NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY

Modeling and Simulation of Spiking Neural Network Models using SpiNNaker

by

Saqib Khan

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the
Computational Science & Engineering
Research Center for Modeling & Simulation (RCMS)

September 6, 2014

Declaration of Authorship

I, Saqib Khan, declare that this thesis titled, 'Modeling and Simulation of Spiking Neural Network Models using SpiNNaker' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated
- Where I have consulted the published work of others, this is always clearly attributed
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work
- I have acknowledged all main sources of help
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself

Signed: ____

Date: ____

“Learning is the process whereby knowledge is created through the transformation of experience.”

David Kolb

NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

Computational Science & Engineering
Research Center for Modeling & Simulation (RCMS)

Master of Science

by Saqib Khan

Simulating Spiking Neural Networks (SNN) models is a research field attracting the interest of researchers from various fields, from biology to computer science. The final objective is understanding the mechanisms defining the human brain working. Multiple neural models have been proposed, each with their peculiarities, from the very complex and biologically realistic Hodgkin-Huxley neuron model to the very simple leaky integrate-and-fire neuron. Researchers can, depending on the objective, choose which model to use in their simulation. For an efficient simulation of large population of neurons using these models, there need to be a real parallel system architecture and biologically realistic simulator. This research work revolves around using a universally accepted biologically accurate NEST (software simulator) and SpiNNaker (hardware based simulator). During this research, Hodgkin Huxley model has been implemented for the first time on SpiNNaker using fixed point notation and its results have been verified with those from NEST. Similarly, a newly proposed AJ neural model has been implemented for the first time over NEST and SpiNNaker and we successfully verified its results with those from MATLAB. The research contributed in devising implementation libraries for these two models for researchers interested to simulate neural populations over SpiNNaker and NEST using these models.

Acknowledgements

I would like to express my thanks and sincere gratitude to my advisor Dr. Muhammad Mukaram Khan for his continuous support, helpful advice and valuable guidance throughout this work. His emphasis for excellence kept me well-directed and focused. I am immensely grateful to Dr. Jamil Ahmad for his valuable inputs. His vast research experience of the field has assisted me right to the completion. I am thankful to Mr. Tariq Saeed for his support during my implementation. My sincere appreciation goes to Dr. Adnan Maqsood for providing necessary resources and facilities to carry out this project in time.

My sincere gratitude to the SpiNNaker team at the University of Manchester, including Dr. Alexander Rast, Dr. Francesco Galluppi and Prof. Steve Furber, for their valued time/effort and prompt response which helped me complete my work in time. Finally, I am extremely gratified and indebted to my family members for their enormous support, persistent encouragement and earnest prayers throughout my studies.

Contents

Declaration of Authorship **i**

Abstract **iii**

Acknowledgements **iv**

List of Figures **viii**

1 Introduction **1**
1.1 Background 1
1.2 Motivation 3
1.3 Aim & Objectives 4
1.4 Contributions 4
1.5 Thesis Structure 5

2 Neural Network Modeling **6**
2.1 Introduction 6
2.2 The Brain 6
2.3 Neuron 7
2.4 Axon 7
2.5 Dendrite 8
2.6 Synapse 8
2.7 Mathematical Neuron Modeling 8
2.7.1 Neuron Electrophysiology 8
2.7.2 The Hodgkin-Huxley Model 9
2.7.2.1 Dynamics 10
2.8 The Integrate-and-Fire Model 12
2.9 The Izhikevich Model 13
2.10 Summary 14

3 Neural Network Simulators **16**
3.1 Introduction 16
3.2 Categorization of SNN Simulators 16
3.2.1 Hardware Simulators 17

3.2.2 Software Simulators . . . 19
3.3 Review of Main Simulators Developed . . . 20
3.3.0.1 Hardware . . . 20
SyNAPSE . . . 20
Blue Brain Project . . . 21
FACETS BrainScalesS . . . 21
Neurogrid . . . 22
VLSI Chips from Zurich . . . 22
3.3.0.2 Softwares . . . 22
Brian . . . 22
Neuron . . . 22
Nest . . . 23
Genesis . . . 23
SpikeFunDigiCortex . . . 23
NEF Nengo . . . 23
3.3.0.3 SpiNNaker . . . 24
3.4 Neuron Models Available on SpiNNaker . . . 25
3.4.1 Izhikevich Model . . . 26
3.4.2 Leaky Integrate-and-Fire Model . . . 26
3.4.3 Poisson Spike Source Generator Neuron . . . 27
3.4.4 Spike Source Neuron . . . 27
3.4.5 Spike Source Live Neuron . . . 27
3.4.6 NEF Interface Neurons . . . 28
3.4.7 Plasticity Models Available . . . 28
3.4.8 Needs of Implementation of More Neuronal Models . . . 28
3.5 Summary . . . 29

4 Implementation of Spiking NN models over NEST **30**
4.1 Introduction . . . 30
4.1.1 A NEST simulation consists of following main components . . . 31
4.1.1.1 Nodes . . . 31
4.1.1.2 Events . . . 31
4.1.1.3 Connections . . . 32
4.1.1.4 Classes . . . 32
4.1.1.5 Model Variations . . . 32
4.1.1.6 Data management . . . 32
4.2 Need/Requirement of PyNN . . . 33
4.2.1 Using PyNN . . . 33
4.2.1.1 Data Conversion . . . 34
From PyNN to SLI . . . 34
From SLI to PyNN . . . 35
4.2.1.2 Data Handling . . . 36
4.2.2 Hodgkin-Huxley model over NEST . . . 38
4.2.2.1 Using PyNN- Hodgkin-Huxley Model . . . 39
4.2.3 Fixed Point Unit vs Floating Point Unit Implementation of HH Model . 43
4.3 Summary . . . 46

5 Implementing Spiking Neural Network Model over SpiNNaker 47
5.1 Introduction . . . 47
5.2 Feature of SpiNNaker . . . 47
5.2.1 Architecture: . . . 48
Chip Inter-Connection . . . 49
5.3 Pacman . . . 51
5.3.1 Splitter . . . 51
5.3.2 Grouper . . . 52
5.3.3 Mapper . . . 52
5.3.4 Routing . . . 52
5.3.5 SpiNNaker File Generator . . . 52
5.4 Implementing Hodgkin Huxley on SpiNNaker . . . 52
5.5 32-BIT FIXED POINT IMPLEMENTATION OF THE HODGKIN-HUXLEY MODEL . . . 54
5.5.1 Implementation Constraints . . . 55
5.5.1.1 Elementary Mathematical Operations Only . . . 55
5.5.1.2 32-bit Fixed-Point Representation . . . 55
5.5.1.3 Limited Local Memory . . . 55
5.5.1.4 Limited Time to Process a Neuron . . . 55
5.5.1.5 Synaptic Data Only Available on Input Event . . . 55
5.5.2 Implementation Rules . . . 56
5.5.2.1 Defer Event Processing with Annotated Delays . . . 56
5.5.2.2 Solve Differential Equations using the Euler Method . . . 56
5.5.2.3 Represent most Variables using 16-bit Values . . . 56
5.5.2.4 Pre-Compute Constant Parameters where Possible . . . 56
5.5.2.5 Compute Non-Polynomial Functions by Lookup Table . . . 57
5.5.2.6 Exploit free Operations such as Shifting . . . 57
5.6 The Hodgkin-Huxley Model . . . 57
5.6.1 Choice of Scaling Factors . . . 58
5.6.2 The Transformation of Equations . . . 59
5.7 Summary . . . 62

6 Conclusion & Future Work 63
6.1 Conclusion . . . 63
6.2 Future Work . . . 64
6.3 Summary . . . 64

Bibliography 67

List of Figures

2.1 [1]. 7
2.2 Equivalent Circuit Representation of Cell Membrane [2] 9
2.3 Action potential generation in the Hodgkin-Huxley model [3] 11
2.4 The schematic diagram of the leaky integrate-and-re model by Gerstner [] 13

3.1 Difference Between Abstract-Time Simulator and Discrete-Time Simulator. [3] 17
3.2 Relation Between Simulation Time and Biological Time. [4] 18
3.3 Hierarchical Representation of Hardware Neural Network Simulators. [5] 19
3.4 Hierarchical Representation of Software Neural Network Simulators. [6] 20
3.5 Comparison Between the Spikking Neural Network Simulator [7] . . . 25

4.1 Structure of the simulation system and its main parts are a simulation kernel, simulation language interpreter (SLI) and some auxiliary modules. The simulation language interpreter integrates all parts and acts as interface to the user. 31
4.2 Input current I of an excitory neuron. 41
4.3 Input current I of an inhibitory neuron. 41
4.4 Membrane Voltage of an excitory neuron. 42
4.5 Membrane Voltage of an inhibitory neuron. 42
4.6 Fixed Point Unit vs Floating Point Unit Implementation of HH Model. 43
4.7 Action Potential using Floating Point Unit. 43
4.8 Action Potential using Fixed Point Unit. 44
4.9 Equilibrium function for three variables m,n,h using Floating Point Unit. 45
4.10 Equilibrium function for three variables m,n,h using Fixed Point Unit. 45
4.11 Time Constant for three variables m,n,h using Floating Point Unit. . 46

5.1 Block Diagram of the Full SpiNNaker Chip [] 49
5.2 Two-Dimensional Grid of SpiNNaker Chips with the Needed Connections (in Green) to Form the Toroidal Shape. [] 50
5.3 Hexagonal shaped SpiNNaker chip network. [] 50
5.4 Block Diagram of the Partition and Configuration Software (PACMAN). [] 51

5.5 Figure 4.5:A general event-driven function pipeline for neural networks and Variable retrieval recovers values stored from deferred-event processes as well as local values. Polynomial evaluation computes simple functions expressible as multiply and accumulate operations. These then can form the input to lookup table evaluation for more complex functions. Polynomial interpolation improves achieved precision where necessary, and then finally the differential equation solver can evaluate the expression (via Euler method integration). Each of these stages is optional (or evaluates to the identity function) []. 57
5.6 Output of Hodgkin-Huxley Model Simulation on SpiNNaker 59
5.7 Output of Hodgkin-Huxley Model on SpiNNaker Emulator 60

Dedicated to My Family & Friends.

Chapter 1

Introduction

1.1 Background

The neuronal networks in our brains can be described as weighted, directed graphs, with neurons as nodes and synaptic connections as edges. Neurons communicate by sending and receiving point events (spikes) through their connections (synapses)[8]. In the mammalian cortex, each neuron sends its output to about 10^4 other neurons and receives input from almost the same number of neurons. Just 1 mm^3 cortex contains some 10^5 neurons with 10^9 connections [9][8]. This represents a threshold size for simulations, as a realistic number of synapses per neuron can be combined with realistic sparseness (connection probability 0.1). Brain function emerges from the spatio-temporal patterns of neuronal spike activity, but the principles are poorly understood. Progress in understanding brain function therefore depends on simulation studies of large cortical networks. In large neuronal networks, we can neglect the geometric and biophysical complexity of individual nerve cells and describe neurons as point-like objects with a dynamic state governed by a set of differential equations. The most common state variable is the membrane potential V, which is affected by spikes that arrive at the neurons synapses. Whenever V crosses a threshold value Vth, the neuron produces a spike, which is transmitted to all adjacent neurons with a delay of a few milliseconds. Each connection can have a different delay and weight. Weights may evolve as a result of neuronal activity, a phenomenon known as synaptic plasticity, the biological substrate of learning. The spikes of an individual neuron are rare and occur at rates of 150 Hz, whereas the rate of incoming spikes is of the order of 100 kHz due to some 10^4 incoming connections. Simulating large-scale neuronal networks poses several challenges [10]:

- 10^9 to 10^{12} connections must be stored; this requires a specific representation.
- A large number of spikes must be buffered until they are transmitted across the network.

- Simulation results must be reproducible down to the level of membrane potentials and spike times.
- The object-oriented implementation must be appropriate for the problem domain and allow network and machine level optimizations such as efficient caching.

In this contribution, we describe how the neural simulation tools NEST [2] and SpiNNaker address these issues to efficiently simulate neuronal networks of more than 105 neurons and 109 synapses [8]. Many methods investigated to speed up simulation basically go in two directions:

- Developing more computationally efficient neuronal models and training rules. This has been explored mostly by psychologists [11]; the approach is a trade-off between simulation performance and precision. The Hodgkin-Huxley model [12] introduced in 1952 is probably the best known model for such neurons. It is biologically plausible, yet the most sophisticated and computationally expensive model. It takes about 1200 floating-point operations to simulate 1ms of activity. The simplest model i.e. Leaky Integrate and Fire model takes less time to compute (5 floating-point operations per 1ms), but it is less precise and can reproduce fewer firing patterns than the Hodgkin-Huxley model. This indicates that neuronal dynamics involves trade of between speed and accuracy [13].
- Using more computationally powerful hardware, a lot of hardware architectures, from FPGAs [14] to the IBM Blue Gene supercomputer [15], have been investigated both theoretically and experimentally to support the simulation of neural networks.

General-purpose supercomputer systems such as Blue Gene or Beowulf [14]clusters are extremely powerful and easier to program than dedicated hardwired devices such as FPGAs etc. However, firstly, their standard communication systems are usually not efficient enough to meet the high communication demands of neural networks; and secondly, their large physical size and power consumption make them almost impossible to use in embedded neural network applications such as robots. On the other hand, dedicated hardware such as FPGAs and VLSI implementations lack scalability and flexibility [16]. Different applications have variable network sizes. For example, some simple applications such as a control system requires fewer neurons while others such as brain simulation require a larger population of neurons and more connections. Thus linear scalability can help to maintain constant simulation speed by expanding the size of the hardware when the scale of the neural network increases. As computational neuroscience is still developing, the best neuronal model has not yet been discovered. Different models, connection patterns and learning rules are investigated [17]. Therefore, the neuromorphic hardware needs to be reconfigurable and general-purpose to support different neural applications. In this context, what is expected is dedicated neuromorphic hardware which is not

hardwired and offers programmability in order to support a variety of neural network applications.

Neural networks (both biological and artificial) are naturally parallel. Neurons in such a system operate concurrently. Information is stored in a distributed way among synaptic connections. Most traditional computer systems are sequential [18]. This fundamental difference in the structure is one of the most important reasons why it is so inefficient to simulate neural networks on traditional computers. This problem is not only with the speed but also with the memory requirements. As computer science research moves in the direction of multi-core systems, parallel architectures have become a very hot research topic [15]. Our solution is to produce areal" parallel system, that uses a parallel machine to simulate parallel neural networks.

Such a system should be efficient in computation and communication, yet low power, compact and scalable.

1.2 Motivation

The SpiNNaker project [19] is motivated by an idea of simulating large scale neural populations in real time. The aim of the SpiNNaker project is to provide a scalable and massively-parallel computing system as a general-purpose platform for the parallel simulation of large-scale neural systems. The SpiNNaker architecture is an attempt to capture the possible features that ideal parallel machines should have for neural network simulation, as discussed above. Each SpiNNaker chip is a chip multiprocessor (CMP) containing up to 20 ARM968 processors and other components such as a router, communication controllers, etc. There are two different types of memory system associated with each chip, and each processor has an internal RAM block called the tight-couple memory (TCM) which is fast but small, and a block of external SDRAM which is large in capacity but much slower. The TCMs provide instant access to application code and variables, while the SDRAM stores large data sets with comparatively low access rates, for example, the synaptic connectivity data. Each SpiNNaker chip also has 6 self-timed external links by which multiple chips can be linked together to expand the scale of the system and a multicast mechanism is provided for efficient one to many communications. Small chip area and low power consumption are also taken into account in the design [20].

This thesis focuses on the software design and implementation on SpiNNaker system. There are many neural simulators, such as Brian [GB09], NEURON [NEU] and PCSIM [PNS09] that run on desktop computers to make neural simulation straight forward to users. With a dedicated parallel machine such as SpiNNaker, software support is necessary to enable users to run previous experiments on the SpiNNaker system as easily and efficiently as on a desktop

computer. To develop such software, the most important problem to be solved is to map large-scale neural networks efficiently onto SpiNNaker. To achieve high performance, the design of the software and its modeling algorithms must emphasize efficiency.

1.3 Aim & Objectives

The aim of this research is to integrate different neuronal models into system library of SpiNNaker using spiking neural network models. We also aim to efficiently map these implementations over NEST (software simulator) and SpiNNaker (hardware simulator), thus users can choose from these to be used in their simulations.

We achieved our goals by pursuing the following objectives in mind:-

- **Performance:** Achieving real-time (or near to real-time) performance of spiking neural networks models without compromising its accuracy.
- **Scalable Mapping:** Designing a flexible and scalable mapping technique to map underlying neural network over various software and hardware architectures.
- **Computation:** To speed up the computation, fixed-point arithmetic is used during the implementation, and a dual-scaling factor scheme is developed to reduce the loss of precision.
- **Storage:** Neuronal and synaptic information are distributed to different processors or chips. The data structure is organized very carefully and a compressed storage scheme is used to save space.

1.4 Contributions

The research has contributed in providing:

- A neural network modeling scheme over NEST and SpiNNaker (Chapter 3)
- Fixed Point Implementation of spiking neuronal models to improve processing performance over SpiNNaker system (Chapter 4)
- First time implementation of Hodgkin-Huxley neural network model and AJ neural network model over SpiNNaker system (Chapter 4)
- Validation of our implementation methodology using NEST simulator for the functional verification (Chapter 5).

1.5 Thesis Structure

The thesis comprises six chapters including this one. Other chapters cover the following:

- **Chapter 2** gives brief introduction of modeling theory of neuronal models.
- **Chapter 3** reviews neural network simulators and models implemented on it.
- **Chapter 4** discuses design and implementation of Hodgkin-Huxley model and AJ model over NEST simulator
- **Chapter 5** presents the approaches for building a neural system for the Hodgkin-Huxley model on a single ARM968 processor. This includes the approaches for modeling neuronal dynamics, neural representations and implementation on SpiNNaker system.
- **Chapter 6** concludes the thesis and possible evolution in future

Chapter 2

Neural Network Modeling

2.1 Introduction

Computational neuroscientists are trying to mimic the functionality of brain and research into neurobiology has been carrying out for centuries therefore, understanding of brain and biological neural networks have been developed.Now-a-days, computational neuroscientists are using the knowledge electrochemical processes of neurons in the brain and their connectivity pattern for building models and rigorous mathematical methods are being used for describing the brain activities [21]. There are also modeling theories about neural coding, connectivity, and learning along with neuronal dynamic models. This chapter gives a brief overview of the biological research as well as computational modeling of neural networks.

2.2 The Brain

The structure of brain is composed of three most important parts which includes are [22]:

- **Brainstem**It provides main motor and sensor functions to face and neck because it is the lowest part of the brain and is connected to the spinal cord.
- **Cerebellum**It is responsible for integration and coordination between sensory perception and motor control because it is located behind the brain stem.
- **Cerebrum**The major portion of brain is consists of cerebrum with an outer layer known as cortex and inner layer is composed of only white matter. The focus of the most research in neuroscience is cortex and it is responsible for process visual information either static or moving objects, recognition of pattern, planning and controlling muscle movements.

2.3 Neuron

The key component in the cortex is neuron cell and their inter connection therefore scientists found cortex is versatile and so powerful in the structure of brain.It is the basic building block of the nervous system that is responsible for communicating information in both chemical and electrical forms. There are several different types of neurons responsible for different tasks in the human body. Sensory neuron carries massage from a sense organ to the brain. Motor neuron transmits information from the brain to the muscles of the body. Associative neurons are responsible for communicating information between different neurons in the body.More specifically, a neuron cell is different in that it has interaction with other neurons by receiving or sending electrical pulses (spikes). A neuron has three main parts; membrane, axon and dendrite in 2.1 [23].

