University of Nevada, Reno

NCS: Neuron Models, User Interface, and Modeling

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computer Science and Engineering

by

Devyani Tanna

Dr. Frederick C. Harris, Jr., Thesis Advisor

August, 2014

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

DEVYANI TANNA

Entitled

NCS: Neuron Models, User Interface, And Modeling

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Dr. Frederick C. Harris, Jr., Advisor

Dr. Sergiu M. Dascalu, Committee Member

Dr. Yantao Shen, Graduate School Representative

David W. Zeh, Ph.D., Dean, Graduate School

August, 2014

Abstract

Neuroscientists conduct experiments at different scales from molecules to system level to gain insights into various brain functions. When *in vivo* and *in vitro* experiments are hard to perform, neural simulators are very helpful. The NeoCortical Simulator (NCS) is a neural simulator designed to run on a heterogeneous cluster of CPUs and NVIDIA GPUs. With the use of neural simulators, there is always a trade-off between biological accuracy and time. Previously, NCS had one built-in neuron model for researchers to use. In order to provide choices for accuracy and execution time, two more built-in neuron models have been added to NCS. The back-end of NCS is written using C++11 and CUDA. Prior to NCS7, input files were used as an input. They did not allow loops and files tended to be large. To overcome the issues with input file and to allow scientists with varying level of programming skills to utilize NCS, a Graphical User Interface and a Python interface have been added to NCS. Furthermore, the new Report interface will allow users to view output by the simulator in real time and the MongoDB database will allow researchers to share their models and collaborate with others in the community. This thesis presents the design and implementation of NCS6 along with the newer version of NCS, NCS7, and its Python interface, Graphical User Interface, Report interface, database, and modeling information.

Dedication

For Neha, Mom, and Dad

Acknowledgments

I would like to thank my advisor, Dr. Frederick Harris, and my committee members Dr. Sergiu Dascalu and Dr. Yantao Shen for their time and suggestions. Also, special thanks to Dr. Frederick Harris and Dr. Laurence Jayet Bray for introducing me to the the field of Computational Neuroscience, encouraging me to do research, and providing guidance and support during my undergraduate and graduate studies at University of Nevada, Reno.

I would like to thank Roger Hoang, Torbjorn Loken, Nathan Jordan, and everyone at Brainlab and HPCVIS for all the help and moral support. Also, huge thanks to my family and friends for their love, encouragement, and never-ending support.

Contents

List of Figures

Chapter 1

Introduction

The brain is the most complex organ to study. The field of Computational Neuroscience allows scientists to study complex brain structures using neural simulators when *in vivo* and *in vitro* experiments are hard to perform. Additionally, *in vivo* and *in vitro* experiments are only ideal for small scale experiments. Brain Computation Laboratory at University of Nevada, Reno has it's own simulator called NeoCortical Simulator (NCS). NCS has three in-built neuron models and it is designed to run on the heterogeneous cluster of CPUs and NVIDIA GPUs for large-scale simulation. NCS is able to simulate 1 million neurons and 100 million synapses model in quasi real time [27].

There are three well known neuron models used in the field: Izhikevich(IZH) [32], Leaky-integrate-and-fire(LIF), and Hodgkin-Huxley(HH) [52]. Those models provide varying amount of biological accuracy with IZH being the most simplistic, then LIF, and HH being the most biologically accurate neuron model. Every version of NCS has provided Integrate-and-Fire neuron model with conductance based synapses as described in Section 2.1 referred to as LIF model in rest of the paper. In addition to that, Izhikevich neuron model has been added in NCS6, and Hodgkin-Huxley neuron model has been added in the current version of NCS, NCS7. Furthermore, users have an option to create their own plugin interface for different neuron models.

The backend of NCS is written using C++11 and CUDA. Prior to NCS7, the simulator accepted text files as an input. It did not allow loops, tended to be large, lots of the code was same for each part with minor change in parameter values, and one change required changes in multiple places in the code. Due to this, errors were easily introduced in the code and the code was hard to maintain. To overcome the issues with input file and to allow scientists with varying level of programming skills to utilize NCS, a Graphical User Interface and a Python interface have been added to NCS7. There is also a Reporting interface for users to view output by the simulator and database to save models so researchers can easily collaborate and share models with community.

This thesis presents the design and implementation of NCS6 along with it's Python interface, Graphical User Interface, Report interface, database, and modeling information. The rest of the thesis is as follows: Chapter 2 which appeared as a journal paper [27], explains the design and implementation of NCS6 and gives information for built-in LIF and IZH neuron models. For this work I wrote models for LIF and IZH neurons and helped with writing and editing of the paper. Chapter 3 which has been submitted as a conference paper [3], covers the design of Web based front end of NCS as well as a walk through of model creation using Graphical interface. For this paper I interviewed neuroscience and biomedical engineering students about their expectations for the user interface for the brain simulator, helped with the design of the user interface, and directed the writing and editing of the paper. Chapter 4 which has been submitted as a conference paper [2], shows the design of MongoDB database for NCS and Graphical interface for reports. For this paper I designed the MongoDB database and created the first prototype, helped with design of JSON schemas, wrote sections of the paper, and oversaw the writing and editing of the paper. Chapter 5 presents equations for Hodgkin-Huxley neuron model and the details of model creation using Python interface as well as information about the online documentation for NCS. I co-designed the initial Python interface and led the review and modifications for the next update. Additionally, I wrote the current version of the documentation for NCS which appears at *ncs.io/docs/*. Chapter 6 provides conclusions and plans for future work for NCS.

Besides the contributions listed in the previous paragraph, I have also assisted the research in Computational Neuroscience through a variety of other items. These include:

- Advising and assisting several senior project groups. Three of these resulted in publications [2, 3, 13].
- Performing NCS demonstrations at NIPS 2012[33] and NIPS 2013[28].
- Collecting data for and running emotional speech processing experiments as well as editing and rewriting the text [10].
- I also had the opportunity to design and teach the majority of the lectures for CS 491C Topics: Computational Neuroscience.

Chapter 2

A Novel CPU/GPU Simulation Environment for Large-Scale Biologically Realistic Neural Modeling

Roger V. Hoang, Devyani Tanna, Laurence C. Jayet Bray, Sergiu M. Dascalu, and Frederick C. Harris, Jr. A Novel CPU/GPU Simulation Environment for Large-Scale Neural Modeling. Frontiers in Neuroinformatics, 7, 2013.

Abstract

Computational Neuroscience is an emerging field that provides unique opportunities to study complex brain structures through realistic neural simulations. However, as biological details are added to models, the execution time for the simulation becomes longer. Graphics Processing Units (GPUs) are now being utilized to accelerate simulations due to their ability to perform computations in parallel. As such, they have shown significant improvement in execution time compared to Central Processing Units (CPUs). Most neural simulators utilize either multiple CPUs or a single GPU for better performance, but still show limitations in execution time when biological details are not sacrificed. Therefore, we present a novel CPU/GPU simulation environment for large-scale biological networks, the NeoCortical Simulator version 6 (NCS6). NCS6 is a free, open-source, parallelizable, and scalable simulator, designed

to run on clusters of multiple machines, potentially with high performance computing devices in each of them. It has built-in leaky-integrate-and-fire (LIF) and Izhikevich (IZH) neuron models, but users also have the capability to design their own plug-in interface for different neuron types as desired. NCS6 is currently able to simulate one million cells and 100 million synapses in quasi real time by distributing data across eight machines with each having two video cards.

2.1 Introduction

Many different scales of experiments in neuroscience research attempt to clarify the complex functions of the nervous system. From the genetics of single molecules to the behavioral research of cognitive neuroscience, studies lead to a better understanding of neural networks, such as the brain. When *in vivo* and *in vitro* experiments are impossible to perform due to the complexity of structures, computational neuroscience provides new opportunities. Its unique access to any brain region as well as its different levels of abstraction allow biologically-realistic neural simulations, and thus additional neuroscience findings. However, neural simulations have always involved a trade-off between execution time and biophysical realism. Even as neuron models are simplified and approximated, the neural regions of interest may require an unreasonable amount of running time. To further drive computational neuroscience research, computer scientists and engineers have created more optimized simulation programs and more advanced hardware architecture, respectively.

Biologically, most simulation environments already have built-in spiking neuron models. These models, described as hybrid systems, satisfy a set of differential equations that describe the continuous evolution of several state variables and discrete events [12]. The well-known ones are Hodgkin-Huxley (HH), Izhikevich (IZH), and leaky integrate-and-fire (LIF) neuron models. The HH model quantifies the process of spike generation with a set of four differential equations [52], formalizing their measured findings of the giant axon of a squid. This model uses the voltage dependence and the dynamics of Sodium and Potassium channels, which captures many biological details while losing computational efficiency. The IZH model is a powerful engine, capable of replicating much of the dynamics phenomena observed in neurons. It uses a mathematical formulation derived from the treatment of a neuron as a dynamical system, resulting in a membrane voltage expression [32]. This is an intermediate model, which is computationally efficient while still capturing a large variety of response properties of real neurons. The LIF model is comprised of a subthreshold leaky-integrate dynamic, a firing threshold, and a reset mechanism, which gives an approximation of the subthreshold dynamics of the membrane potential with a simple linear differential equation [52]. It is beneficial for analytic calculations and is efficient in numerical implementations. However, the approximation is not sufficient to include most of the response patterns seen in real neurons.

Computationally, most simulators (e.g. NEURON [14], NEST [16], GENESIS [8, 9], BRIAN [11]) were designed to run one or more of these models on a single Central processing Unit (CPU). Over the years, they have evolved to support simulations on multiple CPUs for extensibility and higher performance. These enhancements, in combination with parallel computing [7, 35], have become a necessity to cope with the higher computational and the communication demands of neuroapplications. Recently, a number of developers have investigated the possibility of simulating spiking neural networks on a single Graphical Processing Unit (GPU) [1, 4, 5, 17, 18, 19, 24, 25, 30, 31, 37, 38, 43, 45, 48, 50, 51, 54, 55] or on multiple GPUs [12]. All these current simulators have shown significant improvements over their CPU only counterparts by integrating the utilization of GPUs. However, these approaches have had limitations. Two of the main limitations are that researchers either utilize an Izhikevich neuron model while running the simulation on GPU, or if they utilize more than one neuron model (e.g. HH and IZH) their model focuses on small-scale networks. Additionally, they are not capable of running simulations on heterogeneous cluster of GPUs.

To reduce execution times without sacrificing biological details, we have developed a new version of our brain simulator. Here, we present a new CPU/GPU Simulation Environment for Large-Scale Neural Modeling, called the NeoCortical Simulator (NCS) version 6. Previous versions of NCS were designed to run on a CPU or cluster of CPUs. Every version of NCS has implemented a hybrid spiking neuron model. Sub-threshold dynamics are determined by channels that follow the HH formalism. When the voltage crosses a specified threshold value, the membrane potential follows a user-specified spike shape pattern, similar to an LIF neuron. This hybrid model is referred to as an LIF model in the rest of this paper. For a detailed history of NCS and related equations, please refer to our Brain Computation Laboratory's website:http://www.cse.unr.edu/brain/. In addition to the hybrid LIF spiking neurons , NCS6 implements the simplified IZH equations [32] as a separate neuron type. The Compute Unified Device Architecture (CUDA) by NVIDIA [44] provides an instruction set and tools to developers to work in a GPU environment. NCS utilizes CPUs and CUDA-capable GPUs for simulation. Computationally, shared-memory multiprocessor architectures and recent experiments with clustered GPUs indicate that we will soon be able to simulate a million cells in real time without sacrificing biological detail. In this manuscript, Section 2.2 explains how NCS has been designed, Section 2.3.1 gives a validation of its implementation, and Section 2.3.2 shows a representation of its high performance. Furthermore, we provide a brief comparison between NCS and other simulation environments in Section 2.3.4. In Section 2.4 we conclude with a summary of the paper and our planned future work.

2.2 Design

2.2.1 Simulation Composition

At the detailed level, every simulation is comprised of four primary types of elements: neurons, synapses, stimuli, and reports. Neurons represent the cell body and must report two values at each time-step: the spiking state and the membrane voltage. Synapses represent a unidirectional connection between a presynaptic neuron and a postsynaptic neuron. When the presynaptic neuron fires, the synapse introduces

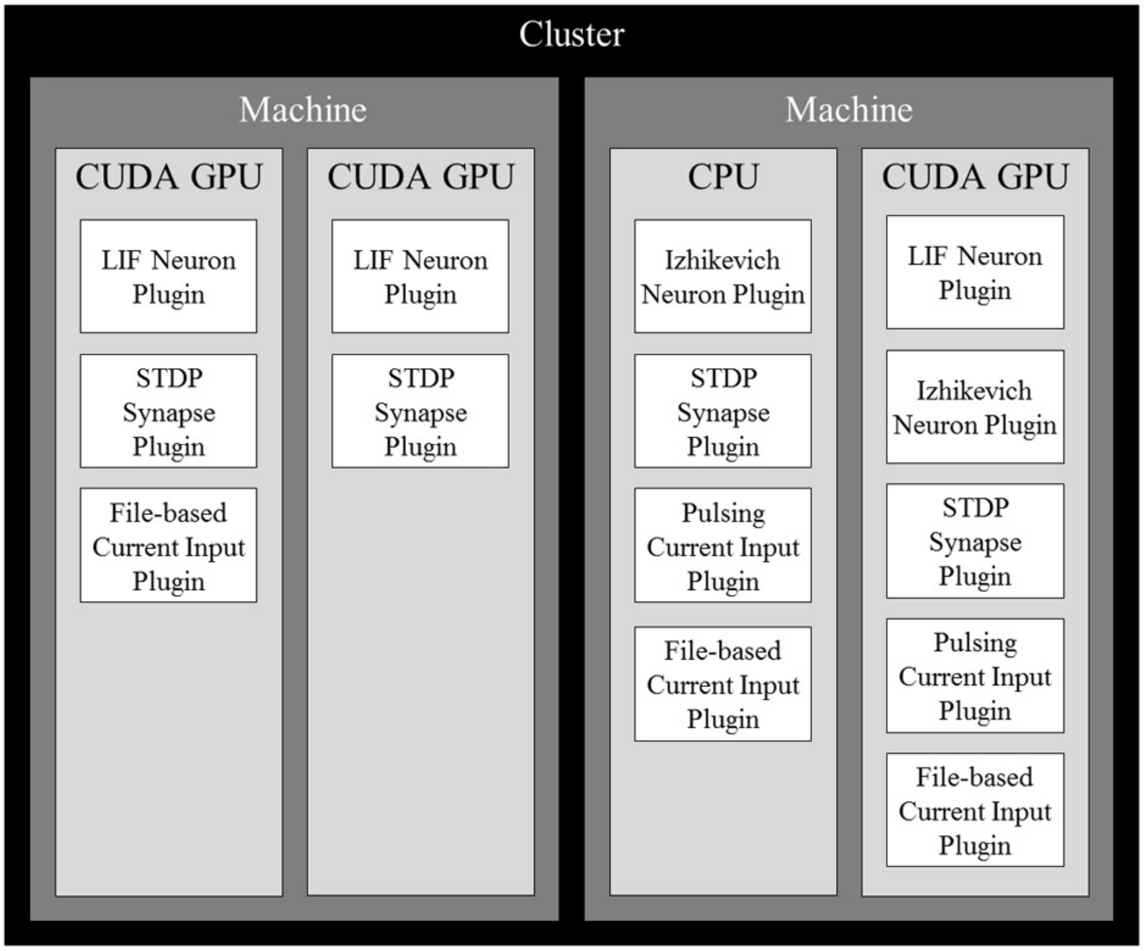


Figure 2.1: An example of a complete distribution of simulation elements in NCS6. Elements are distributed across devices based on the devices' computational power and their dependencies. Synapses and inputs associated with a particular neuron are linked to it on a device. Within devices, elements are organized into contiguous sections by type that are simulated by plugins of their specific type.

a synaptic current into the postsynaptic cell after some specified delay. Stimuli are connected to a neuron and represent a type of external input, able to either clamp the membrane voltage to some level or inject some amount of current. Reports connect to a set of elements (e.g. cell group) and are used to extract output information (e.g. voltage) from those elements and generate the result to some arbitrary data sink.

