca,i}}{dt} = -\frac{g_{k-Ca,i}}{\tau_{k-Ca,i}} + \delta_{k-Ca,i} \sum_{\mu} \delta(t - t_i^u) \quad (5e)$$

where α determines the rate of change for v,v_r is the resting potential;$v_t$is the threshold potential, $I_{a,i}(t)$controls the fraction of endogenously active cells,ε_k is the potassium reversal potential, $g_{k,i}$ and $g_{k-ca,i}$ are the potassium conductance,$\tilde{I}_i$ and $\tilde{I}_{\varepsilon,i}$ describe the synaptic input currents to the neuron. The time evolution equation of $\tilde{I}_i$, $g_{k,i}$, $g_{k-Ca,i}$ and $\tilde{I}_{\varepsilon,i}$ are given by equations 5b-5e.A Runge-Kutta4thorder solver, created in C++, was used to solve the differential equations for the network.

The time period of these ultra-slow spontaneous oscillations is much larger than previously reported and these ultra-slow spontaneous oscillations are not affected by the parameters of the standard IF model. It is likely that these oscillations are controlled by other biochemical processes and/or network

structure in the neuronal culture. In order to overcome this difficulty, I introduce an additional equation,

$$\frac{d\varphi}{dt} = -\frac{\varphi}{\tau_\varphi} + \alpha \sum_{\mu,i=0}^{i=n} \delta(t - t_i^u) \quad (6)$$

that describes the generation and dissipation of an inhibiting property (φ).The generation of the inhibiting property is proportional to the firing activity and represented by a parameter α and the dissipation of the inhibiting property is represented by a decay coefficient, τ_φ when the inhibiting property reaches a higher threshold value, the neurons within the subnetwork stop firing. The neurons start to fire again when it has dissipated to a lower threshold value.

It has been previously reported [45] that for some large networks, the synchronized bursting events might be classified as belonging to several distinct subnetworks with each subnetwork representing a synchronized bursting event with a well-defined spatio-temporal internal structure. From the spike sorting results [19], I postulate here that there are two or more such distinct subnetworks. The generation of the inhibiting property in each subnetwork is proportional to the firing rate but the decay rate is different depending on the local subnetwork properties. In this manner we will have neurons in each subnetwork firing at a different rate. For the purpose of the present simulations, the neurons are divided into 2 subnetworks. By proper adjustment of the parameters α and τ_φ we can obtain a subnetwork of neurons that fires continuously and another that fires only at the peaks and not at the

troughs. The simulation algorithm described in section 3.2 will be utilized for the simulation of these ultra-slow spontaneous oscillations.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Simulation Algorithm

Sets of C++ libraries were created for the simulation algorithm. Libraries containing the equations describing the dynamics of the membrane were included in the main file of the simulation programs. The developed libraries were tested for speed and accuracy of simulation. The speeds of simulations for optimized and OpenMP were compared for different sets of simulations. The results of the simulations were recorded on to the hard disk using the CSV format. Information such as the number of spikes, membrane potential and variable changes during the simulations were selected as the output of the simulation code.

4.1.1 Comparison of Single Neuron Simulations with Exact Solutions

The accuracy of the simulation algorithm was tested by comparing the simulation results with the exact analytical solutions for the basic IF neuron model and also for the QIF neuron model. For the basic IF model, the error is taken as the difference between the results of the numerical simulation of

$$\frac{dv}{dt} = \frac{1}{\tau}(-(v - v_r) + RI) \tag{7}$$

and the exact analytical solution given by

$$v(t) = (RI + v_r) - (RI + v_r - v_0)e^{-(\frac{t_c}{\tau})} \quad (8)$$

For equation 7, the following values were used: τ = 20ms, v_r = -65mV and RI = 15mV and for equation 8, $v_0 = v_r$ and $t_c = t - t_s$ where t_s is the time when the spike last occurred. The threshold voltage for spike generation is set at -55mV. When a spike occurs, the membrane voltage at the current time step will be reset back to the resting potential at -65mV.

Figure 8 shows a comparison of the results of the membrane voltage generated by the numerical simulation and the exact analytical solution. The mean square error obtained from the numerical simulation using the Runge-Kutta method compared to the exact analytical solution is 7.9113×10^{-10} showing that the numerical simulation is highly accurate.

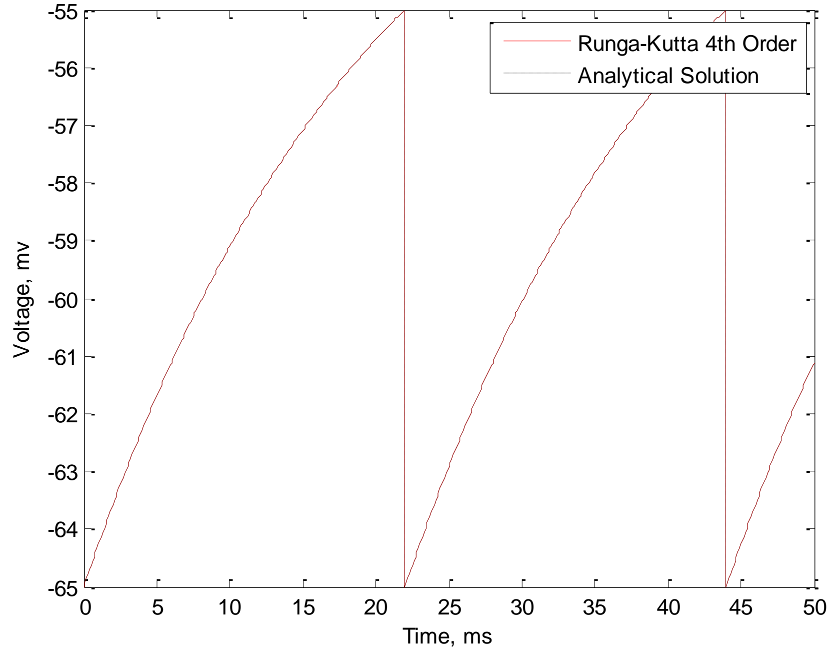

Figure 8 Comparison of simulated membrane voltage of single IF neuron model with exact solution

The QIF model used for the comparison is described by

$$\frac{dv}{dt} = \frac{1}{\tau}(\alpha_0(v - v_r)(v - v_t) + RI) \tag{9}$$

and the exact analytical solution given by

$$\boldsymbol{v(t)} = \left(\frac{(\boldsymbol{4ac - b^2})\tan\left(\frac{t_c\sqrt[2]{(4ac-b^2)}}{2\tau}\right) + (\boldsymbol{2av_0 + b})\sqrt[2]{(\boldsymbol{4ac - b^2})}}{\boldsymbol{2a}\left(\sqrt[2]{(\boldsymbol{4ac - b^2})} - (\boldsymbol{2av_0 + b})\tan\left(\frac{t_c\sqrt[2]{(4ac-b^2)}}{2\tau}\right)\right)} - \frac{\boldsymbol{b}}{\boldsymbol{2a}} \right) \tag{10a}$$

$$a = \alpha_0 \tag{10b}$$
$$b = -\alpha_0(v_t + v_r) \tag{10c}$$
$$c = \alpha_0 v_t v_r + RI \tag{10d}$$

For the numerical simulations, the following values were used: $\alpha_0 = 0.2$, $\tau = 20$mV, v_r = -65mV and RI = 15mV and for the analytical solution, $v_0 = v_r$andt_c = t-t_s where t_c = t-t_s is the time when the spike last occurred. The threshold voltage for spike generation is set at 55 mV and the peak of the spike set at 0mV. When a spike occurs, the membrane voltage at the current time step will be reset back to the resting potential at -65mV.

Figure 9 shows a comparison of the results of the membrane voltage generated by the numerical simulation and the exact analytical solution. The mean square error obtained from the numerical simulation using the Runge-Kutta method compared to the exact analytical solution is 1.1457 x 10^{-5} showing that the numerical simulation is highly accurate.

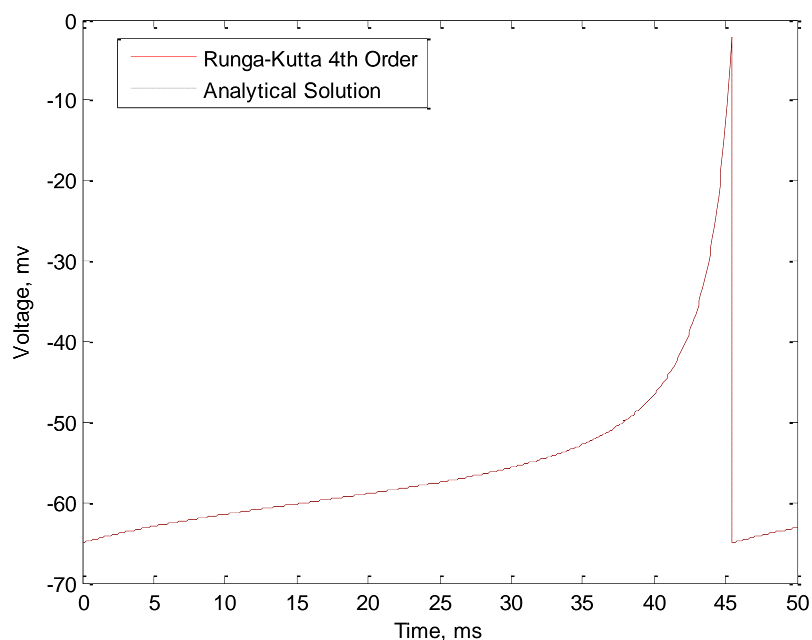

Figure 9 Comparison of simulated membrane voltage of single QIF neuron with the exact analytical solution

4.1.2 Network Simulation

A network of neurons was created to test for the propagation of error due to the interconnectivity of neurons. Table 2 shows the parameters of the network.

	Network Parameter
Number of Neuron	5000
Connection Per Neuron	1000
Neuron Type	Inhibitory

Table 2 Network parameters for IF and QIF networks

For the simulation of a network of IF neurons, equation 7 is numerically evaluated using the Runge-Kutta4th order method. In the network, a presynaptic neuron spike will produce a potential jump J, of -1mV at the postsynaptic neurons. The sum of the spikes for a simulation of 2000ms is calculated for the network. The results are compared with those obtained from a similar simulation using equation 8 (the exact analytical solution of equation 7). Figure 10 shows the number of spikes computed from network simulations for the IF model using equations 7 and 8. The number of spikes computed from both cases is identical, showing that there is no propagation of errors in the simulation.

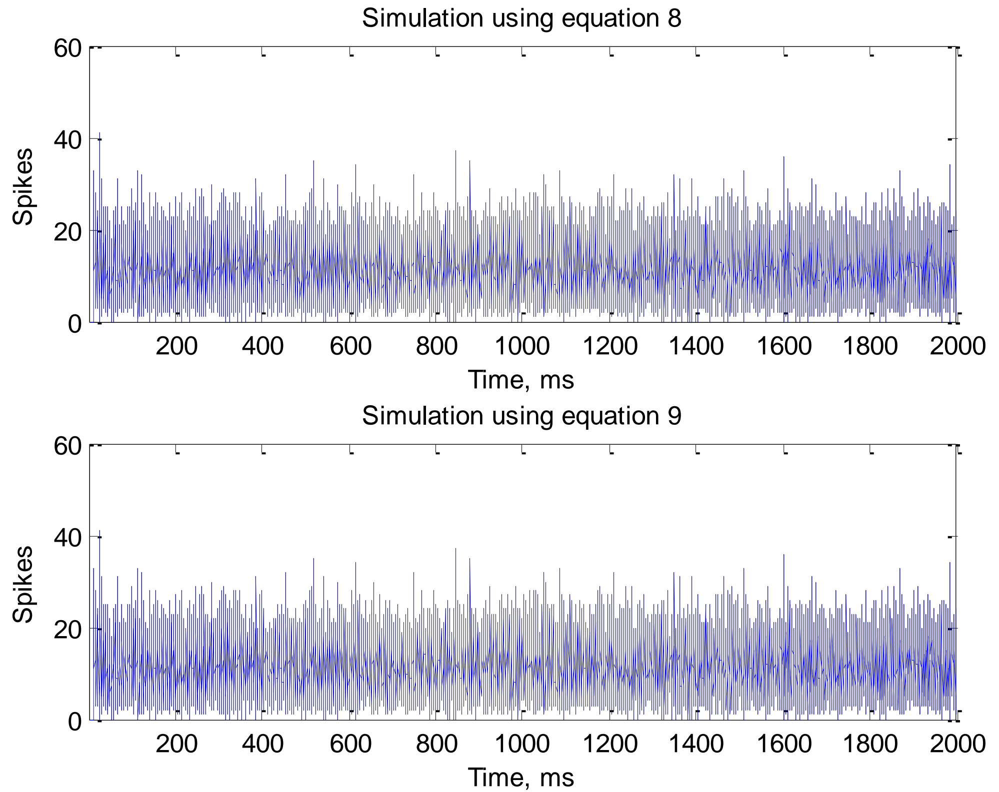

Figure 10 Comparison of results from simulation of a network of IF neurons

Figure 11 shows the number of spikes computed from network simulations for the QIF model using equations 9 and 10. The number of spikes computed from both cases is identical, showing that there is no propagation of errors in the simulation.