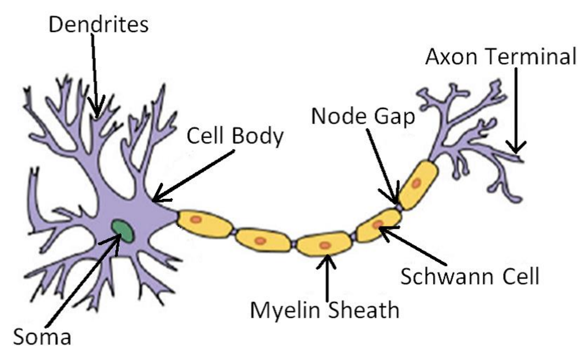


FIGURE 2.1: [1].

2.4 Axon

The long wire like structure that acts as output terminal of neuron is called axon. The axon may branch and send multiple fibers to attach to the other neurons. These fibers are called axon terminals [24].

2.5 Dendrite

The cell body of the neuron contains multiple fibers called dendrites. The dendrites of a neuron may range from a few short fibers to a huge mass of entangled bushes. A typical neuron can have 104 to 105 dendrites. The axons from one neuron can be connected to the cell body of another neuron directly or through dendrites. The dendrites of many neurons contain thousands of little extensions called dendritic spines [25].

2.6 Synapse

The point of functional contact between two neurons is called synapse. In between the contact points, there is a space of about 20 nanometers called synaptic cleft. When a synapse is active, the vesicles open and release neurotransmitters into the synaptic cleft. The synapse can be either excitatory or inhibitory depending upon the type of neurotransmitter. The excitatory synapses increase the activation of target neurons while the inhibitory synapse reduces their activation [26].

2.7 Mathematical Neuron Modeling

2.7.1 Neuron Electrophysiology

An ionic movement around the membrane causes the neuronal activity and it is due to four types of ions which are: sodium, potassium, calcium, and chloride. Basically, their concentrations vary inside and outside of a cell and electrochemical gradient drives their movements [26].

Actually ions move in opposite direction, towards either inside or outside of cell due to effect of concentration and electric potential gradient. The equilibrium phenomenon is achieved when the ionic concentration and the electric potential gradient would equal but in opposite direction because net cross-membrane current is zero [27]. On the contrary side, ions flow through the cell membrane which separates the interior part of cell from the extracellular and difference between membranes creates the phenomenon of membrane potential. Assuming the membrane potential is V, the equilibrium potential (Nernst potential) of K^+ is E_K and the net K^+ current is $I_K(A = cm^2)$ [28], we have:

$$I_K = g_K(V - E_K) \tag{2.1}$$

where the positive parameter g_K is the K_+ conductance. As indicated in Equation 2.2, K_+ ions are driven by the difference between the membrane potential V and the equilibrium potential E_K; the same equation also applies to other ions. The electrical properties of membranes can be represented by the equivalent circuits shown in Figure 2.5. According to Kirchhoffs law [29], we have:

$$I = C(dv/dt) + I_{Na} + I_{Ca} + I_K + I_{Cl} \tag{2.2}$$

or in the standard dynamical system form:

$$C(dv/dt) = I - I_{Na} - I_{Ca} - I_K - I_{Cl} \tag{2.3}$$

where I is the total current; C is the membrane capacitance (C = 1:0F=cm2), and dv/dt is the derivative of the voltage variable V with respect to time t. If there are no additional current sources such as synaptic current or current injections via an electrode [30], then I = 0.

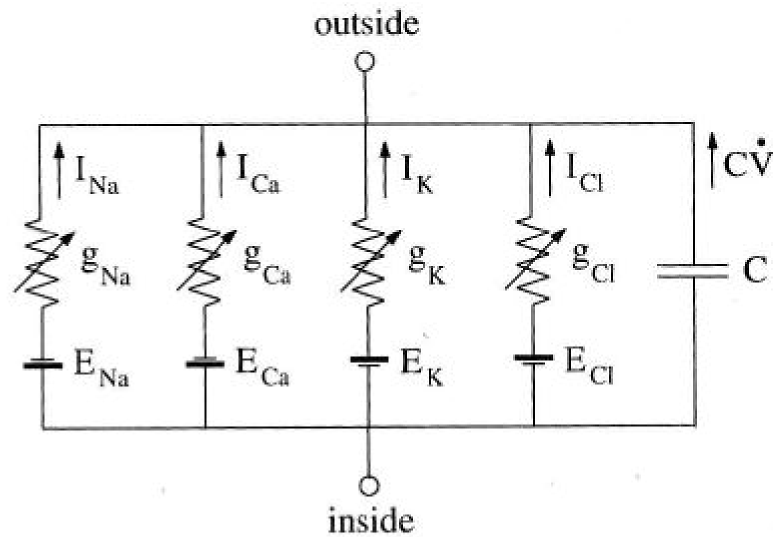


FIGURE 2.2: EQUIVALENT CIRCUIT REPRESENTATION OF CELL MEMBRANE [2]

2.7.2 The Hodgkin-Huxley Model

Hodgkin and Huxley [31]performed a series of experiments on the giant axon of the squid and succeeded in measuring ion currents and described their dynamics by a set of nonlinear differential equations [HH52]. This is one of the most important qualitative models in computational neuroscience. Three types of current are taken into consideration [Izh07]: The K_+ current with four activation gates (resulting in the term n^4), the Na^+ current with three activation gates and one inactivation gate (resulting in the term m^3h), and the Cl^- Ohmic leak current (note that most

neurons in the central nervous system have additional currents). The complete Hodgkin-Huxley equations are [13]:

$$\frac{du}{dt} = -g_{Na}m^3h(u-E_{Na}) - g_K n^4(u-E_K) - g_L(u-E_L) + I(t) \tag{2.4}$$

$$E_{Na} = 115mV, E_K = -12mV, E_L = 10.6mV, g_{Na} = \frac{120ms}{cm^2}, g_K = \frac{36ms}{cm^2}, g_L = \frac{0.3ms}{cm^2}$$

$$\frac{dm}{dt} = \alpha_m(u)(1-m) - \beta_m(u)m$$

while

$$\alpha_m(u) = \frac{2.5-0.1u}{e^{-0.1u+2.5}-1} \text{ and } \beta_m(u) = 4 \times e^{\frac{-u}{18}}$$

$$\frac{dn}{dt} = \alpha_n(u)(1-n) - \beta_n(u)n$$

while

$$\alpha_n(u) = \frac{0.1-0.01u}{e^{-0.1u+1}-1} \text{ and } \beta_n(u) = 0.125 \times e^{\frac{-u}{80}}$$

$$\frac{dh}{dt} = \alpha_h(u)(1-h) - \beta_h(u)h$$

while

$$\alpha_h(u) = \frac{0.07u}{e^{\frac{-u}{20}}} \text{ and } \beta_h(u) = \frac{1}{e^{3-0.1u}+1}$$

The three variables n, m, and h are activation gates (or gating variables) for the three variables n, m, and h are activation gates (or gating variables) for K^+, Na^+, and Cl^- respectively [GK02], [Izh07]. The g_K, g_Na, and g_L are the conductance variables. The membrane capacitance is $C = \frac{1.0F}{cm^2}$ and the applied current is $I = 0A/cm^2$. The parameters that Hodgkin and Huxley used were based on a voltage scale that was shifted by approximately 65 mV, making the resting potential zero for convenience [32]. The shifted Nernst equilibrium potentials are:

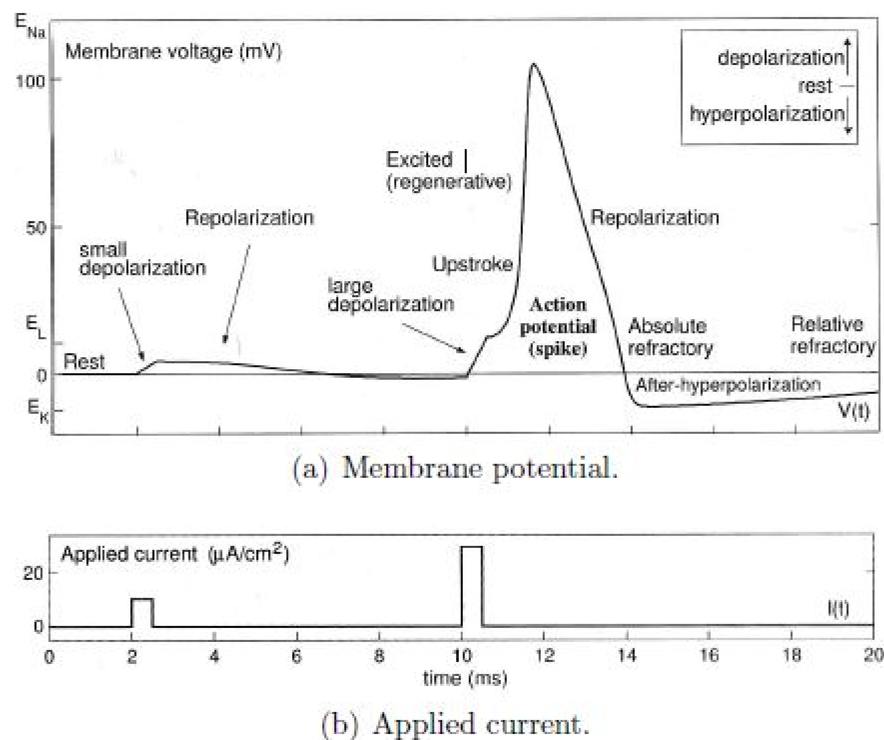
$$E_k = 12mv; E_{Na} = 120mv; E_L = 10:6mv$$

and the typical values of the conductance are: $g_K = \frac{36mS}{cm^2}$; $g_{Na} = \frac{120mS}{cm^2}$; $g_L = 0.3mScm^2$. Now we look into the dynamics the Hodgkin-Huxley model to see how an activation potential (spike) is generated. When the membrane potential V equalises rest value V_rest(0mV in the Hodgkin-Huxley model and about -65mV in reality) [33]. All types of currents balance each other and the rest state is stable.

2.7.2.1 Dynamics

When a small pulse of current I is applied as shown in Figure, the membrane potential V raises. This causes the variable m to be increased, hence increasing the conductance of the sodium (Na+) channels. The influx of positive sodium currents to the cell body then pushes the membrane potential even higher. The effect of the input current I is thus amplified significantly, causing rapid increase of V. If the membrane potential is not big enough to generate a spike, only a positive perturbation of the membrane potential (a small depolarization) is produced as

shown in Figure. This small depolarization is immediately pulled back to the resting value by a small net current. If the amplitude of input current I is much larger, a spike is generated. The sodium conductance is shut off due to the effect of h, when the membrane potential is high. The outflow of potassium (K+)currents then pulls down the membrane potential V. In this case, the on going outflow of K+currents causes V to go below its rest value $V_r eset$, which is called the after-hyperpolarization progress. This is followed by an absolute refractory period, which prevents the system from producing another spike, because the Na+ currents are still depressed and take time to recover. After a long relative refractory period, the membrane potential V goes back to its rest value and the system reaches a new stable state. Generally speaking, the duration of a spike is about1 ms and the amplitude is about 100mv (based on a rest value of 0mv) [34].



(a) Membrane potential.

(b) Applied current.

FIGURE 2.3: ACTION POTENTIAL GENERATION IN THE HODGKIN-HUXLEY MODEL [3]

2.8 The Integrate-and-Fire Model

The detailed high-dimensional Hodgkin-Huxley model is biological plausible, but is complex to analyze and difficult to implement in hardware. As a result, simplified models are desired. As a first step, the objective is to reduce the four-dimensional Hodgkin-Huxley model to a two-dimensional model.The key idea of the reduction is to eliminate two of the four variables in the Hodgkin-Huxley model. This is based on two qualitative observations [GK02].Firstly, in the Hodgkin-Huxley model, the time scale of the dynamics of the activation gate m is much faster than others variables n, h, and V . As a result,m can be treated as an instantaneous variable and can therefore be replaced by it steady-state value m0; this is called a quasi steady-state approximation. Secondly, then and h in the Hodgkin-Huxley model can be replaced by a single effective variable, since their time scales are roughly the same [GK02]. Based on these assumptions, several two-dimensional models have been proposed, such as the Morris-Lecar model and the Fitz Hugh-Nagumo model.These models are conductance-based models, in which the variables and parameters have well-defined biological meanings, and can be measured experimentally. However, the conductance-based models are still complex to analyze. The simple phenomenal models, on the other hand, are not biological meaningful, but address most key properties of neurons and are less computationally intensive.The three key properties of a neuron that a phenomenological model usually addresses are [19]:

- The ability to generate spikes when the membrane potential crosses a well-defined threshold.
- A reset value to initialize the membrane potential after firing.
- A certain refractory period to depress the neuron from generating another spike immediately.

Phenomenal models, which capture these features, are easier to implement and analyze, hence they are more popular in computational neuroscience.Amongthem, the leaky integrate-and-fire (LIF) model [Ste67, Tuc88] as well as its generalized versions (such as the nonlinear integrate-and fire model) are probably the best know spiking neuronal models. A schematic diagram of the LIF model is shown in Figure 2.7. It is an integrate-and-fire model with a leak" term added to the membrane potential to solve the memory problem. The basic circuit of the LIF model is comprised of a capacitor C in parallel with a resistor R driven by a current I[35]. Based on the circuit, we have:

$$I = \frac{V}{R} + C\frac{dV}{dt} \tag{2.5}$$

If we introduce a time constant $\tau_m = RC$ of the leaky integrator"[36], we get a standard form of the LIF model:

$$\tau_m \frac{dV}{dt} = -V + RI \tag{2.6}$$

where V is the membrane potential and τ_m is the membrane time constant. In this model, if the membrane potential V reaches the threshold value V_{thresh}, the neuron V_{res} and then V is reset to a certain value V_{reset}. In the general version the LIF model also incorporates an absolute refractory period t_{abs}. If the neuron fired at time t, we stop the neuron dynamics for a period of tabs and start the dynamics again at time $t + t_{abs}$ with $V = V_{reset}$. The LIF model is simple enough to implement and easy to analyze. However, it has a severe drawback - it is too simple to reproduce the versatile firing patterns of real neurons[37].

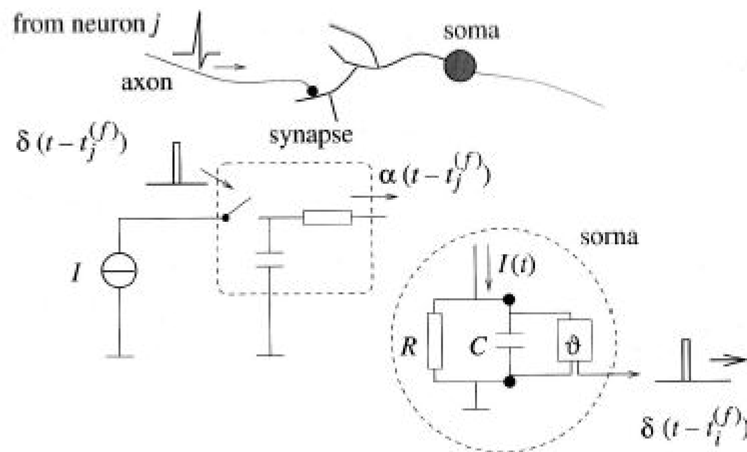


FIGURE 2.4: THE SCHEMATIC DIAGRAM OF THE LEAKY INTEGRATE-AND-RE MODEL BY GERSTNER []

2.9 The Izhikevich Model

Another important phenomenal model is the Izhikevich model [Izh03]. This uses the bifurcation theory to reduce the high-dimensional conductance-based model to a two dimensional system with a fast membrane potential variable v and as low membrane recovery variable u [Izh07]. The Izhikevich model is based on a pair of coupled differential equations [38]:

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{2.7}$$

$$\frac{du}{dt} = a(bv - u) \tag{2.8}$$

if v ¿ = 30mV; then v = c; u = u + d

Where t is time in ms, I is the synaptic current (in mV), v represents the membrane potential (in mV). u represents a membrane recovery variable (also in mV), which reflects the negative effects on the membrane potential caused by some factors such as the active of K+and the inactive of Na^+ ionic current. a, b, c, and d are adjustable parameters [39]:

- a is the time scale of the recovery variable u. Smaller values results in slower recovery. A typical value is $a = 0.02$.
- b is the sensitivity of the recovery variable u to the membrane potential v.Greater values couple v and u more strongly. A typical value is $b = 0.2$.
- c is the after-spike reset value of the membrane potential v. A typical value is $c = -65mV$.
- d is the after-spike offset of the recovery variable u. A typical value is $d = 2$.

It should be noted that the threshold value of this model is typically between $-70mV$ and $-50mV$ and is dynamic. In this model, when the membrane potential v exceeds the threshold value, the neuron spikes with a $30mV$ apex of membrane potential v. The membrane potential v is limited to $30mV$. If the membrane potential v goes above the limitation, it is firstly reset to $30mV$. Then the membrane potential v and the recovery variable u are both reset according to equation 2.10 [40].

There are two important features that make this model ideal for the real-time simulation of a large-scale network. Firstly, the Izhikevich model is computationally simple compared to the Hodgkin-Huxley model in that it takes only 13 floating-point operations to simulate 1 ms of modeling (with 1 ms resolution),but can reproduce firing patterns of all known types of cortical neuron. In comparison, the Hodgkin-Huxley model takes 1200 floating-point operations for 1 ms of modeling. Secondly, one of the most important advantages of the Izhikevich model over the leaky integrate-and-fire model is that the former is capable of reproducing rich firing patterns. With the choice of neuron parameters a, b, c,and d, the Izhikevich model can generate all six known classes of firing pattern.

2.10 Summary

This chapter is helpful in understanding a neural simulation application, it is important to understand neural network modeling and its computational modeling. Our brain is made of billions of neurons - functionally independent processing units with a tremendous amount of connectivity. The neuron's behavior is dictated by its electro-physiological properties controlled by chemical

ions inside and around its cell body. Stimuli to a neuron in the shape of neurotransmitters cause an action potential - a spike or pulse. Neurons communicate with each other using these spikes. All our body movements, responses to our senses and learning/ memories are controlled with these spikes. Much is known about the neural and learning dynamics in the nervous systems. However, a lot remains to be discovered, especially the emergent behaviors of neural networks. Many mathematical models have been proposed based on empirical hypotheses to capture the neural dynamics in the nervous systems.

Chapter 3

Neural Network Simulators

3.1 Introduction

This chapter presents an introduction to the tools and strategies used in the simulation of Spiking Neural Networks (SNN). The description gives, first, a theoretical overview on a possible way of categorizing neural network simulators. The second part of the chapter then focuses on the main simulators developed in the field and describes them relative to the categorization scheme proposed. This chapter gives the reader the grounds for a critical comparison between the various simulators developed in that field and the one in which this thesis work has been carried out[32].

3.2 Categorization of SNN Simulators

In this section a categorization of the SNN simulators is proposed. The first characteristic splits the whole class in two halves: hardware and software simulators.Hardware simulators are those which include the development of dedicated hardware (e.g. a Printed Circuit Board, a special-purpose chip, etc.). Software simulators are those developed to run on a standard computational unit (e.g. a Personal Computer, a computer cluster, etc.). In addition there are hybrid simulators which require both the development of custom hardware and software to complete the simulator (e.g. FPGA device, custom built chip including a computational unit, etc.). Since, for this class of simulators, the development of new hardware is required to provide a computational substrate for the software simulator, they can be classified as hardware neural network simulators [19].