While each component type has some required constraints, the majority of the internal behavior is determined by the more specific subtype being simulated. For example, one neuron could be specified to simulate a LIF neuron while another neuron could be specified to simulate an IZH cell. The underlying equations governing the behavior are completely different between the two, but they can still be used within the same simulation. The only requirements are that they each receive an external stimulation and/or a synaptic current, and that they each report a firing state and/or a voltage.

2.2.2 Simulation Environment and Distribution

To improve the simulation run times, NCS6 is designed to run on clusters of multiple machines, potentially with different computing devices in each computer. These devices include CUDA-capable GPUs, and CPUs. Even within the same device class, the computational power of different devices can be drastically different. To better facilitate load-balancing, a relative computational power rating is assigned to each device. The current method for determining this quantity is to multiply the device's clock rate by the number of computational cores.

After determining the computational power of each device, a cost estimate for each neuron is computed. Since the number of synapses outnumbers the number of neurons and inputs by several orders of magnitude, we use the number of synapses as the cost estimation. Neurons are then sorted in decreasing order of computational cost and distributed across all available computing devices in the cluster so the device with the lowest computational load (total computational cost / computational power) receives the next neuron. Once all neurons in the simulation are distributed, all contributing synapses and stimuli are also placed on the same devices as their targeted neurons. With all elements distributed across all devices, they are further partitioned by their subtypes, each of which being managed by a plugin. Figure 2.1 shows an example of a complete distribution.

2.2.3 Data Scopes and Structures

Due to the distributed nature of NCS6, elements may be referenced in a number of scopes that mirror the environment's hierarchy: plugin, device, machine, and global (cluster). After the distribution is finished, every element is assigned a zero-based ID for each scope. IDs are padded between plugins so that data words for structures allocated in other scopes are related to only one plugin. In general, this means that IDs are padded to a factor of 32 (the number of bits in a word) between plugins. It is important to note that IDs are only unique within the same element type; that is, there can be both a neuron and a synapse with a global ID of 0.

Depending on which elements need access to other elements, certain key data structures are allocated and accessed using different scopes. Data that is specific to an element subtype is stored at the plugin scope. Because synapses may need to access the membrane voltage from their postsynaptic neurons in order to determine their synaptic current contributions, membrane voltages are stored and accessed using device level IDs. The reason is all postsynaptic neurons and their associated synapses reside on the same device due to the way they are distributed. However, because the spiking state of a synapse depends on the spiking state of the presynaptic neuron, the spiking state of neurons is accessed using a global level ID when updating synaptic spiking states.

2.2.4 Simulation Flow and Parallelization

The basic flow of a simulation is as follows: for each time-step, the current from stimuli and synapses is computed and used to update the state of every neuron. The resulting spiking state of each neuron is then used to determine the spiking state of their associated synapses in later time-steps.

To facilitate maximum utilization of computing devices, the simulation is partitioned into several stages that can be executed in parallel as long as the requisite data for a given stage is ready. Figure 2.2 illustrates this division of work (dark boxes) along with the required data (light boxes) needed to simulate a particular stage and the data that is produced once that stage has been updated. A publisher-subscriber system is used to pass data buffers from one stage to the next. During the simulation, a stage attempts to pull all necessary data buffers from their associated publishing stages. The stage is blocked until all the data is ready. Once it obtains all the required data buffers, it advances the simulation by a single time-step and publishes its own data buffer while releasing all the others that it no longer needs. When all subscribers to a data buffer release it, the data buffer is added back to its publisher's resource pool for reuse. For any given stage, a limited number of publishable buffers are used to prevent a stage from consuming all computational resources and getting unnecessarily ahead of any other stages. For example, without limiting the buffer count, because the input update stage requires no data from any other sources, the stage could simulate all time-steps before a single neuron update is allowed to occur, effectively adding a serial time cost to the overall run time.

Within a single stage, further granularity is gained by parallelizing across subtypes. As an example, if a device simulates both LIF Neurons and Izhikevich Neurons, the plugins updating each can be executed in parallel. Due to padding from the ID assignments, updates should affect completely separate regions of memory, including operations on bit vectors. Exceptions to this, such as when an input writes to a device-indexed input current for its target neuron, are handled by using atomic operations or by aggregating partial buffers generated by each plugin. The method chosen depends on the type of device and its memory characteristics. While plugins are allowed to update ahead of one another, the results for from a stage at a given time-step will not be published to subscribers until all plugins (in that stage) have updated up to that time-step.

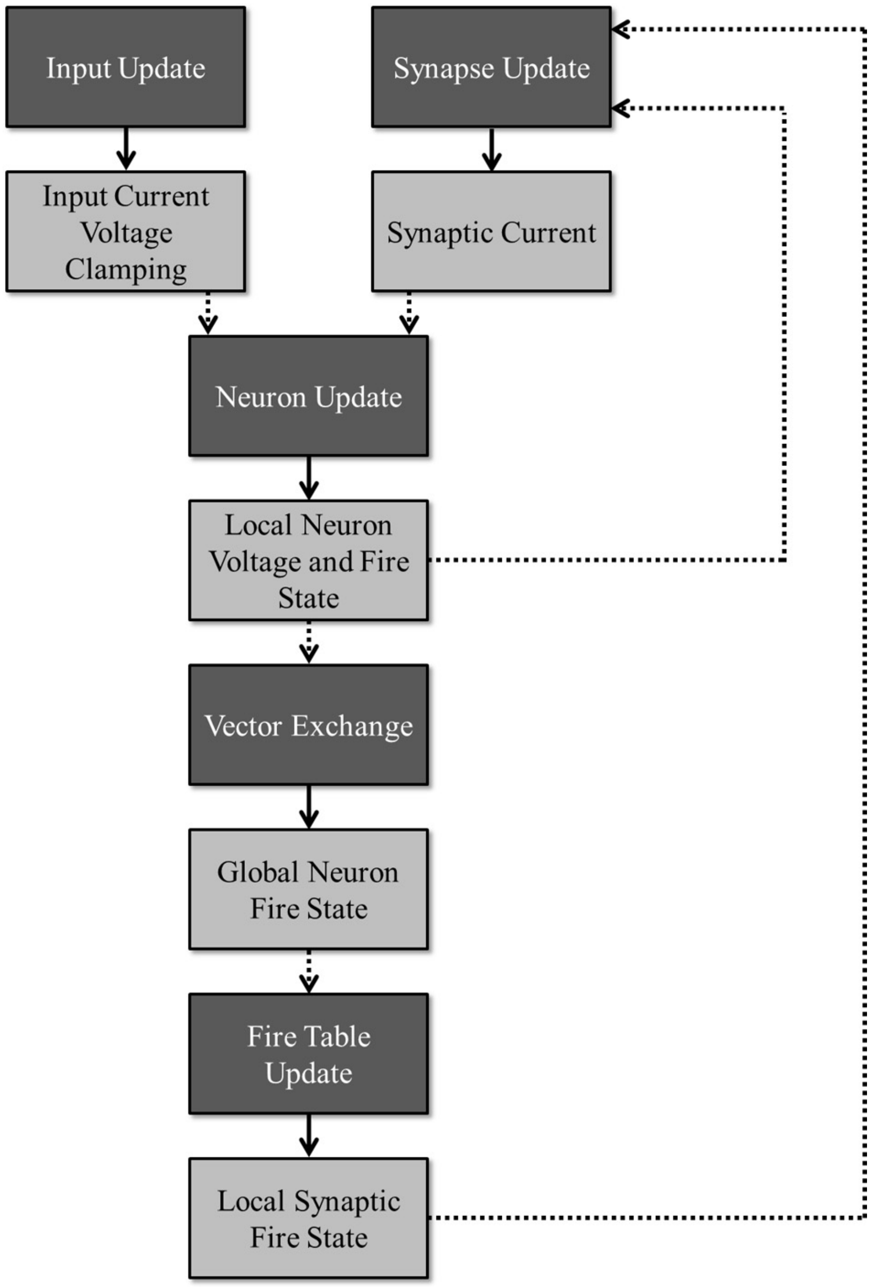


Figure 2.2: Division of work: the dark boxes represent stages that can run concurrently as long as the necessary data has been received for a given time step. Each stage produces an output (denoted by the lighter boxes) that is consumed by the stage denoted by the dotted arrows.

Input Update. The purpose of the input update stage is to compute the total input current to each neuron on the device as well as any voltage clamping that should be done. The input current is represented by an array of floating point values, one for each neuron (including padding) on the device, initialized to zero at the beginning of each time-step. The voltage neurons are clamped and stored in a similar fashion where a bit vector is used to select which neurons should actually be clamped.

Inputs are expected to be updated by input plugins designed to handle their subtype. Other than the device-level Neuron ID for each Input that is statically determined at the beginning of the simulation, input plugins rely on no other data from any other stage of the simulation. As such, they are allowed to simulate ahead of the rest of the system as long as it has empty buffers that can be written to and published.

Neuron Update. Unlike the input update stage, the neuron update stage has two dependencies: the input current per neuron published from the input update stage and the synaptic current per neuron published by the synapse update stage. Given these two pieces of information, this stage is expected to produce the membrane voltage and spiking state of every neuron on the device. Like the input current, the membrane voltage is represented by an array of floating point values. On the other hand, the spiking state is represented by a bit vector.

Similar to inputs, neurons are expected to be updated by neuron plugins designed to handle their subtypes. Despite receiving and writing data out into device-level structures, neuron plugins operate purely in plugin space. This is possible due to the fact that each plugin is given a contiguous set of device-level IDs during the distribution. As a result, device-level data passed into each plugin is simply offset accordingly to yield the appropriate plugin-level representation.

Vector Exchange. The result of the neuron update stage is the firing state of every neuron residing on the device. However, synapses are distributed purely based on the postsynaptic neurons and as such the presynaptic neurons could possibly reside on a different device. Thus, to determine synaptic spiking, the state of every neuron in the simulation must be gathered first. Again, the publisher-subscriber scheme is used to pass data asynchronously. However, rather than passing data between stages, it is used to pass data between different data scopes.

Figure 2.2 shows the flow of the neuron spiking information across a cluster. When the device-level vector exchanger receives a local firing vector, the data is published to the machine-level vector exchanger. Within this exchanger, the local vector is copied into a global vector allocated in the system memory. Once all local device vectors are copied for a single time-step, the complete machine-level vector is broadcast to all the other machines in the cluster. After all machines in the cluster finish broadcasting, the complete global firing vector is published back to the device-level vector exchangers where it is copied back into the appropriate type of device memory before being published out to any subscribing stages.

Firing Table Update. With the firing state of every neuron in the simulation, a device can determine when all of its synapses will receive the firing based on a per-synapse delay value. Given the potential range of delays, this information is stored within a synaptic firing table. A row of the table is a bit vector representing the firing state of every synapse on the device. The number of rows in the table depends on the maximum delay of all local synapses. When this stage receives the global neuron fire vector, each synapse checks its associated presynaptic neuron for a firing state. If it is firing, the synapse adds its delay to the current time-step to determine the appropriate vector which is then modified by setting its bit to 1.

After updating the table for a single time-step, the table row associated to that step can be published. However, up to N time-steps ahead of the current time can be published, where N is the minimum delay across all local synapses. This allows devices to simulate ahead of one another to a point rather than being completely locked in step. Additionally, the publication of these extra buffers at the beginning of the simulation allows the data to start flowing through the simulation.

Synapse Update. Given the firing state of each synapse on the device, the synapses themselves can be updated. Like the input update stage, the synapse update stage