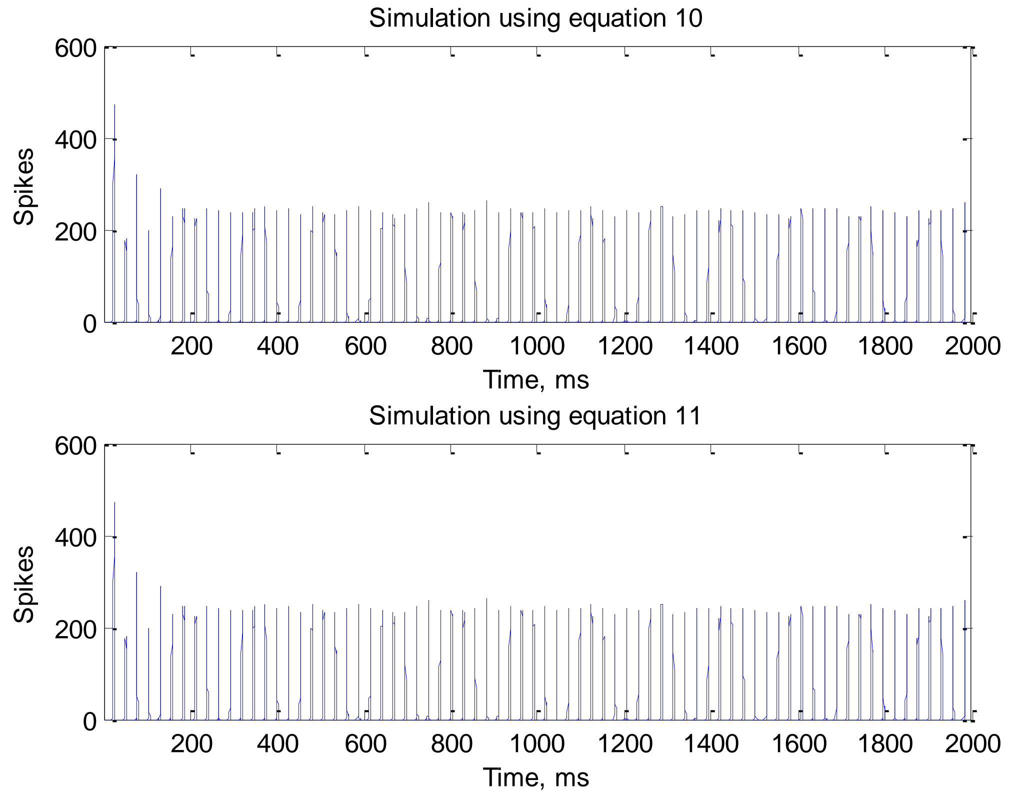

Figure 11 Results from simulation of QIF neurons network using different methods

4.1.3 Effect of Parallel Processing on Simulation Speed

Numerical simulations were carried out for networks of 10000 and 20000 QIF neurons with and without OpenMP implementation. The parameters for the simulations are given in Table 3.

	Network Parameter
Number of Neuron	Varies
Connection Per Neuron	0.2 * Number of Neuron
Neuron Type	Inhibitory
External Input	Fixed

Table 3 Network parameters for comparison of time taken for simulation (QIF)

Figure 12 shows a comparison of the time taken for the simulation. With OpenMP implementation, a 5.05% speed increase for the simulation of 10000

neurons and 3.71% speed increase for the simulation of 20000 neurons can be observed.

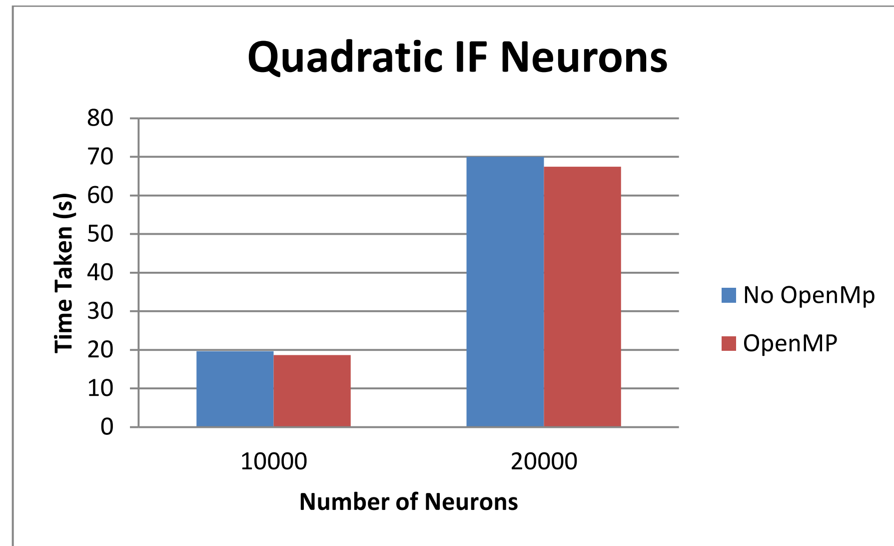

Figure 12 Speed of simulation for a network of QIF Neurons

Test for speed of simulations were also carried out for a network of QIF neurons with AHP. The parameters in Table 3 were used. An increase of 23.33% in the simulation speed for 10000 neurons and 18.26% increase in simulation speed for 20000 neurons can be observed.

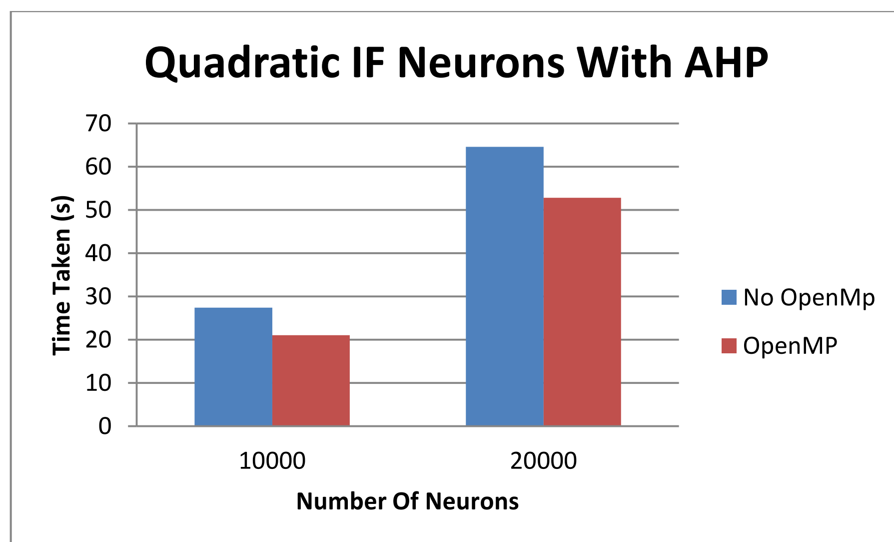

Figure 13 Speed of simulation for a network of QIF with AHP neurons

The overall increase in simulation speed is much higher for the QIF with AHP neurons compare to QIF neurons only. This is due to the overhead needed by OpenMP to create individual threads for the execution of the simulations. Processing time is used up to create threads and forks in the program for parallel processing. Time is also taken up when the data for threads is combined at the end of the parallel fork. Containing only a single differential equation, the network of QIF neurons only receive minor speed improvement for the simulation due to the time taken by OpenMP. Conversely, the simulation for a network of QIF with AHP neurons shows a more significant improvement in computational speed. The overhead imparted with the use of OpenMP for this simulation is similar to the simulation of QIF neurons. With a more complex equation, simulation of a network of QIF with AHP neurons benefit more from the implementation parallel processing using OpenMP.

4.1.4 Simulations of a Network of Inhibitory and Excitatory Neurons

Using the current simulation algorithm, I attempt to reproduce the results as reported by Brunel [37] for a network of inhibitory and excitatory IF neurons. The parameters in Table 4 are used for the simulation. The amplitude for the inhibitory post synaptic potential and external noise are varied in the simulation. The amplitude of inhibitory post synaptic potential is varied using the term g as shown in Table 4.

Parameters	Value
PSP amplitude for excitatory synapses	0.1mV
PSP amplitude for inhibitory synapses	-g*0.1mV
Synaptic Delay	2ms
Firing threshold	20mV
Resting potential	10mV
Membrane time constant	20ms
Number of excitatory neurons	10,000
Number of inhibitory neurons	2,500
Inhibitory Connections	250
Excitatory Connection	1000

Table 4 Network parameter for simulation of network of inhibitory and excitatory neurons

Figure 14 to Figure 17 show the activity patterns as reported by Brunel [3] can be reproduced using the current simulation algorithm.

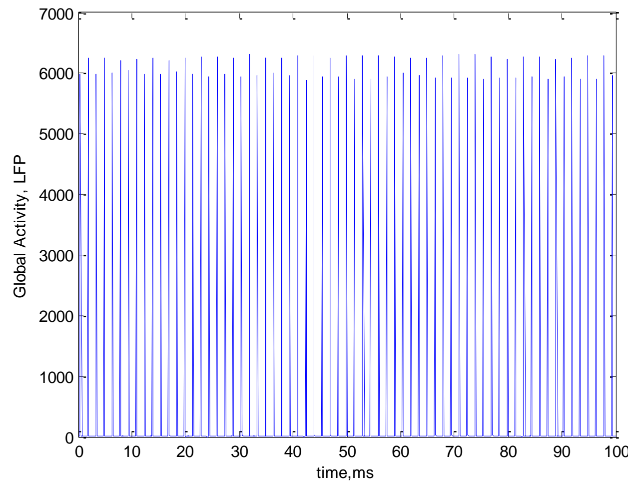

Figure 14 Synchronous Regular State (SR), g = 3, External noise =2 mV. The activity time bin is 10ms

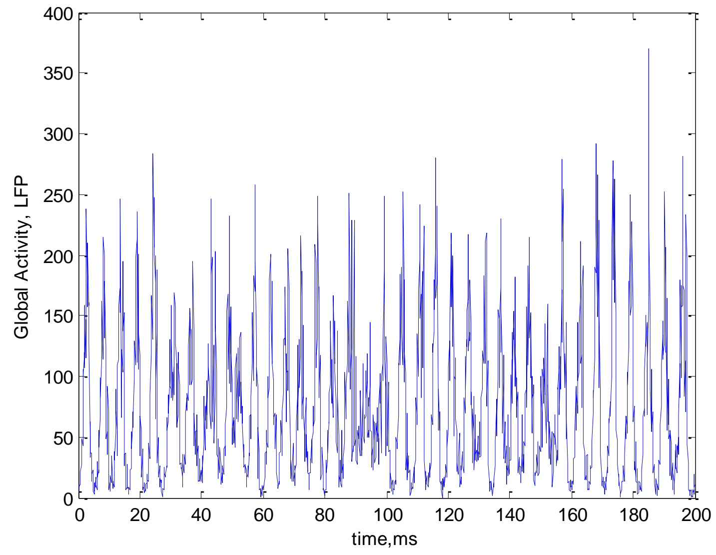

Figure 15 Synchronous Irregular State (SI), Fast Oscillation, g = 6; External noise = 3 mV. The activity time bin is 10ms

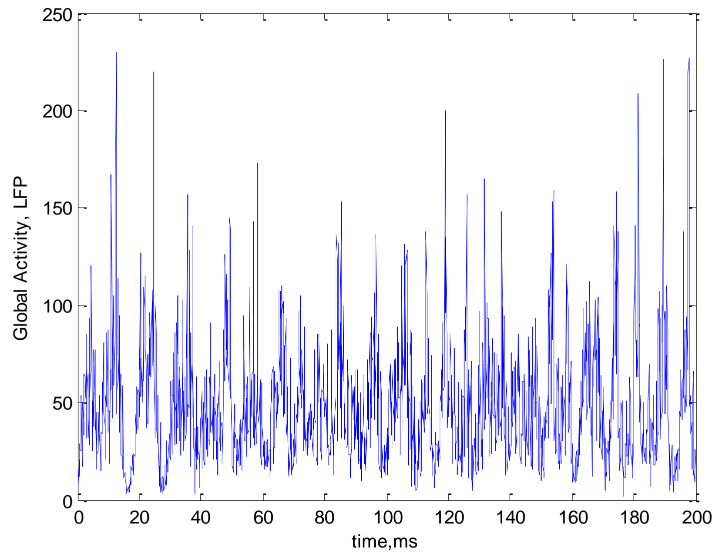

Figure 16 Asynchronous Irregular State (AI), g = 5; External noise = 2 mV. The activity time bin is 10ms

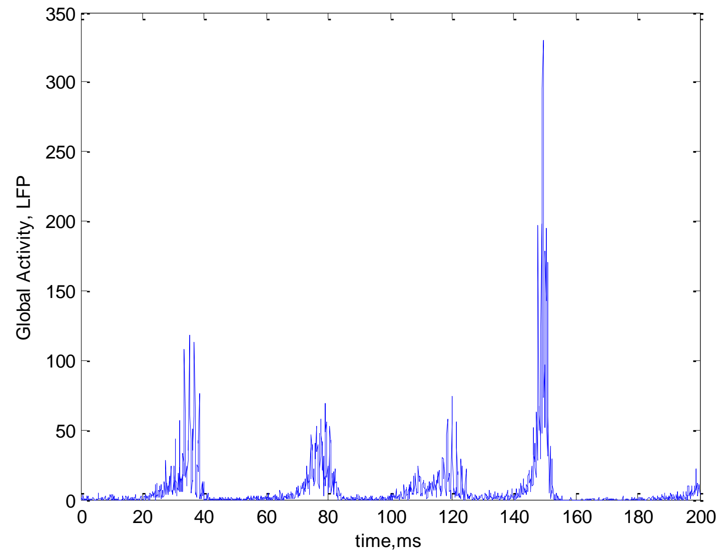

Figure 17 Synchronous Irregular State (SI), Slow Oscillation, g = 4.5; External noise = 1.3 mV. The activity time bin is 10ms

4.2 Effects of the Parameters of the IF Model on the Global Oscillations in a Network of Homogenous Neurons

Simulations for a network of homogenous neurons are conducted and the parameters of the IF model are varied to study the effect of parameters on the network oscillations.

4.2.1 Network Size

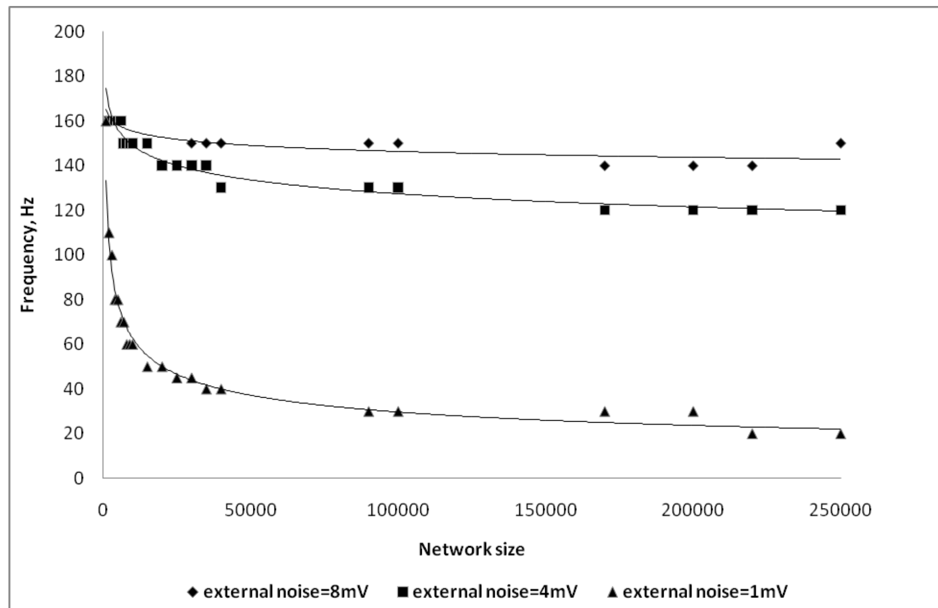

Figure 18 Frequency of network oscillation for different network size

The size of the neuronal network is increased while other parameters of the network are kept constant. The other parameters of the network are $\varepsilon = 0.2$, $\tau = 20ms$, $D = 2ms$, $J_{ij} = 0.1mV$, and $\mu_{ext} = 25mV$. Figure 18 shows the variation of the network oscillation frequency with network size 3 noise levels of 1mV, 4 mV and 8 mV. The frequency of network oscillations decreases sharply as the number of neurons N in the network increases for N less than 50,000. For N greater than 100,000, the frequency of network oscillations did not change significantly. It may be observed that the frequency of network oscillations increases with external noise from 1 mV to 8 mV.