3.2.1 Hardware Simulators

Hardware simulators are those which include the development of dedicated hardware to simulate SNN. This class includes simulators with specific software which runs on dedicated hardware. This class comprises analogue and digital hardware simulators, differentiated by how the neurons are implemented: if neurons are implemented directly with analogue components (transistors, capacitors, resistors, etc.) then it is an analogue hardware simulator. Alternatively, if the simulator is based on digital circuits which run dedicated software of a specific neuron model, and the values of the physical quantities can assume only discrete values, then this is a digital hardware simulator. The literature also describes hardware projects which involve a combination of the two techniques. Often the distribution of the spikes across a system may use digital circuits, while the neural model is simulated using analogue components. In this case the technique used to build the circuit which emulates the neural model is used as reference to identify the type of simulator. Both these classes can be subdivided on the basis of the number of neuron models which the simulator is able to run: single and multiple neuron models. Single neuron model simulators are those which, besides the re-configurability of the synapses and the variability of neuron parameters, permit only one specific neuron model to be run (e.g.leaky integrate-and-fire, adaptive exponential leaky integrate-and-fire, etc.). Multiple neuron model hardware simulators are those which permit different neuron models to run on the same hardware in a reconfigurable network. This means that the same chip permits different neuron models to run in different runs of a simulation. While both analogue and digital simulators are able to run a single neuron model simulation, only digital hardware simulators are likely to be able to simulate multiple neuron models; in digital hardware it is easy to modify the part connected to the computation of the state variable update of the model developed (e.g. modifying the connection between multipliers, adders and other basic components), while in analogue hardware it is very hard to reconfigure the circuitry (e.g. transistors, capacitors and resistances) to a second neuron model. Over all these taxonomies of simulators there is a relation with time: the simulation can be in continuous time (analogue hardware), in discrete time (analogue and digital hardware) or in abstract time (digital hardware) (Rast, 2010). Continuous and discrete time simulations are generally well-known paradigms[41].

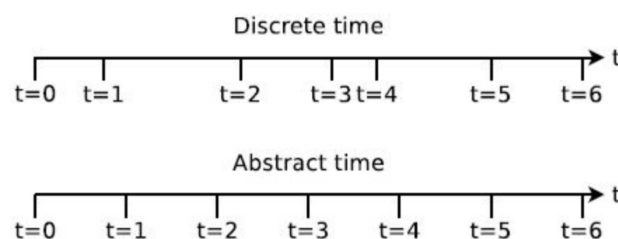


FIGURE 3.1: DIFFERENCE BETWEEN ABSTRACT-TIME SIMULATOR AND DISCRETE-TIME SIMULATOR. [3]

Simulators running in continuous time have the physical values of the simulation always meaningful. Discrete time simulators have the time divided into events in which the values of the simulation are meaningful. An abstract time simulator has temporal slots in which the boundaries have a definition in the real life time. Between these boundaries the computation takes place to move the simulation one step forward integrating the differential equation(s) across one time step. The difference between the abstract time simulator and the discrete time simulator can be described through Fig.2.1: a discrete time simulator presents computation intervals which may not be constant in time. An abstract time simulator has, instead, time-constant computation intervals. Therefore, a discrete time simulator may become an abstract time simulator if the computation interval is kept constant in all the slots. Time relation is a characteristic that transcends the categorization of the simulators. These, in fact, can be classified with respect to biological time (Fig.2.2) [42]:

- Real-Time simulators: the time of the simulation corresponds to biological time.Therefore one millisecond of simulation corresponds to one millisecond in biological time. This is a strict constraint.
- Accelerated time: the time of the simulation runs faster than biological time. Therefore in one millisecond the simulation will model more than one millisecond of activity of the biological model.
- Non real-time simulators: the time of the simulation runs slower than biological time. Therefore in one millisecond the simulation will model less than one millisecond of activity of the biological model.

A hierarchical definition of the classes of hardware simulator is graphically described in Fig.2.3.

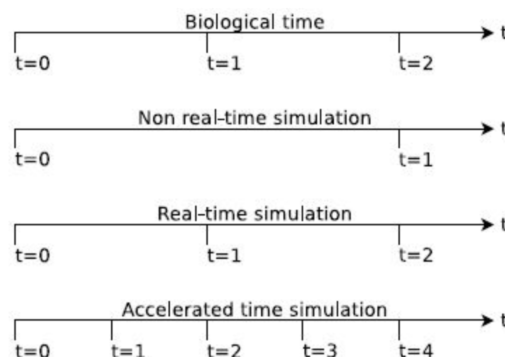


FIGURE 3.2: RELATION BETWEEN SIMULATION TIME AND BIOLOGICAL TIME. [4]

3.2.2 Software Simulators

Software simulators are those developed to run on a standard computational unit (e.g. A Personal Computer, a computer cluster, etc.). The neuron model implemented in these types of simulators is represented by some lines of code which implement the mathematical model (in terms of Ordinary Differential Equations - ODE) of a biological neuron. Since the neuron model here is implemented in software, it is very likely that those simulators are able to simulate multiple neuron models (perhaps even within the same run of a simulation). However it is possible that simulators implement only single neuron model, in which case, the simulator has usually been strongly optimized for a specific neuron model. Software simulators can be clock driven (synchronous simulators) or event driven (asynchronous simulators) and always run in discrete time (Brette et al., 2007) or in abstract time (Rast, 2010). Also, in this case, there is a time relationship which transcends the categories of simulator presented until now. Usually the simulation is slow as the number of neuron increases, because the simulator substrate has a finite computational power shared between all the neurons: the greater the number of neurons simulated, the greater the computation demands of the simulator and hence the greater time taken to perform a time step in the simulation. Eventually, for medium scale neural networks, the time relation goes below the real-time boundary (in other words, it becomes slower than real time)[43].

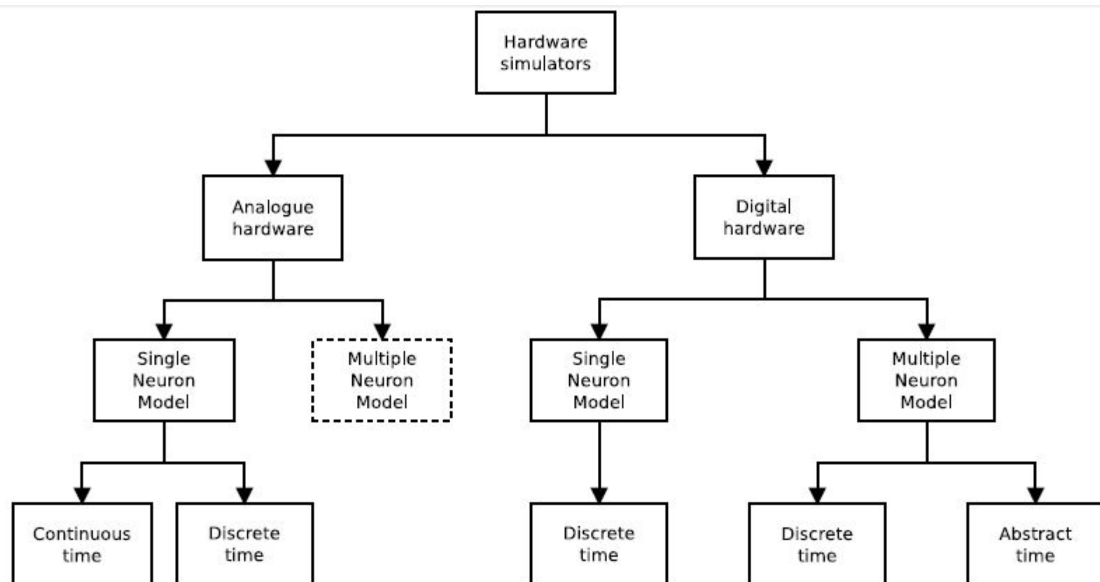


FIGURE 3.3: HIERARCHICAL REPRESENTATION OF HARDWARE NEURAL NETWORK SIMULATORS. [5]

3.3 Review of Main Simulators Developed

In this section a review of some existing simulators is presented; these have usually been developed as part of neuro scientific projects. To reflect the differentiation made in the previous section, here, hardware and software simulators are presented separately[44].

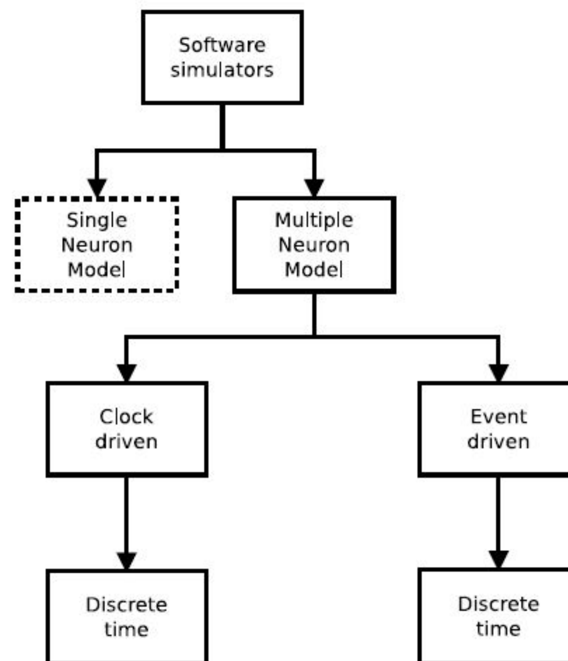


FIGURE 3.4: HIERARCHICAL REPRESENTATION OF SOFTWARE NEURAL NETWORK SIMULATORS. [6]

However, this section presents only a summary of the reviewed simulators. Complete descriptions of each simulator are available in the cited references. Moreover, more complete reviews have been published in numerous articles (e.g. Brette et al. (2007), Misra and Saha (2010), Draghici (2000) and Zhu and Sutton (2003)) [45].

3.3.0.1 Hardware

SyNAPSE SyNAPSE is the acronym of the System of Neuromorphic Adaptive Plastic Scalable Electronics project. The goal of this project is to design a neuromorphic chip which is able to replicate a mammalian brain in size, functionality and power consumption: it should be able to recreate 1010 neurons with 1014 synapses consuming 1KW of electrical power and occupying 2dm3 (liters) of space (Seo et al., 2011; Ananthanarayanan et al., 2009). The simulator uses digital components to simulate a leaky integrate-and-fire neuron model. Spikes are transmitted from one neuron to subsequent ones using a communication crossbar (Merolla et al., 2011) [46].

Four different chips have been produced (Seo et al., 2011):

1. A base design chip with binary synapses and standard leaky integrate-and fire neurons.

2. A slim neuron variant, which fixes spiking, learning parameters and the structure of the network in a two-layer learning network.

3. A 4-bit synapse variant which allows modification of synaptic weights in a way closer to biology.

4. A low leakage variant to reduce power dissipated by neurons. This has some cost in terms of the minimum operating voltage of the memory array.

In summary, this project can be classified as hardware, digital, real-time simulatorwith learning capabilities. The neural model is integrated over steps of 0.1 ms with a discrete time paradigm.

Blue Brain Project The Blue Brain Project aims to provide a computational substrate for molecular-level simulations that present biological realism. The goal of this platform is to simulatethe brains of mammalians with a high level of biological accuracy and, ultimately, tostudy the emergence of biological intelligence (Markram, 2006). The platform Blue Gene/L is provided by IBM and comprises 8,192 CPU nodes each being a PowerPC 440 running at 700 MHz, with a peak performance of 22.4TFLOPS and 2 TB of memory (Markram, 2006). As described before, the software running the simulation involves biological details,therefore this simulator can be classified as: hardware neural network simulator, running slower than real-time with a discrete time paradigm. The capabilities incorporated in this simulator involve learning, as well as other biological details [23].

FACETS BrainScalesS The FACETS (Fast Analog Computing with Emergent Transient States) project delivered wafer-scale integration of neuromorphic chips which simulate adaptive exponential leaky integrate-and-fire neurons (Schemmel et al., 2010). In addition it is possible to reconfigure the circuit such that the adaptation part can be deactivated and hence the neuron can behave as a standard leaky integrate-and-fire neuron. Short-term plasticity and long-term plasticity mechanisms are implemented on the synapses. The distribution of spikes uses digital components and digital interfaces may be used to route spikes between two wafers or to a computer. The system runs 104 times faster than real-time, consequently, this simulator can be classified as multiple-model analogue hardware simulator with learning capabilities running faster than real-time. The FACETS project is completed and the outcome forms the basis of the current BrainScalesS project [22].

Neurogrid The core of this simulator is a neuromorphic analogue chip simulating 256 x 256 leaky integrate-and-fire neurons in real-time (Silver et al., 2007). The distribution of spikes across the system uses digital interconnections which propagate spikes from one layer to the subsequent using an Address Event Representation (AER) protocol (Boahen, 2000). This protocol defines the transmission of neural events (action potentials) in simulators by sending the address of the element which had emitted it. In the Neurogrid project, an external FPGA is required to program arbitrarily neuron interconnectivity. In summary, this is a single-model hardware simulator running in real-time without learning capabilities [47].

VLSI Chips from Zurich The chips built by the group at the Institute for NeuroInformatics (INI) in Zurich implement analogue leaky integrate-and-fire neurons with 28 plastic and 4 non-plastics synapses per neuron, running in real-time (Indiveri et al., 2009). The learning rule implemented in hardware is related to the standard STDP learning rule, but uses bi-stable state synapses: one state provides a high synaptic weight, while the other state provides a low weight. The distribution of the spikes in the architecture takes advantage of the AER protocol. This simulator can be classified as a single-model, analogue, hardware simulator with learning capabilities running in real-time [48].

3.3.0.2 Softwares

Brian Brian is a software neural simulator written in Python (Goodman and Brette, 2008). It is able to simulate multiple neuron models, but much slower than real-time, especially when simulating complex neural networks. It is a clock-driven simulator, where all events take place on a fixed time grid (t = 0, dt, 2dt, 3dt, ..). Learning features are available for the simulation. In summary, this is a software simulator which allows multiple neuron models in the same simulation and implements learning features. It uses a discrete, clock-driven, time paradigm, running slower than real-time [2].

Neuron Neuron is a software simulator for creating and using models of biological neurons and neural circuits (Brette et al., 2007). It is supported by a complete development environment to describe characteristics of neurons and neural circuits. To advance simulations in time, users have a choice between built-in clock-driven methods (a backward Euler and a Crank-Nicholson variant both using fixed time step) and event-driven methods (fixed or variable time step which may be system-wide or local to each neuron, with second order threshold detection). In summary, this is a software simulator which allows multiple neuron models in the same simulation and implements learning features. It uses a discrete clock-driven and event-driven time paradigm, running slower than real time

Nest The purpose of the simulator Nest was to be the reference implementation to supportthe development of neural network simulators (Brette et al., 2007). The networks this simulator is able to run can easily grow up to 105 neurons and beyond, with realisticconnectivity. This simulator supports heterogeneity of neurons and synapses in a single simulation. It implements both a global time-driven simulation mechanism and an event driven algorithm, so that spikes are not fixed on the discrete grid. In summary, Nest is a software simulator which allows multiple neuron models in the same simulation and implements learning features. It uses a discrete clock-driven and event-driven time paradigm, running slower than real time [19].

Genesis Genesis (Wilson et al., 1989) (GEneralNEuralSImulation System) is a simulator thataims to reproduce the biological behavior of neural systems with details ranging from bio-chemical reactions to large scale neural networks. Genesis was the first simulator able to cope with large scale neural networks and the main application is connected with the simulation of biological neural systems. In summary Genesis is a software simulator which allows multiple neuron models in the same simulation and implements learning features[27].

SpikeFunDigiCortex This project aims to build a large-scale biologically-realistic neural network simulatorfor a standard PC (Dimkovic, 2011). The simulator engine is called Digi-Cortex,while the graphical user interface is called SpikeFun. Currently it implements onlythe Izhikevich neuron model with 30 compartments per neuron and AMPA, GABA and NMDA synapses; learning capabilities are based on the standard STDP rule. The simulator uses a discrete time paradigm and its speed depends on the number of neurons simulated. However, even for small scale simulations, the speed of the simulation appears to be slower than real-time. In summary this project can be classified as a software simulator, running a single neuron model in discrete time with learning capabilities [2].

NEF Nengo The Neural Engineering Framework (NEF) is a mathematical background which allows the use of neural networks in the field of control theory (Eliasmith and Anderson, 2004). Using values encoded as neuron spiking rates it is possible to evaluate functions (even non-linear) with the purposes of computing more complex algorithms. Nengo is the software simulator which implements the principles of the NEF. The simulator uses one main neural model, the leaky integrate-and-fire neuron model, and sets of encoders and decoders to represent numeric values. Numeric values are encoded as collection of spike rates for each population of neurons. The number is converted into spike rates by an encoder population, and backward by a decoder population. The principles of the Neural Engineering Framework allow the possibility of synaptic plasticity with the purpose of training the network to perform a particular function. The learning paradigm featured in this case is, probably, supervised training with a teaching signal which performs error correction. In summary, this simulator allows multiple neuron models,

runs slower than real time, has a discrete clock-based time paradigm, and incorporates learning features [3].

3.3.0.3 SpiNNaker

The acronym SpiNNaker stands for Spiking Neural Network Architecture (Furber et al., 2006) which is a hardware-based, real-time, universal, neural network simulator following an event-driven computational approach (Furber et al., 2012; Rast et al., 2010c). This project involves the design of a chip and the development of dedicated software to simulate neural networks (Jin et al., 2008). This system tries to mimic the features of biological neural networks and SpiNNaker system provides some features that have not been proposed by other simulators. For each simulator, the features developed have been marked with a tick. Onlysimulators including software development have been categorized using the Eventdriven and Clock-driven classes. For Hardware simulators the two categories are marked as Non Addressable.