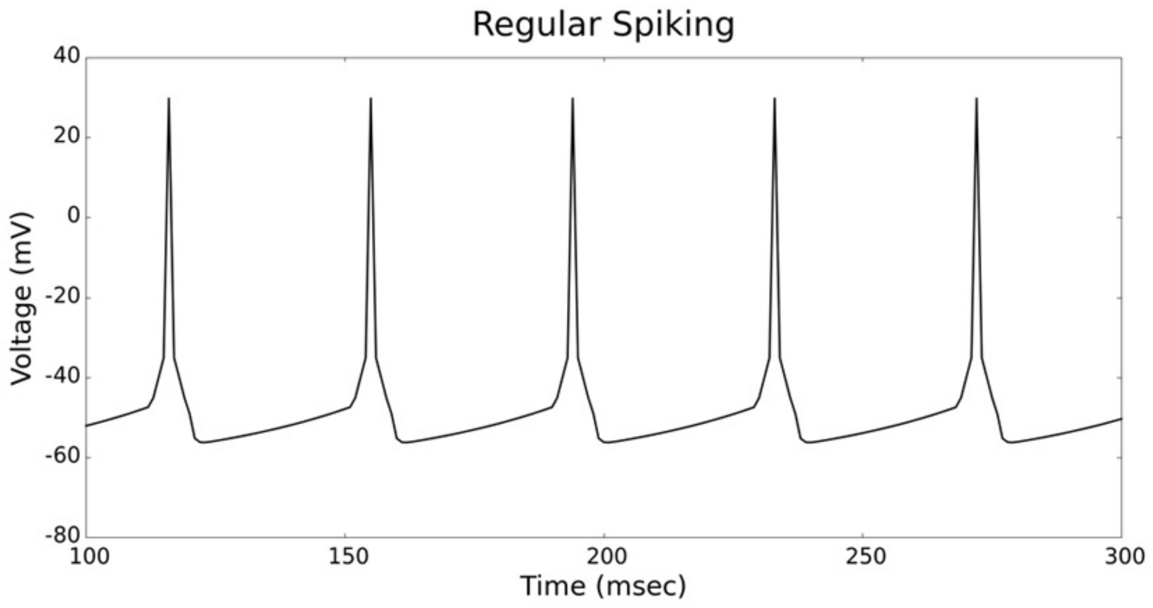


Figure 2.3: LIF Neuron Model: Regular Spiking Firing Patterns

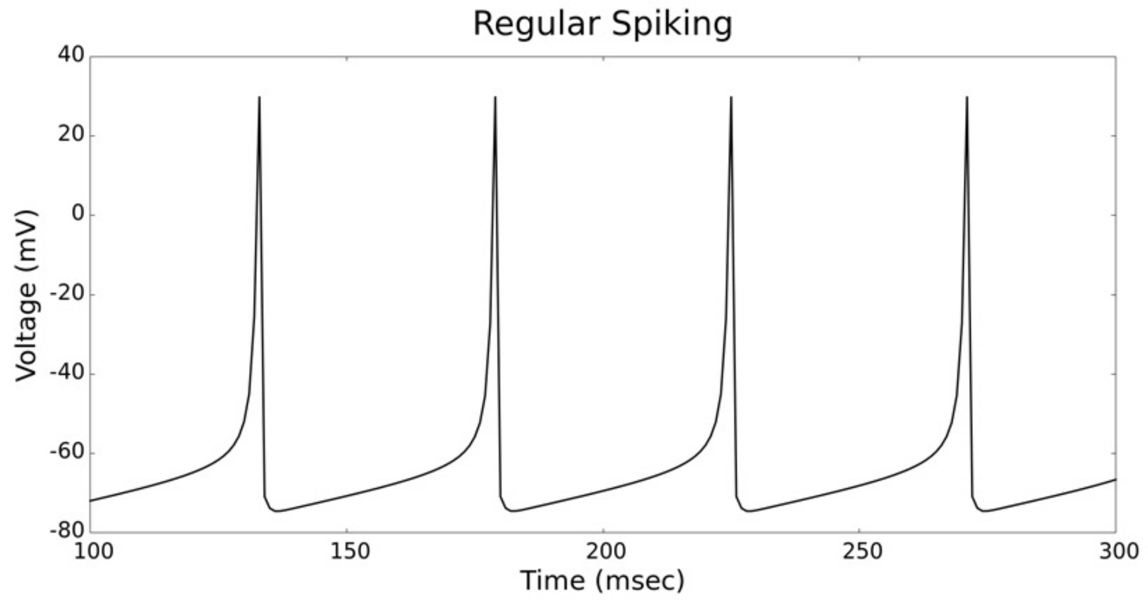


Figure 2.4: IZH Neuron Model: Regular Spiking Firing Patterns

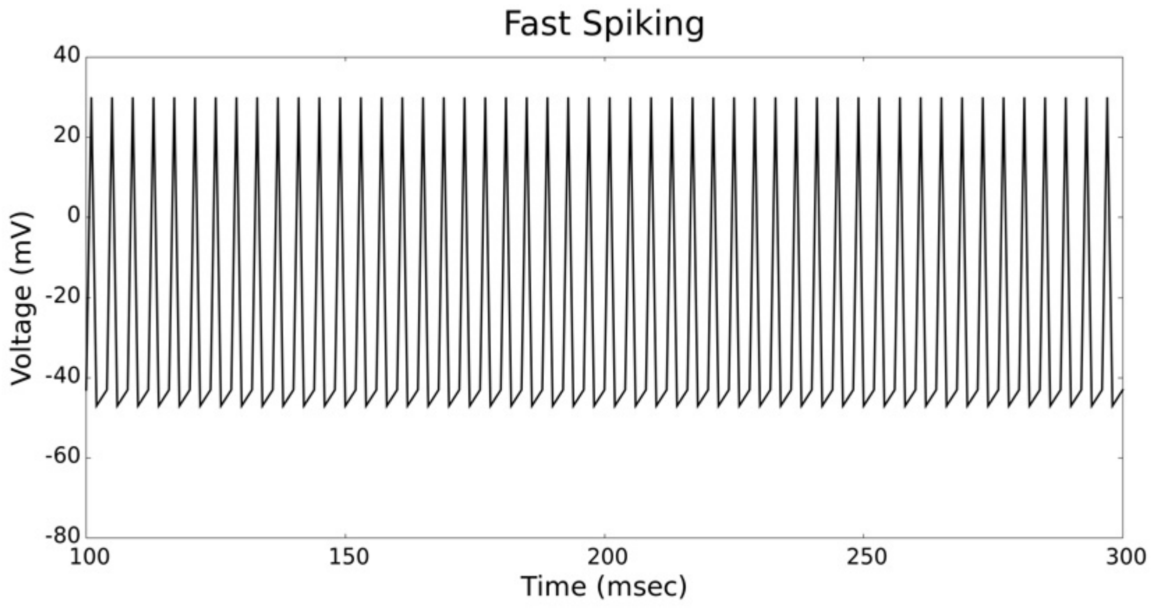


Figure 2.5: LIF Neuron Model: Fast Spiking Firing Patterns

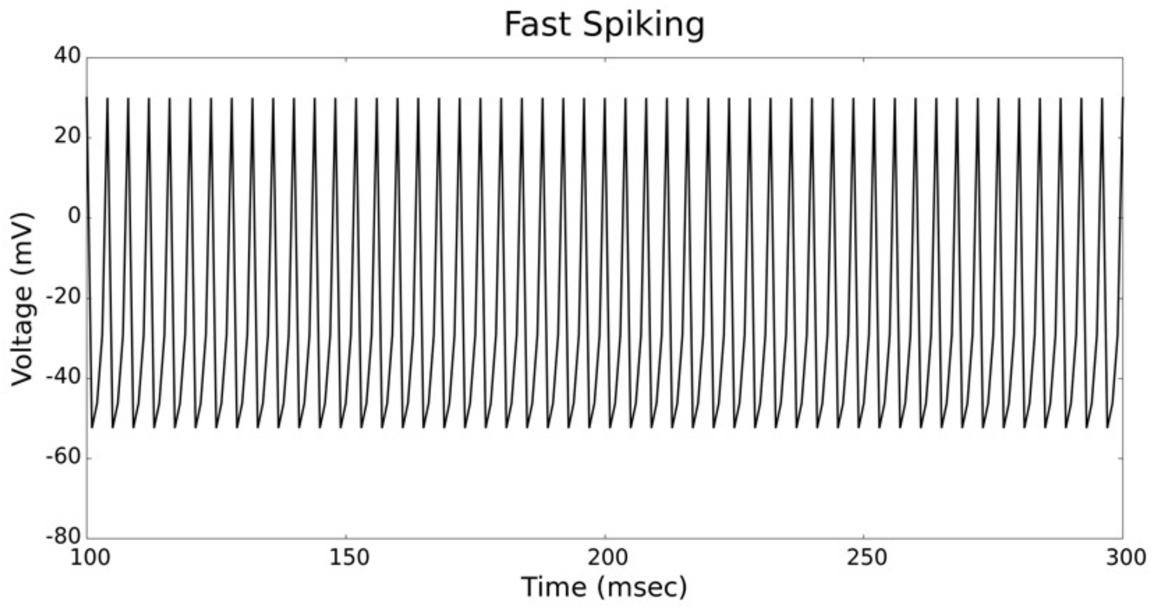


Figure 2.6: IZH Neuron Model: Fast Spiking Firing Patterns

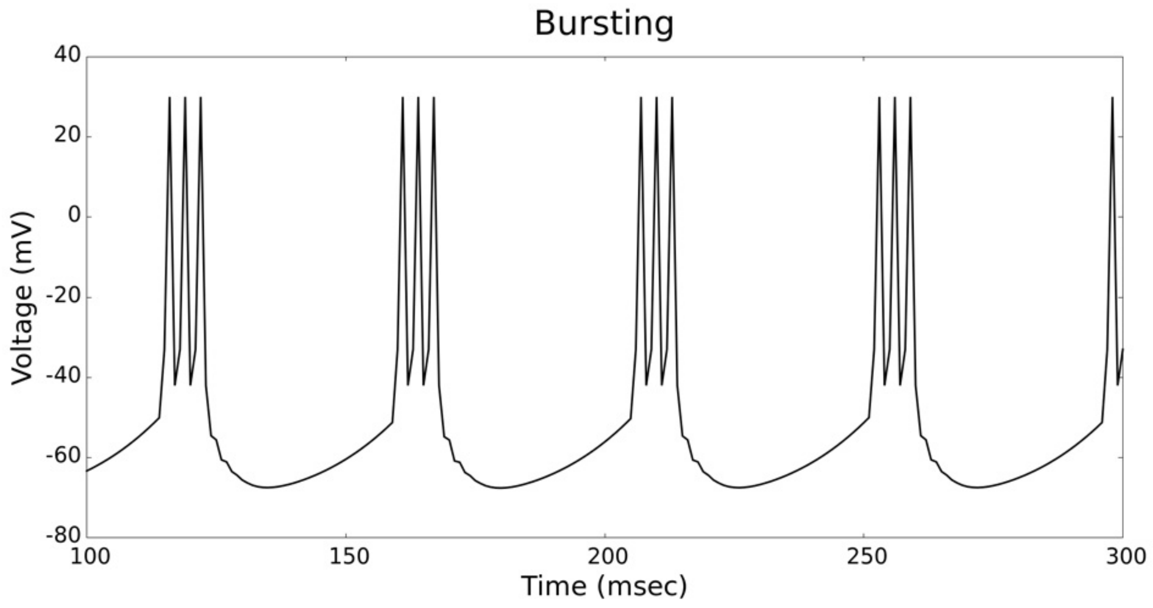


Figure 2.7: LIF Neuron Model: Bursting Firing Patterns

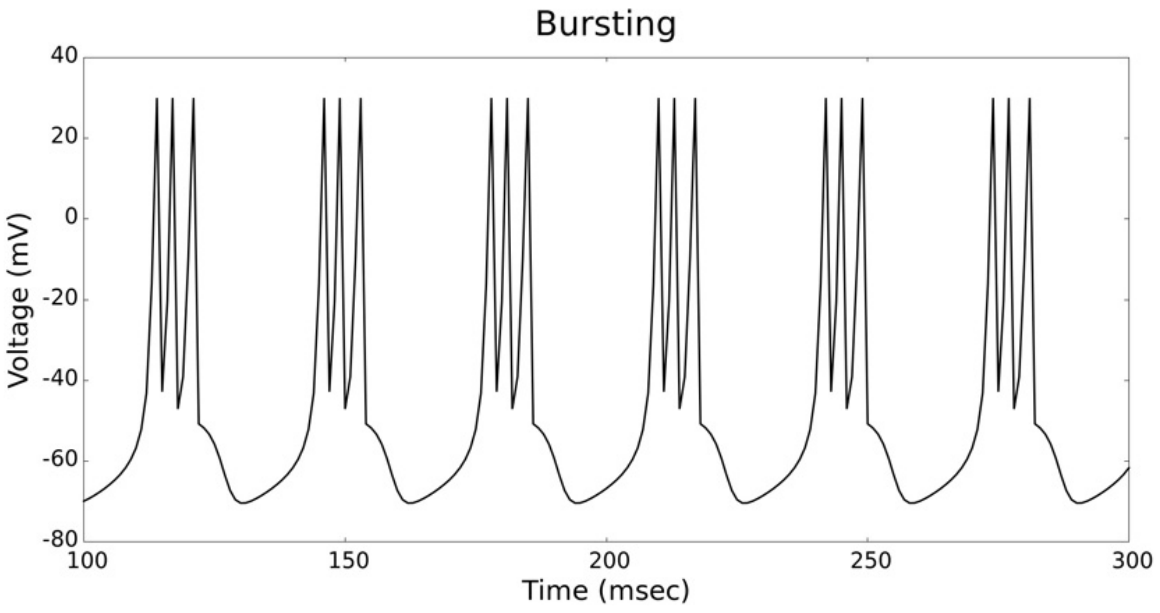


Figure 2.8: IZH Neuron Model: Bursting Firing Patterns

produces the total synaptic current per device-level neuron also represented by an array of floating point values. In terms of operating spaces, synapse plugins update synapses that operate at both the plugin and device levels, reading from the synaptic fire vector while writing to the synaptic current vector.

Reports. Reports gather information regarding some aspect of the simulation. They are specified by the user as a set of elements and values that should be extracted from them as the simulation progresses. Because these elements can be scattered across multiple devices and/or different machines and the data required can reside on one of several different scopes, every machine, device, and plugin are given a unique identifier. Following the distribution, every element that must be reported on can be located by the appropriate ID based on the data scope and the identifier within the data source.

With these two values, the appropriate data can be extracted during the simulation. To accomplish this, a single reporter is instantiated on each machine, which contains at least one element that should be collected. Then, a reporter subscribes to each publisher of the data through a more generalized publisher-subscriber interface. This interface allows a reporter to access data arrays along with the memory type using a string identifier. At each time-step, the reporter extracts data from all of its subscriptions and aggregates them as necessary. A separate MPI communication group is then used to further aggregate these data across the entire cluster asynchronously before being written out to a file or some other data sink.

Instead of using a built-in reporter type, a plugin-type interface is devised to provide flexibility in terms of data extraction, aggregation, communication, and output techniques without overly complicating the resulting code. For instance, a reporter that counts the number of neuron firings may choose to minimize data bus traffic on CUDA devices by implementing the count directly on the device and retrieving the single value rather than by downloading the entire buffer to the system memory first before operating on it. Implementations of the reporter interface are given access to an MPI communication group along with the element IDs and source identifiers to accomplish the aforementioned tasks.

2.2.5 CUDA Implementation

Every CUDA plugin in any stage of the simulation flow uses a separate CUDA stream to enqueue work for the GPU, sleeps while waiting for kernel execution to finish, and publishes the results to subscribing stages when the results are ready. Each stream operates independently on separate pieces of data, allowing the CUDA scheduler to execute kernels from different streams concurrently in order to maximize hardware utilization.

Unlike the computationally-straightforward Izhikevich model, the LIF model as specified by NCS presents a number of challenges when implementing it in CUDA. To begin with, LIF neurons can be composed of multiple compartments that affect one another and have different synaptic connections. To maintain minimal data transfer, all compartments of a single LIF neuron are decomposed into neuron-like objects that must be distributed to the same device, localizing cross-compartment interactions to that device. Since each compartment is modeled like a neuron, compartment-specific connections are realized as well.

An additional complexity of the LIF neuron comes from the ability for a compartment to have one or more channels that alter its current based on a number of different attributes. The solution to this comes from applying the simulation flow breakdown to this smaller subproblem. Each unique channel type is implemented as a plugin to the larger LIF plugin in order to minimize branching within a single kernel. At each time-step, the channel plugins concurrently modify a current buffer. This buffer is then published to the compartment updater, which in turn publishes the compartments newly updated state for use by the channel plugins in the subsequent time-step.