4.2.2 External Noise

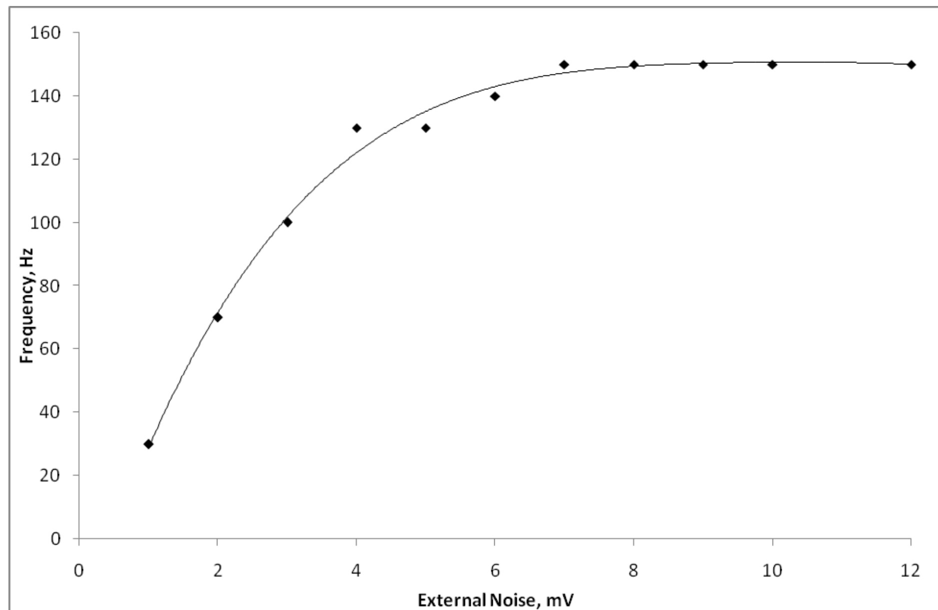

Figure 19 Frequency of network oscillation for different external noise

For this simulation, N was chosen to be 130,000 neurons to eliminate the effect of network size on the frequency of network oscillations. The effect of different external noise levels to the network is investigated. The following parameters are kept constant: N=130,000 neuron, τ = 20ms, ε = 0.2, D = 2ms, J_{ij} = 0.1mV, and μ_{ext} = 25mV. Figure 19 shows that the frequency of network oscillation increases with external noise until 8 mV after which it remains relatively constant.

4.2.3 Connection Probability

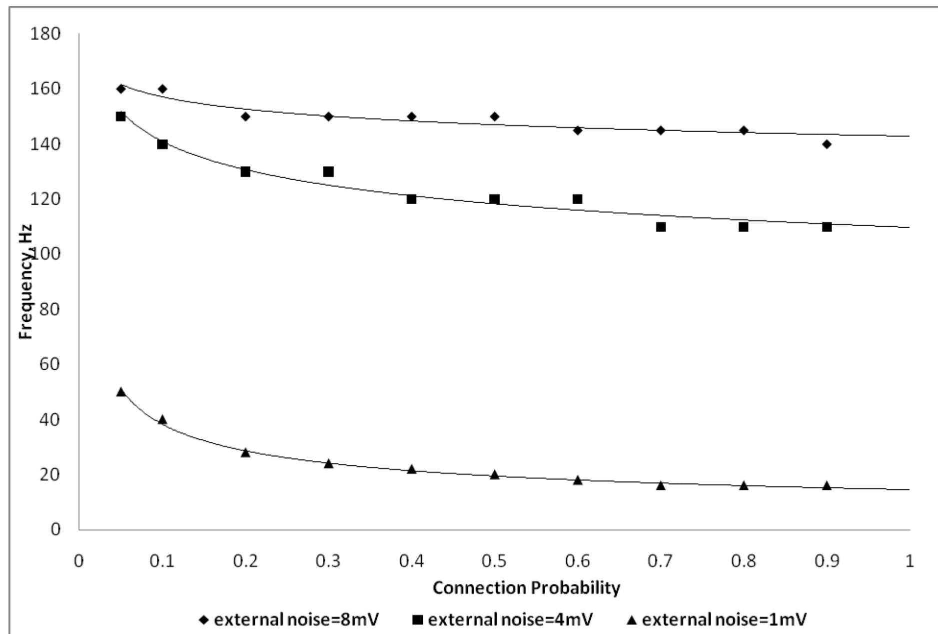

Figure 20 Frequency of network oscillation for different connection probability

Different connection probability for the network is simulated. A network of 130,000 neurons is chosen to eliminate the effect of N on the network oscillations. For each curve, with the exception of connection probability, the other parameters remain the same: τ = 20ms, D = 2ms, J_{ij} = 0.1mV, and μ_{ext} = 25mV. Three curves are generated for external noise, σ_{ext} = 1mV, σ_{ext} = 4mV and σ_{ext} = 8mV. Figure 20 shows that the frequency of oscillation decreases slightly when the network varies from sparsely to highly connected.

4.2.4 Synaptic Delay

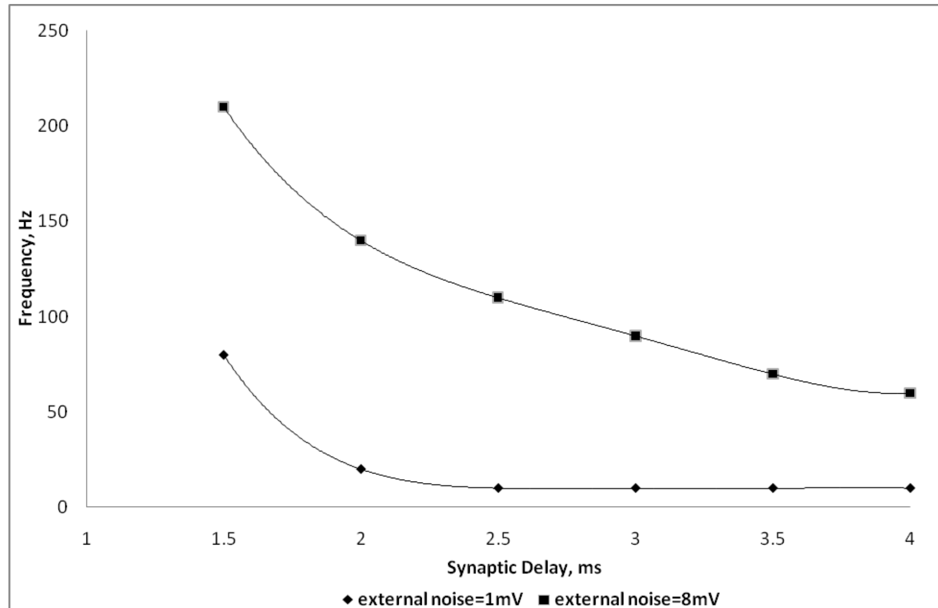

Figure 21 Frequency of network oscillation for different synaptic delay

The synaptic delay of the network is varied. The network size of 130,000 with connection ratio of 0.2 is used. Other parameters of the network are τ = 20ms, J_{ij} = 0.1mV and μ_{ext} = 25mV. External noise level of σ_{ext} = 1mV and σ_{ext} = 8mV is used for the simulation. Figure 21 shows the results of the simulation. At low synaptic delays, the frequency of the oscillation is very high. With higher delay times, the frequency of network oscillation decreases.

4.2.5 Synaptic Weight

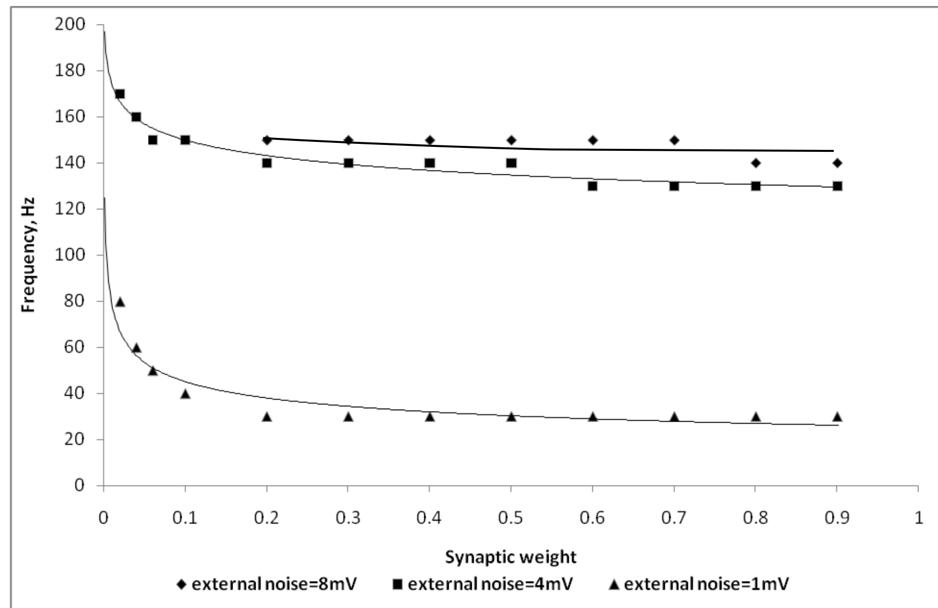

Figure 22 Frequency of network oscillation for different synaptic weight

The synaptic weights of the network are varied from 0.02 to 0.9. The parameters of the network are N = 130,000, $\varepsilon = 0.2$, $\tau = 20ms$, $J_{ij} = 0.1mV$ and $\mu_{ext} = 25mV$. External noise level of σ_{ext} = 1mV, 4mV and 8mV are used for the simulation. The oscillation frequency remains relatively constant for higher synaptic weights. At lower synaptic weights, the stability of the oscillation is affected causing a noise-like network activity to occur. Figure 22 shows the result for the simulation.

4.3 Simulation of Spontaneous Activity

Spontaneous activity in cultures of cortical neurons using the model described in chapter 3.3 will be simulated. By varying the number of endogenous neurons in the network, regular spontaneous activity can be obtained.

To simulate ultra-slow spontaneous oscillations in cortical cultures [18], a network of 2 subnetwork of neurons will be created. An inhibiting property is introduced in each subnetwork. If the properties of the 2 subnetworks are different, it is possible to simulate one subnetwork that will fire continuously while another subnetwork fires periodically

4.3.1 Spontaneous Activity

A network to simulate spontaneous activity is created based on the model described by equation 5a and with parameters given in Table 5. The parameters for the neurons and synapses are as described by Latham et al. [5]. In the model, an applied current I_a is introduced to each of the neurons. The applied current is chosen randomly from a uniform distribution between 0 and I_{max}. A fraction of the neurons will be endogenously active based on the value chosen for I_{max}, $\tilde{I}$ and $\tilde{I}$ describe the synaptic currents between the neurons while g_k and g_{k-ca} represent the potassium conductance for the slow and fast after hyperpolarization currents.

Parameter	Value
T_m	10ms
v_r	-65mV
v_t	-50mV
α	1/15
ε_k	-80mV
τ_s	3ms
r_s	0.1mS
τ_K	30ms
δ_k	1mS
τ_{k-Ca}	3000ms
δ_{k-Ca}	0.2mS

Table 5 Neuron parameters for simulation of spontaneous activity

Parameter	Value
Number of Neuron	10000
Connection Per Neuron	2000
Excitatory Neuron	20%
Inhibitory Neuron	80%
V_{EPSP}	1mV
V_{IPSP}	-1.5mV
Connectivity Bias	Varied
Imax	Varied

Table 6 Network parameters for simulation of spontaneous activity

The network parameters for cortical cultures are listed in Table 6. w_{ij} (Latham et al. [5]) is calculated from V_{psp} using

$$w_{ij} = \frac{V_{psp,ij}}{\varepsilon_j} \frac{1}{r_s} \frac{\tau_m}{\tau_s} exp\left[\frac{\log(\tau_m/\tau_s)}{\tau_m/\tau_s - 1}\right] \quad (11)$$

with parameter values given in table 5.Neurons in the network are allowed to connect to all other neurons except itself. Connectivity bias is a ratio which describes the tendency of the network to favor connections between 2 different types of neuron.

Based on the connectivity bias for inhibitory and excitatory neurons, connection probability P_{xT_j} (Latham et al. [5]) can be calculated using,

$$P_{ET_j} = \frac{K_{T_j}}{N_E + N_I B_{T_j}} \quad (12)$$

$$P_{IT_j} = \frac{K_{T_j} B_{T_j}}{N_E + N_I B_{T_j}} \quad (13)$$

where B_{T_j} is the connection bias, K_{T_j} is the mean number of post synaptic connection, N_E is the number of excitatory connection and N_I is the number of inhibitory connection.The network connection is then randomly selected based on the probability calculated. I_{max} and connectivity bias are varied to find a set of parameters that is capable of reproducing activity patterns that are observe in the experiment.

Connection biases are varied from 0.8-1.2 for inhibitory and excitatory neurons. For each of the connection patterns, the I_{max} values are varied and the effect on the generated spontaneous activity are observed. All the differential equation are implemented in C++ template and solve using Runge-Kutta 4^{th}order numerical scheme.