Simulator name	Synapse	Facets - BrainscaleS	Neurogrid	VLSI from Zurich	Brian	Neuron	Nest	Genesis	SpikeFun	NEF - Nengo	Blue Brain Project	SpiNNaker
Non real time					✓	✓	✓	✓	✓	✓	✓	
Real time	✓		✓	✓								✓
Accelerated time		✓										
Event-driven	N.A.	N.A.	N.A.	N.A.		✓	✓	✓	?		?	✓
Clock-driven	N.A.	N.A.	N.A.	N.A.	✓	✓	✓	✓	?	✓	?	
Abstract time												✓
Discrete time	✓				✓	✓	✓	✓	✓	✓	✓	
Continuous time		✓	✓	✓								
Learning capabilities	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
Multiple neural model		✓			✓	✓	✓	✓		✓	✓	✓
Single neural model	✓		✓	✓					✓			
Digital	✓				✓	✓	✓	✓	✓	✓	✓	✓
Analogue		✓	✓	✓								
Software					✓	✓	✓	✓	✓	✓	✓	✓
Hardware	✓	✓	✓	✓							✓	✓

FIGURE 3.5: COMPARISON BETWEEN THE SPIKKING NEURAL NETWORK SIMULATOR [7]

3.4 Neuron Models Available on SpiNNaker

Various neural models have been implemented for the software simulator. Below there is, a list of the most relevant, with the mathematical model (in terms of Ordinary DifferentialEquation(s) - ODE) that each of these implement:

3.4.1 Izhikevich Model

A model of the Izhikevich neuron (Izhikevich, 2003) has been implemented for the SpiNNaker software simulator (Jin et al., 2008). The ODEs representing this mathematical model are:

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u - I$$
$$\frac{du}{dt} = a(bv - u)$$

$if v = 30mV then v = c, u = u + d$

Where the state variable v is the neuron membrane potential, while the state variable u is the neuron recovery variable. The condition $v = 30mV$ is the spiking condition. Whenever this condition is met, the neuron fires (emits an action potential) and then goes back to the reset state expressed in the same formula. The constants 0:04, 5 and 140 have been chosen by Izhikevich (2004); using these it is possible to simulate at least 20 biologically meaningful spiking behaviors. a, b, c and d are the four parameters of the Izhikevich neuron model (Izhikevich, 2003):

- The parameter a describes the time-scale of the recovery variable u. Smaller values result in slower recovery.
- The parameter b describes the sensitivity of the recovery variable u to the subthreshold fluctuations of the membrane potential v. Greater values couple v and u more strongly resulting in possible sub-threshold oscillations and low-threshold spiking dynamics.
- The parameter c describes the after-spike reset value of the membrane potential v caused by the fast high-threshold K+conductance.
- The parameter d describes after-spike reset of the recovery variable u caused by slow high-threshold Na+ and K+conductance. The mathematical equations are integrated twice in each time step to avoid the mathematical instabilities suggested by Izhikevich (2010): a biologically unrealistic oscillation of the membrane potential due to an integration step too long.

3.4.2 Leaky Integrate-and-Fire Model

A model of the Leaky Integrate-and-Fire neuron (Dayan and Abbott, 2001) has also been implemented for the SpiNNaker system (Rast et al., 2010b). The mathematical model of this neuron is:

$$\tau\frac{dv}{dt} = V_L - V + R_m \times I_{input}$$

The spike condition is defined by the condition $V >= V_{threshold}$. Whenever the neuron membrane potential satisfies this condition, it emits a spike and then the membrane potential resets to V_{reset}. Where:

V is the neuron membrane potential. τ is the membrane time constant of the neuron. If the input is null, the membrane potential exponentially relaxes with this time constant. V_L is the resting potential of the cell. If the input is null, the membrane potential of the neuron decays exponentially to this value; R_m is the membrane resistance; I_{input} is the current injected in the neuron; $V_{threshold}$ is the threshold potential for a spike emission. V_{reset} is the reset value for the membrane potential after the neuron emits spike.

This neuron model has been implemented in various versions according to the precision required for the simulation: the first implementation used 16 bits in fixed point precision, with 8 bits representing the integer part and 8 bits representing the decimal part. A second implementation has been produced with 32 bit precision: 16 bits to represent each of the integer and decimal parts. This implementation replaced the 16 bit one.

3.4.3 Poisson Spike Source Generator Neuron

This neuron generates a spike train according to a Poisson process: the Inter-Spike Interval (ISI) is an instance of a Poisson random variable generated during the simulation. Limitations in the implementation, due to the fixed-point arithmetic, limit the rate of the spike train in the interval between 25Hz ¡= rate ¡= 1000Hz [4].

3.4.4 Spike Source Neuron

This neuron generates spikes according to a pattern saved in the SDRAM memory chip. Due to limitations in the memory available for this purpose, the input pattern has to be less than 8MB.

3.4.5 Spike Source Live Neuron

This neuron generates spikes according to data input from a host computer through theEthernet channel. This neuron works in partnership with the SpikeServer software that will be introduced in the next chapter.

3.4.6 NEF Interface Neurons

To import the Neural Engineering Framework (NEF) (Eliasmith and Anderson, 2004) into the SpiNNaker simulator, a value encoder neuron and a value decoder neuron are required (Galluppi et al., 2012b), as described earlier in section 2.3.2

3.4.7 Plasticity Models Available

In the SpiNNaker architecture synaptic weights are available for computation only when a spike event is received. Therefore, to trigger potentiation when the postsynaptic neuron emits an action potential, an implementation model is required to store this information until the subsequent presynaptic spike is received. This model is called the Deferred Event Model (Rast et al., 2009), and has been applied to reproduce two learning rules. The first algorithm has been developed according to the Spike-Timing-Dependent Plasticity (STDP) general rule (Bi and Poo, 1998; Jin et al., 2009). A second algorithm has been developed to reproduce a simplified learning rule: the spike-pair STDP (also known as nearest-neighbour STDP) [5].

This research discovers that there are also ranges of other popular neuronal models such as the Hodgkin-Huxley model which can be implemented on SpiNNaker. It also leads to the point that different models can be integrated into the system library, and then users can choose which model to use in their simulation. Moreover, for better utilization of SpiNNaker system, it is necessary we built novel model which is comparative to renowned model but less in computation cost and it can be verified using software simulator such as NEST. This is the main contribution of author for his dissertation.

3.4.8 Needs of Implementation of More Neuronal Models

This research discovers that there are also ranges of other popular neuronal models such as the Hodgkin-Huxley model which can be implemented on SpiNNaker. It also leads to the point that different models can be integrated into the system library, and then users can choose which model to use in their simulation. Moreover, for better utilization of SpiNNaker system, it is necessary we built novel model which is comparative to renowned model but less in computation cost and it can be verified using software simulator such as NEST. This is the main contribution of this thesis.

3.5 Summary

The purpose of this chapter was to put neural network hardware & software simulators in historical context and reviewed the main research developments in neural network simulator. It considered the architectures: basic platforms for neural networks that have capabilities of simulation. There is no reason to believe that previous hardware designed for biological simulation and it performed well in computational applications. This chapter discussed that one domain where neural networks may be particularly valuable is embedded systems. Such applications usually demand real-time adaptability for simulating the neural network modes. It also inspected that software neural simulator is very important for model exploration and validation. Here the discussion used the SpiNNaker chip as a specific example to introduce as Spiking Neural Network architecture. The study examined that there are range of some popular neuronal models was implemented on this architecture so there is need of implementation of Hodgkin-Huxley model on this architecture.

Chapter 4

Implementation of Spiking NN models over NEST

4.1 Introduction

In Computational Neuroscience, simulations are used to investigate models of the nervous system at functional or process levels. Consequently, a lot of effort has been put into developing appropriate simulation tools and techniques, and a plethora of simulation software, specialized for the single neuron or small sized networks is available. Recently, however, there has been growing interest in large scale simulations, involving some 104 neurons while maintaining an acceptable degree of biological detail. Thus, there is need for simulation software, possibly parallel, which supports such simulations in a flexible way. Here, we describe the Neural Simulation Technology (NEST) initiative, a collaborative effort to develop an open simulation framework for biologically realistic neuronal networks. The system is distinguished by following features.

- It aims at large structured networks of heterogeneous, biologically realistic elements at different description levels.
- It employs an iterative incremental development strategy, maintaining a running system at any time.
- The software is developed as a collaborative effort of several research groups. Since NEST is a research tool,it has to continuously adapt to the ever changing demands of the researchers who are using it.
- In an electrophysiological experiment, a number of different devices are used to stimulate and observe the neuronal system. Accordingly, devices are explicitly mapped to the simulation.

- It is written in an object-oriented style (C++). An abstract base class defines a core interface which has to be implemented by each module. The various modules are combined by a module loader, which is part of a simulation language interpreter (SLI).

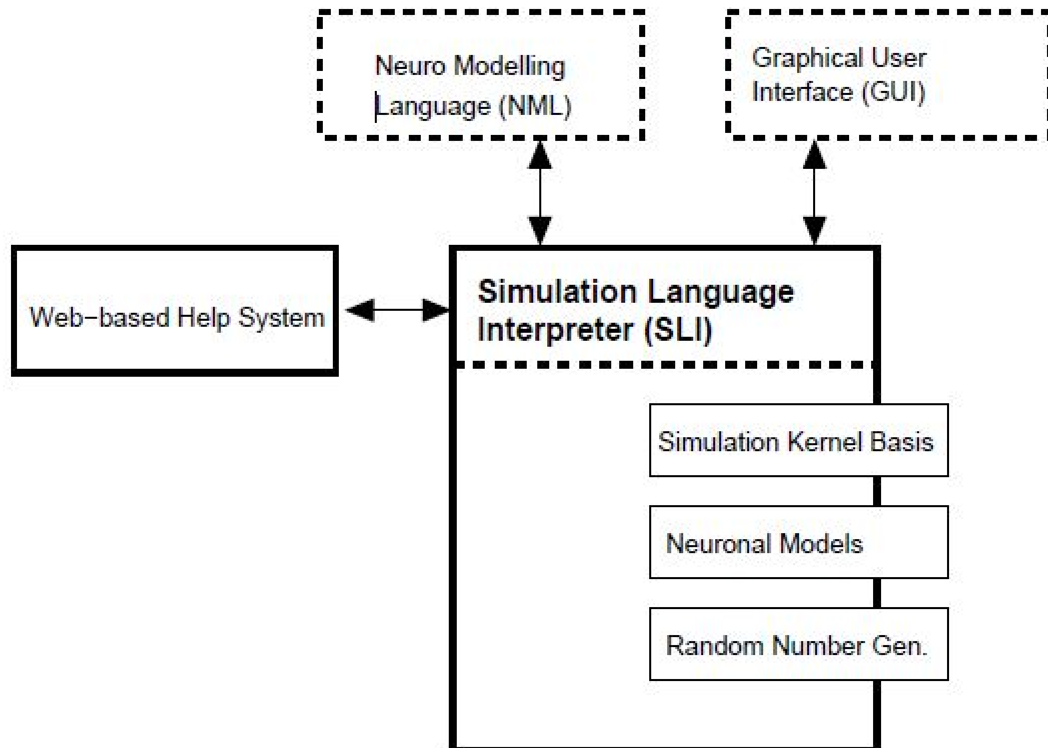


FIGURE 4.1: STRUCTURE OF THE SIMULATION SYSTEM AND ITS MAIN PARTS ARE A SIMULATION KERNEL, SIMULATION LANGUAGE INTERPRETER (SLI) AND SOME AUXILIARY MODULES. THE SIMULATION LANGUAGE INTERPRETER INTEGRATES ALL PARTS AND ACTS AS INTERFACE TO THE USER.

4.1.1 A NEST simulation consists of following main components

4.1.1.1 Nodes

Nodes are all neurons, devices, and also sub-networks. Nodes have a dynamic state that changes over time and that can be influenced by incoming events.

4.1.1.2 Events

Events are pieces of information of a particular type. The most common event is the spike-event. Other event types are voltage events and current events.

4.1.1.3 Connections

Connections are communication channels between nodes. Only if one node is connected to another node, can they exchange events.

4.1.1.4 Classes

NEST offers a number ofmechanisms to structure and organize not only your simulations, but also your simulation data using Python interface.

4.1.1.5 Model Variations

To change minor aspects of a model. For example, in one version it has homogeneous connections and in another randomized weights.

4.1.1.6 Data management

To run simulations with different parameters or other variations and forget to record which data file belonged to which simulation. Pythons class mechanisms provide a simple solution.

To organize the model into a class, by realizing that each simulation has five steps which can be factored into separate functions:

- Define all independent parameters of the model. Independent parameters are those that have concrete values which do not depend on any other parameter.
- Compute all dependent parameters of the model. These are all parameters or variables that have to be computed from other quantities (e.g. the total number of neurons).
- Create all nodes (neurons, devices, etc.)
- Connect the nodes.
- Simulate the model.

We translate these steps into a simple class layout that will fit most models:

4.2 Need/Requirement of PyNN

The first user interface for NEST was the simulation language SLI, a stack-based language derived from PostScript.However, programming in SLI turned out to be difficult to learn and researchers asked for a more convenient programming language for NEST. When it was decided to use Python as the new simulation language, it was almost unknown in Computational Neuroscience. In fact, Matlab was far more common, both for simulations and for analysis. But the Python has a number of advantages over commercial software like Matlab such as it is a powerful interactive programming language with a surprisingly concise and readable syntax and supports many programming paradigms such as object-oriented and functional programming installed by default on all Linuxand Mac-OS based computers. Moreover, a number of neuroscience laboratories meanwhile use Python for simulation and analysis, which further supports this choice.But there is also a drawback that is whenever a new feature is implemented in the application, the interface to Python must be changed as well.Similarly on high performance computer Python is not available for NEST simulation. In order, to avoid two different interfaces, there is need of common interface that enables simulation scripts to be run on any supported system.

Thus researcher developed PyNN (pronounced pine), a common interface for specifying models that would work across multiple systems would retain the benefits of simulator/hardware diversity while reducing or removing the translation barrier and enables simulation scripts to be run on any supported system.Itis both an application programming interface (API) for the Python programming language and an implementation of the interface for a number of systems. A simulation script is written in Python, with neuronal network modeling functions and classes provided by PyNN.It takes care of translating the neuron, synapse and network models into the required form for a given simulator,consistent handling of physical units, and consistent handling of random numbers, and provides a high-level, object-oriented interface to enable structured development of large-scale, complex models.

4.2.1 Using PyNN

A neural network in NEST consists of two basic element types: Nodes and connections. Nodes are neurons, devices or subnetworks. Devices are used to stimulate neurons or to record from them. Nodes can be arranged in subnetworks to build hierarchical networks like layers, columns, and areas. After starting NEST, there is one empty subnetwork, the so-called root node. New nodes are created with the command Create(), which takes the model name and optionally the number of nodes as arguments and returns a list of handles to the new nodes. These handles are integer numbers, called ids. Most PyNN functions expect or return a list of ids. Thus it is easy to apply functions to large sets of nodes with a single function call. Nodes are connected using

Connect(). Connections have a configurable delay and weight. The weight can be static or dynamic. Different types of nodes and connections have different parameters and state variables. To avoid the problem of fat interfaces, we use dictionaries with the functions GetStatus() and SetStatus() for the inspection and manipulation of an elements configuration. The properties of the simulation kernel are controlled through the commands GetKernelStatus() and SetKernelStatus(). PyNN contains the submodules $raster_plot$ and $voltage_trace$ to visualize spike activity and membrane potential traces. They use Matplotlib internally and are good templates for new visualization functions.

4.2.1.1 Data Conversion

From PyNN to SLI The data conversion between PyNN and SLI exploits the fact that most data types in SLI have an equivalent type in PyNN. The function $sli_push()$ calls PyObjectToDatum() to convert a Python object py_object to the corresponding SLI data type. PyObjectToDatum() determines the type of py_object in a cascade of type checks (e.g. $PyInt_Check()$, $PyString_Check()$, PyFloatCheck()). The listing below shows how this technique is used for the conversion of the PyNN type float and for arrays of doubles:

```
void Datum* PyObjectToDatum(PyObject *py_object)
{
if (PyFloat_Check(py_object))
{
return new DoubleDatum(PyFloat_AsDouble(
py_object));
}
if (PyArray_Check(py_object)) 10 {
intsize = PyArray_Size(py_object);
PyArrayObject *array;
array = (PyArrayObject*) py_object;
assert(array != 0);
switch (array->descr->type_num)
{
case PyArray_DOUBLE:
{
double *begin = (double*) array->data;
return new DoubleVectorDatum(
new std::vector<double>(
begin, begin+size));
}} }
```

From SLI to PyNN To convert a SLI data type to the corresponding PyNN type, it avoids the cascade of type checks, since all SLI data types are derived from a common base class, called Datum. This would add a pure virtual conversion function convert() to the class Datum. Each derived class (e.g. DoubleDatum, DoubleVectorDatum) then overloads this function to implement its own conversion to the corresponding PyNN type. This approach is shown for the SLI type DoubleDatum in the following listing. The function get() is implemented in each Datum and returns its data member.

```
void  Datum::use_converter(DatumConverter& converter)
{converter.convert_me(* this);
} PyObject*DoubleDatum::convert()
{return PyFloat_FromDouble(get());}
```

The function $use_converter()$ is defined in the base class Datum and inherited by all derived classes.

```
class DatumConverter
{public: virtual void convert_me(Datum&);
virtual void convert_me(DoubleDatum&)=0;
virtual void convert_me(DoubleVectorDatum&)=0;};
```

The PyNN specific part of the conversion is encapsulated in the class DatumToPythonConverter, which derives from DatumConverter and implements the $convert_me()$ functions to actually convert the SLI types to Python objects.

```
void DatumToPythonConverter::convert_me(
DoubleDatum&dd)
{py_object = PyFloat_FromDouble(dd.get());}
```

The diagram in Figure illustrates the sequence of events in $sli_pop()$: First, $sli_pop()$ retrieves a SLI Datum d from the operand stack (not shown). Second, it creates an instance of DatumToPythonConverter and calls its convert() function, which then passes itself as visitor to the $use_converter()$ function of $d.Datum :: use_converter()$ calls the DatumToPythonConverters $convert_me()$ function that matches the type of d. The function $convert_me()$ then creates a new Python object from the data in d and stores it in the DatumToPythonConverters member variable py_object, which is returned to $sli_pop()$.