A final challenge to modeling NCS neurons is due to the behavior of firings. Rather than sending a single impulse across a synapse when the neuron fires, a waveform is sent over a potentially large number of time-steps. Repeated firings over a short time period produce multiple waveforms that are summed together. To enable this memory of firings in CUDA, the synaptic update plugin behavior is decomposed into a few steps. A synapse begins by checking the fire table to see if a firing has been received. If so, it pushes the event composed of a waveform iterator onto a list. That list along with the list from the previous update are then updated, computing the total synaptic current for a single neuron at the same time. If an event has not yet iterated across its entire waveform, it is pushed onto a new list that is published for the next time-step.

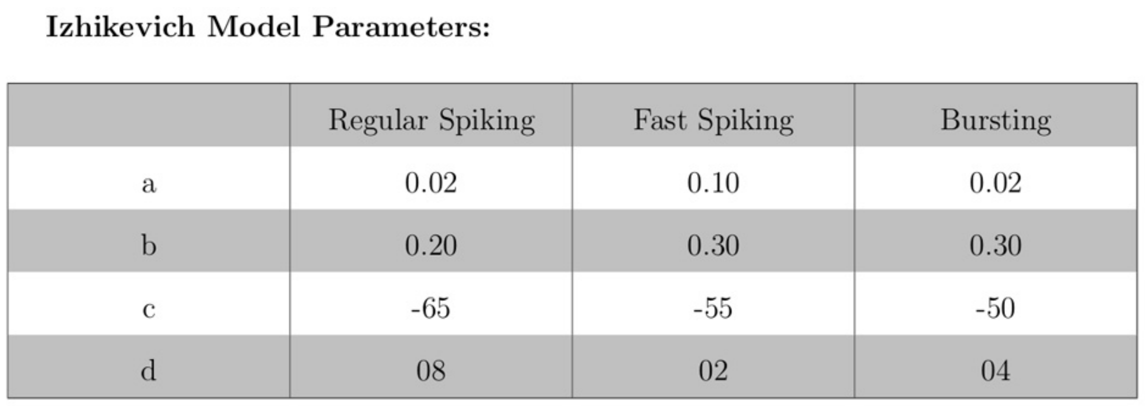
2.3 Results

The results of this manuscript are presented in the form of: neuron model validation, NCS performance, existing models using NCS, and a comparison of simulation environments.

2.3.1 Neuron Model Validation

The validation of our neuron models is crucial to the reliability of modeling studies. We have compared membrane potential traces using our two types of neurons models in response to current injection with electrophysiological data [15] and the well-known Izhikevich firing patterns [32]. As examples, we looked at three major types of neuronal firing patterns: regular spiking (RS), fast spiking (FS), and bursting (B). For the LIF neuron model, we used different types of channels and parameters. Channels included voltage-dependent and calcium-activated potassium channels. For the IZH neuron model, we used specific values for the parameters a, b, c, and d, which are given in Figure 2.9.

Figures 2.3 and 2.4 show the firing patterns of simulated regular spiking neurons using the LIF and the IZH neuron models, respectively. Figures 2.5 and 2.6 show the firing patterns of simulated fast spiking neurons using the LIF and the IZH neuron models, respectively. Figures 2.7 and 2.8 show the firing patterns of simulated bursting neurons using the LIF and the IZH neuron models, respectively. All six figures



Izhikevich Model Parameters:

	Regular Spiking	Fast Spiking	Bursting
a	0.02	0.10	0.02
b	0.20	0.30	0.30
c	-65	-55	-50
d	08	02	04

Figure 2.9: IZH neuron model: specific values used for parameters a, b, c, and d.

graph a sample of the simulation from 100 to 300 msec. Overall, our two neuron models were validated by closely replicating spike shapes and spike frequencies from electrophysiological data [15] and the well-known Izhikevich firing patterns [32] for three major types of neurons: RS, FS, and B. Note: our two models are not limited to these three types; all neural patterns can be replicated.

2.3.2 NCS Performance

Based on recent development and enhancements of NCS, we are capable of running large-scale neural models (100,000 - 1,000,000 neurons) faster than most simulators by distributing data across multiple GPUs. Considering a synapse to neuron ratio of 100 (e.g. 500,000 neurons and 50 million synapses), NCS runs any models up to almost 1 million neurons in real-time, for example, 1s simulation = 1s (IZH) or 2s (LIF) real-time, as presented in Figures 2.10 and 2.11 for the hybrid and Izhikevich neurons, respectively. In the NCS performance figures, eight machines were used with each having two video cards (GTX 680s, GTX 480s, GTX 460s, or Tesla C2050s) with a time-step of 1 ms. From one to ten-second simulations, NCS has shown no loss of performance over time, as shown in Figures 2.12 and 2.13. However, the loss of performance can occur in models containing more than 50 million synapses due to the high computation power required by synapses. The limit in terms of communications occurs when the size of the neuron vector is too large for the network to handle. In the case of GigE(1000Mbps) simulating at 1 ms intervals, we have 1 Mb per update,

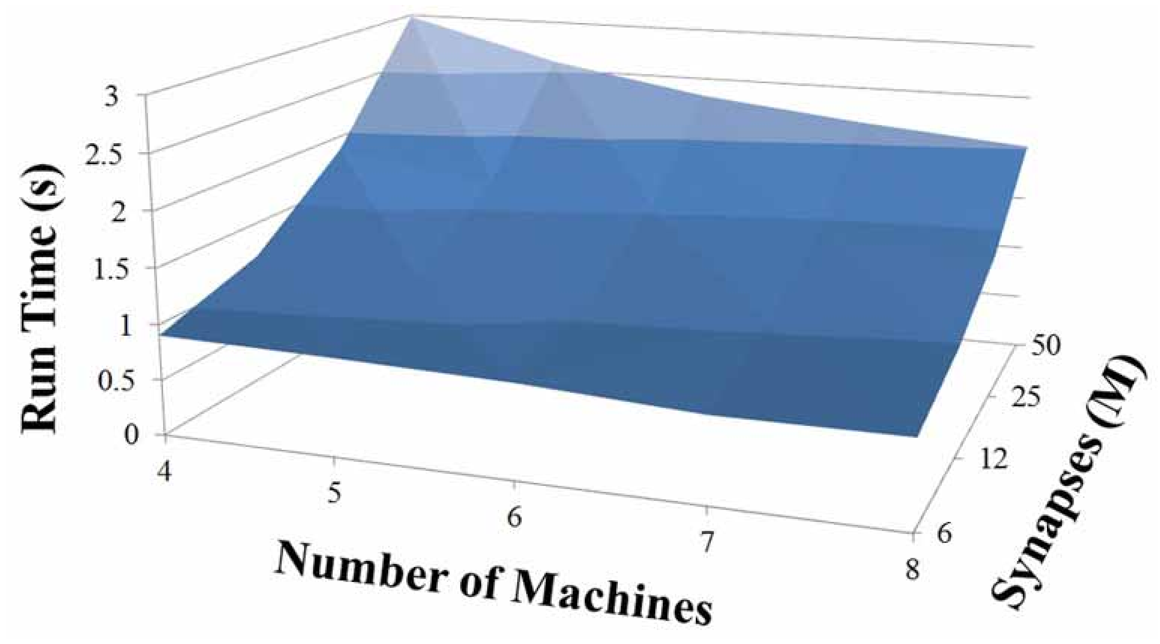


Figure 2.10: LIF Neuron Model: 1s Simulation

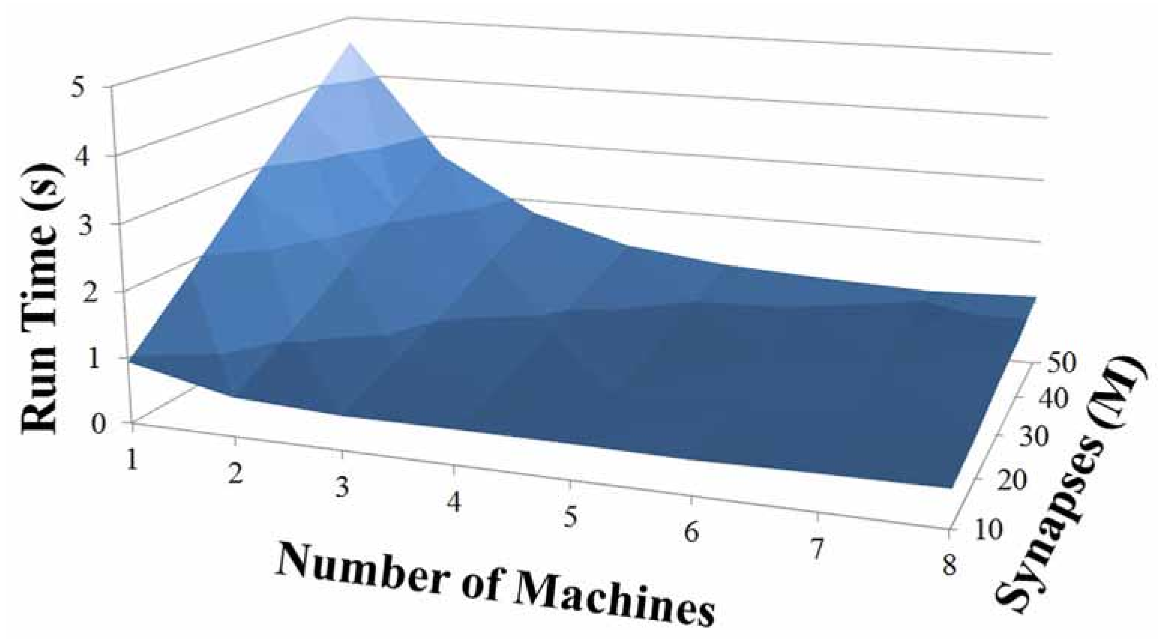


Figure 2.11: IZH Neuron Model: 1s Simulation

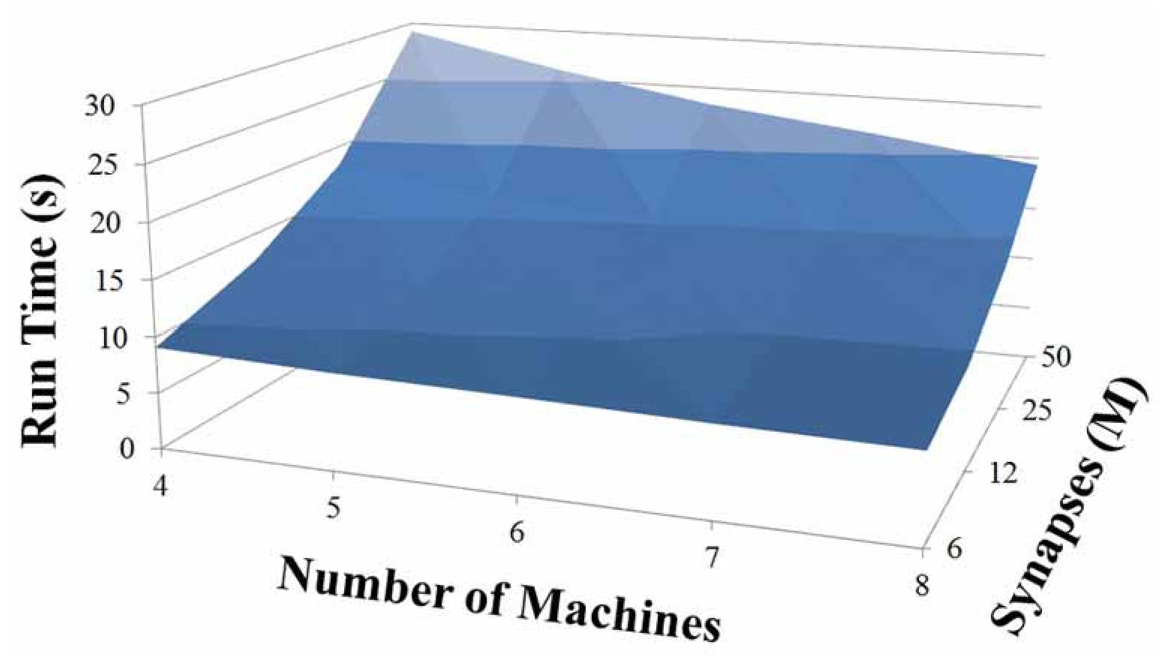


Figure 2.12: LIF Neuron Model: 10s Simulation

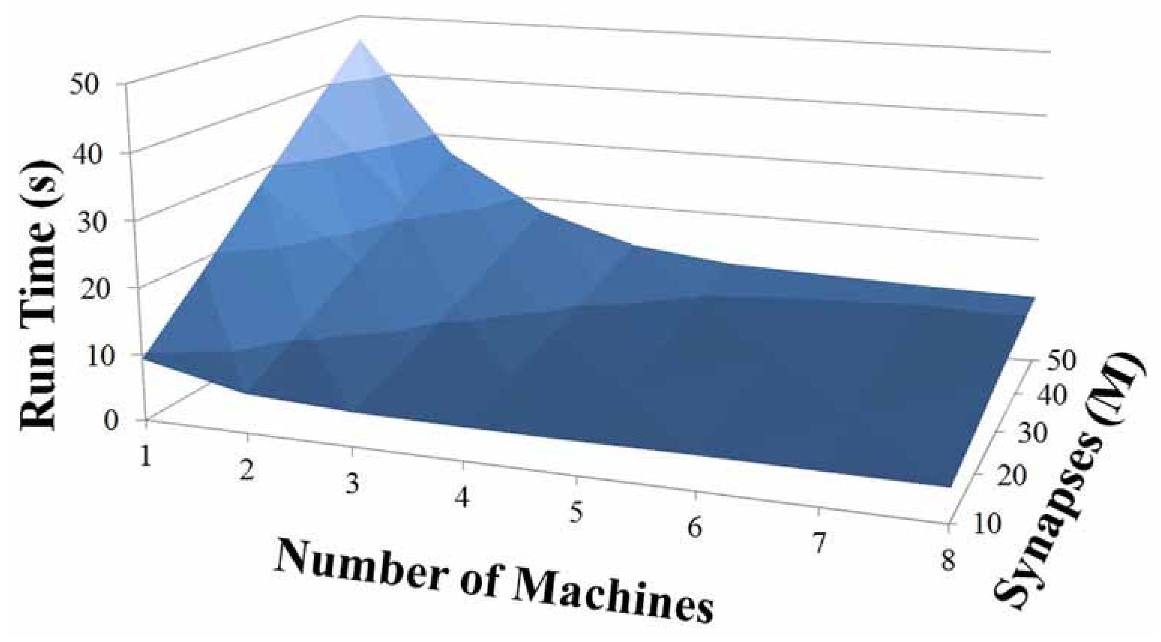


Figure 2.13: IZH Neuron Model: 10s Simulation

which represents 1 million cells (1 bit per cell). Additionally, there is MPI packet overhead. Currently, the main reason for loss of performance in very large models is due to memory constraints of the GPUs and not due to network limitations.