Figure 23 shows the different types of activity patterns obtained from the simulation with different connection bias. Periodically repeating firing of neurons can be observed in Figure 23 (a) and (b) when the inhibitory bias is low. Figure 23 (c) shows a very low level of activity. Figure 23 (d), (e) and (i) show a constant activity pattern. Figure 23 (f) and (h) show bursting activity

patterns. Figure 23 (g) shows large activity at the start that decays to a lower activity.

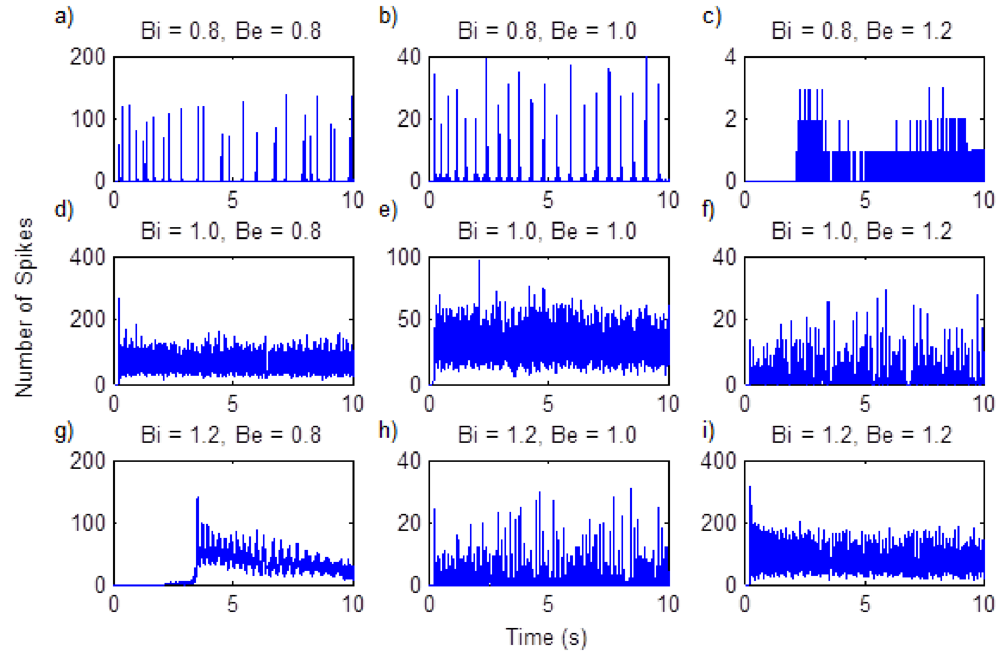

Figure 23 Simulated activity for different Be and Bi values. Imax of 4 is used for the simulation. The activity time bin is 10ms

The activity patterns of networks with connection bias, Bi = 0.8 are similar to the experimental results by Mok et al.[18] except that the simulated period of inactivity in between steady network firing is much smaller compare to the period detected in the experiments as shown in Figure 24.

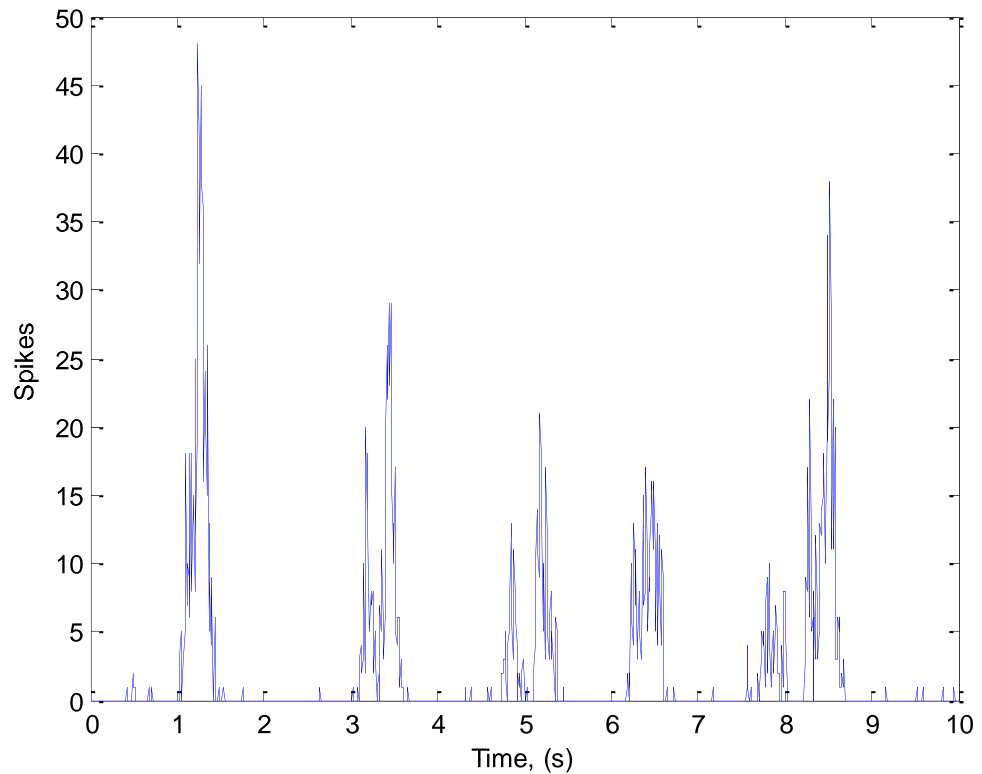

Figure 24 Experiment data showing spontaneous bursting activity for 10s in a culture of cortical neurons. The activity time bin is 10ms (Mok et al.[19])

Simulations were carried out for Bi = 0.8 and I_{max} from 3.6 to 5. For I_{max} of 3.6 and lower there is no network activity due to the low number of endogenously active neurons. Figure 25 shows that for I_{max} values of 4.0 and 3.8, the number of spikes of the network is very low. For I_{max} values of 5.0 and 4.5, the activity patterns show steady firing with a period of inactivity around 2s to 3s. The activity pattern simulated using network parameters Bi = 0.8 and Be = 1.2 are the closest to the activity pattern as shown in the experimental results of Figure 24.

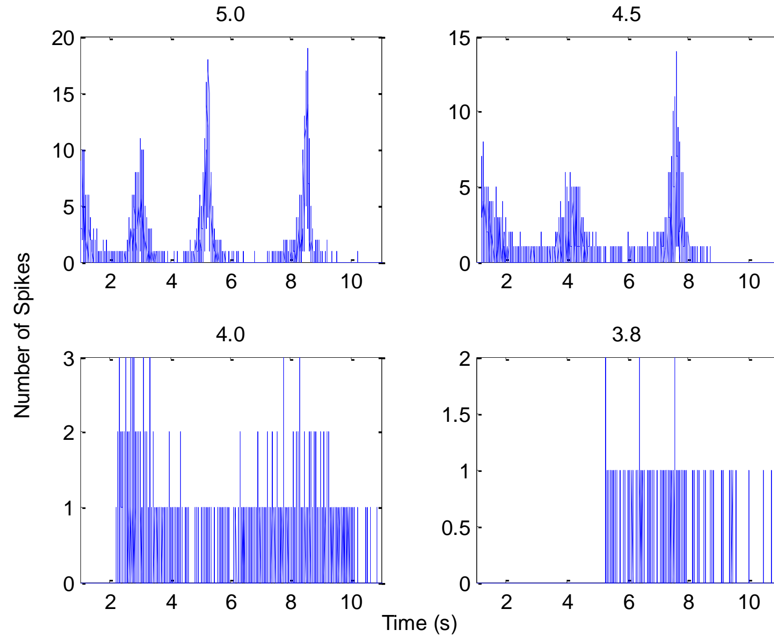

Figure 25 Activity for network with Bi = 0.8,Be = 1.2 Imax = 5 to 3.8. The activity time bin is 10ms

4.3.2 Simulation of Ultra-Slow Spontaneous Oscillations

A network with 2 subnetworks is created for the purpose of simulating the ultra-slow oscillations as observed by Mok et al.[18]. Neurons are assigned randomly to the subnetworks. An inhibiting property is assigned to each subnetwork. The inhibiting property is described by equation 6 in section 3.3. Parameters in table8, 9 and 10 are used for the simulation.

Parameter for inhibiting property	Value
α	0.001-0.01
Upper Threshold for φ	4-5
Lower Threshold for φ	0.1
τ_{φ}	1000-50000
Number of neuron in each subnetwork 1	90% to 10%
Number of neuron in each subnetwork 2	10% to 90%

Table 7 Parameters for the inhibiting properties of a subnetwork

The number of neurons in the 2 subnetworks is varied from 50-50% to 10-90%. The subnetwork accumulates the inhibitory property when the neurons in a subnetwork fire. If the inhibitory property reaches the upper threshold value, the neurons in the subnetwork will become inactive until the value of inhibitory property falls below the lower threshold value. All external currents to the neurons within the subnetwork will be set to 0 during this inactive period. The membrane potential of the neurons in this subnetwork will fall towards the resting potential during the period of inactivity.

The parameters of subnetwork size, α and τ_φ affect the shape and pattern of the generated ultra-slow oscillation. The size of subnetwork determines the number of neurons that are active during the peak and trough of the oscillations. The time period where both subnetworks are active is determined by the time the inhibitory property takes to reach the upper threshold. The size of the subnetwork, τ_φ and α affects the accumulation rate of inhibitory property. The period of inactivity for the subnetwork is dependent on τ_φ. With no external current acting on the neurons, no activity will be generated during the time the inhibitory property takes to reach the lower threshold. For the above simulation, a small τ_φ is assigned to subnetwork 2 to simulate a neuron group that is active all the time.

By adjusting these 3 parameters, oscillatory patterns with different periods can be achieved. Figure 26 shows simulation results that are comparable to activity detected in experiment. When both subnetworks are active, about 150 spikes per time bin can be obtained from the simulation. Activity when only 1 of the subnetwork is active is much lower, at about 45 spikes per time bin.

Figure 27 shows the changes of the inhibitory property for subnetwork 1 and subnetwork 2. The inhibitory property of subnetwork 1 accumulates and reaches the upper threshold. Then it decays to the lower threshold when neurons are inactive. The inhibitory property of subnetwork 2 stays at about 0.7 due to the fast decay rate.

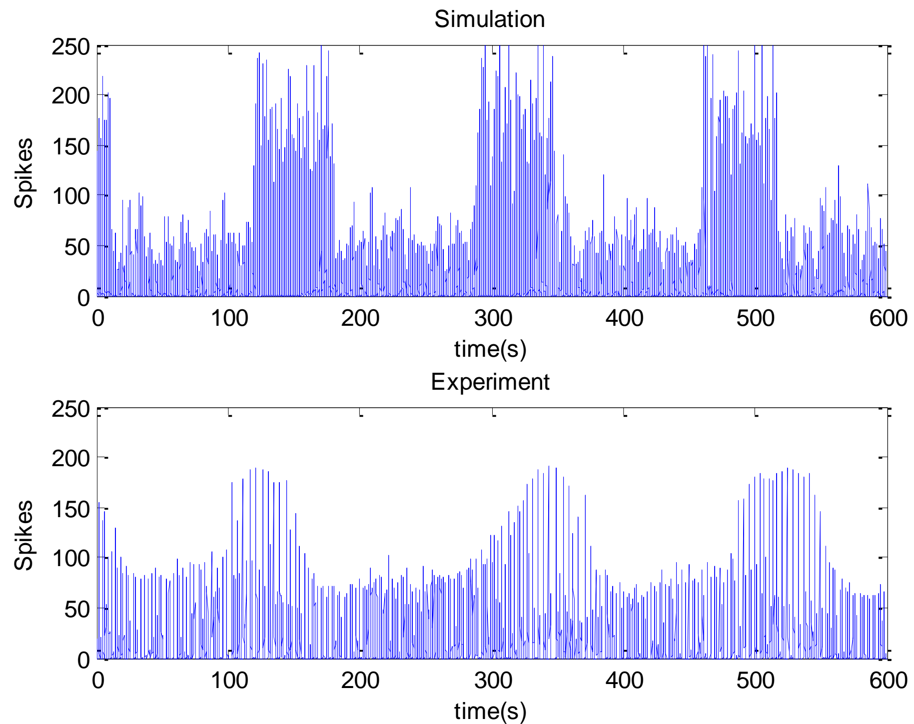

Figure 26 Simulation results showing 5 peaks in 800s. Parameters for the simulation of activity: α = 0.07, upper threshold = 3.8.Subnetwork 1: number of neuron = 2000, τ_φ = 30000. Subnetwork 2: number of neurons = 8000, τ_φ = 1000. The activity time bin is 10ms

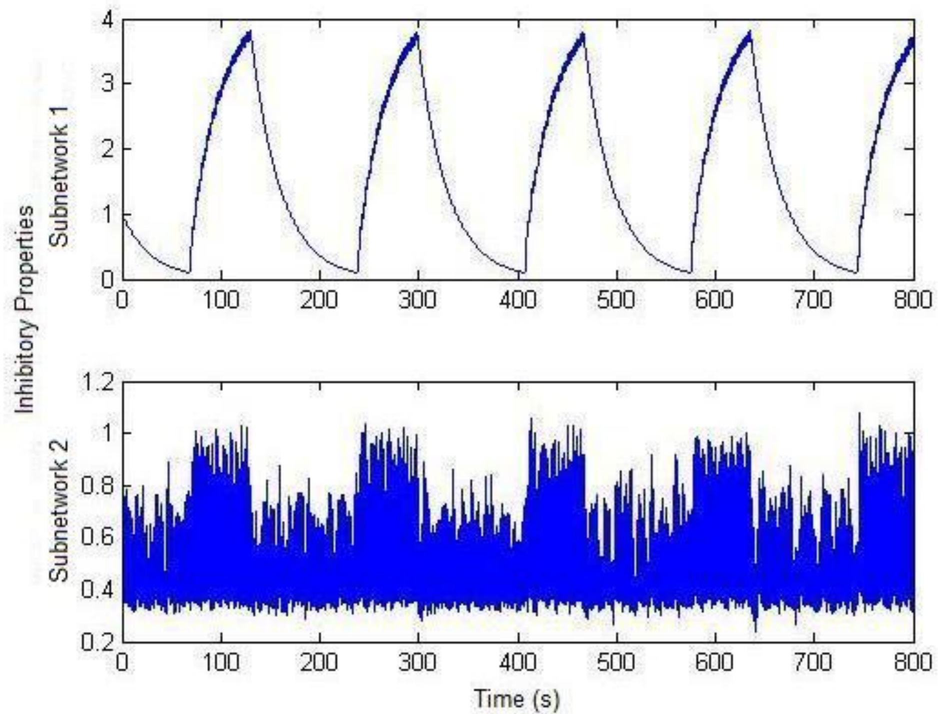

Figure 27 Changes of inhibitory properties, φ in subnetwork 1 and subnetwork 2 with parameters for the simulation of activity: α = 0.07, upper threshold = 3.8. Sub network 1: number of neuron = 2000, τ_{φ} = 30000. Subnetwork 2: number of neurons = 8000, τ_{φ} = 1000

Figure 28 shows simulation results with another set of parameters. Different patterns of activity can be obtained by changing the number of neurons in subnetwork 1 and subnetwork 2. The simulation results show patterns that are comparable to the activity pattern observed in culture 2.