4.2.1.2 Data Handling

Recorded data in PyNN is always associated with the Population or Assembly from which it was recorded. Data may either be written to file, using the $write_{d}ata()$ method, or retrieved as objects in memory, using $get_{d}ata()$. Handling of recorded data in PyNN makes use of the NEO package, which provides a common Python data model for neurophysiology data (whether real or simulated). The $get_{d}ata()$ method returns a Neo Block object. This is the top-level data container, which contains one or more Segments. Each Segment is a container for data sharing a common time basis - a new Segment is added every time the $reset()$ function is called. A Segment can contain lists of AnalogSignal, AnalogSignalArray and SpikeTrain objects. These data objects inherit from NumPys array class, and so can be treated in further processing (analysis, visualization, etc.) in exactly the same way as NumPy arrays, but in addition they carry metadata about units, sampling interval, etc.

```
import pyNN.neuron as sim  # can of course replace `neuron` with `nest`, `brian`, etc.
import matplotlib.pyplot as plt
import numpy as np

sim.setup(timestep=0.01)
p_in = sim.Population(10, sim.SpikeSourcePoisson(rate=10.0), label="input")
p_out = sim.Population(10, sim.EIF_cond_exp_isfa_ista(), label="AdExp neurons")

syn = sim.StaticSynapse(weight=0.05)
random = sim.FixedProbabilityConnector(p_connect=0.5)
connections = sim.Projection(p_in, p_out, random, syn, receptor_type='excitatory')

p_in.record('spikes')
p_out.record('spikes')                    # record spikes from all neurons
p_out[0:2].record(['v', 'w', 'gsyn_exc']) # record other variables from first two neuron

sim.run(500.0)

spikes_in = p_in.get_data()
data_out = p_out.get_data()

fig_settings = {
    'lines.linewidth': 0.5,
    'axes.linewidth': 0.5,
    'axes.labelsize': 'small',
```

```
    'legend.fontsize': 'small',
    'font.size': 8
}
plt.rcParams.update(fig_settings)
plt.figure(1, figsize=(6,8))

def plot_spiketrains(segment):
    for spiketrain in segment.spiketrains:
        y = np.ones_like(spiketrain) * spiketrain.annotations['source_id']
        plt.plot(spiketrain, y, '.')
        plt.ylabel(segment.name)
        plt.setp(plt.gca().get_xticklabels(), visible=False)

def plot_signal(signal, index, colour='b'):
    label = "Neuron %d" % signal.annotations['source_ids'][index]
    plt.plot(signal.times, signal[:, index], colour, label=label)
    plt.ylabel("%s (%s)" % (signal.name, signal.units._dimensionality.string))
    plt.setp(plt.gca().get_xticklabels(), visible=False)
    plt.legend()

n_panels = sum(a.shape[1] for a in data_out.segments[0].analogsignalarrays) + 2
plt.subplot(n_panels, 1, 1)
plot_spiketrains(spikes_in.segments[0])
plt.subplot(n_panels, 1, 2)
plot_spiketrains(data_out.segments[0])
panel = 3
for array in data_out.segments[0].analogsignalarrays:
    for i in range(array.shape[1]):
        plt.subplot(n_panels, 1, panel)
        plot_signal(array, i, colour='bg'[panel%2])
        panel += 1
plt.xlabel("time (%s)" % array.times.units._dimensionality.string)
plt.setp(plt.gca().get_xticklabels(), visible=True)

plt.show()
```

4.2.2 Hodgkin-Huxley model over NEST

The NEST simulates the dynamics of the membrane voltage, the membrane current and the Hodgkin-Huxley variables h,m,n of the ordinary differential equations. The neuron model has no spatial structure, i.e., only a single compartment is simulated. Following are main attribute of its implementation

- Compartment Characteristics: This section defines the compartment physics like its diameter, the membrane capacitance Cm, the axial resistance Ra, the sodium conductance GNa, the potassium conductance GK and the leakage conductance Gl. It is also possible to alter the concentration of potassium and sodium inside and outside the compartment with the four variables Ki, Ko, Nai and Nao.
- Applying External Current/Voltage: The external excitation of the compartment is normally a step like function of current or voltage. It is helpful in taggling between voltage and current .The beginning of the simulation indicates the time at which the current or voltage step is applied. The duration of the stimulus should be entered in the field marked 'duration'.
- Simulation Parameters: The begin and end values control the total interval of a simulation. This allows you to continue the integration of the Hodgkin Huxley model even after the stimulation has stopped.
- Voltage Clamp: The characteristics of Hodgkin-Huxley cells may best be studied by considering some voltage clamp experiments. In voltage clamp experiments, the voltage is fixed. Hence the capacitive current Ic is set to zero. As in the voltage clamp experiments, the potassium current is outward and the sodium current inward. In particular, the sodium current reacts rapidly to an increase in the potential whereas the potassium current reacts more slowly.
- Current Clamp: It is also called space clamp since it shunts the inner axial resistance Ra. Injected current is therefore uniformly distributed over the part of axon which is investigated. The effect is that an axon which normally has some spatial characteristics will be transformed to an axon that behaves like one single big compartment.
- Spike Variation: The spike shapes can mainly be controlled by the intensity of the current that flows into the cell (simulated here by the externally applied current). The amount of time needed to produce a new spike after a first one is called the ”refractory period”. The shorter this time, the less time the spike has to attain its peak value. In the extreme case, it oscillates around some value between the resting state and the peak value.

4.2.2.1 Using PyNN- Hodgkin-Huxley Model

PyNN with a simulation of a neuron receiving input from an excitatory and an inhibitory population of neurons. Each pre-synaptic population is modeled by a Poisson generator, which generates a unique Poisson spike train for each target. The simulation adjusts the firing rate of the inhibitory input population such that the neurons of the excitatory population and the target.

```
from nest import *
import nest.voltage_trace as plot
t_sim = 100000.0 , n_ex = 16000
n_in = 4000 , r_ex = 5.0 ,epsc = 45.0
ipsc = 45.0, d = 1.0, lower = 5.0 upper = 25.0 , prec = 0.05

neuron = Create("hh_neuron")
noise= Create("poisson_generator", 2)
voltmeter = Create("voltmeter")
spikedetector = Create("spike_detector")
SetStatus([ noise [0]], [{ "rate" : n_ex*r_ex }])
SetStatus(voltmeter, [{ "interval" : 1000.0,"withgid" : True}])
```

First, we import all necessary modules for simulation, analysis and plotting neuron fire at the same rate.Second, the parameters for the simulation are set. Third, the nodes are created using Create(). Its arguments are the name of the neuron or device model and optionally the number of nodes to create. If the number is not specified, a single node is created. Create() returns a list of ids for the new nodes, which we store in variables for later reference.Fourth, the excitatory Poisson generator (noise[0]) and the voltmeter are configured using SetStatus(), which expects a list of node handles and a list of parameter dictionaries. The rate of the inhibitory Poisson generator is set . For the neuron and the spike detector we use the default parameters. Fifth, the neuron is connected to the spike detector and the voltmeter, as are the two Poisson generators to the neuron.

```
Connect(neuron, spikedetector)
Connect(voltmeter, neuron)
ConvergentConnect(noise, neuron,[epsc, ipsc], [d, d])
```

The command Connect() has different variants. Plain Connect() just takes the handles of pre-synaptic and postsynaptic nodes and uses the default values for weight and delay. Convergent-Connect() takes four arguments: A list of presynaptic nodes, a list of postsynaptic nodes, and

lists of weights and delays. It connects all pre-synaptic nodes to each postsynaptic node. All variants of the Connect() command reflect the direction of signal flow in the simulation kernel rather than the physical process of inserting an electrode into a neuron. For example, neurons send their spikes to a spike detector, thus the neuron is the first argument to Connect(). By contrast, a voltmeter polls the membrane potential of a neuron in regular intervals, thus the voltmeter is the first argument of Connect().

To determine the optimal rate of the neurons in the inhibitory population, the network is simulated several times for different values of the inhibitory rate while measuring the rate of the target neuron. This is done until the rate of the inhibitory neurons is determined up to a given relative precision, such that the target neuron fires at the same rate as the neurons in the excitatory population. The algorithm is implemented in two steps: First, the function $output_rate()$ is defined to measure the firing rate of the target neuron for a given rate of the inhibitory neurons.

```
SetStatus([ noise [1]], [{"rate": rate}])
SetStatus(spikedetector, [{"n_events": 0}])
Simulate(t_sim)
n_events = GetStatus(spikedetector,"n_events")[0]
r_target = n_events*1000.0/t_sim
print "r_in = %.4f Hz," % guess,
print "r_target = %.3f Hz" % r_target
return r_target
```

The function takes the firing rate of the inhibitory neurons as an argument. It scales the rate with the size of the inhibitory population and configures the inhibitory Poisson generator (noise[1]). The spike-counter of the spike detector is reset to zero. It simulates the network using Simulate(), which takes the desired simulation time in milliseconds and advances the network state by this amount of time. During the simulation, the spike detector counts the spikes of the target neuron and the total number is read out at the end of the simulation period. The return value of $output_rate()$ is an estimate of the fi ring rate of the target neuron in Hz. Finally, we plot the target neurons membrane potential as a function of time.

$plot.from_device(voltmeter, timeunit = "s")$

A transcript of the simulation session and the resulting plot are shown in following figures

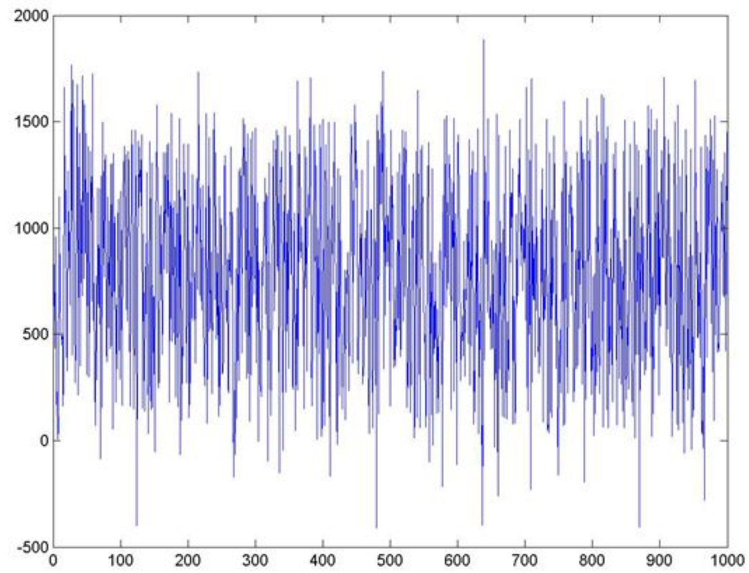


FIGURE 4.2: INPUT CURRENT I OF AN EXCITORY NEURON.

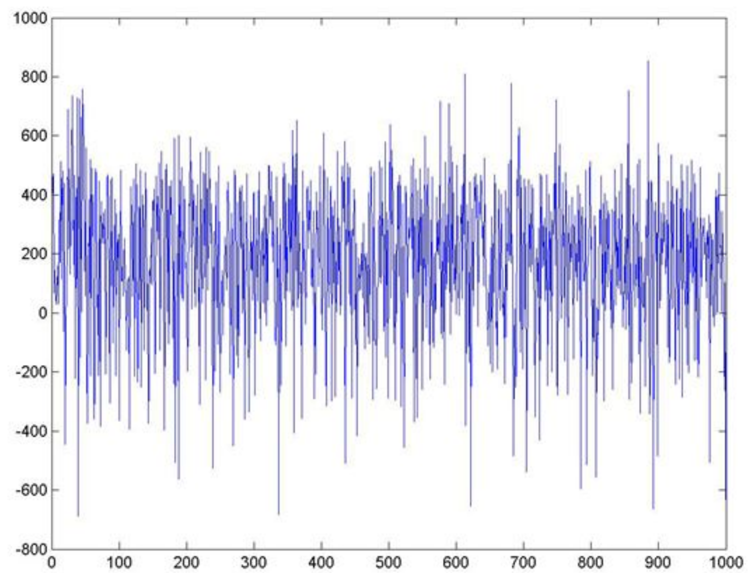


FIGURE 4.3: INPUT CURRENT I OF AN INHIBITORY NEURON.

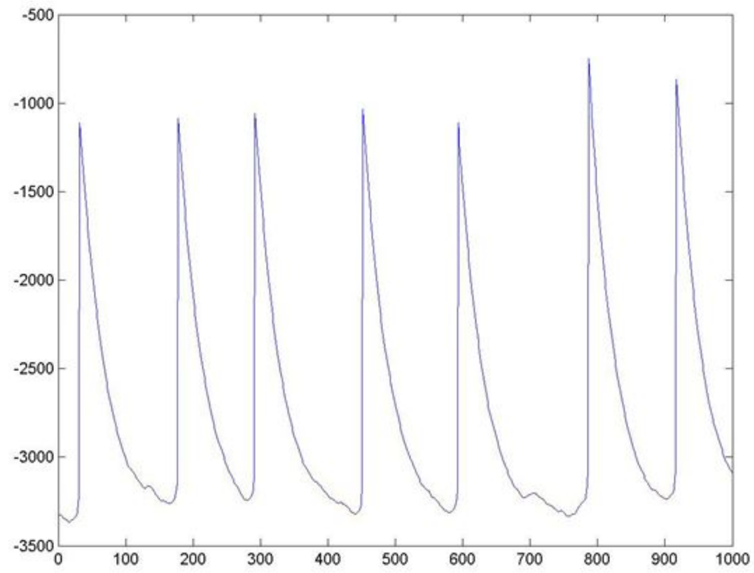


FIGURE 4.4: MEMBRANE VOLTAGE OF AN EXCITORY NEURON.

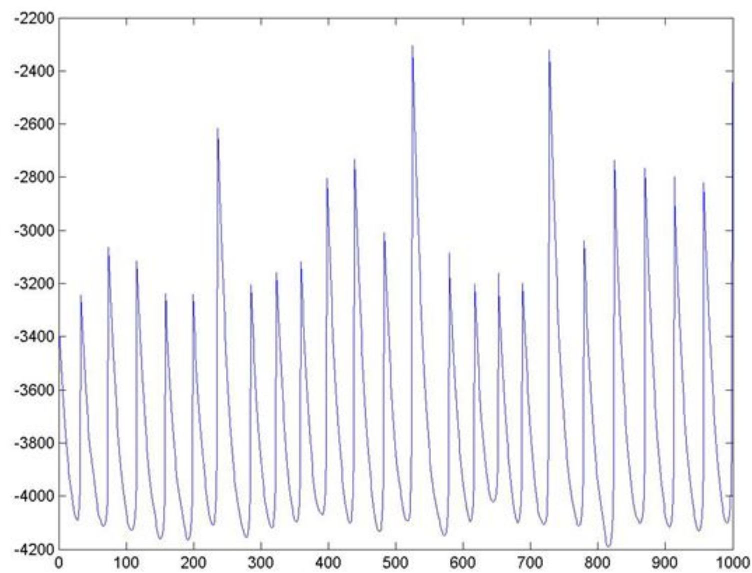


FIGURE 4.5: MEMBRANE VOLTAGE OF AN INHIBITORY NEURON.

4.2.3 Fixed Point Unit vs Floating Point Unit Implementation of HH Model

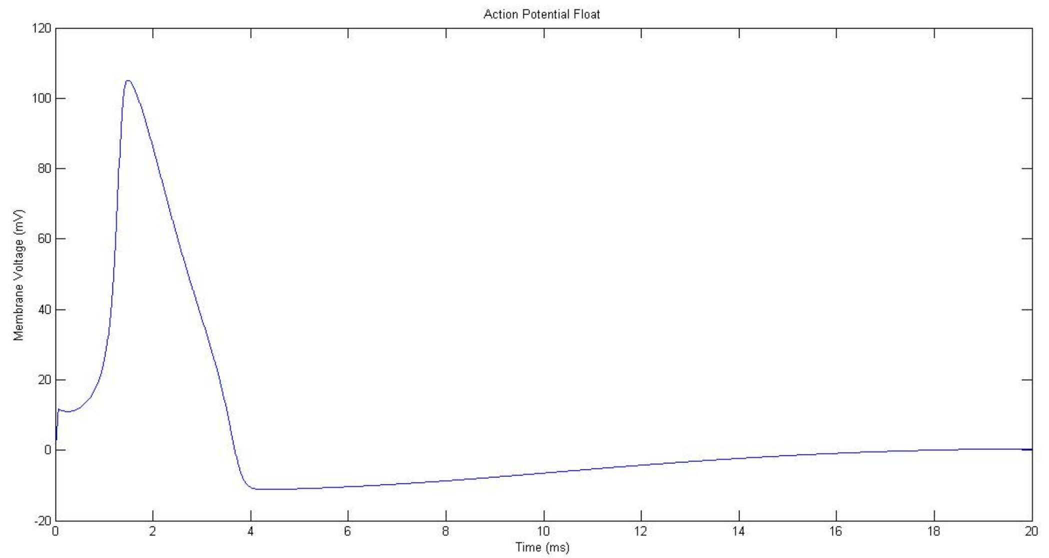


FIGURE 4.6: FIXED POINT UNIT VS FLOATING POINT UNIT IMPLEMENTATION OF HH MODEL.

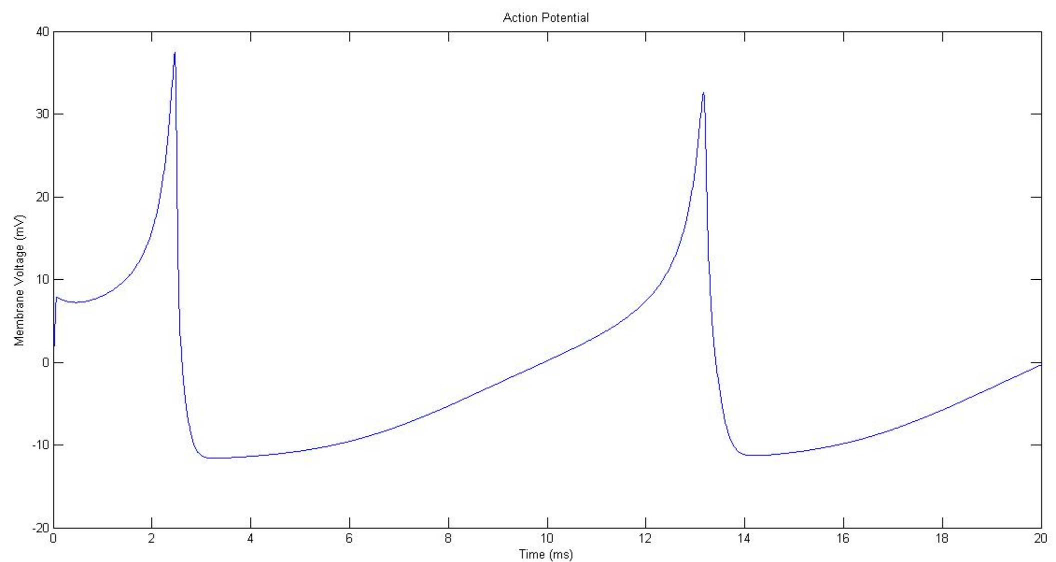


FIGURE 4.7: ACTION POTENTIAL USING FLOATING POINT UNIT.

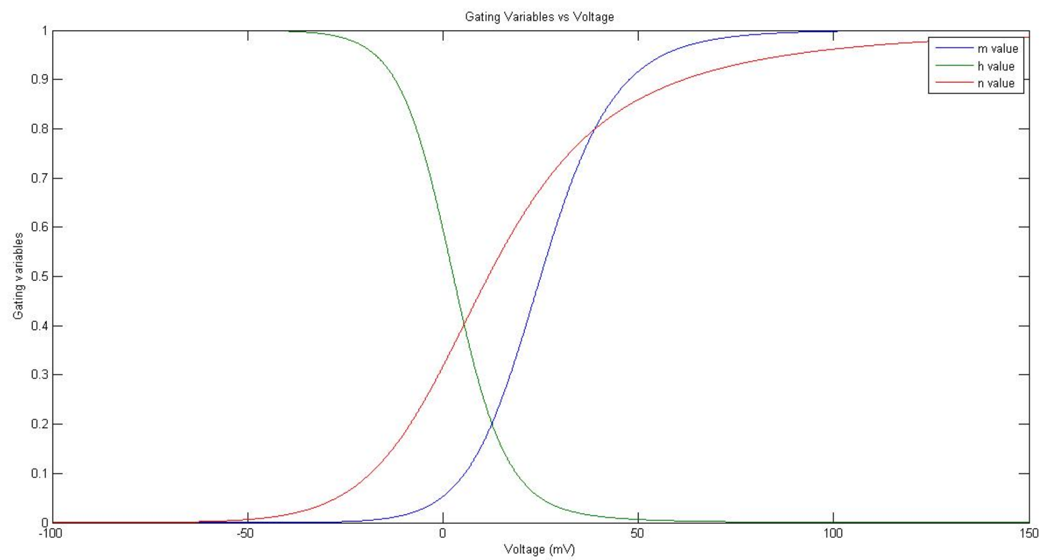


FIGURE 4.8: ACTION POTENTIAL USING FIXED POINT UNIT.

In Figure : Action Potential using floating point unit has been initiated by current pulse before t=0. The time course of the membrane potential shows the action potential (positive peak) followed by a relative refractory period where the potential is below the resting potential. In the spike response framework, the time course of the action potential for t ¿ 0. The overall effect of the sodium and potassium currents is a short action potential followed by a negative overshoot.The amplitude of spike is 100 mV and the spike has been initiated by a short current pulse of 1 millisecond duration applied at t¡0. In the Figure 1.2 the action potential using fixed point unit has been taken by NEST (software based simulator) and the voltage scales is different from voltage scale of floating point unit because of data over flow. In fact Hodgkin-Huxley model exceeds its memory of 64 bits data range using fixed point unit, thus positive peak of membrane voltage followed by refractory period after 3 millisecond.