2.3.3 Existing Models using NCS

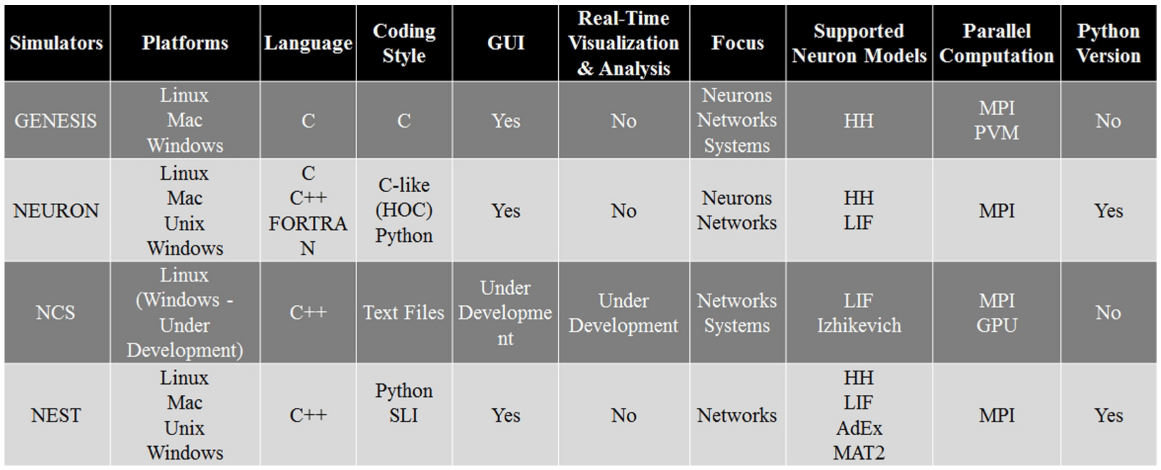
For details regarding existing models using NCS, related research projects, and publications please refer to our Brain Computation Laboratory's website: http://www.cse.unr.edu/brain/.

2.3.4 Comparison of Simulation Environments

As every simulation environment have their own advantages and disadvantages, we have compared NCS with three well-known simulators, NEURON, GENESIS, and NEST. This comparison, presented in Figure 2.14, can be useful for scientists to decide which simulator is better suited for their modeling experiments. Specifically, it describes the four simulation environments' features, such as platforms, back-end language, front-end coding style, GUI, appropriate applications, supported neuron models, type of parallel computation, and possible python version. Overall, NCS is currently well suited for large-scale neural networks and average biological details which can be simulated with LIF and IZH models. The input language for NCS is a text file and it requires minimum computer programming experience.

2.4 Discussion and Future Work

NCS6 is a new, free, open-source, parallelizable, and scalable simulator, designed to run on clusters of multiple machines, potentially with high performance computing devices in each of them. Simulator, tutorial slides, models, documentation, and conference posters are available for download at http://www.cse.unr.edu/brain/ncs. It has built-in LIF and IZH neuron models that replicate biological neural firing patterns based on experimental data [15]. All firing patterns can be reproduced with realistic spikes shapes and spikes frequencies. If users are not satisfied with these available



Simulators	Platforms	Language	Coding Style	GUI	Real-Time Visualization & Analysis	Focus	Supported Neuron Models	Parallel Computation	Python Version
GENESIS	Linux Mac Windows	C	C	Yes	No	Neurons Networks Systems	HH	MPI PVM	No
NEURON	Linux Mac Unix Windows	C C++ FORTRAN	C-like (HOC) Python	Yes	No	Neurons Networks	HH LIF	MPI	Yes
NCS	Linux (Windows - Under Development)	C++	Text Files	Under Development	Under Development	Networks Systems	LIF Izhikevich	MPI GPU	No
NEST	Linux Mac Unix Windows	C++	Python SLI	Yes	No	Networks	HH LIF AdEx MAT2	MPI	Yes

Figure 2.14: Simulation Environments Comparison

models, they also have the flexibility to design their own plug-in interfaces for different neuron types. NCS6 is currently able to simulate one million cells and 100 million synapses in quasi real time by distributing data across these heterogeneous clusters of CPUs and GPUs. A variety of models have been created and simulated with NCS, and they have shown interesting findings on high-level behaviors (e.g. navigation). The advantages of using NCS6 are its computational power, its biological capabilities at multiple levels of abstraction, and its minimum computer programming demand. NCS6's main limitations include its lack of biophysical parameters, its only availability on LINUX platforms, and the absence of a GUI. Therefore, our current work consists of increasing the biological details behind NCS6 without affecting simulation time. NCS6 should be soon available on Windows, and be able to run on openCL-capable devices. Additionally, our main focus has been on developing a real-time visualization and analysis tool to make the use of NCS6 convenient to a broader community.

Appendix

NCS Cell Equations

At a cellular level, NCS solves a limited and slightly reordered form of the Hodgkin-Huxley model that is similar to equation (2.1). However, during the numerical integration a constant membrane leak is added. This is explained further below.

$$C_N \frac{dV}{dt} - I_M - I_A - I_{AHP} - I_{input} - I_{syn} + I_{leak} = 0 \tag{2.1}$$

The currents expressed in this equation fall into several different categories that are only briefly explained here. To begin, both I_M and I_{AHP} contribute to the membrane voltage by controlling spike-frequency adaptation. These are small ionic currents that have a long period of activity when the membrane voltage is between rest and threshold. I_M is the Noninactivating Muscarinic Potassium Current and is defined by

$$I_M = \bar{g}_M S m^P (E_k - V) \tag{2.2}$$

Where S is a non-dimensional strength variable added to NCS and P is the power that the activation variable m is raised to. This is essentially decreasing the slope of the activation variable. The change of that activation variable is defined as

$$\frac{dm}{dt} = \frac{m_\infty - m}{\tau_m} \tag{2.3}$$

Where

$$\tau_m = \frac{\epsilon}{e^{\left(\frac{V - V_{1/2}}{\omega}\right)} + e^{-\left(\frac{V - V_{1/2}}{\eta}\right)}}$$

$$m_\infty = \frac{1}{1+e^{-\left(\frac{V-V_{1/2}}{\xi}\right)}}$$

ϵ is the scale factor.

$V_{1/2}$ satisfies the equation $m_\infty(V_{1/2})$=0.5.

ω,η, and ξ are slope factors affecting the rate of change of the activation variable m.

Notice that equation (2.2) is different from the traditional equation shown below in equation(2.4). This reverse of the driving force explains the sign changes in equation (2.1).

$$I_M = \bar{g}_M m_m (V - E_K) \tag{2.4}$$

I_{AHP} is the current provided by the other small spike-adaptation contributing channel. These are voltage independent potassium channels that are regulated by internal calcium.

$$I_{AHP} = \bar{g}_{AHP} S m^P (E_K - V) \tag{2.5}$$

Where S is a non-dimensional strength variable added to NCS and P is the power that the activation variable m is raised to. The change of that activation variable is defined as

$$\frac{dm}{dt} = \frac{m_\infty - m}{\tau_m} \tag{2.6}$$

$$\tau_m = \frac{\epsilon}{f(Ca)+b}$$

$$m_\infty = \frac{f(Ca)}{f(Ca)+b}$$

Where

ϵ is a scale factor.

b is the backwards rate constant, defined as CA_Half_Min in the NCS documentation.

$f(Ca)$ is the forward rate constant defined by equation (2.7).

$$f(Ca) = \kappa \left[Ca\right]_i^{\alpha} \tag{2.7}$$

Internal calcium concentrations are calculated at the compartment level in NCS. Physiologically the calcium concentration of a cell increases when an action potential fires. After the action potential has ended the internal concentration of calcium will diffuse throughout the cell where it is taken up by numerous physiological buffers. In NCS this diffusion/buffering phenomena is modeled by a simple decay equation defined by equation (2.8).

$$\left[Ca\right]_i (t+1) = \left[Ca\right]_i (t) \left(1 - \frac{dt}{\tau_{Ca}}\right) \tag{2.8}$$

Where

dt is the simulation time step.

τ_{Ca} is the defined time constant for the Ca decay.

When an action potential fires in NCS the internal calcium concentration is increased by a static value specified in the input file.

The third and final channel type modeled in NCS is the transient outward potassium current or K_a. This channel requires hyperpolarization for its activation; meaning that the channel will open during inhibitory synaptic input. This is defined by equation (2.9).

$$I_K = \bar{g}_M S m^P h^C \left(E_K - V\right) \tag{2.9}$$

Where as before S is a non-dimensional strength variable added to NCS, P is the power that the activation variable m is raised to and C is the power that the inactivation variable h is raised to. The change of activation and inactivation variables is defined by equations (2.10) and (2.11).

$$\frac{dm}{dt} = \frac{m_\infty - m}{\tau_m} \tag{2.10}$$

$$\frac{dh}{dt} = \frac{h_\infty - m}{\tau_h} \tag{2.11}$$

Where

$$m_\infty = \frac{1}{1+e^{-\left(\frac{V - V_{1/2m}}{\xi}\right)}}$$

$V_{1/2m}$ satisfies the equation $m_\infty(V_{1/2m}) = 0.5$.

ξ is slope factor affecting the rate of change of the activation variable m.

$$h_\infty = \frac{1}{1+e^{-\left(\frac{V - V_{1/2h}}{\eta}\right)}}$$

$V_{1/2h}$ satisfies the equation $h_\infty(V_{1/2h})$=0.5.

η is slope factor affecting the rate of change of the inactivation variable h.

τ_m and τ_h are voltage dependent. NCS allows this dependence to be defined using an array of values for both voltages and time constants. This is defined by equation (2.12).

$$\tau(V) = \begin{cases} \tau(1) & \text{if V < V(1),} \\ \tau(2) & \text{if V < V(2),} \\ \vdots & \\ \tau\,(n) & \text{if V < V(n)} \\ \tau\,(n+1) & \text{else} \end{cases} \tag{2.12}$$

The leakage current is voltage-independent and is modeled by equation (2.13). Notice that the driving force is expressed using the normal convention. This is the reason the leakage current is subtracted in the membrane voltage equation rather than added, as seen in the traditional membrane voltage equations.

$$I_{leak} = g_{leak}\left(V - E_{leak}\right) \tag{2.13}$$

The synaptic currents are calculated by

$$I_{syn} = \bar{g}_{syn} PSG\left(t\right)\left(E_{syn} - V\right) \tag{2.14}$$

The numerical integration scheme employed by NCS is similar to an Eulerian method. However, as mentioned above a constant leak term is added to the discretized form of equation (2.1). To begin the current values defined above are summed

$$I_{Total} = I_M + I_A + I_{AHP} + I_{input} + I_{syn} - I_{leak} \tag{2.15}$$

The new voltage is then calculated as a combination of the defined membrane resting potential, the previously calculated membrane potential, the membrane resistance, capacitive time constant and total currents.

$$V\left(t+1\right) = V_{rest} + \left(V\left(t\right) - V_{rest}\right)\left(1 - \frac{\Delta}{\tau_{mem}}\right) + \Delta\frac{I_{Total}}{C_n} \tag{2.16}$$

Rearranging for clarity

$$V\left(t+1\right) = V\left(t\right) + \left(V_{rest} - V\left(t\right)\right)\frac{\Delta}{\tau_{mem}} + \Delta\frac{I_{Total}}{C_n} \tag{2.17}$$

Where

$C_n = \frac{\tau_{mem}}{R_{mem}}$

R_{mem} is the defined resistance of the membrane.

τ_{mem} is the defined capacitive time constant of the membrane.

Notice the form of equation (2.1) in a simple Eulerian integration scheme would be

$$V\left(t+1\right)=V\left(t\right)+\Delta\frac{I_{Total}}{C_n} \tag{2.18}$$

The addition of the middle term in equation (2.17) numerically drives the membrane voltage of the cell back to a predefined resting potential.

When the voltage crosses a specified threshold value $v_{threshold}$, the membrane potential follows a user-specified spike shape pattern. During this time, the internals of each channel are updated; however, they have no effect on the value of the memberane potential. At the end of the pattern, calculations resume using equation(2.17).

Chapter 3

NeoCortical Builder: A Web Based Front End for NCS

Jakub Berlinski, Cameron Rowe, Daniel M. Chavez, Nathan M. Jordan, Devyani Tanna, Roger V. Hoang, Sergiu M. Dascalu, Laurence C. Jayet Bray, and Frederick C. Harris, Jr. NeoCortical Builder: A Web Based Front End for NCS. In Proceedings of the 27th International Conference on Computer Applications in Industry and Engineering (CAINE-2014), 2014. Submitted.

Abstract

The NeoCortical Builder (NCB) is a web-based application for building brain models and creating simulation input and output parameters. NCB is also able to launch simulations on the NeoCortical Simulator (NCS). NCS is a neural network simulator which takes various brain models with corresponding simulation parameters and provides output. NCB was designed to streamline the process of model creation and the creation of simulation input and output parameters. NCB has the ability to drastically change the way various types of scientists interact with brain simulators. NCB creates a modern graphical interface to streamline the creation of brain simulations.

3.1 Introduction

Brains are complex systems which are difficult to fully understand. The problem is brains are extremely complicated to model and in many different brain modeling applications the interface is confusing or outdated. One example of a competing product is NEURON [14]. NEURON also simulates brain models but its interface is outdated and sometimes confusing. Along with having an interface, NEURON also allows users to use text based input files. NCS has tried different ways to create brain models and simulation parameters, such as name and duration, for brain simulations. The latest method involved using text based input files. These files were complicated to construct as they required very specific syntax that the user would need to know. NCB solves this problem by providing a modern graphical user interface which the user will interact with to create their brain model as well as their simulation parameters. This graphical user interface will be implemented as part of a bigger front end web application.

Web based applications are becoming more prevalent because the Internet is more reliable and faster than ever before. Ease of use and accessibility are key features of web based applications and NCB provides that for any user that uses it. To run NCB all a user needs is access to a device capable of running a modern web browser. The goal of NCB as a web based application is to provide the end user with a portable way to create brain models.

The rest of this paper is structured as follows: Section 3.2 covers what NCS is and the technologies used to create NCB. Section 3.3 covers the design of NCB. Section 3.4 goes into detail of model creation. Section 3.5 discusses conclusions and future plans for NCB.

3.2 Background

3.2.1 NCS

The NeoCortical Simulator is a neural simulation tool developed at the University of Nevada, Reno [21]. NCS has the ability to model millions of neurons in real time. It allows neuroscientists to build, test, and visualize a brain model that they design using NCB. NCS can run on a single computer or across a cluster of computers. Most importantly, NCS uses both CPUs and NVIDIA GPUs to generate the maximum computational power [27].

The goal of NCS is to create an approachable neuro simulator that can be used by any neuroscientist without the need to know how to write any code. Using NCS, users will be able to quickly launch simulations and receive their outputs in real time with the click of a button. NCS also aims to be an extensible application where different parts of a model are computed with different inputs that the user defines. This allows NCS to have many more combinations of components which create the most realistic simulation possible. Finally NCS aims to do all of this in the most efficient manner. Harnessing the power of GPUs, NCS will be able to use the maximum amount of resources to provide the simulation of neurons in the most efficient way [26].