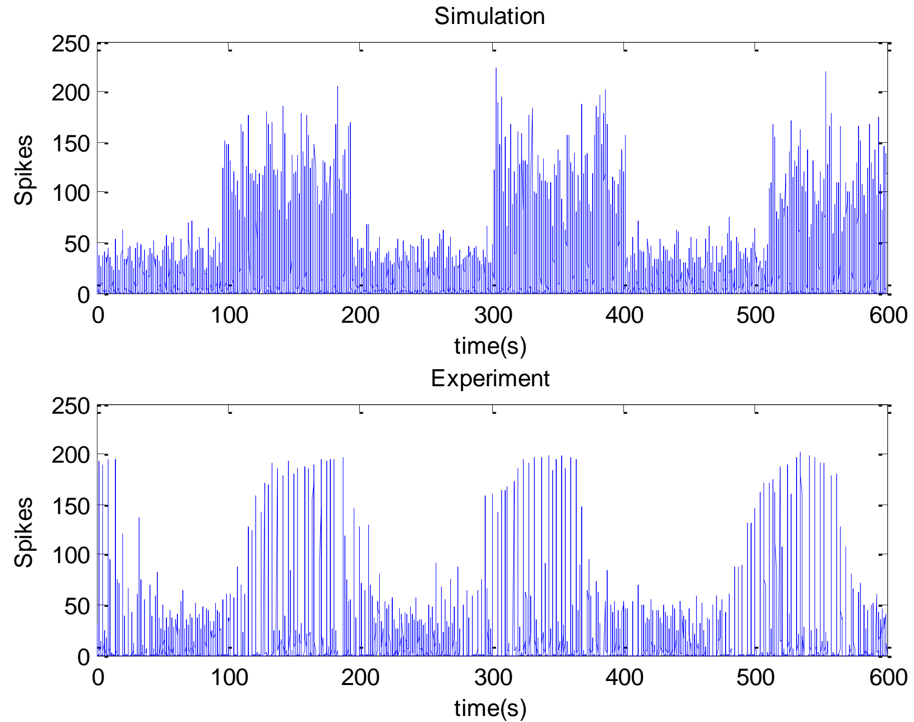

Figure 28 Simulation results showing 3 peaks in 800s. Parameters for the simulation of activity: α = 0.07, upper threshold = 3.8. Subnetwork 1: number of neuron = 3000, τ_φ = 30000. Subnetwork 2: number of neuron = 7000, τ_φ = 1000. The activity time bin is 10ms

Ultra-slow oscillations can be reproduced in simulations by introducing an inhibition property with a slow time constant to one of the subnetwork. The inhibition of subnetwork cannot be detected from the experiments conducted by Mok et al. [18].Further experiments must be carried out to better understand the underlying biochemical process that causes the Ultra-slow oscillation.

CHAPTER 5

CONCLUSIONS

A simulation algorithm was developed for a network consisting of IF neurons using objects in C++. Function classes are used to describe the differential equations in the simulations. Network models are represented in C++ based on the classes. The simulation algorithm takes advantage of the parallel processing capabilities of current computer CPU's to improve the speed of computation.

Using the above simulation algorithm, the effects of the different parameters of the IF model on the global oscillations of a network of homogenous neurons were investigated. The results show that the frequency of the network global oscillations are affected by the leakage term, the network size, the connection probability, the external noise, the synaptic delay and the synaptic weight.

Ultra-slow spontaneous oscillations were simulated using the model by Latham et al. [5] together with an additional equation that describes the generation and dissipation of an inhibiting property for 2subnetwork of neurons. The results of the simulations show activity patterns that are comparable to ultra-slow spontaneous oscillations observed in cortical cultures of rat neurons by Mok et al. [18].

REFRENCES

[1] A. Vallabhaneni, T. Wang and B. He.*Neural Engineering*, pp.85-121 2005.

[2] L.F. Tan, C.S. Ng, J.Q. Ng, and S.Y. Goh. “A brain-computer interface with intelligent distributed controller for wheelchair”in *4th Int. Conf. on Biomedical Engineering*, Kuala Lumpur, IFMBE Proc.,2008, pp. 641–644.

[3] L.F. Abbott. “Theoretical neuroscience rising”*Neuron*, vol. 60, issue 3, pp. 489–495, Nov 2008.

[4] W. Gerstner and W.M. Kistler.*Spiking Neuron Models Single Neurons, Populations, Plasticity*.Cambridge University Press, 2002.

[5] P. E. Latham, B. J. Richmond, P. G. Nelson, and S.Nirenberg.“Intrinsic dynamics in neuronal networks.I.Theory.”*J. of Neurophysiology*, vol 83, issue 2, pp. 808–827, Feb 2000.

[6] A. Husain, W.O. Tatum, P.W. Kaplan. *Handbook of EEG interpretation*.Demos Medical, 2008.

[7] E. O. Mann and O. Paulsen.“Mechanisms underlying gamma (’40’) network oscillations in the hippocampus - a mini-review”,*Progress in Biophysics and Molecular Biology*, vol 87, issue 1, pp67 – 76, Jan 2005.

[8] X.X. Jia and A. Kohn."Gamma rhythms in the brain",*PLoS Biology*, vol. 9, issue 4, April 2011.

[9] L. Zhu, K.L. Blethyn, D.W. Cope, V. Tsomaia, V. Crunelli, and S. W. Hughes. "Nucleus- and species-specific properties of the slow (<1 hz) sleep oscillation in thalamocortical neurons",*Neuroscience*, vol. 141, issue 2, pp. 621–636, Aug 2006.

[10] M. Steriade, A. Nuñez, and F. Amzica."A novel slow (< 1 hz) oscillation of neocortical neurons in vivo: depolarizing and hyperpolarizing components."*J. Neuroscience*, vol 13, issue 8 pp. 3252–3265, Aug 1993.

[11] F. Amzica and M. Steriade."Short- and long-range neuronal synchronization of the slow (< 1 hz) cortical oscillation",*J. of Neurophysiology*, vol. 73, issue 1, pp. 20–38, Jan 1995.

[12] M. Penttonen, N. Nurminen, R. Miettinen, J. Sirviö, D. A. Henze, J. Csicsvári, and G. Buzsáki. "Ultra-slow oscillation (0.025 hz) triggers hippocampal afterdischarges in wistar rats"*Neuroscience*, vol 94, issue 3, pp. 735–743, Oct 1999.

[13] M.O. Cunningham, M.A. Whittington, A. Bibbig, A. Roopun, F.E.N. LeBeau, A. Vogt, H. Monyer, E.H. Buhl, and R.D. Traub. "A role for fast rhythmic bursting neurons in cortical gamma oscillations in vitro",*Proc. of the National Academy of Sciences of the United States of America*, vol. 101, no. 18, pp. 7152–7157, Apr 2004.

[14] M. V. Sanchez-Vives and D. A. McCormick."Cellular and network mechanisms of rhythmic recurrent activity in neocortex",*Nature Neuroscience*, vol. 3, issue 10, pp.1027–1034, Oct 2000.

[15] C.P. Chen, L. Chen, Y.S. Lin, S.Q. Zeng, and Q.M. Luo."The origin of spontaneous synchronized burst in cultured neuronal networks based on multi-electrode arrays",*Biosystems*, vol. 85, issue 2, pp. 137–143, Aug 2006.

[16] G. Zhu, X.N. Li, J.B. Pu, W.J. Chen, and Q.M. Luo. "Transient alterations in slow oscillations of hippocampal networks by low-frequency stimulations on multi-electrode arrays",*Biomedical Microdevices*, vol 12, issue 1, Nov 2009.

[17] P.E. Latham, B.J. Richmond, S. Nirenberg, and P.G. Nelson."Intrinsic dynamics in neuronal networks. II. Experiment"*J. of Neurophysiology*, vol. 83, issue 2, pp. 828–835, Feb 2000.

[18] S. Y. Mok, Z. Nadasdyb, Y.M. Lim, S.Y. Goh."Ultra-slow oscillations in cortical networks in vitro",*Molecular and Cellular Neuroscience*, vol. 206, Mar 2012.

[19] S. Y. Mok, private communication, 2011.

[20] B. W. Knight. "Dynamics of encoding in a population of neurons",*The J. of General Physiology*, vol. 59, issue 6, pp. 734–766, Jun 1972.

[21] E.M.Izhikevich. *Dynamical systems in neuroscience*. MIT Press, Cambridge,MA, 2006.

[22] D. Hansel and G. Mato. "Existence and stability of persistent states in large neuronal networks",*Physical Review Letters*, vol. 86, issue 18, pp. 4175–4178, Apr 2001.

[23] A. L. Hodgkin and A. F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve",*J. of Physiology*, vol. 117, issue 4, pp. 500–544, Aug 1952.

[24] C. Morris and H. Lecar. "Voltage oscillations in the barnacle giant muscle fiber"*Biophysical J.*, vol. 35, issue 1, pp. 193–213, Jul 1981.

[25] E. M. Izhikevich. "Which model to use for cortical spiking neurons?",*IEEE Transactions on Neural Networks*, vol. 15, issue 5, pp. 1063–1070, Sep 2004.

[26] C. J. Lobb, Z. Chao, R. M. Fujimoto, and S. M. Potter."Parallel event-driven neural network simulations using the hodgkin-huxley neuron model",*Proc. of the 19th Workshop Principles of Advanced and Distributed Simulation*, *PADS 2005*, pp. 16–25, 2005.

[27] W. C. Gerken, L. K. Purvis, and R. J. Butera."Genetic algorithm for optimization and specification of a neuron model",*Proc. 27th Annual Int. Conf.*

of the Engineering in Medicine and Biology Society, IEEE-EMBS 2005, pp. 4321–4323, 2005.

[28] B. Han and T. M. Taha.“Neuromorphic models on a gpgpu cluster”, *Proc. Int. Neural Networks Joint Conf.*, *IJCNN*, pp. 1–10, 2010.

[29] T. M. Taha, P. Yalamanchili, M. Bhuiyan, R. Jalasutram, C. Chen, and R. Linderman.“Neuromorphic algorithms on clusters of playstation 3s,*Proc. Int. Neural Networks Joint Conf.*, *IJCNN*, pp. 1–8, 2010.

[30] K. MacLeod and G. Laurent. “Distinct mechanisms for synchronization and temporal patterning of odor-encoding neural assemblies”, *Science*, vol. 274, issue 5289, pp. 976–979, Nov 1996.

[31] D. Desmaisons, J. D. Vincent, and P. M. Lledo. “Control of action potential timing by intrinsic subthreshold oscillations in olfactory bulb output neurons”*J Neuroscience*, vol. 19, issue 24, pp. 10727–10737, Dec 1999.

[32] W.M. Kistler, J.L.V. Hemmen, and C.I. De Zeeuw.“Time window control: a model for cerebellar function based on synchronization, reverberation, and time slicing”,*Progress in Brain Research*, vol. 124, pp. 275–297, 2000.

[33] L.F. Abbott and C.V. Vreeswijk.“Asynchronous states in networks of pulse-coupled oscillators”,*Physical Review E*, vol. 48, issue 2, pp. 1483–1490, Aug 1993.

[34] D. Hansel, G. Mato, and C. Meunier. “Synchrony in excitatory neural networks”,*NeuralComputation*, vol. 7, issue 2, pp. 307–337, Mar 1995.

[35] M.V. Tsodyks and T. Sejnowski.“Rapid state switching in balanced cortical network models”,*Network: Computation in Neural Systems*, vo. 6, issue 2, pp. 111–124, Oct 1995.

[36] N. Brunel and V. Hakim.“Fast global oscillations in networks of integrate-and-fire neurons with low firing rates”,*Neural Computation*, vol. 11, issue 7, pp. 1621–1671, Oct 1999.

[37] N. Brunel. “Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons”,*JComputational Neuroscience*, vol. 8, issue 3, pp. 183–208, Jun 2000.

[38] A. Burkitt. “A review of the integrate-and-fire neuron model: II. inhomogeneous synaptic input and network properties”, *Biological Cybernetics*, vol. 95, issue 2, pp. 97–112, Jul 2006.

[39] X. J. Wang and G. Buzsáki.“Gamma oscillation by synaptic inhibition in a hippocampal interneuronal network model”,*J. Neuroscience*, vol. 16, issue 20, pp. 6402–6413, Oct 1996.

[40] D. Hansel and G. Mato. “Asynchronous states and the emergence of synchrony in large networks of interacting excitatory and inhibitory neurons”,*Neural*Computation, vol. 15, issue 1, pp. 1–56, Jan 2003.

[41] W. Gerstner. “Time structure of the activity in neural network models”*Physical Review E*, vol. 51, issue 1, pp. 738–758, Jan 1995.

[42] R.D. Traub, D. Contreras, M.O. Cunningham, H. Murray, F.E.N. LeBeau, A. Roopun, A. Bibbig, W.B. Wilent, M.J. Higley, and M.A. Whittington. “Single-column thalamocortical network model exhibiting gamma oscillations, sleep spindles, and epileptogenic bursts”,*J.Neurophysiology*, vol. 93, issue 4, pp. 2194–2232, Apr 2005.

[43] M. Tsodyks, A. Uziel, and H. Markram.“Synchrony generation in recurrent networks with frequency-dependent synapses”,*J. ofNeuroscience*, vol. 20, RC50, pp 1-5, Jan 2000.