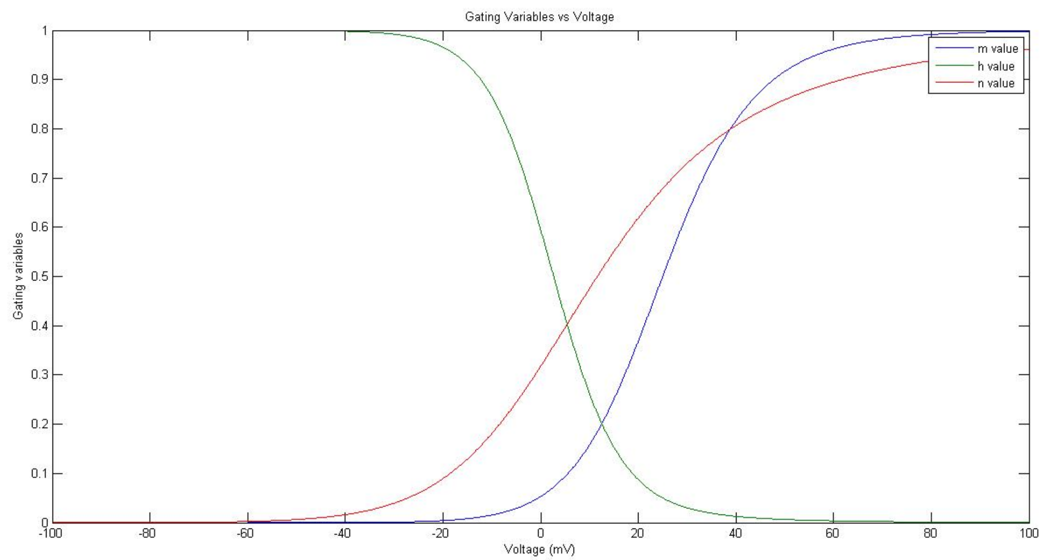


FIGURE 4.9: EQUILIBRIUM FUNCTION FOR THREE VARIABLES M,N,H USING FLOATING POINT UNIT.

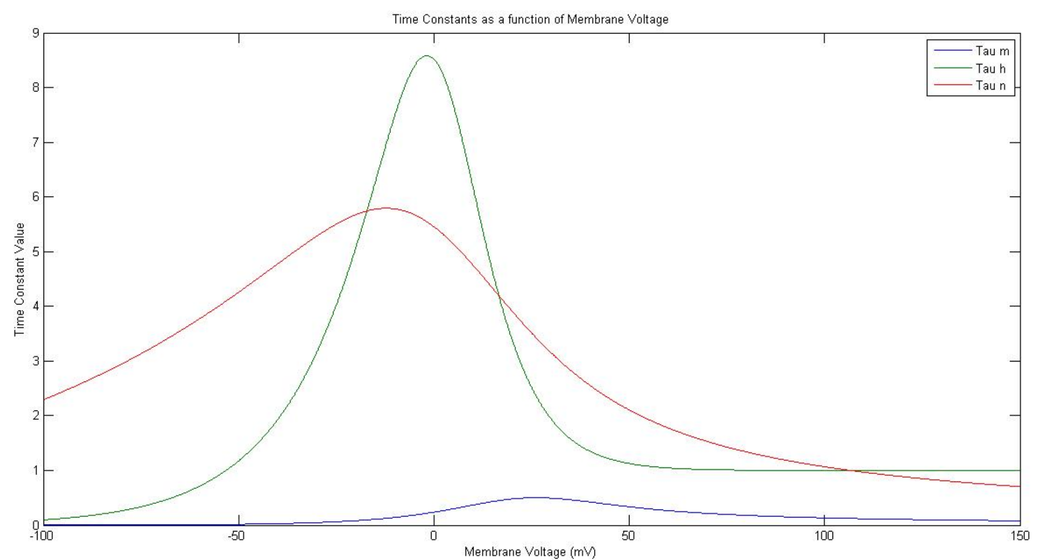


FIGURE 4.10: EQUILIBRIUM FUNCTION FOR THREE VARIABLES M,N,H USING FIXED POINT UNIT.

As both Figure 4.9 and Figure 4.10 respectively depicts that values of m and n increases as voltage increases while h decreases. Therefore, the rise of membrane voltage depends upon some external input and the conductance of sodium channel increases due to increasing value of m. Consequently, positive sodium ions flow into the cell and raise the membrane potential even further.

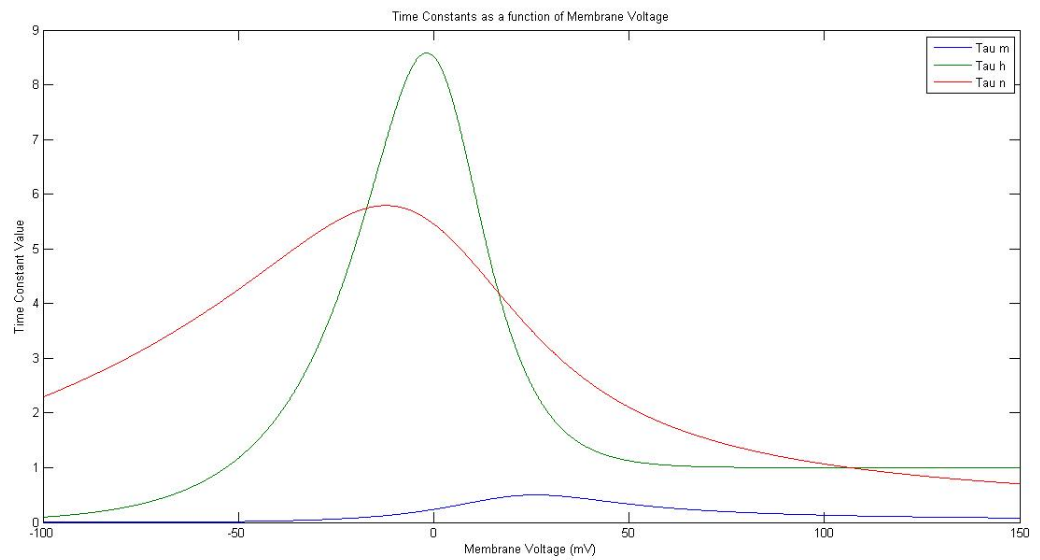


FIGURE 4.11: TIME CONSTANT FOR THREE VARIABLES M,N,H USING FLOATING POINT UNIT.

4.3 Summary

The aim of this chapter is to implement the Hodgkin-Huxley model using NEST simulator. We used PyNN simulation script for erasing the translation barrier in order to run on hardware architecture. We derived a scheme of fixed point implementation of Hodgkin-Huxley model and evaluated its precision with floating point unit implementation by executing it on NEST simulator. From analytical and experimental results we observed that that our proposed scaling factor scheme computationally efficient, requires less memory space and is more scalable floating point scheme.

Chapter 5

Implementing Spiking Neural Network Model over SpiNNaker

5.1 Introduction

In SpiNNaker, the treatment of spikes is a key innovation implemented with application-specific hardware: a multicast, packet-switched and self-timed communication fabric with on-chip routers. To maintain flexibility and generality, the neuronal models run in software on embedded ARM968 processors. These neuronal models communicate by means of spike packets directly supported by the SpiNNaker architecture. The SpiNNaker test chips in 2009 with the batch arriving in Manchester in December. Here, we offer our research thesis and running spiking neurons based on Hodgkin-Huxley model and novel model is known as AJ model which is comparative to Izhikevichs model in function but computationally is less expensive than it.

5.2 Feature of SpiNNaker

This system tries to mimic the features of biological neural networks in various ways:

- **Native Parallelism:** Each biological neuron is a primitive computational element within a massively parallel system. Likewise, SpiNNaker uses parallel computation.
- **Spiking communications:** In biology, neurons communicate through spikes.The SpiNNaker architecture uses source-based and address event representation (AER) packets to transmit the equivalent of neural signals (i.e. action potentials). Each AER packet identifies the event source through an addressing scheme.The time of the event is basically identified by the time of the packet itself.

- **Event-Driven Behavior:** Neurons are very power efficient, and consume muchless power than modern hardware. To reduce power consumption, the hardwareis put into sleep state when idle, awaiting an interrupt.
- **Distributed Memory:** In biology, neurons use only local information for processing coming stimuli. The SpiNNaker architecture features a hierarchy of memories.
- **Re-Configurability:** In biology, synapses are plastic. This means that neural connectivity change both in shape and strength. The SpiNNaker architectureallows on-the-fly reconfiguration.

5.2.1 Architecture:

This architecture is a real-time simulator running on an event driven abstract-time paradigm using multiple neural models during the same simulation, and incorporating learning capabilities. The core of this simulator is the SpiNNaker chip (Furber, 2011): a full-custom ASIC chip with 18 ARM 968 cores, running at 200MHz with low power consumption specifications and extended instruction set for digital signal processing. No floating point unit has been embedded in the architecture of this chip to comply with the low power specifications and the space available on the die. Consequently, all computation implemented in the software must be based only on fixed-point operations. Additionally, the division operation is not part of the instruction set of the ARM architecture, and therefore must be implemented in software, or avoided if possible. Figure 3.1 describes the chip by functional blocks, and Figure 3.2 labels each major component on the die.

A full-custom router (Plana et al., 2007) stands between the cores and the input/output links where it can receive network packets from any source and route network packets to their correct destination(s). A Network on Chip (system NoC) interfaces the cores with the peripherals: System RAM, System ROM, MII (Ethernet) interface, Watchdog, System controller, PLLs and PL340. There are three types of memory available to each core:

- Tightly Coupled Memory (TCM) which is part of each ARM core and is dividedin two parts: instruction TCM (ITCM) which is 32 KB and data TCM (DTCM) which is 64 KB. This memory is integrated into each core, and therefore each core accesses its own TCM.
- System RAM, which is integrated into the SpiNNaker chip and shared between all the processors. Its size is 32KB.
- SDRAM, which is a mobile laptop memory chip external to the SpiNNaker chip and accessible through the PL340 interface, shared between all the cores. The size of this memory is 128 MByte.

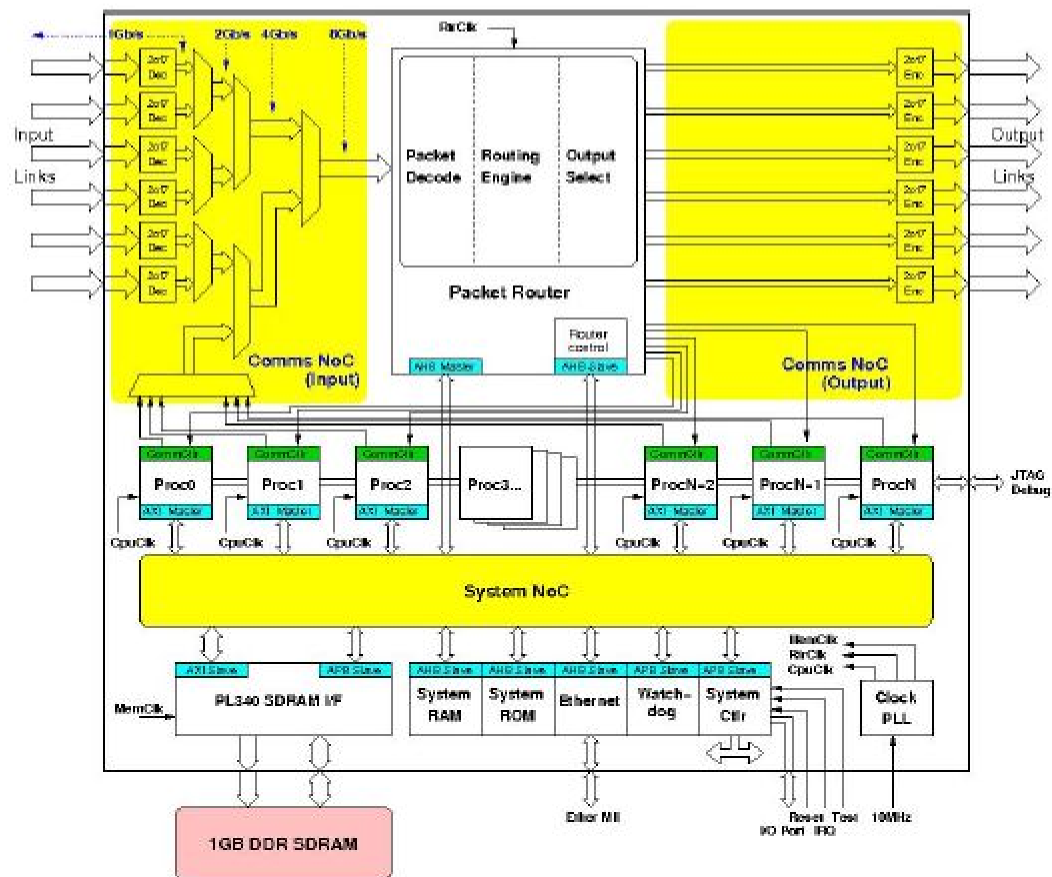


FIGURE 5.1: BLOCK DIAGRAM OF THE FULL SPINNAKER CHIP []

Chip Inter-Connection The SpiNNaker chip has six external links, as described in Figure 5.1, to connect to six other SpiNNaker chips in a 2-dimensional network of up to 256 x 256 chips. The extremities of such a grid can be wrapped to form a toroidal network. Alternatively, it is possible, also, to visualize the 2-dimensional array of chips as a hexagonal-shaped network and Two-dimensional grid of SpiNNaker chips with the needed connections (in green) to form the toroidal shape as shown in Figure 5.2.

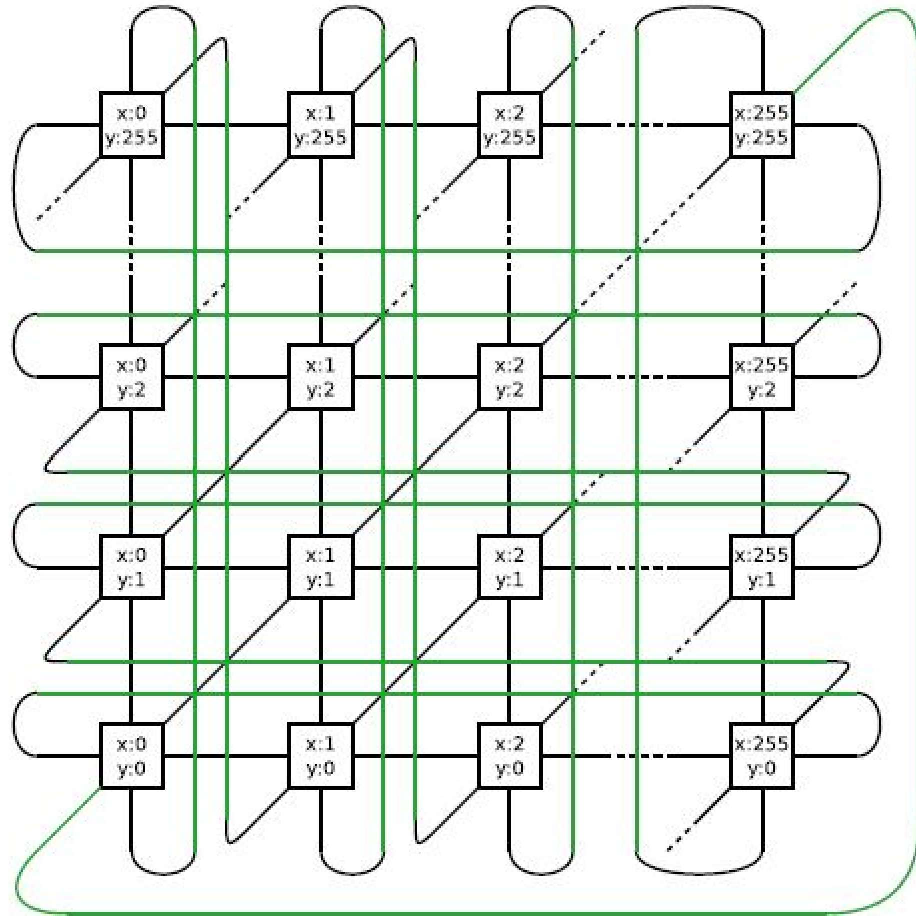


FIGURE 5.2: TWO-DIMENSIONAL GRID OF SPINNAKER CHIPS WITH THE NEEDED CONNECTIONS (IN GREEN) TO FORM THE TOROIDAL SHAPE. []

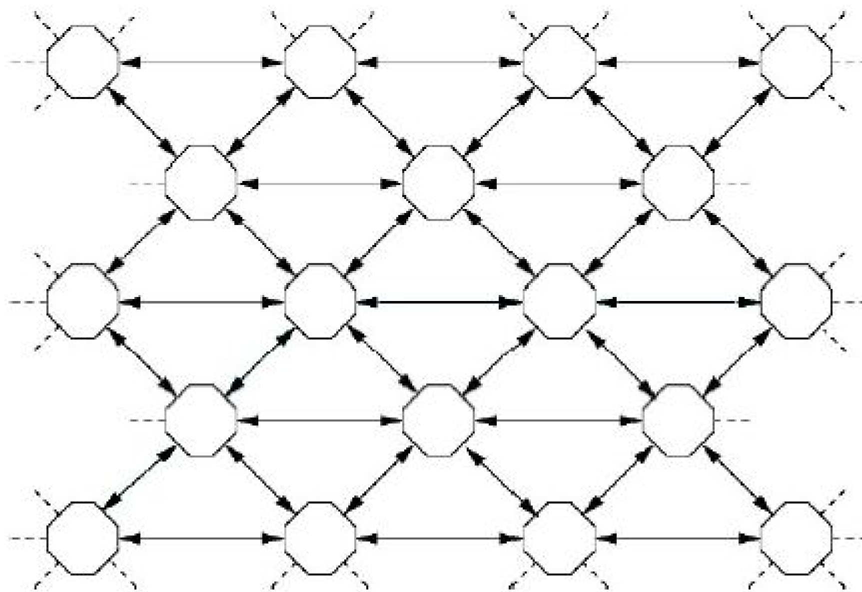


FIGURE 5.3: HEXAGONAL SHAPED SPINNAKER CHIP NETWORK. []

5.3 Pacman

Pacman is the acronym for Partition and Configuration Manager (Galluppi et al.,2012a). This software converts a neural network description (using the standard PyNN language (Davison et al., 2008)) into binary files which need to be loaded into the SpiNNaker system to run the simulation. The block diagram of this software is depicted in Figure 5.4.