3.2.2 Technologies

Many web technologies were used in the creation of the NeoCortical Builder. The core technologies that were used to build NCB are HTML5[41] and CSS[40]. The following technologies were also used: JavaScript, JQuery, AngularJS, Bootstrap, X-editable, and Flask.

- JavaScript is an object-oriented computer programming language which is used by NCB to create interactive effects within the web application [42]. For example, JavaScript would detail what would happen when a certain element is selected by the user. Additionally, JavaScript is used as a way to hold brain models and simulations on the back end.

- JQuery is a library for javascript which simplifies DOM manipulation [49]. NCB uses JQuery to edit the HTML of the web page to show or hide various different elements. JQuery is being replaced with AngularJS because AngularJS is a more robust library for manipulating the web page.

- AngularJS is a javascript library which allows for single web page applications. Usually when creating a web page there are many different pages that are accessed when using the site. For example selecting "about" on a web page will load a whole new HTML file for that page. Angular allows one big html file where elements are hidden or shown instead of having multiple files [23]. NCB uses AngularJS as a framework to set up a single page web application.

- Bootstrap is a set of design templates which use HTML and CSS to create various types of elements on the web page [53]. NCB uses many different bootstrap design templates from web page structuring to the design of buttons and dropdown menus.

- X-editable is a javascript library which allows DOM elements to be made editable. The elements can then be edited through either a pop-up or in-line within the webpage [46]. In NCB this was one of the most important features because it allowed users to edit the various brain model and simulation parameters quickly and easily.

- Flask is a python based web server framework [47]. NCB uses flask for communication with the running simulation.

3.3 Design Overview

The front end web application for NCS will be made up of many different parts as shown in Figure 3.1. The NeoCortical Builder will be in charge of two major components, the brain builder tab will be used to create brain models and the simulation builder tab which will create simulation runtime parameters. The next two tabs will be done by the NeoCortical Repository team (NCR) [2]. The first of their two tabs will be the reports tab which will show various graphs and other information pertaining to the running simulation. Their second tab will be the model repository tab which will be in charge of holding all the brain models in various databases. The final section will be the virtual robot tab which will use the simulation output to affect a robot.

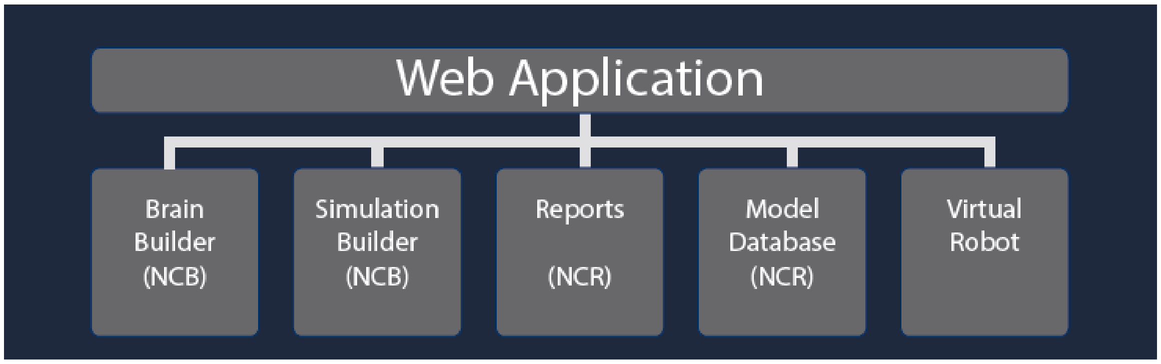


Figure 3.1: The complete web application framework.

3.3.1 Brain Builder

The brain builder tab is split into a three column layout as shown in Figure 3.2. The left column is where the pre-built models will be loaded from various model databases. Additionally, the models in this tab may be filtered instantly by typing into the filter input box as shown in Figure 3.3. Selecting a model will expand a quick view which details the model parameters as shown in Figure 3.4. In this column there is also a button which prompts the user to import models from a file or export models to the user's computer. There is an options button which allows the user to customize the colors of various portions on this tab.

The middle column is where the model currently being worked on is shown. In this column there is a button which allows the user to add various brain model elements including neurons, groups, and aliases shown in Figure 3.5. After an element is added the user is able to select the red minus button to remove that element. Groups are able to be nested within each other and when this occurs a breadcrumb trail is created across the top of the column shown in Figure 3.6. Additionally, there is a second button which allows the user to edit the current model parameters including name, description, and author. Selecting a model in this column will populate the right column.

The right column is where all the selected model's parameters are shown. The parameters are split into three main categories. The categories are cell-group parame-

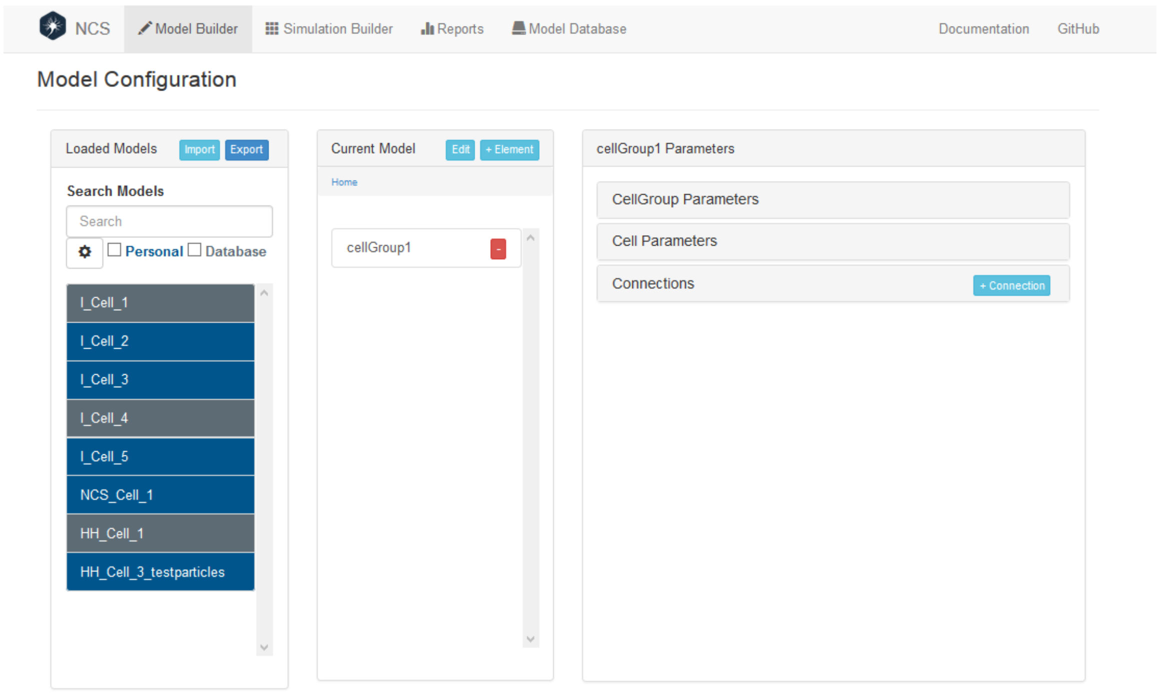


Figure 3.2: The model builder tab within the NCB front end web application.

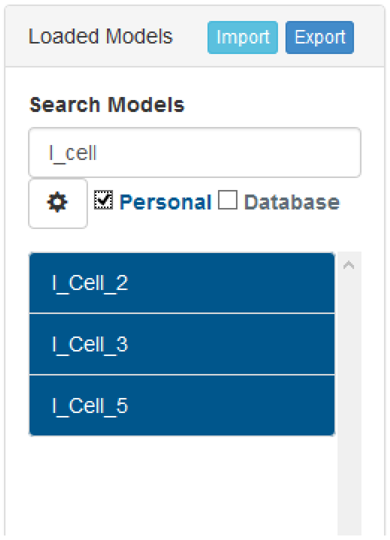
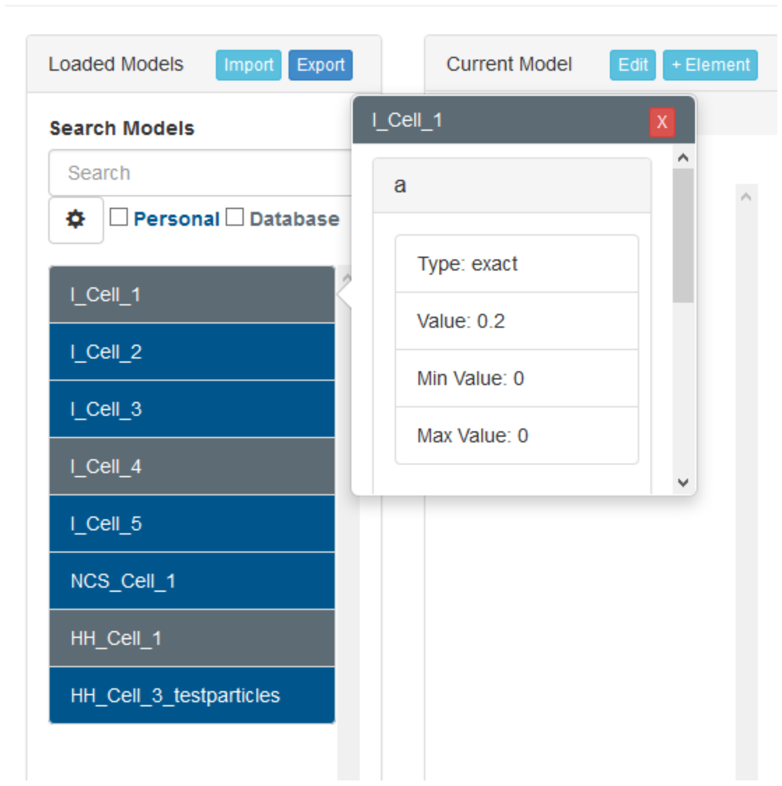


Figure 3.3: Filtering the models in the left menu.



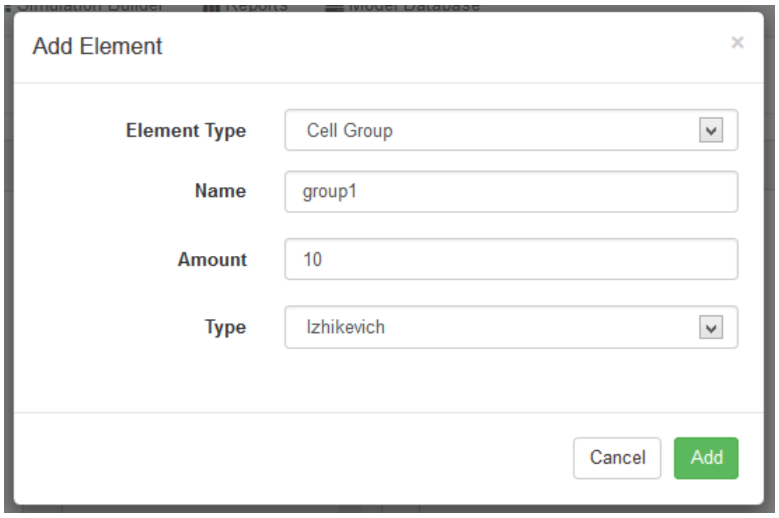
Type: exact

Value: 0.2

Min Value: 0

Max Value: 0

Figure 3.4: The popover when a user selects a pre-built model in the left menu on the model builder tab.



Element Type: Cell Group

Name: group1

Amount: 10

Type: Izhikevich

Figure 3.5: When the user selects the "+ Element" button they are shown this modal which asks for various element details.

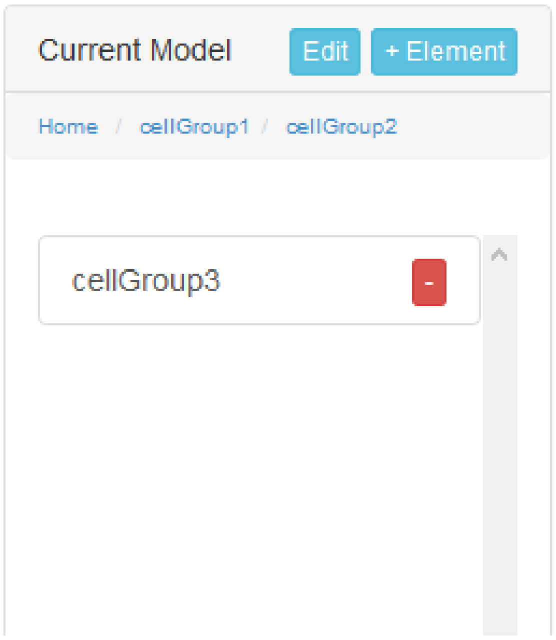
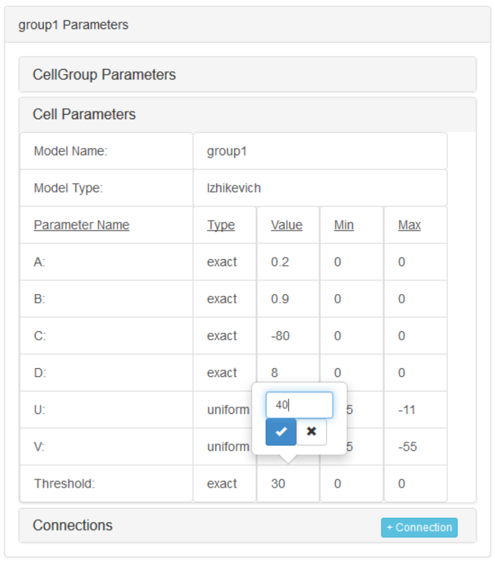


Figure 3.6: Multiple cell-groups nested within each other.