[44] P. Kudela, P.J. Franaszczuk, and G.K. Bergey.“Changing excitation and inhibition in simulated neural networks: effects on induced bursting behaviour”,*Biological Cybernetic*, vol. 88, issue 4, pp. 276–285, Apr 2003.

[45] V. Volman, I. Baruchi, and E. Ben-Jacob.“Manifestation of function-follow-form in cultured neuronal networks”,*Physical Biology*, vol.2, issue 2, pp. 98–110, Jun 2005.

[46] T.J. Fountain. *Parallel Computing: Principles and Practice*. Cambridge University Press, 2006.

[47] L. Chai, Q. Gao, and D.K. Panda. “Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core

system."*Proc. Seventh IEEE Int. Symp.Cluster Computing and the Grid CCGRID 2007*, pp. 471–478, 2007.

[48] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Inc, 2004.

[49] J. Wensch and B. Sommeijer."Parallel simulation of axon growth in the nervous system",*Parallel Computing*, vol. 30, issue 2, pp. 163–186, 2004.

[50] S. Pang, L. Chen, Y. Yin, A. Duan, J. Zhou, and L. Hu."A parallel numerical study of transient heat transfer and fluid flow of weld pool during laser keyhole welding", *Advanced Materials Research*, vol. 97-101, pp. 3001-3004, Mar 2010.

[51] D. Gao and T.E. Schwartzentruber. "Optimizations and openmp implementation for the direct simulation monte carlo method",*Computers and Fluids*, vol. 42, issue 1, pp. 73–81, Nov 2011.

[52] H. Jang, A. Park, and K. Jung."Neural network implementation using cuda and openmp",*Proc.2008.Digital Image Computing: Techniques and Applications*, *DICTA*, pp. 155–161, 2008.

[53] G. Svenk. *Object-oriented programming: using C++ for engineering and technology*. Cengage Learning, 2003.

[54] H. Chen, B. Yuan, D. A. Baxter, and J. H. Byrne.“Parallel computation in computer simulation for neural networks”,*Proc. IEEE Region 10th Conf. Computers, Communications Control and Power Engineering*, *TENCON ’02*, volume 1, pp. 641–644, 2002.

[55] J.M. Bower and D. Beeman.“Constructing realistic neural simulations with genesis”,*Methods in molecular biology*, vol. 401, pp. 103–125, 2007.

[56] L.N. Long and A. Gupta.“Scalable massively parallel artificial neural networks”,*J. of Aerospace Computing, Information and Communication*, vol. 5, issue 1, pp. 3–15, Jan 2008.

[57] W.H. Tseng, S.Y. Lu, and H. Mei.“On the development of a brain simulator”,*Lecture Notes in Computer Science,*LNAI(PART 2), pp.258–267, 2010.

[58] D.G. Zill. *A First Course in Differential Equations: With Modeling Applications*. Cengage Learning, 2008.

[59] L.R Petzold, U.M. Ascher. *Computer methods for ordinary differential equations and differential-algebraic equations*.SIAM, 1998.

[60] R.Q.Quiroga, Z. Nadasdy, and Y. Ben-Shaul.“Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering”,*Neural Computation*, vol. 16, issue 8, pp. 1661–1687, Aug 2004.

[61] S. Gibson, J.W. Judy, and D. Markovic."Comparison of spike-sorting algorithms for future hardware implementation"*Proc. 30th Annual Int. Conf. of the IEEE Engineering in Medicine and Biology Society*, *EMBS 2008*, pp. 5015–5020, 2008.

APPENDIX A. EQUATIONS:

Equation used in simulation of inhibitory and excitatory neurons:

1	Integrate and Fire Model	$\frac{dV_i(t)}{dt} = \frac{1}{\tau}[-(V_i(t) - V_r) + RI_i(t) + J_{ext}r]$
2	External noise generation: Average Part	$\mu_{ext} = J_{ext}C_{ext}\nu_{ext}\tau$
3	External noise generation: Fluctuating Part	$\sigma_{ext} = J_{ext}\sqrt[2]{C_{ext}\nu_{ext}\tau}$
4	Internal input current	$RI_i(t) = \tau \sum_j w_{ij}J_j \sum_k \delta(t - t_j^k - D)$

Equation used for simulation of spontaneous activity:

1	Neuron model for Simulation of Spontaneous activity	$T_m \frac{dv_i(t)}{dt} = \alpha(v_i(t) - v_r)(v_i(t) - v_t) + I_{a,i}(t) - (g_{k,i} + g_{k-ca,i})(v_i(t) - \varepsilon_k) - (v_i(t)\tilde{I}_i - \tilde{I}_{\varepsilon,i})$
	Synaptic Input	$\frac{d\tilde{I}_i}{dt} = -\frac{\tilde{I}_i}{\tau_s} + r_s \sum_{j,u} w_{ij}\delta(t - t_j^u)$
	Synaptic Input	$\frac{d\tilde{I}_{\varepsilon,i}}{dt} = -\frac{\tilde{I}_{\varepsilon,i}}{\tau_s} + r_s \sum_{j,u} w_{ij}\delta(t - t_j^u)$
	AHP current	$\frac{dg_{k,i}}{dt} = -\frac{g_{k,i}}{\tau_{K,i}} + \delta_{k,i} \sum_{\mu} \delta(t - t_i^u)$
	AHP current	$\frac{dg_{k-Ca,i}}{dt} = -\frac{g_{k-Ca,i}}{\tau_{k-Ca,i}} + \delta_{k-Ca,i} \sum_{\mu} \delta(t - t_i^u)$
	Equation specifying weight of synapse	$w_{ij} = \frac{V_{psp,ij}}{\varepsilon_j}\frac{1}{r_s}\frac{\tau_m}{\tau_s} exp\left[\frac{\log(\tau_m/\tau_s)}{\tau_m/\tau_s - 1}\right]$
	Probability of connection from excitatory postsynaptic neuron	$P_{ET_j} = \frac{K_{T_j}}{N_E + N_I B_{T_j}}$
	Probability of connection from inhibitory postsynaptic neuron	$P_{IT_j} = \frac{K_{T_j}B_{T_j}}{N_E + N_I B_{T_j}}$

Equation used for testing of simulation algorithm:

1	Basic IF model driven by constant current	$\frac{dv}{dt} = \frac{1}{\tau}(-(v - v_r) + RI)$
2	Analytical solution of Basic IF model	$v(t) = (RI + v_r) - (RI + v_r - v_0)e^{-(\frac{t_c}{\tau})}$
3	Quadratic IF Model driven by constant current	$\frac{dv}{dt} = \frac{1}{\tau}(\alpha_0(v - v_r)(v - v_t) + RI)$
4	Analytical Solution of Quadratic IF Model	$v(t) = \left(\frac{(4ac - b^2)\tan\left(\frac{t_c\sqrt[2]{(4ac-b^2)}}{2\tau}\right) + (2av_0 + b)\sqrt[2]{(4ac - b^2)}}{2a\left(\sqrt[2]{(4ac - b^2)} - (2av_0 + b)\tan\left(\frac{t_c\sqrt[2]{(4ac-b^2)}}{2\tau}\right)\right)} - \frac{b}{2a}\right)$ $a = \alpha_0$ $b = -\alpha_0(v_t + v_r)$ $c = \alpha_0 v_t v_r + RI$

APPENDIX B:

Example code showing usage of model

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <ctime>
#include <cmath>
#include <cstring>
#include <sstream>
#include <deque>

using namespace std;
using namespace std::tr1;

class LinIF{

        private:
                double v_thres;
        public:
                double v;
                double ge;
                double gi;
                double ref;
                double Er;
                double tau;

                LinIF(double e):
                        v(e),
                        ge(0),
                        gi(0),
                        ref(0),
                        Er(-65),
                        tau(20),
                        v_thres(-55){}

                ~LinIF(){}

                double operator()(double y, double t){
                        return (1/tau) * (-(y-Er) + 15);
                }

                bool update(){
                        if (v >= v_thres){
```

```
                v = -65;
                return true;
            }
            else
                return false;
        }
};
class QIF{

    public:
        double v;
        double ge;
        double gi;
        double ref;
        double Er;
        double tau;
        double v_thres;

        QIF(double e):
            v(e),
            ge(0),
            gi(0),
            ref(0),
            Er(-65),
            tau(20),
            v_thres(0){}

        ~QIF(){}

        double operator()(double y, double t){
            return (1/tau) * (0.2*(y-Er)*(y-(-55)) + 10) ;
        }

        bool update(){
            if (v >= v_thres){
                v = -65;
                return true;
            }
            else
                return false;
        }
};
template<typename function>double runge_kutta_4th(function equation,
double initial, double t, double dt){
    double k1 = equation(initial, t);
    double k2 = equation(initial + 0.5 * k1 * dt, t + 0.5 *dt);
    double k3 = equation(initial + 0.5 * k2 * dt, t + 0.5 *dt);
    double k4 = equation(initial + k3 * dt, t + dt);
    return initial + (k1 + 2*k2 + 2*k3 + k4)/6 * dt;
}
```

```
template<typename function>double euler(function equation, double initial,
double t, double dt){
	return initial + equation(initial, t) * dt;
}

int main()
{
	LinIF testRK(-65);
	LinIF testEU(-65);
	QIF testQRK(-65);
	QIF testQEU(-65);
	ofstream out_RK("outRK.csv");
	ofstream out_EU("outEU.csv");
	ofstream out_QRK("outQRK.csv");
	ofstream out_QEU("outQEU.csv");
	out_RK << testRK.v << endl;
	out_EU << testEU.v << endl;
	out_QRK << testQRK.v << endl;
	out_QEU << testQEU.v << endl;
	for (int i = 0; i < 500; i++){
		testRK.v = runge_kutta_4th(testRK, testRK.v, 1, 0.1);
		testRK.update();
		testEU.v = euler(testEU, testEU.v, 1, 0.1);
		testEU.update();
		out_RK << testRK.v << endl;
		out_EU << testEU.v << endl;
		testQRK.v = runge_kutta_4th(testQRK, testQRK.v, 1, 0.1);
		testQRK.update();
		testQEU.v = euler(testQEU, testQEU.v, 1, 0.1);
		testQEU.update();
		out_QRK << testQRK.v << endl;
		out_QEU << testQEU.v << endl;
	}
return 0;
}
```

```
};
```

Neuron with AHP Implementation

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <ctime>
#include <cmath>
#include <cstring>
#include <sstream>
#include <deque>

using namespace std;
using namespace std::tr1;

class gSynIn{
	public:
		double g;
		double spk;

	private:
		double t_syn;
		double Ai;

	public:
		gSynIn():
			g(0),
			spk(0),
			t_syn(3),
			Ai(0.17)
			{}

		~gSynIn(){}

		double operator()(double y, double t){
			return (1/ t_syn) * (-y + (spk * Ai)) ;
		}

		void reset(){
			spk = 0;
		}
};
class gSynEx{
	public:
		double g;
		double spk;
```

```
		private:
			double t_syn;
			double Ai;

		public:
			gSynEx():
				g(0),
				spk(0),
				t_syn(3),
				Ai(0.25)
				{}

			~gSynEx(){}

			double operator()(double y, double t){
				return (1/ t_syn) * (-y + (spk * Ai)) ;
			}

			void reset(){
				spk = 0;
			}
	};
	class neuron_membrane_dynamics{
		public:
			double gkca;
			double gk;
			double Ia;
			double Ia_mean;
			double v;
			double gi;
			double ge;
			double Ei;
			double Ee;
			double E_syn;
			double c;
			double v_rest;
			double v_thres;
			double E_pot;
			double v_apex;
			double v_repol;

		public:
			neuron_membrane_dynamics(double Ia_m):
				gkca(0),
				gk(0),
				ge(0),
				gi(0),
				Ia_mean(Ia_m),
				Ia(0),
				Ei(-80),
```

```
                        Ee(0),
                        E_syn(-60),
                        c(10),
                        v_rest(-65),
                        v_thres(-50),
                        E_pot(-80),
                        v_apex(20),
                        v_repol(-80)
                        {
                                v = v_rest;
                                Ia = Ia_mean;
                        }

                ~neuron_membrane_dynamics(){}

                double operator()(double y, double t){
                        return (1/c) *  ((y - v_rest)*(y - v_thres)/( v_thres - 
v_rest)+(-gi*(y-Ei))+(-ge*(y-Ee)));
                }

                bool update(){
                        if (v >= v_apex){
                                v = v_repol;
                                return true;
                        }
                        else
                                return false;
                }

};
template<typename function>double runge_kutta_4th(function equation, 
double initial, double t, double dt){
        double k1 = equation(initial, t);
        double k2 = equation(initial + 0.5 * k1 * dt, t + 0.5 *dt);
        double k3 = equation(initial + 0.5 * k2 * dt, t + 0.5 *dt);
        double k4 = equation(initial + k3 * dt, t + dt);
        return initial + (k1 + 2*k2 + 2*k3 + k4)/6 * dt;
}
template<typename function>double euler(function equation, double initial, 
double t, double dt){
        return initial + equation(initial, t) * dt;
}

int main()
{
        gSynIn tempI;
        gSynEx tempE;
        neuron_membrane_dynamics testMembrane(0);
        ofstream out_M("outM.csv");
        out_M << testMembrane.v << endl;
```

```
	for (int i = 0; i < 100; i++){
		tempI.g = runge_kutta_4th(tempI, tempI.g, 1, 1);
		tempI.spk = 0;
		tempE.g = runge_kutta_4th(tempE, tempE.g, 1, 1);
		tempE.spk = 0;
		testMembrane.gi = tempI.g;
		testMembrane.ge = tempE.g;
		testMembrane.v = runge_kutta_4th(testMembrane,
testMembrane.v, 1, 1);
		testMembrane.update();
		out_M << testMembrane.v << endl;
		if(i == 2){
			tempI.spk = 1;
		}
		//if(i == 2){
		//	tempE.spk = 1;
		//}
	}	return 0;
}
```

Network of neurons solve numerically and analytical with OpenMP implementation

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <ctime>
#include <cmath>
#include <cstring>
#include <sstream>
#include <random>
#include <deque>
#include <complex>
#include <omp.h>

using namespace std;
using namespace std::tr1;

class QIF{

	public:
		double v;
		double Er;
		double tau;
		double v_thres;
		double I;