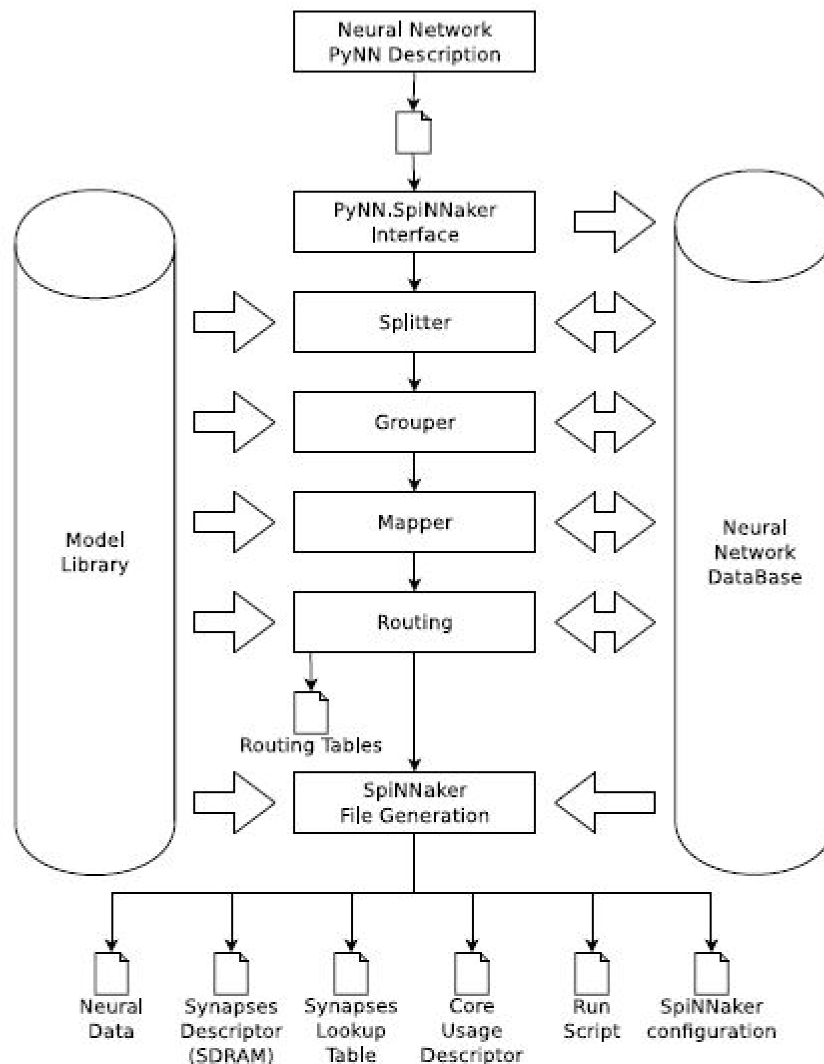


FIGURE 5.4: BLOCK DIAGRAM OF THE PARTITION AND CONFIGURATION SOFTWARE (PACMAN). []

5.3.1 Splitter

This component splits populations and projections into parts which can be mapped onto single processors.

5.3.2 Grouper

This component groups populations which contain fewer neurons than those which could fit into a single core. In this way it is possible to use the SpiNNaker system for neural simulation efficiently.

5.3.3 Mapper

This component maps the outcome of the grouper on the SpiNNaker cores available, considering the topology of the network of chips and the functioning processors available.

5.3.4 Routing

This component assigns the addressing space to the populations of neurons and computes the routing tables for the SpiNNaker system to map the projections between the populations.

5.3.5 SpiNNaker File Generator

This component generates files to describe each single neuron, each single synapse and all the auxiliary files to run the simulation on the SpiNNaker system. The outcome of this process is the binary files to be loaded on SpiNNaker, or used to run the simulation.

5.4 Implementing Hodgkin Huxley on SpiNNaker

Neuronal activity is a result of ionic movement, caused by two electrochemical gradientsconcentration and electric potential gradientsaround the cell bodys membrane. The two electrochemical gradient forces drive ions in opposite directions, toward either the inside or the outside of the cell. Several different types of mathematical models have been developed to describe neuronal dynamics. Alan Lloyd Hodgkin and Andrew Huxley developed a well-known biologically plausible conductance-based model (Hodgkin-Huxley) that describes ion currents dynamics by a set of nonlinear differential equations. In conductance based models, the variables and parameters have well-defined biological meanings, and can be measured experimentally. However, analyzing the conductance- based models is complicated. It quantitatively describes the subcellular level ionic behaviors and the membrane current underlying the generation and propagation of neural spike. The Hodgkin-Huxley model is one of the most biological plausible models in computational neuroscience and their model is a complicated nonlinear ODE system consisting of

four equations describing the membrane potential, activation and inactivation of different ionic gating variables respectively. The HH equations are given as follows.

$$\frac{du}{dt} = -g_{Na}m^3h(u - E_{Na}) - g_Kn^4(u - E_K) - g_L(u - E_L) + I(t) \tag{5.1}$$

$$E_{Na} = 115mV, E_K = -12mV, E_L = 10.6mV, g_{Na} = \frac{120ms}{cm^2}, g_K = \frac{36ms}{cm^2}, g_L = \frac{0.3ms}{cm^2}$$

$$\frac{dm}{dt} = \alpha_m(u)(1 - m) - \beta_m(u)m$$

while

$$\alpha_m(u) = \frac{2.5 - 0.1u}{e^{-0.1u+2.5} - 1} \text{ and } \beta_m(u) = 4 \times e^{\frac{-u}{18}}$$

$$\frac{dn}{dt} = \alpha_n(u)(1 - n) - \beta_n(u)n$$

while

$$\alpha_n(u) = \frac{0.1 - 0.01u}{e^{-0.1u+1} - 1} \text{ and } \beta_n(u) = 0.125 \times e^{\frac{-u}{80}}$$

$$\frac{dh}{dt} = \alpha_h(u)(1 - h) - \beta_h(u)h$$

while

$$\alpha_h(u) = \frac{0.07u}{e^{\frac{-u}{20}}} \text{ and } \beta_h(u) = \frac{1}{e^{3-0.1u} + 1}$$

Accurate brain modeling requires not only the neuronal dynamics but also a comprehensive map of structural connection patterns in the human brain. Connectivity is closely related to the neural coding problem: information is coded and propagated within the neural network through structural links such as synapses and fiber pathways. Based on connectivity patterns discovered in laboratory experiments, researchers have built several mathematical models of connectivity. The resulting connectivity knowledge was used to create large-scale neural network models, including Hodgkin Huxley model to simulate brain activity.

We selected the Hodgkin Huxley to demonstrate real-time simulation with 1 ms resolution on SpiNNaker. We derived a 16-bit fixed-point arithmetic implementation to save both computation and storage space, as well as to avoid having a floating point unit in the ARM968 cores. We take advantage of knowing the membrane potentials exact range of values (-80 = v = 380). By using a dual scaling factor scheme, we can reduce the precision lost during the conversion and hence maintain a good precision level. We also optimize the implementations performance and accuracy by:

- Expanding the width from 16 to 32 bits during the computation to achieve better precision.
- Transforming the equations to allow the use of efficient ARM instructions.
- Adjusting the parameters and pre computing Equation 1 as much as possible; and programming in ARM assembly language.

In our implementation, one iteration of Hodgkin-Huxley equations takes six fixed-point arithmetic and two shift operations. This is more efficient than the original implementation, which requires 13 floating-point operations. In a practical implementation, the complete subroutine for computing Hodgkin-Huxley equations can be performed in as few as 20 instructions if the neuron doesnt fire. If the neuron does fire, it takes 10 more instructions to reset the value and send a spike event. An event address mapping (EAM) scheme is used to achieve minimum communication. The EAM scheme keeps synaptic weights at the post-synaptic end (neurons receiving the spike) and set a relationship (in a mapping table) between the spike event and the address of the synaptic weight. Hence, no synaptic weight information needs to be carried in a spike packet. When a neuron fires, the packet propagates through a series of multicast routers according to the preloaded routing table in each router, and finally arrives at the destination processing cores. The EAM scheme employs the two memory systems, DTCM and SDRAM, to store and access efficiently synaptic connections. DMA operations transfer each weight block from the SDRAM to the local DTCM, before computing the update following the Hodgkin-Huxley equations. A core can easily find the synaptic weights associated with the fired neuron. It does this by matching the incoming spike packet with entries in the mapping table, which is organized as a binary tree. The neural simulation involves a 500-neuron network with an excitatory inhibitory ratio at 4:1. Each neuron is randomly connected to 25 other neurons. We randomly selected 12 excitatory and three inhibitory neurons as biased neurons, each receiving a constant input stimulus of 20 mV.The spike raster plots compare a fixed-point arithmetic simulation in Matlabwith the same neural model executing on a real SpiNNaker chip.

5.5 32-BIT FIXED POINT IMPLEMENTATION OF THE HODGKIN-HUXLEY MODEL

The Hodgkin-Huxley model of spiking neuron is selected as the neuronal model used in the SpiNNaker system. Floating-point numbers are used in the original Hodgkin-Huxley model, however, fixed point operations are more efficient than their floating-point alternatives and the ARM processor does not have a Floating Point Unit (FPU). As a result, we propose to use fixed-point operations instead of floating-point operations in the system. 16-bit fixed-point arithmetic is used to further speed-up the processing and save storage space. Pearson et al. have implemented spiking neural networks based on fixed point leaky-integrate-and-fire (LIF) model on FPGA [16, 17]. We are modelling spiking neural networks of a different neuron model on different hardware with a new approach of using two scaling factors.

5.5.1 Implementation Constraints

Implementation of the Hodgkin-Huxley model must consider SpiNNakers hardware architecture. In particular, the following details of the hardware act as model design constraints:

5.5.1.1 Elementary Mathematical Operations Only

The ARM968 has basic add and subtract, logical, shift, and multiply operations, but does not have native support for division, transcendental functions, and other nonlinear computations. Therefore, the Hodgkin-Huxley model must express its processing in terms of simple polynomial mathematical operations.

5.5.1.2 32-bit Fixed-Point Representation

Similarly, the ARM has no floating-point unit. The Hodgkin-Huxley model needs to translate any floating-point quantities into fixed-point numbers, while determining a position for the decimal point, hence assigning a fractional precision.

5.5.1.3 Limited Local Memory

SpiNNakers individual processors have 64k data memory and 32k instruction memory each. This effectively prohibits having synaptic data local at all times and limits the number of parameters. Memory management must therefore attempt to store as much information as possible on a per neuron rather than per-synapse basis using HH model.

5.5.1.4 Limited Time to Process a Neuron

To stay within the real-time update requirement, each neuron of HH model must be able to update its state in the time between external events. If a processor is modelling multiple neurons, this means updating at worst in 1 NRmax, where N is the number of neurons modelled and Rmax is the maximum event rate.

5.5.1.5 Synaptic Data Only Available on Input Event

Because of the limited memory, SpiNNaker stores synaptic data off-chip and brings it to the local processor only when an input event arrives. Synapse processing must therefore be scheduled for a fixed time after the input, and can only depend on information knowable at the time the input arrived.

5.5.2 Implementation Rules

To meet SpiNNakers hardware constraints with an efficient, accurate model it is necessary to introduce a set of design rules that help to define the Hodgkin-Huxley model implementation. These rules are indicative but not forcing, while Hodgkin-Huxley model generally obeys this pattern.

5.5.2.1 Defer Event Processing with Annotated Delays

The deferred-event model is a method to allow event reordering. Under this scheme only perform minimal processing at the time of a given event, storing state information of Hodgkin-Huxley model in such a way as to be available to a future event, so that processes can wait upon contingent future events. Future events thus trigger state update relevant to the current event.

5.5.2.2 Solve Differential Equations using the Euler Method

The Euler method is the simplest general way to solve nonlinear differential equations. Thus, using it for Hodgkin-Hulxey model, the processor updates the equations using a small fixed time step t, using the formula $X(t+_t) = X(t) + dx/dt(t+_t)$. The time step is programmable (nominally 1 ms in our model), allowing us to select the time step to optimize the precision/timing margin tradeoff.

5.5.2.3 Represent most Variables using 16-bit Values

Various studies indicate that 16-bit precision is adequate for most neural models. Since the ARM contains efficient 16-bit operations it makes sense to conserve memory space and use 16-bit variables throughout. Intermediate values, however, may use 32 bits to avoid unnecessary precision loss. Thus we have to go for 32 bits for HH model implementation.

5.5.2.4 Pre-Compute Constant Parameters where Possible

By an astute choice of representation, it is often possible to transform a set of parameters in a neural equation into a single parameter that can be pre-computed and simplifies the computation remaining. For example, in the expression x(t) = Aekt, we can use the substitution logab = logcblogca , choose 2 for c and arrive at x(t) = A(2(log2e)kt), which allows us to pre-compute a new constant ? = klog2e and determine x with simple shift operations.

5.5.2.5 Compute Non-Polynomial Functions by Lookup Table

Lookup tables provide a simple, and in fact the only general way of computing an arbitrary function. The ARM takes at most 2N instructions to compute a LUT-based function with N variables. Memory utilization is a concern: a 16-bit lookup table is impractical; however, 8- or 12-bit lookup tables are feasible, occupying 512 or 8192 bytes for 16- bit values respectively. Where Hodgkin-Huxley model need greater precision we implement various interpolations.

5.5.2.6 Exploit free Operations such as Shifting

Most ARM instructions can execute conditionally, many can shift an operand before doing the instruction, and there are built-in multiply-accumulate instructions. Taking advantage of such free operations is an obvious optimization. Using these rules, we build up a generalized function pipeline to represent a neural process that is adequate for HH model as shown in Figure **??**.

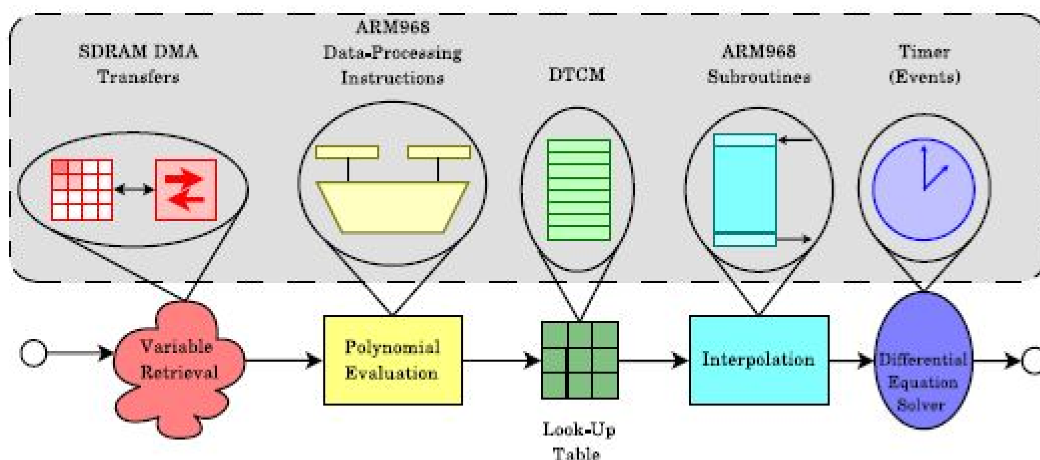


FIGURE 5.5: FIGURE 4.5:A GENERAL EVENT-DRIVEN FUNCTION PIPELINE FOR NEURAL NETWORKS AND VARIABLE RETRIEVAL RECOVERS VALUES STORED FROM DEFERRED-EVENT PROCESSES AS WELL AS LOCAL VALUES. POLYNOMIAL EVALUATION COMPUTES SIMPLE FUNCTIONS EXPRESSIBLE AS MULTIPLY AND ACCUMULATE OPERATIONS. THESE THEN CAN FORM THE INPUT TO LOOKUP TABLE EVALUATION FOR MORE COMPLEX FUNCTIONS. POLYNOMIAL INTERPOLATION IMPROVES ACHIEVED PRECISION WHERE NECESSARY, AND THEN FINALLY THE DIFFERENTIAL EQUATION SOLVER CAN EVALUATE THE EXPRESSION (VIA EULER METHOD INTEGRATION). EACH OF THESE STAGES IS OPTIONAL (OR EVALUATES TO THE IDENTITY FUNCTION) [].

5.6 The Hodgkin-Huxley Model

The Hodgkin-Huxley model uses this instruction pipeline. Many of the techniques it uses are common to the reference. The basic approach applies to virtually any spiking model with

voltage-variable differential-equation dynamics: an illustration of the universal design of the software as well as the hardware.

- Variable Retrieval: Hodgkin-Huxely neuron uses the deferred-event model to place input spikes into a circular array of current bins representing the total input in a given time step Arrival of an input triggers a DMA operation which retrieves a source neuron-indexed synaptic row. Completion of the DMA triggers a second stage of deferral, adding the synaptic weight (representing the current injection) to the bin corresponding to the synapses delay. The Timer event that occurs when that delay expires recovers the total current from the bin and the neurons associated state block, containing the voltage variable and internal parameters. We pre-compute natural frequency fn = 1/t from time constant t in order to avoid division

5.6.1 Choice of Scaling Factors

To approximate the floating point arithmetic by the fixed-point arithmetic we need to adopt scaling factors. The choice of scaling factors is essential in the floating-point to fixed-point transformation. To choose a proper scaling factor, firstly we investigate the range of variables and the parameters relevant to the transformation. According to the experimental results from the simulation on Hodgkin-Huxley model, the value of membrane potential v during computing is in the range -80 to 380, where 380 is the value before reset (representing the -65mV resting potential at 0mV as a reference voltage. By restoring the membrane potential back to its original value, we get Vrest = -65mV after it reaches 380). A 16-bit half word can represent a signed integer number in the range -32768 to 32768. Hence we get

$$-32786 < vp < 32767(-80 < v < 380) \tag{5.2}$$

Where p is the scaling factor and according to 5.2, we get $p < 86$. In this case, we only consider values for p that are powers of 2 so that they can be implemented simply by shifting. Since a greater value of p always leads to better precision (see experimental results in Table I and Table II below), we choose p = 64. If we select any value for p greater than 64, the membrane potential v may overflow during computation. However, ARM968E-S is a 32-bit processor. Some 32-bit operations are therefore as efficient as some 16-bit operations. This allows us to expand some operations from 16-bit to 32-bit during computation to gain better numerical precision without losing performance, and we can still keep variables in the data structure in 16-bit format. In this way, a greater value of p can be applied to produce better precision without increasing the computation time and the storage space. Although the value of the membrane potential v during

computing is in the range -80 to 380 as we described above, the final value of the membrane potential v hold in the data structure will be in the range -80 to 380. We get

$$-32768 < vp < 32767(-80 < v < 380) \tag{5.3}$$

p = 256 can be selected to satisfy the equation.So far, we have considered only the variable which has the greatest numerical value. Some parameters have very small floating-point values also need to be considered since they may cause a decrease of the precision if the value of the scaling factor is not big enough. There are six parameters $a_m,_m,?a?_n?,?_n,a_h,??_h and$ four constants 0.07,0.125,0.01 and 0.25 that we have to care about a and range roughly from 0.02 to 0.1 and from 0.2 to 0.25 respectively when modeling different types of neuron. The scaling factor p = 256 is probably just enough for these values. However, to get a better performance, some changes to the presentation of equations, which makes the precision worse when the equations are transformed to fixed-point using the scaling factor p = 256. The solution we have adopted here is to use two scaling factors", and p2 with a small and large value respectively. We apply the smaller scaling factor p1 to parameters, variables and constants with values greater than 0.5 and the larger scaling factor p2 to those with values less than 0.5. p2= 65536 is selected because it is both large enough and efficient to implement using multiply-accumulate operations (detailed below). So we get p1 = 256, p2 = 65536

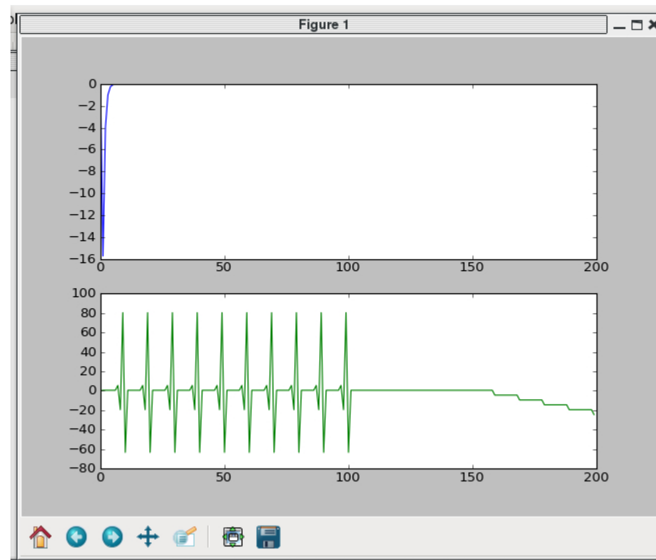


FIGURE 5.6: OUTPUT OF HODGKIN-HUXLEY MODEL SIMULATION ON SPINNAKER

5.6.2 The Transformation of Equations

In order to get an extremely fast processing speed, a few changes are made to the presentation of equations. These changes are made based on two objectives:

FIGURE 5.7: OUTPUT OF HODGKIN-HUXLEY MODEL ON SPINNAKER EMULATOR

- Pre-computing as much as possible
- Reducing the number of operations as much as possible

Continuous-time differential equations can be implemented in discrete time by the following equations,

Cdv/dt=t(-F+I)

$$F(V,m,h,n) = g_N am^3 h(V - V_N a) + g_K n^4 (V - V_K) + g_L (V - V_L) \tag{5.4}$$

Where, ? is time step which can be small to achieve adequate numerical precision. We set ? = 1 for 1 ms resolution. In the ARM architecture, there is a signed multiply-accumulate operation (32 ¡= 32 x 16 + 32) ”SMLAWB”, where ”B” means use the bottom half of the register (bits [15:0]). An operation with the form of (ax*b)/(x + c) can be implemented by one ”SMLAWB” instruction when x = 216 and b is a 16-bit value. It takes only one CPU cycle to obey this instruction in the ARM.We transform equations to the following:

After applying scaling factors p1 andp2, equations turn out to be: To setup a new data structure for each neuron:

```
structNeuronState
```

```
{
signed short Param_V;
signed short Param_EL;
signed short Param_EN;
signed short Param_EK;
signed short Param_GL;
signed short Param_GN;
signed short Param_GK;
signed short Param_M;
signed short Param_N;
signed short Param_H;
signed short Param_InverseCM;  } NeuronStates;
```

In this data structure, scaling factors p1 and p2 have been applied. Hence variables and parameters are fixed-point numbers and pre-computed. Constants in equation are also pre-computed. In ARM assembly code, when p2 = 216 (65536), 1 ms simulation of equation consists of:

$$vP1 = P1(1/C)[P2(-F.P1)/P2+P1(I)]$$

$$F(V,m,h,n) = g_N a m^3 h(V - V_N a) + g_K n^4 (V - V_K) + g_L (V - V_L) \tag{5.5}$$

```
1. A = (Vp1  (VLp2 ) * VG p2.
2. A =  A <<(16  log2p1).
3. A = (Vp1   VN p1 )
4. A = A <<(16  log2p2).
5. A = (Vp1   Vk p1 )
6. A = A <<(16  log2 p2)
7. A = A+ Ip1 one "ADD" operation.
8. vp, = Aup1 one "SUB" operation.
9. A =[(p1/C2)(p2(-F. p1) ] / p2+Ip1,
10. A = A >> log2p2, one shift operation
```

Where, Arepresents the partial result of each step. In step 1, vp1 is stored in the bottom 16 bits of a register. When the instruction is obeyed, what it actually does is multiplying p2 (32 bits) by vp1 (16 bits) in the bottom 16 bits of a register and only the top 32 bits of the multiplication result is pre reserved. If p2 = 216, the division operation of /p2 is done automatically as the bottom 16 bits is dismissed during the multiplication. The result is added to 6p1 finally. However, in step 4 p1 ? 216, so we need a shift operation in step 3 to fit the condition. In step 5, Vp1 and Vkp1is kept in the top 16 bits of two different registers respectively. The computational

result of step 6 is in the most significant 16 bits. As a result, a shift operation is required in step 7.In this approach, 1 ms simulation only takes 6 fixed-point mathematical operations plus 2 shifting operations. Obviously it is more efficient than the original which took 1200floating-point operations.

5.7 Summary

We discovered that Hodgkin-Huxley model was not implemented yet, so we opted Hodgkin-Huxley model for implementation on SpiNNaker architecture. We derived a 32-bit fixed-point arithmetic implementation to save both computation and storage space in the ARM968 processor. We observed that by expanding the fixed point unit causes the loss in precision of model. Our present scaling factor of fixed point athematic is computationally efficient because in original implementation requires 13 floating-point operations and there are eight instructions units in a single floating point operation. In a fixed point unit implementation, the complete subroutine for computing Hodgkin-Huxley equations can be performed in 20 instructions if the neuron does not fire. If the neuron does fire, it takes 10 more instructions to reset the value and send a spike event.

Chapter 6

Conclusion & Future Work

6.1 Conclusion

The research work described in this thesis demonstrated the feasibility of, and provides the implementation details for, modeling neural network model on a scalable chip multiprocessor system and NEST. During the study, a number of problems were solved by developing novel approaches which may also be applicable to other, larger, neural hardware models. In this research we modeled and simulated Hodgkin-Huxley model on hardware and software simulator. We derived a 32 bit fixed point unit implementation scheme for Hodgkin-Huxley model implementation on ARM processor and validate the result by executing the model on NEST simulator. We maximum expanded the fixed point arithmetic of ARM processor in order to get better performance without losing precision. We evaluated the performance parameters like number of instruction units carried out in fixed point unit implementation and floating point unit implementation and compared results of our proposed scaling factor scheme with original model. We observed that data structure in 32 bit format was taking minimum number of instruction sets than floating point operation. Because in original implementation requires 13 floating-point operations and there are eight instructions units in a single floating point operation. In a fixed point unit implementation, the complete subroutine for computing Hodgkin-Huxley equations can be performed in 20 instructions if the neuron does not fire. If the neuron does fire, it takes 10 more instructions to reset the value and send a spike event. NEST simulator was used for validation of our proposed scheme scaling factor. PyNN script simulation erased the translation barrier for executing the model over SpiNNaker architecture and NEST simulator. For ASIC implementation (i.e. on SpiNNaker massively parallel CMP system), the current problem is transformed into PyNN specific format. It is because SpiNNaker middleware (Pacman) takes the input for a non-spiking model in the PyNN format.

6.2 Future Work

The research discovers several potential issues related to the real-time simulation of neural networks, leading to further investigation:

- Spiking neural networks are the type of neural network that SpiNNaker was originally designed for. The Hodgkin-Huxley and AJ neuronal models are used as an example during the study. There must be novel model developed and they can also be implemented on SpiNNaker for its better utilization. The implementation of a neuronal model is dependent on other parts of the system, making it easy to extend the library of models.
- Learning rules can also be investigated and implemented using this new implementation on SpiNNaker architecture.
- Neuron to processor mapping for simulating spiking neural network model is needed.
- Application can be run with the implementation of new models on SpiNNaker so the testing and verification of model and system can be examined.
- New neuronal functions such as neuronal dynamics, short term plasticity and conductance-based synapses need to be implemented on SpiNNaker and neural library must be regularly updated to support new theories.

6.3 Summary

This thesis explores algorithms as well as software implementation for simulation of neural networks on the SpiNNaker chip multiprocessor system and software neural simulator- NEST. The main focus are on implementing of spiking neural network models dealing with minimizing the processing time, and saving memory usage. A typical neural networks model has been investigated: Hodgkin-Huxley model. The modeling schemes are either fully implemented or validated, or analytically studied and evaluated.

In this thesis, initially, a brief introduction to the modeling theory of spiking neural networks is presented in Chapter 2. This chapter is helpful in understanding a neural simulation application, it is important to understand neural network modeling and its computational modeling. Our brain is made of billions of neurons - functionally independent processing units with a tremendous amount of connectivity. The neuron's behavior is dictated by its electro-physiological properties controlled by chemical ions inside and around its cell body. Stimuli to a neuron in the shape of neurotransmitters cause an action potential - a spike or pulse. Neurons communicate with each other using these spikes. All our body movements, responses to our senses and

learning/ memories are controlled with these spikes. Much is known about the neural and learning dynamics in the nervous systems. However, a lot remains to be discovered, especially the emergent behaviors of neural networks. Many mathematical models have been proposed based on empirical hypotheses to capture the neural dynamics in the nervous systems.

There have been many attempts to build engineering systems for simulating large-scale neural networks. They were reviewed in Chapter 3.The purpose of this chapter was to put neural network hardware & software simulators in historical context and reviewed the main research developments in neural network simulator. It considered the architectures: basic platforms for neural networks that have capabilities of simulation. There is no reason to believe that previous hardware designed for biological simulation and it performed well in computational applications. This chapter discussed that one domain where neural networks may be particularly valuable is embedded systems. Such applications usually demand real-time adaptability for simulating the neural network modes. It also inspected that software neural simulator is very important for model exploration and validation. Here the discussion used the SpiNNaker chip as a specific example to introduce as Spiking Neural Network architecture. The study examined that there are range of some popular neuronal models was implemented on this architecture so there is need of implementation of Hodgkin-Huxley model on this architecture.

Most of the solutions have their particular benefits as well as downside. Based on the study of neural network models and existing engineering systems, a new system called SpiNNaker was proposed accordingly to these requirements. SpiNNaker provides a general-purpose and high-performance platform for large-scale neural network simulation. The proposed SpiNNaker architecture raises the research topic of understanding how to map neural models onto such a system a topic investigated in the rest of the chapters. For modeling and simulation on ARM processor, it is necessary to execute the model on single processor. Thus chapter 4 aims to run the Hodgkin-Huxley model on NEST simulator and the result of single processor system verified and validate with the original implementation of model. Chapter 5 discusses building a neural system using the Hodgkin-Hulxey model on a single ARM968 processor. The first problem addressed was to determine how to simulate the Hodgkin-Huxley equations efficiently on the ARM968 using 32-bit fixed- point arithmetic. The aim of this chapter is to implement the Hodgkin-Huxley model using NEST simulator. We used PyNN simulation script for erasing the translation barrier in order to run on hardware architecture. We derived a scheme of fixed point implementation of Hodgkin-Huxley model and evaluated its precision with floating point unit implementation by executing it on NEST simulator. From analytical and experimental results we observed that that our proposed scaling factor scheme computationally efficient, requires less memory space and is more scalable floating point scheme.

A scheme fixed point implementation was used to achieve an accurate resolution without sacrificing performance and by converting the presentation of the equations, and using ARM specific

instructions.The last chapter concludes the thesis with the direction of future prospects. We discovered that Hodgkin-Huxley model was not implemented yet, so we opted Hodgkin-Huxley model for implementation on SpiNNaker architecture. We derived a 32-bit fixed-point arithmetic implementation to save both computation and storage space in the ARM968 processor. We observed that by expanding the fixed point unit causes the loss in precision of model. Our present scaling factor of fixed point athematic is computationally efficient because in original implementation requires 13 floating-point operations and there are eight instructions units in a single floating point operation. In a fixed point unit implementation, the complete subroutine for computing Hodgkin-Huxley equations can be performed in 20 instructions if the neuron does not fire. If the neuron does fire, it takes 10 more instructions to reset the value and send a spike event.

Bibliography

[1] Lawrence Livermore National Laboratory Blaise Barney. Parallel Computing Lectures. https://computing.llnl.gov/tutorials/parallel_comp/. [Online; accessed 23-Dec-2013].

[2] Paul A Merolla, John V Arthur, Bertram E Shi, and Kwabena A Boahen. Expandable networks for neuromorphic chips. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 54(2):301–311, 2007.

[3] Timothée Masquelier, Rudy Guyonneau, and Simon J Thorpe. Spike timing dependent plasticity finds the start of repeating patterns in continuous spike trains. *PloS one*, 3(1): e1377, 2008.

[4] Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeffrey L Krichmar, Alex Nicolau, and Alex Veidenbaum. Efficient simulation of large-scale spiking neural networks using cuda graphics processors. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 2145–2152. IEEE, 2009.

[5] Martin Pearson, Ian Gilhespy, Kevin Gurney, Chris Melhuish, Benjamin Mitchinson, Mokhtar Nibouche, and Anthony Pipe. A real-time, fpga based, biologically plausible neural network processor. In *Artificial Neural Networks: Formal Models and Their Applications–ICANN 2005*, pages 1021–1026. Springer, 2005.

[6] Benjamin W Walt and Lon-Chan Chit. Efficient mapping of neural networks on multicomputers. *Urbana*, 51:61801.

[7] Maryam Alavi and John C Henderson. An evolutionary strategy for implementing a decision support system. *Management Science*, 27(11):1309–1323, 1981.

[8] Rafic A Ayoubi and Magdy A Bayoumi. Efficient mapping algorithm of multilayer neural network on torus architecture. *Parallel and Distributed Systems, IEEE Transactions on*, 14 (9):932–943, 2003.

[9] Sophie Achard and Ed Bullmore. Efficiency and cost of economical brain functional networks. *PLoS computational biology*, 3(2):e17, 2007.

[10] Jay B Angevine and Carl W Cotman. *Principles of neuroanatomy*. Oxford University Press, 1981.

[11] Tom Binzegger, Rodney J Douglas, and Kevan AC Martin. A quantitative map of the circuit of cat primary visual cortex. *The Journal of Neuroscience*, 24(39):8441–8453, 2004.

[12] Guy E Blelloch and Charles R Rosenberg. Network learning on the connection machine. In *IJCAI*, pages 323–326. Citeseer, 1987.

[13] Kwabena A Boahen. Point-to-point connectivity between neuromorphic chips using address events. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 47(5):416–434, 2000.

[14] Guo-qiang Bi and Mu-ming Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *The Journal of neuroscience*, 18(24):10464–10472, 1998.

[15] WJ Bainbridge, Luis A Plana, and Stephen B Furber. The design and test of a smartcard chip using a chain self-timed network-on-chip. In *Proceedings of the conference on Design, automation and test in Europe-Volume 3*, page 30274. IEEE Computer Society, 2004.

[16] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.

[17] Gully APC Burns and Malcolm P Young. Analysis of the connectional organization of neural systems associated with the hippocampus in rats. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 355(1393):55–70, 2000.

[18] Santiago Ry Cajal. *Texture of the Nervous System of Man and the Vertebrates: I*, volume 1. Springer, 1999.

[19] Muhammad Mukaram Khan, David R Lester, Luis A Plana, A Rast, Xin Jin, Eustace Painkras, and Stephen B Furber. Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 2849–2856. IEEE, 2008.

[20] Sharon M. Crook, G Bard Ermentrout, Michael C. Vanier, and James M. Bower. The role of axonal delay in the synchronization of networks of coupled cortical oscillators. *Journal of computational neuroscience*, 4(2):161–172, 1997.

[21] Barry W Connors and Michael J Gutnick. Intrinsic firing patterns of diverse neocortical neurons. *Trends in neurosciences*, 13(3):99–104, 1990.

[22] Catherine E Carr and Masakazu Konishi. Axonal delay lines for time measurement in the owl's brainstem. *Proceedings of the National Academy of Sciences*, 85(21):8311–8315, 1988.

[23] Iain S Duff, Roger G Grimes, and John G Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software (TOMS)*, 15(1):1–14, 1989.

[24] John C Eccles. *The understanding of the brain.* McGraw-Hill, 1973.

[25] Daniel J Felleman and David C Van Essen. Distributed hierarchical processing in the primate cerebral cortex. *Cerebral cortex*, 1(1):1–47, 1991.

[26] Shou King Foo, Paramasivan Saratchandran, and Narasimhan Sundararajan. Parallel implementation of backpropagation neural networks on a heterogeneous array of transputers. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 27(1):118–126, 1997.

[27] Steve Furber and Steve Temple. Neural systems engineering. *Journal of the Royal Society interface*, 4(13):193–206, 2007.

[28] Steve Furber, Steve Temple, and Andrew Brown. On-chip and inter-chip networks for modeling large-scale neural systems. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4–pp. IEEE, 2006.

[29] Dan Goodman and Romain Brette. Brian: a simulator for spiking neural networks in python. *Frontiers in neuroinformatics*, 2, 2008.

[30] Wulfram Gerstner, Raphael Ritz, and J Leo van Hemmen. A biologically motivated and analytically soluble model of collective oscillations in the cortex. *Biological cybernetics*, 68(4):363–374, 1993.

[31] Dan FM Goodman and Romain Brette. The brian simulator. *Frontiers in neuroscience*, 3 (2):192, 2009.

[32] Mark A Glover, Alister Hamilton, and Leslie S Smith. An analog vlsi integrate-and-fire neural network for sound segmentation. In *NC*, pages 86–92, 1998.

[33] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

[34] Wulfram Gerstner, Richard Kempter, J Leo van Hemmen, and Hermann Wagner. A neuronal learning rule for sub-millisecond temporal coding. *Nature*, 383(LCN-ARTICLE-1996-002):76–78, 1996.

[35] Roger W Sperry. Cerebral organization and behavior. *Science*, 133(3466):1749–1757, 1961.

[36] Xin Jin, Mikel Luján, Muhammad Mukaram Khan, Luis A Plana, Alexander D Rast, Stephen R Welbourne, and Stephen B Furber. Algorithm for mapping multilayer bp networks onto the spinnaker neuromorphic hardware. In *Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on*, pages 9–16. IEEE, 2010.

[37] Xin Jin, Mikel Luján, Luis A Plana, Alexander D Rast, Stephen R Welbourne, and Steve B Furber. Efficient parallel implementation of multilayer backpropagation networks on spinnaker. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 89–90. ACM, 2010.

[38] Xin Jin, Alexander Rast, Francesco Galluppi, Mukaram Khan, and Steve Furber. Implementing learning on the spinnaker universal neural chip multiprocessor. In *Neural information processing*, pages 425–432. Springer, 2009.

[39] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th conference on Computing frontiers*, pages 87–96. ACM, 2008.

[40] Sun-Yuan Kung and JN Hwang. A unified systolic architecture for artificial neural networks. *Journal of Parallel and Distributed Computing*, 6(2):358–387, 1989.

[41] Muhammad Mukaram Khan, David R Lester, Luis A Plana, A Rast, Xin Jin, Eustace Painkras, and Stephen B Furber. Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 2849–2856. IEEE, 2008.

[42] MM Khan, E Painkras, X Jin, LA Plana, JV Woods, and SB Furber. System level modelling for spinnaker cmp system. In *Proc. 1st International Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO09)*, 2009.

[43] AK Kreiter and W Singer. Oscillatory neuronal responses in the visual cortex of the awake macaque monkey. *European Journal of Neuroscience*, 4(4):369–375, 1992.

[44] Vipin Kumar, Shashi Shekhar, and Minesh B. Amin. A scalable parallel formulation of the backpropagation algorithm for hypercubes and related architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 5(10):1073–1090, 1994.

[45] Jens Langner. *Development of a parallel computing optimized head movement correction method in positron-emission-tomography*. PhD thesis, University of Applied Sciences, 2003.

[46] Rémy Lestienne. Determination of the precision of spike timing in the visual cortex of anaesthetised cats. *Biological cybernetics*, 74(1):55–61, 1996.

[47] Joseph Lin, Paul Merolla, John Arthur, and Kwabena Boahen. Programmable connections in neuromorphic grids. In *Circuits and Systems, 2006. MWSCAS'06. 49th IEEE International Midwest Symposium on*, volume 1, pages 80–84. IEEE, 2006.

[48] Misha Mahowald. *VLSI analogs of neuronal visual processing: a synthesis of form and function*. PhD thesis, California Institute of Technology, 1992.