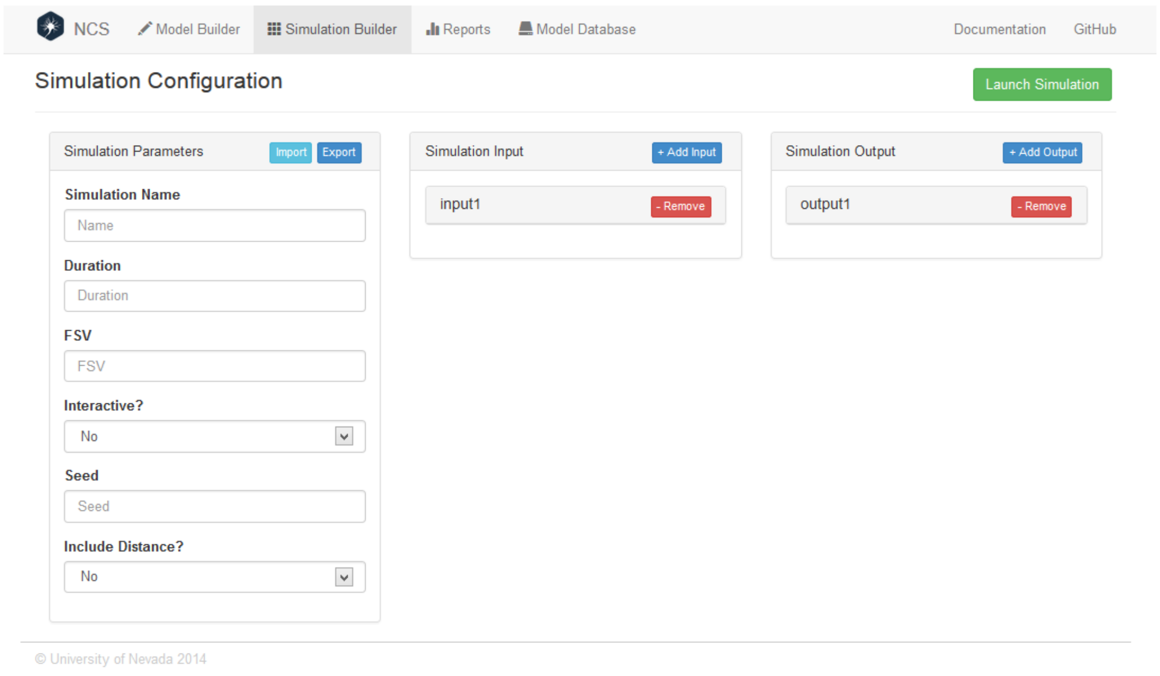


Model Name:	group1			
Model Type:	Izhikevich			
Parameter Name	Type	Value	Min	Max
A:	exact	0.2	0	0
B:	exact	0.9	0	0
C:	exact	-80	0	0
D:	exact	8	0	0
U:	uniform	40	5	-11
V:	uniform		5	-55
Threshold:	exact	30	0	0

Figure 3.7: Editing a model parameter in the right column.

ters, cell parameters, and connections. Certain cells like the Hodgkin-Huxley cell-type may have channels so in these cases a third category will be dynamically created to show this. To add connections the user can select the "+ Connection" button and a pop-up appears prompting the user for various parameters which are needed to create a connection. Furthermore, all parameters will be editable when the user selects them. When a user clicks on an editable parameter a small pop-up appears where the user can then enter their new parameter value as shown in Figure 3.7.

3.3.2 Simulation Builder



Simulation Configuration

Simulation Parameters

Simulation Name: Name

Duration: Duration

FSV: FSV

Interactive?: No

Seed: Seed

Include Distance?: No

Simulation Input: input1

Simulation Output: output1

Figure 3.8: The simulation builder tab within the NCB front end web application.

The simulation builder tab, as shown in Figure 3.8 is also split up into a three column layout. The left column is where the user may add simulation parameters. Some simulation parameters include name, duration, and seed. These simulation parameters are used as a way to identify a specific simulation and also to give NCS some important information about how a simulation should be ran.

The middle column is where the user will input the runtime parameters for the simulation. Some input parameters are stimulus type, frequency, input elements, start and end times, and many more. These parameters indicate where currents will be applied on the built brain model.

The right column is where the user will input the output parameters for the simulation. Some output parameters are output types, report types, output elements, frequency, start and end times, and many more. These parameters indicate specific elements the user wants to observe.

3.4 Usage Scenario

When the user first opens the NCB web page they are shown the front page which is Figure 3.2. As we mentioned in the previous section, there is a three column layout with the first column being the preloaded models, the middle column being the current model, and the right column being the various parameters of the current models selected elements.

The first thing a user would want to do if they were creating a fresh model is to select the add element button. After the user selects that button they are shown the pop-up modal, shown in Figure 3.5, which asks for the type of element to add and various parameters like the name and amount. The user may add as many elements as they need using this method. Figure 3.9 shows a model with two cell-groups.

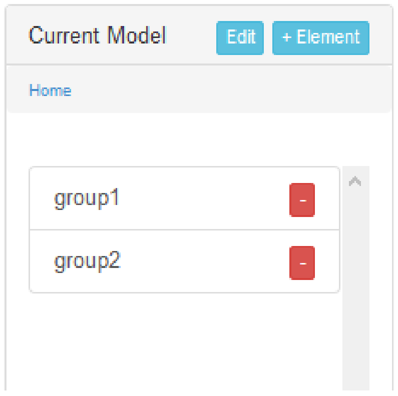
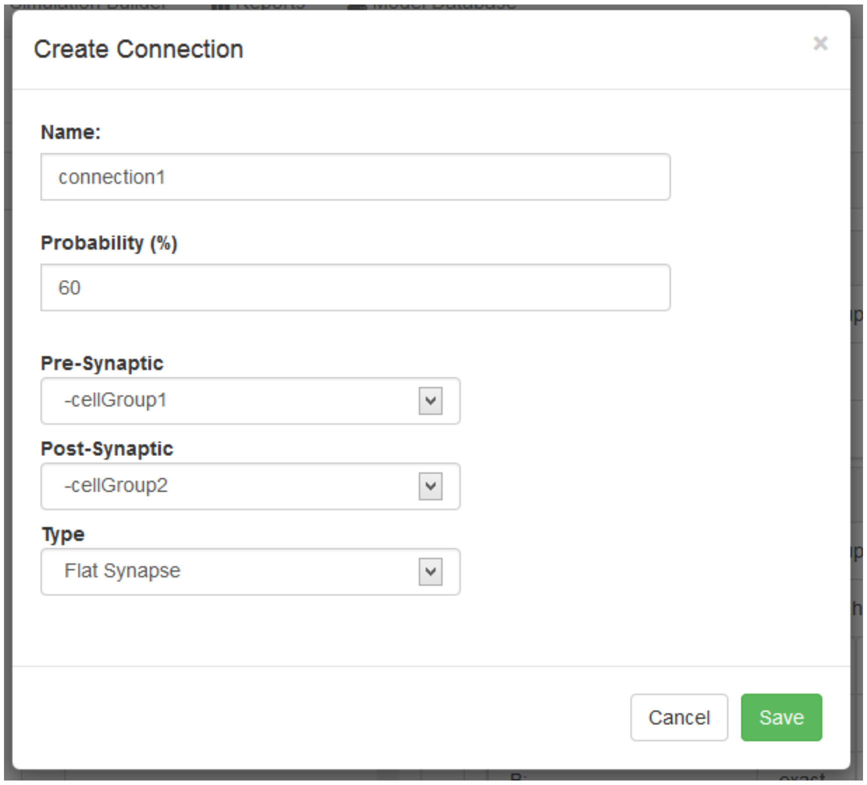


Figure 3.9: Two cell-groups have been added to the current model and they are shown in the center column.

Next the user would edit some parameters in the model they are building. When the user selects a cell group the right column becomes populated with the selected element's parameters. Selecting one of the values brings up a small pop-up where the user may update and save their new value as shown in Figure 3.7. All parameters may be edited to take a specific set of values. When a user selects "exact" they need to only enter one value for that parameter. When a user selects "uniform" they are required to enter two values, a minimum and a maximum. Finally if a user selects the "normal" option they are required to enter a mean and standard deviation value.


Create Connection

Name: connection1

Probability (%) 60

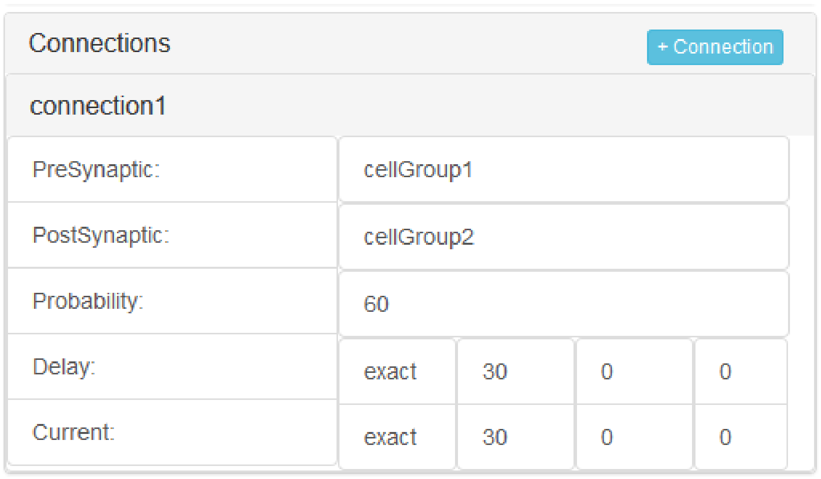
Pre-Synaptic -cellGroup1

Post-Synaptic -cellGroup2

Type Flat Synapse

Figure 3.10: The modal which prompts the user for values for a connection.

Currently in our working model there are two cell-groups that are disconnected from each other. To make a complete brain model the user will need to connect them to each other using a connection. To add a connection between different elements, the user must select the "+ Add Synapse" button and they will be shown the modal in Figure 3.10. Here the user must enter a name for their connection, a probability that the connection will fire, two elements which are being connected, and finally a type of connection. The two types of connections that NCB supports are flat synapses and NCS synapses. After adding a connection, it will appear as a collapsible element within the connections in the right column as shown in Figure 3.11.

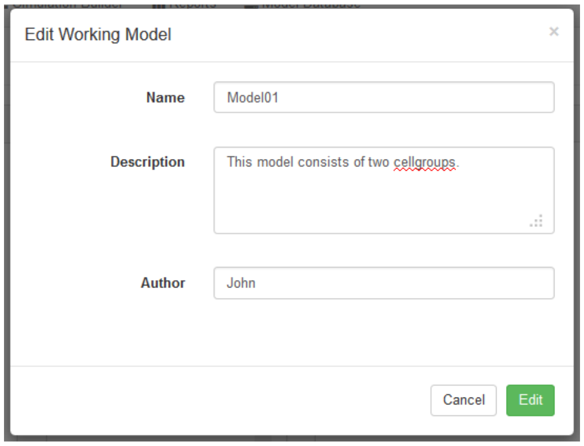


PreSynaptic:	cellGroup1			
PostSynaptic:	cellGroup2			
Probability:	60			
Delay:	exact	30	0	0
Current:	exact	30	0	0

Figure 3.11: An example of a connection between two elements, group1 and group2.

After adding a connection between these two elements the model is simple but complete. The user may want to personalize their current model by giving it a name, a description, and setting the author. To do this the "Edit" button will be selected and a modal pops up where the user may enter these values. This is shown in Figure 3.12. After the model is edited the user may choose to export the model to their computer. Furthermore, if the user does not want to create a new model he or she may import a previously built model. With a complete model the user may navigate to the simulation builder tab and begin editing their simulation parameters.

In the left column on the simulation builder page the user will enter information about the simulation's name, how long the simulation should run, the seed that will be used, and various other information. For the simulation the user will also need to fill out what type of input and output the user wants from the simulation.



Name: Model01
Description: This model consists of two cellgroups.
Author: John

Figure 3.12: The modal which prompts the user for values to personalize their model.

3.5 Conclusion and Future Work

3.5.1 Conclusion

NCB provides many features that make the creation of brain models and simulations easier and more intuitive. With a graphical front end for creating brain models, the user does not need to get into creating convoluted text files that may very quickly get very large and unreadable. NCB makes the power of NCS accessible to any scientist who wants to simulate brain activity. NCB has been designed with simplicity in mind to allow anybody the ability to use it.

3.5.2 Future Work

Although NCB contains many features for creating brain models, it is still only half of the final web application. NCB deals with the creation of brain models and simulations while NCR deals with the saving of the models in a database and displaying the reports from the simulation. In the future both of these projects will be combined to create the final and complete front end web application for NCS.

Chapter 4

NeoCortical Repository and Reports: Database and Repository for NCS

Edson Almachar, Alexander Falconi, Katie Gilgen, Devyani Tanna, Nathan M. Jordan, Roger V. Hoang, Sergiu M. Dascalu, Laurence C. Jayet Bray, and Frederick C. Harris, Jr. NeoCortical Repository and Reports: Database and Repository for NCS. In Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE-2014), 2014. Submitted.

Abstract

In the field of Computational Neuroscience, computer based brain simulators help Neuroscientists in the formulation and examination of theories about the inner workings of the brain in the microscopic and cellular level. Brain simulators offer an opportunity to refine and build upon the compendium of scientific knowledge in Neuroscience. One of these brain simulators is the NeoCortical Simulator (NCS). In operating a computational powerhouse, there is a need of an interface with which a user would interact with to communicate with the simulator. We propose a browser based web application that a user may browse to in order to use a plethora of services for the simulator. Two of these services include a Repository Service which allows a user to save a brain model onto a database and a Reporting Interface which allows

a user to view the data output by the simulator. This paper details the design and implementation of those services, called NeoCortical Repository and Reports (NCR).

4.1 Introduction

The NeoCortical Simulator from the University of Nevada, Reno, is a joint research venture between the colleges of Science, Engineering, and Medicine within the University of Nevada, Reno[26, 27]. It is a tool for researchers to perform CPU/GPU based simulations with biological brain models. The NCS uses brain model data as arguments to the simulator and then outputs data in a parsable text format. Previously, the brain models had to be coded in, which was an inconvenience for researchers unfamiliar with programming. Additionally, the outputs were not conveniently displayed and required extra time and effort to understand the data.

Past projects have expanded on the idea of a friendlier user experience when interacting with the simulator. One of which is the NCS-NeuroML translator [34] which allowed the conversion of NCS input files from the standard NeuroML input language to the native input language via the usage of a java-gnome user interface library thereby streamlining interaction with the simulator. Another is the 3D Neuron visualizer, or NeoCortical View[13], which allowed a user to visualize the current live network state of the simulation in 3D.

To continue this trend of streamlining the experience, we wish to further improve the interface. We propose the NCS Web Application; an intuitive, browser based application that users may browse to on their web browser and begin interacting with the NCS without need of a command line interface. The NCS Web Application is comprised of five main components that encourage a smooth user experience within the NCS. These components are outlined in Figure 4.1.