		QIF(double e):
			v(e),
			Er(-65),
			tau(20),
			v_thres(-20){}

		~QIF(){}

		double operator()(double y, double t){
			return (1/tau) * (0.1*(y-Er)*(y-(-55)) + I) ;
		}

		bool update(){
			if (v >= v_thres){
				v = -65;
				return true;
			}
			else
				return false;
```

```
		}
};
class QIfAnSol{

	private:
		double v_thres;
	public:
		double v;
		double Er;
		double tau;
		double I;
		double prev_t;
		double cur_t;

		QIfAnSol(double e):
			v(e),
			Er(-65),
			tau(20),
			prev_t(0),
			v_thres(-20){}

		~QIfAnSol(){}

		double operator()(double y, double t){
			complex<double>  a(0.1,0),  b(0.1*(55+65),0),  c((-55*-
65*0.1) + I), z0, z1, ans, pans1, pans2, pans3, pans4, pans5;
			complex<double> f4(4,0);
			complex<double> f2(2,0);
			complex<double> ft(t,0);
			complex<double> f1(1,0);
			complex<double> ftau(20,0);
			complex<double> ftest(-5,0);
			z0 = sqrt(f4*a*c-b*b);
			z1 = f2*a*v+b;
			pans1 = ((z0*z0)*tan( z0/(f2*ftau) * ft) + z0 *z1);
			pans2 = (z0-tan( z0/(f2*ftau) * ft)*z1);
			pans3 = (f1/(f2*a));
			pans4 = tan(ftest);
			pans5 = z0/(f2*ftau) * ft;
			ans = pans3 * (pans1/ pans2 - b);
			return ans.real();
		}

		bool update(){
			if (v >= v_thres){
				v = -65;
				//prev_t = cur_t;
				return true;
			}
			else
```

```
                return false;
        }
};
template<typename function>double runge_kutta_4th(function equation,
double initial, double t, double dt){
    double k1 = equation(initial, t);
    double k2 = equation(initial + 0.5 * k1 * dt, t + 0.5 *dt);
    double k3 = equation(initial + 0.5 * k2 * dt, t + 0.5 *dt);
    double k4 = equation(initial + k3 * dt, t + dt);
    return initial + (k1 + 2*k2 + 2*k3 + k4)/6 * dt;
}
template<typename function>double euler(function equation, double initial,
double t, double dt){
    return initial + equation(initial, t) * dt;
}

class Timing {
    public:
        Timing(){
            startt=time(NULL);
            endt=time(NULL);
        }
        ~Timing(){}
        void tic(){
        startt=time(NULL);
        }
        void toc(){
        endt=time(NULL);
        }
        time_t diff(){
        return endt-startt;
        }
    private:
        time_t startt, endt;
};
int main()
{
    Timing timing;
    vector<QIF> EuLif;
    vector<QIF> RkLif;
    vector<QIfAnSol> AnLif;
    vector<vector<bool>> netCon;
    int number = 5000;
    bernoulli_distribution rCon(0.2);
    ofstream out_E("spikeE.csv");
    ofstream out_R("spikeR.csv");
    ofstream out_A("spikeA.csv");
    ofstream out_vE("vE.csv");
    ofstream out_vR("vR.csv");
    ofstream out_vA("vA.csv");
```

```
ranlux4_01 dseed(float(time(0)));
cout<< "initializing\n";
for(int i = 0; i<number; i++){
	vector<bool> temp;
	QIF temp1(-65);
	QIfAnSol temp2(-65);
	for(int j = 0; j<number; j++){
		temp.push_back(rCon(dseed));
	}
	netCon.push_back(temp);
	EuLif.push_back(temp1);
	RkLif.push_back(temp1);
	AnLif.push_back(temp2);
}
cout<< "Simulation\n";
poisson_distribution<int, double>extIn(10);
int s_time = 0;
timing.tic();
out_E << "0" << endl;
out_R << "0" << endl;
out_A << "0" << endl;
out_vE <<EuLif[0].v << endl;
out_vR <<RkLif[0].v << endl;
out_vA <<AnLif[0].v << endl;
double dt = 0.1;
for(int i = 1; i<=20000; i++){
	for(int j = 0; j<number; j++){
		double extI = extIn(dseed) * 2;
		AnLif[j].I += extI;
		EuLif[j].I += extI;
		RkLif[j].I += extI;
	}
	int spikeE = 0;
	int spikeR = 0;
	int spikeA = 0;
	#pragma omp parallel for schedule(guided, 1000)
	for(int j = 0; j<number; j++){
		//RkLif[j].I = 20;
		AnLif[j].v = AnLif[j](AnLif[j].v, dt);
		AnLif[j].I = 0;
		EuLif[j].v = euler(EuLif[j],EuLif[j].v, dt, dt);
		EuLif[j].I = 0;
		RkLif[j].v = runge_kutta_4th(RkLif[j],RkLif[j].v, dt, dt);
		RkLif[j].I = 0;
	}
	for(int j = 0; j<number; j++){
		if(EuLif[j].update()){
			spikeE += 1;
			for(int k = 0; k < number; k++){
```

```
                    if(netCon[j][k] && EuLif[k].v <= -55)
                        EuLif[k].I += -1;
                }
            }
            if(RkLif[j].update()){
                spikeR += 1;
                for(int k = 0; k < number; k++){
                    if(netCon[j][k] && RkLif[k].v <= -55)
                        RkLif[k].I += -1;
                }
            }
            if(AnLif[j].update()){
                spikeA += 1;
                for(int k = 0; k < number; k++){
                    if(netCon[j][k] && AnLif[k].v <= -55)
                        AnLif[k].I += -1;
                }
            }
        }
        out_R << spikeR << endl;
        out_vR <<RkLif[0].v << endl;
        out_E << spikeE << endl;
        out_A << spikeA << endl;
        out_vE <<EuLif[0].v << endl;
        out_vA <<AnLif[0].v << endl;
        timing.toc();
        if (timing.diff()  > 0){
            s_time++;
            timing.tic();
            cout<<  setw(10)  <<  i*dt  <<  "ms\t"  <<  setw(10)  <<
s_time << "s" << endl;
        }
    }
    system("pause");
return 0;
}
```

Simulation of Spontaneous activity

```
#include "stdafx.h"

using namespace std;
using namespace std::tr1;

class neuron_internal_dynamics{
	public:
		double tau;
		double wi;
		double del;
		double z;

		neuron_internal_dynamics(double tau, double del):
			tau(tau),
			wi(0),
			del(del),
			z(0){}

		~neuron_internal_dynamics(){}

		double operator()(double y, double t){
			return -(y/tau) + del * wi;
		}
};

class neuron_membrane_dynamics{
	public:
		double gkca;
		double gk;
		double Ia;
		double Ie;
		double I;
		double v;

	private:
		double t_cell;
		double v_rest;
		double v_thres;
		double E_pot;
		double v_apex;
		double v_repol;

	public:
		neuron_membrane_dynamics(double Ia):
			gkca(0),
```

```
                    gk(0),
                    Ie(0),
                    I(0),
                    t_cell(10),
                    v_rest(-65),
                    v_thres(-50),
                    E_pot(-80),
                    v_apex(20),
                    v_repol(-80),
                    Ia(Ia)
                    {
                            v = v_rest;
                    }

            ~neuron_membrane_dynamics(){}

            double operator()(double y, double t){
                    return (1/t_cell) * ( (v - v_rest)*(v-v_thres)/( v_thres - 
v_rest) + Ia - (gk + gkca) * (v - E_pot) - (v * I - Ie));
            }

            bool update(){
                    if (v >= v_apex){
                            v = v_repol;
                            return true;
                    }
                    else
                            return false;
            }
};

template<typename function>double runge_kutta_4th(function equation, 
double initial, double t, double dt){
        double k1 = equation(initial, t);
        double k2 = equation(initial + 0.5 * k1 * dt, t + 0.5 *dt);
        double k3 = equation(initial + 0.5 * k2 * dt, t + 0.5 *dt);
        double k4 = equation(initial + k3 * dt, t + dt);
        return initial + (k1 + 2*k2 + 2*k3 + k4)/6 * dt;
}

int main()
{
        vector<neuron_membrane_dynamics> membrane_dynamics;
        vector< vector<neuron_internal_dynamics>> internal_dynamics;
        vector< vector<double>> wij;
        bool check;
        clock_t c_start, c_end;
        clock_t c_itv_start, c_itv_end;
        time_t t_curr;
        double t_taken, t_total = 0;
```

```
double applied_current;
double t_sim;
int net_size;
double net_fraction;
double net_in;
double net_ex;
double net_mean;
double net_bias_in;
double net_bias_ex;
double net_probInIn;
double net_probInEx;
double net_probExIn;
double net_probExEx;
double V_pspIn = -1.5;
double V_pspEx = 1;
double V_preIn = -80;
double V_preEx = 0;
double *V_pspj, *V_prei;
double progress;
int t_sim_end;
char *dir_curr;
char dir_name[80];

double temp;

ofstream out_para("parameters.csv");
//cout << left <<setw(40) << "Input Network Size" << ": ";
//cin >> net_size;

//cout << left <<setw(40) << "Input Mean Conenction" << ": ";
//cin >> net_mean;
//cout << left <<setw(40) << "Input Inhibitory Fraction" << ": ";
//cin >> net_fraction;
//cout << left <<setw(40) << "Input Network Bias(in)" << ": ";
//cin >> net_bias_in;
//cout << left <<setw(40) << "Input Network Bias(ex)" << ": ";
//cin >> net_bias_ex;
cout<< left << setw(40) << "Input Applied Current (mV)" << ": ";
cin>> applied_current;
//cout << left <<setw(40) << "Input Simulation End Time (s)" << ": ";
//cin >> t_sim;
net_size = 10000;
net_mean = 2000;
net_fraction = 0.2;
net_bias_in = 1.2;
net_bias_ex = 0.8;
t_sim = 7;
out_para << "Network Size," << net_size << endl;
out_para << "Network mean," << net_mean << endl;
out_para << "Inhibitory Fraction," << net_fraction << endl;
```

```
	out_para << "Network Bias(in)," << net_bias_in << endl;
	out_para << "Applied Current," << applied_current << endl;
	out_para << "Network Bias(ex)," << net_bias_ex << endl;
	out_para << "Simulation End Time," << t_sim << endl;

	cout<< left << setw(40) << "Initiaizing" << endl;
	t_curr = time(0);
	cout<< setw(40) << "Initiaizing Started On" << ": " << ctime(&t_curr);
	c_start = clock();
	double temp1[4] = {30,2000,3,3};	// tau
	double temp2[4] = {1,0.2,0.1,0.1};	// del
	ranlux4_01 dseed(time(0));
	uniform_real<double>  r_real(0, applied_current);
	net_in = net_fraction * net_size;
	net_ex = net_size - net_in;
	net_probInIn = (net_mean * net_bias_in) / (net_ex + net_in * 
net_bias_in);
	net_probInEx = (net_mean * net_bias_ex) / (net_ex + net_in * 
net_bias_ex);
	net_probExIn = net_mean / (net_ex + net_in * net_bias_in);
	net_probExEx = net_mean / (net_ex + net_in * net_bias_ex);
	cout<< net_probInIn << " " << net_probInEx << " " << net_probExIn 
<< " " << net_probExEx << endl;
	bernoulli_distribution r_bInIn(net_probInIn);
	bernoulli_distribution r_bInEx(net_probInEx);
	bernoulli_distribution r_bExIn(net_probExIn);
	bernoulli_distribution r_bExEx(net_probExEx);
	bernoulli_distribution *r_ptr;
	V_pspj = &V_pspIn;
	V_prei = &V_preIn;
	c_itv_start = clock();
	for (int i = 0; i<net_size; i++){
		neuron_membrane_dynamics nmd_temp(r_real(dseed));
		membrane_dynamics.push_back(nmd_temp);
		vector<neuron_internal_dynamics> nid_vtemp;
		for(int j = 0; j<4; j++){
			neuron_internal_dynamics		nid_temp(temp1[j], 
temp2[j]);
			nid_vtemp.push_back(nid_temp);
		}
		internal_dynamics.push_back(nid_vtemp);
		vector<double> temp_wij;
		if (i == net_in){
			V_pspj = & V_pspEx;
			V_prei = &V_preEx;
		}
		for(int j = 0; j<net_size; j++){
			if (i < net_in && j < net_in)
				r_ptr = &r_bInIn;
			else if (i < net_in)
```

```
                        r_ptr = &r_bExIn;
                else if (i >= net_in && j < net_in)
                        r_ptr = &r_bInEx;
                else if (i >= net_in)
                        r_ptr = &r_bExEx;
                if (r_ptr->operator()(dseed) == 1 && j != i)
                        temp_wij.push_back(*V_pspj / (*V_prei + 65) *
55.84311504);
                else
                        temp_wij.push_back(0);
            }
            wij.push_back(temp_wij);
        }
        cout<< left << setw(40) << "Initiaizing Ended On" << ": " <<
ctime(&t_curr);
        cout<< left << setw(40) << "Starting Simulation" << endl;
        t_curr = time(0);
        cout<<setw(40) << "Simulation Started On" << ": " << ctime(&t_curr);
        c_start = clock();
        ofstream out_spikes("spikes.dat");
        ofstream out_in1("in1.dat");
        ofstream out_ex1("ex1.dat"); vector<string> out_data;
        for(int j = 0; j< net_size; j++)
            out_data.push_back("");
        double dt = 0.1;
        t_sim_end = (int) (t_sim * 1000/dt);
        c_itv_start = clock();
        progress = 1;
        for (int i = 0; i < t_sim_end; i++){
            V_prei = &V_preIn;
            int spikes_total = 0;
            #pragma omp parallel for
            for(int j = 0; j< net_size; j++){
                for(int k = 0; k< 4; k++){
                        internal_dynamics[j][k].z                =
runge_kutta_4th(internal_dynamics[j][k], internal_dynamics[j][k].z, 0, dt);
                        internal_dynamics[j][k].wi = 0;
                }
                membrane_dynamics[j].gk     =
internal_dynamics[j][0].z;
                membrane_dynamics[j].gkca=
internal_dynamics[j][1].z;
                membrane_dynamics[j].I      =
internal_dynamics[j][2].z;
                membrane_dynamics[j].Ie     =
internal_dynamics[j][3].z;
                membrane_dynamics[j].v                   =
runge_kutta_4th(membrane_dynamics[j], membrane_dynamics[j].v, 0, dt);
            }
```

```
		#pragma	omp	parallel	for	shared(internal_dynamics,
spikes_total)
		for(int j = 0; j< net_size; j++){
			if (j == net_in)
				V_prei = &V_preEx;
			if(membrane_dynamics[j].update()){
				internal_dynamics[j][0].wi += 1;
				internal_dynamics[j][1].wi += 1;
				spikes_total++;
				for(int k = 0; k < net_size; k++){
					if ( wij[j][k] != 0 ){
						internal_dynamics[k][2].wi	+=
wij[j][k];
						internal_dynamics[k][3].wi	+=
wij[j][k] * *V_prei;
					}
				}
			}
		}
		out_in1 << i << "," << membrane_dynamics[net_in].v << endl;
		out_ex1 << i << "," << membrane_dynamics[net_in + 1].v <<
endl;
		out_spikes << i << "," <<spikes_total << endl;
		if ( ((double) (i +1 ) ) / ((double) t_sim_end) * 100 >=
progress){
			cout<< left << setw(18) << "Percentage" << ": " ;
			cout<< right << setw(18) << progress << "%" << endl;
			progress += 1;
		}
	}
	t_curr = time(0);
	cout<< left <<setw(40) << "Simulation Ended On" << ": " <<
ctime(&t_curr);
	c_end = clock();
	t_taken = difftime(c_end, c_start) / 1000000;
	t_total += t_taken;
	cout<< left << setw(40) << "Time Taken" << ": " << t_taken << "s" <<
endl;
	return 0;
}
```

Simulation of Spontaneous activity and ultr- slow oscillation

```
#include<iostream>
#include<cmath>
#include<fstream>
#include<vector>
#include<time.h>
#include<omp.h>
#include<tr1/random>
#include<iomanip>
#include<string>
 #include <sstream>
 #include <typeinfo>
 #include <stdexcept>

using namespace std;
using namespace std::tr1;

#define pi 3.14159265

class fluctuation{
	public:
		double tau;
		double wi;
		double z;
		double del;
		double ini;

		fluctuation(double tau):
			tau(tau),
			wi(0),
			z(0){}

		~fluctuation(){}

		double operator()(double y, double t){
			return -((y-ini)/tau) - wi;
		}
};

class neuron_internal_dynamics{
	public:
		double tau;
		double wi;
		double del;
		double z;
		bool waste;
```

```
        neuron_internal_dynamics(double tau, double del):
            tau(tau),
            wi(0),
            del(del),
            z(0),
            waste(false){}

        ~neuron_internal_dynamics(){}

        double operator()(double y, double t){
            return -(y/tau) + del * wi;
        }


};

class neuron_membrane_dynamics{
    public:
        double gkca;
        double gk;
        double Ia;
        double Ie;
        double I;
        double v;
        double I_back;
        double wT;

    private:
        double t_cell;
        double v_rest;
        double v_thres;
        double E_pot;
        double v_apex;
        double v_repol;

    public:
        neuron_membrane_dynamics(double Ia):
            gkca(0),
            gk(0),
            Ie(0),
            I(0),
            t_cell(10),
            v_rest(-65),
            v_thres(-50),
            E_pot(-80),
            v_apex(20),
            v_repol(-80),
            Ia(Ia)
            {
                I_back = Ia;
```

```
				v = v_rest;
			}

		~neuron_membrane_dynamics(){}

		double operator()(double y, double t){
			return (1/t_cell) * ( (v - v_rest)*(v-v_thres)/( v_thres - v_rest) + Ia - (gk + gkca) * (v - E_pot) - (v * I - Ie));
		}

		bool update(){
			if (v >= v_apex){
				v = v_repol;
				return true;
			}
			else
				return false;
		}
};

template<typename function>double runge_kutta_4th(function equation, double initial, double t, double dt){
	double k1 = equation(initial, t);
	double k2 = equation(initial + 0.5 * k1 * dt, t + 0.5 *dt);
	double k3 = equation(initial + 0.5 * k2 * dt, t + 0.5 *dt);
	double k4 = equation(initial + k3 * dt, t + dt);
	return initial + (k1 + 2*k2 + 2*k3 + k4)/6 * dt;
}

int main()
{
	vector<neuron_membrane_dynamics> membrane_dynamics;
	vector<neuron_internal_dynamics> waste;
	neuron_internal_dynamics waste1(3*10000, 1);
	neuron_internal_dynamics waste2(1000, 1);
	waste.push_back(waste1);
	waste.push_back(waste2);

	vector< vector<neuron_internal_dynamics>> internal_dynamics;
	vector< vector<double>> wij;
	vector< double > fluc;
	bool check;
	clock_t c_start, c_end;
	clock_t c_itv_start, c_itv_end;
	time_t t_curr;
	double t_taken, t_total = 0;
	double applied_current;
	double t_sim;
	int net_size;
	double net_fraction;
```

```
	double net_in;
	double net_ex;
	double net_mean;
	double net_bias_in;
	double net_bias_ex;
	double net_probInIn;
	double net_probInEx;
	double net_probExIn;
	double net_probExEx;
	double mean1, mean2;
	//double fluc;
	double V_pspIn = -1.5;
	double V_pspEx = 1;
	double V_preIn = -80;
	double V_preEx = 0;
	double *V_pspj, *V_prei;
	double progress;
	int t_sim_end;
	char *dir_curr;
	char dir_name[80];

	double temp;

	double w_thres1,w_thres2;
	vector<double> w_thres;

	ofstream out_para("parameters.txt");
	net_size = 10000;
	net_mean = 2000;
	net_fraction = 0.2;
	net_bias_in = 0.8;
	net_bias_ex = 1.2;
	t_sim = 1000;
	cout<< left << setw(40) << "Input Network Size" << ": " << net_size
<< "\n";
	cout<< left << setw(40) << "Input Mean Conenction" << ": "<<
net_mean <<"\n";
	cout<< left << setw(40) << "Input Inhibitory Fraction" << ": "<<
net_fraction <<"\n";
	cout<< left << setw(40) << "Input Network Bias(in)" << ": "<<
net_bias_in <<"\n";
	cout<< left << setw(40) << "Input Network Bias(ex)" << ": "<<
net_bias_ex <<"\n";
	cout<< left << setw(40) << "Input Simulation End Time (s)" << ": " <<
t_sim <<"\n";

	double por;
	//cout << left <<setw(40) << "Current Flunctuation Peak2Peak" << ": ";
	//cin >> fluc;
	cout<< left << setw(40) << "Ia" << ": ";
```

```
cin>> mean1;
cout<< left << setw(40) << "W Thres 1" << ": ";
cin>> w_thres1;
cout<< left << setw(40) << "W Thres 2" << ": ";
cin>> w_thres2;
cout<< left << setw(40) << "Proportion" << ": ";
cin>> por;
w_thres.push_back(w_thres1);
w_thres.push_back(w_thres2);

out_para << "Network Size," << net_size << endl;
out_para << "Network mean," << net_mean << endl;
out_para << "Inhibitory Fraction," << net_fraction << endl;
out_para << "Network Bias(in)," << net_bias_in << endl;
out_para << "Applied Current," << mean1 << endl;
out_para << "w_thres1," << w_thres1 << endl;
out_para << "w_thres2," << w_thres2 << endl;
out_para << "Por," << por << endl;
out_para << "Network Bias(ex)," << net_bias_ex << endl;
out_para << "Simulation End Time," << t_sim << endl;


cout<< left << setw(40) << "Initiaizing" << endl;
t_curr = time(0);
cout<< setw(40) << "Initiaizing Started On" << ": " << ctime(&t_curr);
c_start = clock();
double temp1[4] = {30,3000,3,3};     // tau
double temp2[4] = {1,0.2,0.1,0.1};    // del
ranlux4_01 dseed(time(0));



net_in = net_fraction * net_size;
net_ex = net_size - net_in;
net_probInIn = (net_mean * net_bias_in) / (net_ex + net_in * net_bias_in);
net_probInEx = (net_mean * net_bias_ex) / (net_ex + net_in * net_bias_ex);
net_probExIn = net_mean / (net_ex + net_in * net_bias_in);
net_probExEx = net_mean / (net_ex + net_in * net_bias_ex);
cout<< net_probInIn << " " << net_probInEx << " " << net_probExIn << " " << net_probExEx << endl;
bernoulli_distribution r_bInIn(net_probInIn);
bernoulli_distribution r_bInEx(net_probInEx);
bernoulli_distribution r_bExIn(net_probExIn);
bernoulli_distribution r_bExEx(net_probExEx);
bernoulli_distribution *r_ptr;


bernoulli_distribution SubNetwork(por);
```

```
uniform_real<double>  r_real1(0, mean1);
uniform_real<double>  r_real2(0, mean2);
uniform_real<double> *u_ptr;

V_pspj = &V_pspIn;
V_prei = &V_preIn;
c_itv_start = clock();


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

        u_ptr = &r_real1;

        neuron_membrane_dynamics                         nmd_temp(u_ptr-
>operator()(dseed));
        if(SubNetwork(dseed))
                nmd_temp.wT = 0;
        else
                nmd_temp.wT = 1;

        membrane_dynamics.push_back(nmd_temp);
        fluc.push_back(nmd_temp.Ia);

        vector<neuron_internal_dynamics> nid_vtemp;
        for(int j = 0; j<4; j++){
                neuron_internal_dynamics                 nid_temp(temp1[j],
temp2[j]);
                nid_vtemp.push_back(nid_temp);
        }
        internal_dynamics.push_back(nid_vtemp);

        vector<double> temp_wij;
        if (i == net_in){
                V_pspj = & V_pspEx;
                V_prei = &V_preEx;
        }
        for(int j = 0; j<net_size; j++){
                if (i < net_in && j < net_in)
                        r_ptr = &r_bInIn;
                else if (i < net_in)
                        r_ptr = &r_bExIn;
                else if (i >= net_in && j < net_in)
                        r_ptr = &r_bInEx;
                else if (i >= net_in)
                        r_ptr = &r_bExEx;
                if (r_ptr->operator()(dseed) == 1 && j != i)
                        temp_wij.push_back(*V_pspj / (*V_prei + 65) *
55.84311504);
                else
```

```
                    temp_wij.push_back(0);
            }
            wij.push_back(temp_wij);
        }

        cout<< left << setw(40) << "Initiaizing Ended On" << ": " <<
ctime(&t_curr);
        cout<< left << setw(40) << "Starting Simulation" << endl;
        t_curr = time(0);
        cout<<setw(40) << "Simulation Started On" << ": " << ctime(&t_curr);
        c_start = clock();
        ofstream out_spikes("spikes.dat");
        ofstream out_in1("in1.dat");
        ofstream out_ex1("ex1.dat");
        ofstream out_fluc1("fluc1.dat");
        ofstream out_fluc2("fluc2.dat");
        ofstream out_w1("w1.dat");
        ofstream out_w2("w2.dat");
        vector<string> out_data;
        for(int j = 0; j< net_size; j++)
            out_data.push_back("");
        double dt = 0.1;
        t_sim_end = (unsigned long) (t_sim * 1000 / dt);
        c_itv_start = clock();
        progress = 0;
        int fluc_index = 0;
        for (unsigned long i = 0; i < t_sim_end; i++){
            double t = double(i) * dt;
            V_prei = &V_preIn;
            int spikes_total = 0;

            for(int j = 0; j<2; j++){
                waste[j].z = runge_kutta_4th(waste[j],waste[j].z ,t,dt);
                waste[j].wi = 0;
                if(waste[j].waste){
                    if(waste[j].z < 0.1)
                        waste[j].waste = false;
                }
                if(waste[j].z >w_thres[j])
                    waste[j].waste = true;
            }
            #pragma omp parallel for
            for(int j = 0; j< net_size; j++){
                if(waste[membrane_dynamics[j].wT].waste){
                    for(int k = 0; k< 4; k++){
                        internal_dynamics[j][k].z = 0;
                        internal_dynamics[j][k].wi = 0;
                    }
                    membrane_dynamics[j].Ia     = 0;
                    membrane_dynamics[j].gk     = 0;
```

```
					membrane_dynamics[j].gkca = 0;
					membrane_dynamics[j].I		= 0;
					membrane_dynamics[j].Ie	= 0;
				}
				else{
					for(int k = 0; k< 4; k++){
						internal_dynamics[j][k].z		=
runge_kutta_4th(internal_dynamics[j][k], internal_dynamics[j][k].z, t, dt);
						internal_dynamics[j][k].wi = 0;
					}
					membrane_dynamics[j].Ia	= fluc[j];
					membrane_dynamics[j].gk	=
internal_dynamics[j][0].z;
					membrane_dynamics[j].gkca =
internal_dynamics[j][1].z;
					membrane_dynamics[j].I		=
internal_dynamics[j][2].z;
					membrane_dynamics[j].Ie	=
internal_dynamics[j][3].z;
				}
				membrane_dynamics[j].v	=
runge_kutta_4th(membrane_dynamics[j], membrane_dynamics[j].v, t, dt);
			}
			#pragma omp parallel for
			for(int j = 0; j< net_size; j++){
				if (j == net_in)
					V_prei = &V_preEx;
				if(membrane_dynamics[j].update()){
					internal_dynamics[j][0].wi = 1;
					internal_dynamics[j][1].wi = 1;
					//cout << j << "\t" << waste[j].z << endl;
					waste[membrane_dynamics[j].wT].wi  +=  0.001
* 7;
					spikes_total++;
					for(int k = 0; k < net_size; k++){
						internal_dynamics[k][2].wi += wij[j][k];
						internal_dynamics[k][3].wi += wij[j][k] *
*V_prei;
					}
				}
			}
			out_spikes << i << " " <<spikes_total << endl;
			out_w1 << i << " " <<waste[0].z  << endl;
			out_w2 << i << " " <<waste[1].z  << endl;
			if  (  ((double)  (i  +1  )  )  /  ((double)  t_sim_end)  *  100   >=
progress){
				cout<< left << setw(18) << "Percentage" << ": " ;
				cout<< right << setw(18) << progress << "%" << endl;
				progress += 0.01;
			}
```

```
        }
        t_curr = time(0);
        cout<< left <<setw(40) << "Simulation Ended On" << ": " <<
ctime(&t_curr);
        c_end = clock();
        t_taken = difftime(c_end, c_start) / 1000000;
        t_total += t_taken;
        cout<< left << setw(40) << "Time Taken" << ": " << t_taken << "s" <<
endl;
        return 0;
}
```