There are multiple development teams in charge of the construction of this web application. The NeoCortical Builder team is responsible for Brain Builder and Simulation builder [3]. Whereas, the development team, NeoCortical Repository and Reports, is concerned with the development of the Model Database and the Graph-

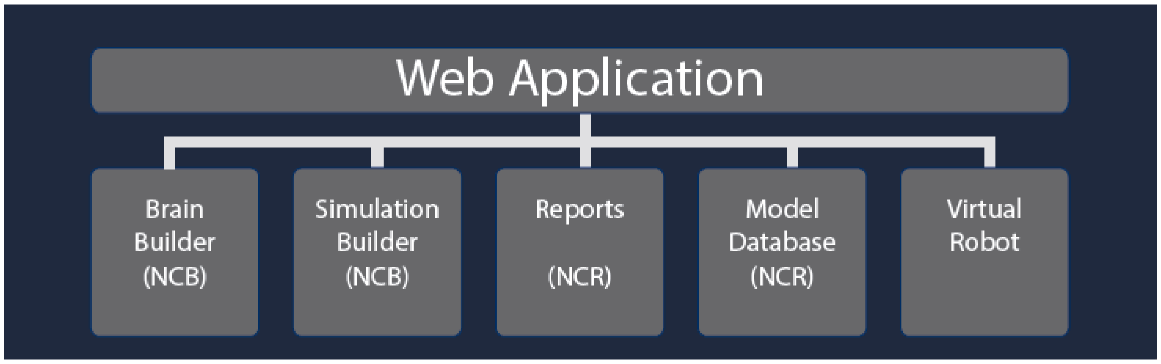


Figure 4.1: The architecture of the NCS Web Application.

ical Reports, and as such, is the main topic of discussion for this paper. The two components developed by NCR are described by the following:

Firstly, when a user chooses to interact with the Database, the user must be aware that NCS has the ability to simulate three biological neuron models: Izhikevich, Leaky integrate-and-fire, and Hodgkin-Huxley. NCR is concerned with the implementation of a Model Database that would need to understand these brain models in order to conduct services like storage, search, and updating a model.

Secondly, further down the cycle in an active simulation, there would have to be an output. NCR is concerned with the implementation of a Reporting Interface comprised of line graphs and raster plots, which are the standard graphing mediums in Neuroscience. These reports make interpretation easy and swift so as to promote efficient use of time and better productivity.

As a result, the implementation of NCR within the NCS mainframe will accomplish a multitude of goals that encourage a user friendly experience with the NCS. To help outline these goals, the design overview of the NCR components and their use cases are detailed in Section 4.2. The User interface, a key asset in the human to computer interaction of the application, is detailed in Section 4.3. The paper wraps up with a discussion of future work in Section 4.4.

4.2 Design Overview

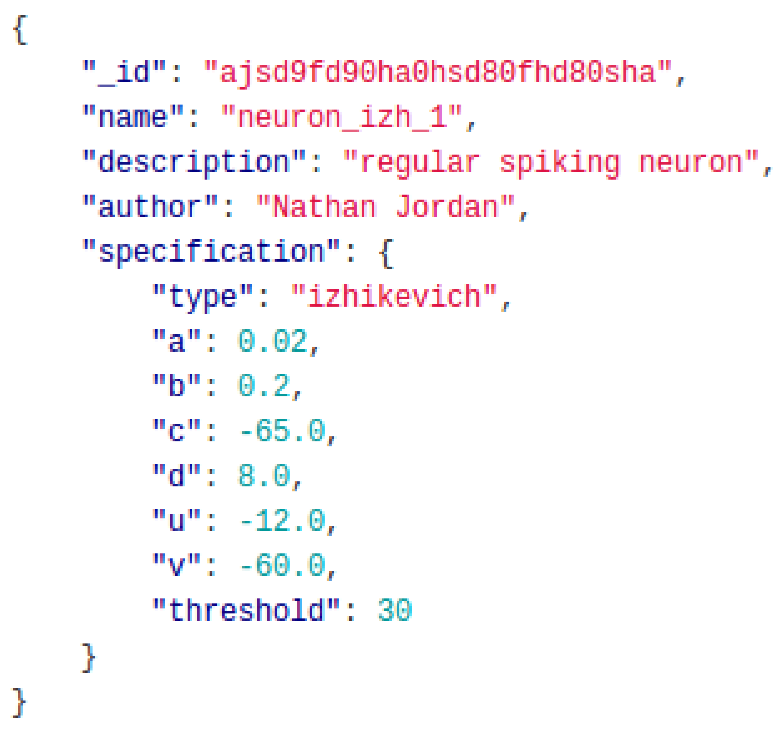
The main functionality of NCR is based on its manifestation as a website. The site is hosted on a web server base constructed using FLASK[47], a python based micro-framework. The model database is designed using MongoDB[36], which is an easily scalable non-relational database. Hosted over the internet, the database can be used to upload or store brain models. The reporting interface is constructed via D3.js[6], a javascript graphing library, and jQueryUI[49], a javascript user interface library. Everything the user does to interact with the NCS would be done so through the users' web browser.

4.2.1 Database Design

The brain model database gives users of the NCS web application the ability to easily collaborate with others and save valuable time. The model database is designed in order to store and query various brain models. MongoDB is an ideal choice for the database since it is free, open source, flexible in terms of schema, and uses JSON-style(Javascript Object Notation) documents as shown in Figure 4.2.

The MongoDB document design is shown in Figure 4.3. Big boxes are documents, and boxes inside boxes are sub-documents. Sub-documents are good for faster queries. Each document belongs to one of the 6 collections in the database: Groups, Neurons, Channels, Synapses, Stimuli, and Reports. By default, MongoDB does not enforce schema. In order to have structured schema and validation layer, MongoKit [39] is used. A database schema is created for each document type, and is used as the format for the models within MongoDB.

The simplistic structure of the search panel provides an intuitive way for a user to find a useful model in the database without the need for searching through various text files and downloads. The database stores models from users who have uploaded or created a model using the Model Builder tab of the application, meaning that users from around the world may publish a model for other users to view.



```
{
    "_id": "ajsd9fd90ha0hsd80fhd80sha",
    "name": "neuron_izh_1",
    "description": "regular spiking neuron",
    "author": "Nathan Jordan",
    "specification": {
        "type": "izhikevich",
        "a": 0.02,
        "b": 0.2,
        "c": -65.0,
        "d": 8.0,
        "u": -12.0,
        "v": -60.0,
        "threshold": 30
    }
}
```

Figure 4.2: JSON document for Izhikevich Neuron

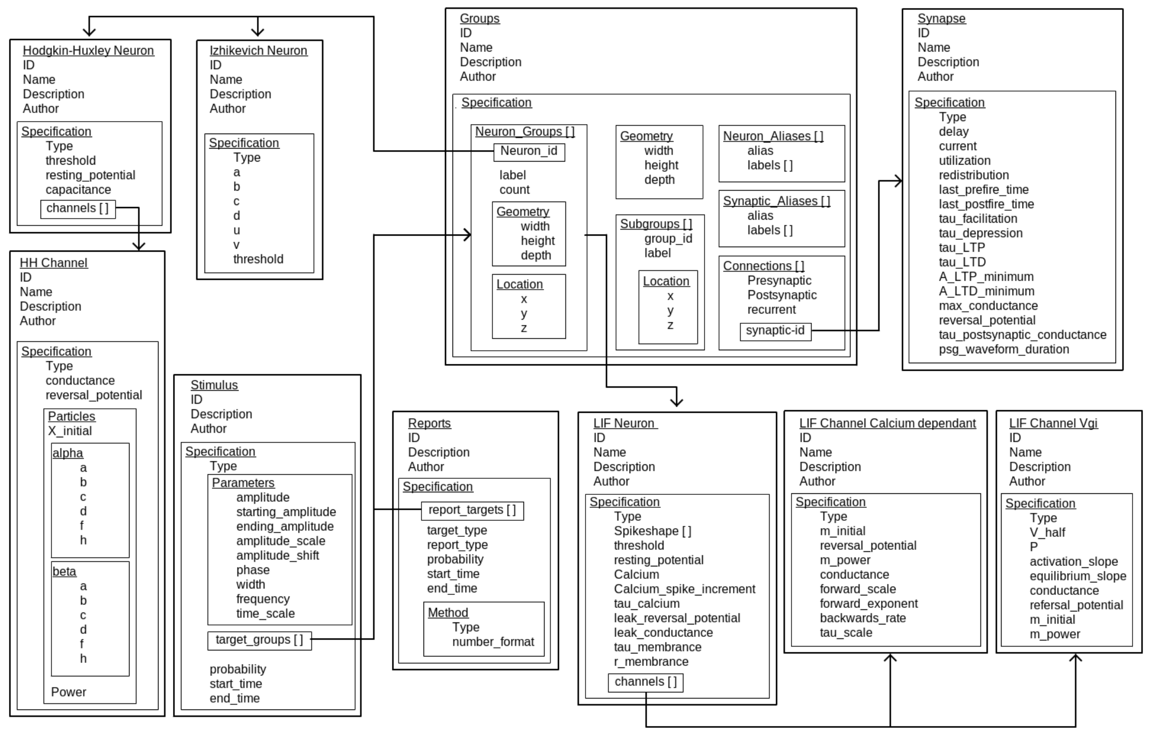


Figure 4.3: MongoDB documents design

4.2.2 Reports - High Level Design

The Graphical Reports tab of the NCS web application aims to provide as much of an intuitive interface as possible in an attempt to maximize comprehension of the data and to minimize the complication of technicality. In order to provide such an interface, a fully dynamic environment is generated that would allow a user to manage and manipulate graphs to their liking. Users may manage graphs by dynamically creating or deleting them. Users may apply spatial manipulations by dragging and placing the graphs from one area of the web page to another, allowing the user to reorganize the graphs to their liking.

When considering a large amount of data, accuracy in representation must be considered when abstracting output from the NCS. Scalable Vector Graphics (SVG) are the main abstraction medium for the data and utilized for dynamic representation. D3.js provides the interface between the data and the SVG representing the data. As data is continuously fed to the client from the server, the SVG's must change dynamically over time. To accomplish this, D3 allows for animated SVG's.

Presentation both on the NCS web application and off are important. When considering the applications ability to save graphs onto disk, accurate representation has to be priority. To facilitate this concern, client side graph capture of the browser generated SVG was an optimal design choice. Users may capture the SVG elements within the webpage and have its context downloaded as an SVG file. For an animated Graphics Interchange Format (GIF), the SVG is continually contextualized into a HTML5 Canvas element. That element is then captured and inserted frame by frame into a GIF object and downloaded as a GIF file.

4.2.3 Reports - Low Level Design

The graphical reporting interface is comprised of multiple components whose functionalities vary widely, but each having an important role in the construction of a reporting window. The components interaction is shown in Figure 4.4.

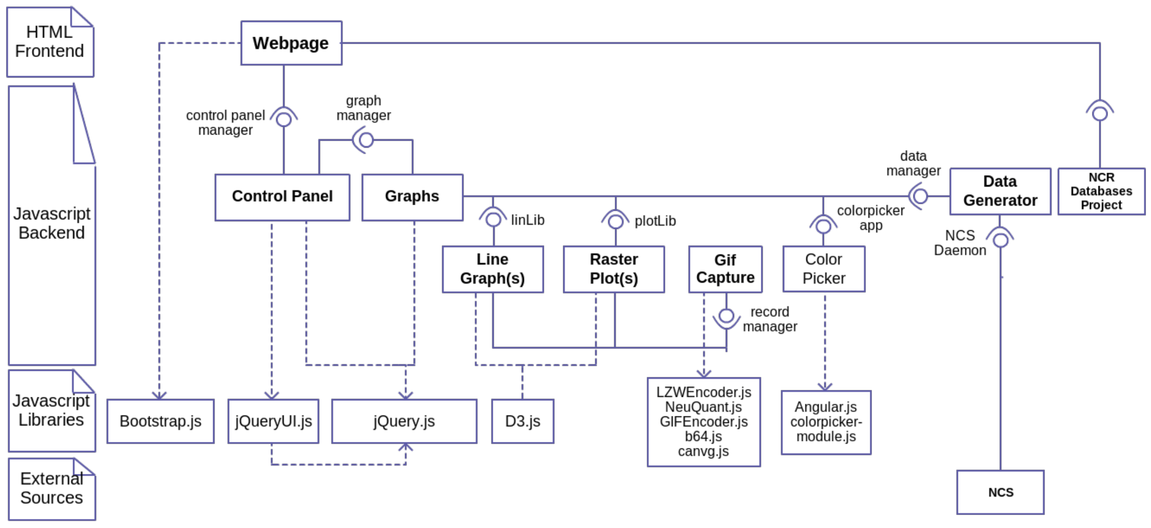


Figure 4.4: The low level design of the reports interface.

Control Panel - The component representing the control panel that would allow a user to manipulate the environment that the graphs will be instantiated into.

Graphs - An singleton that would allow a user to add or remove graphs. This entity is also responsible for maintaining and feeding data to graphs.

Data Generator - An entity that manages the data received from NCS and would continuously inject that data into the line graphs and raster plots.

Line Graph(s) - A D3 based entity that holds and manages the line graph SVG.

Raster Plot(s) - A D3 based entity that holds and manages the raster plot SVG.

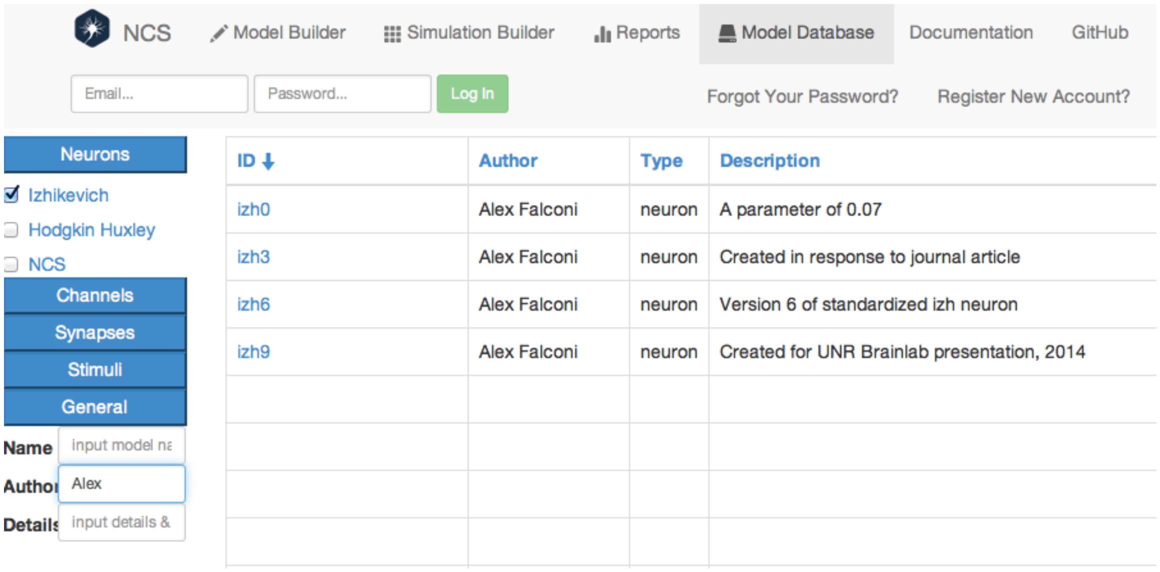
GIF Capture - A service that allows a Line graph or Raster plot to be recorded via GIF or SVG and saved to file.

4.3 User Interface

4.3.1 Repository

The model database tab of the NCS Web Application is used to search through the brain model database using a simplistic search panel. The model list on the tab shows the models in the database that match the search criteria specified by the user. Within the search panel, the user can filter the list based on the model types, which are collected into groups in the search panel. Selecting a model type opens a collapsible list of searchable parameter values. The user may enter an exact value, or a range of values delimited by a dash.

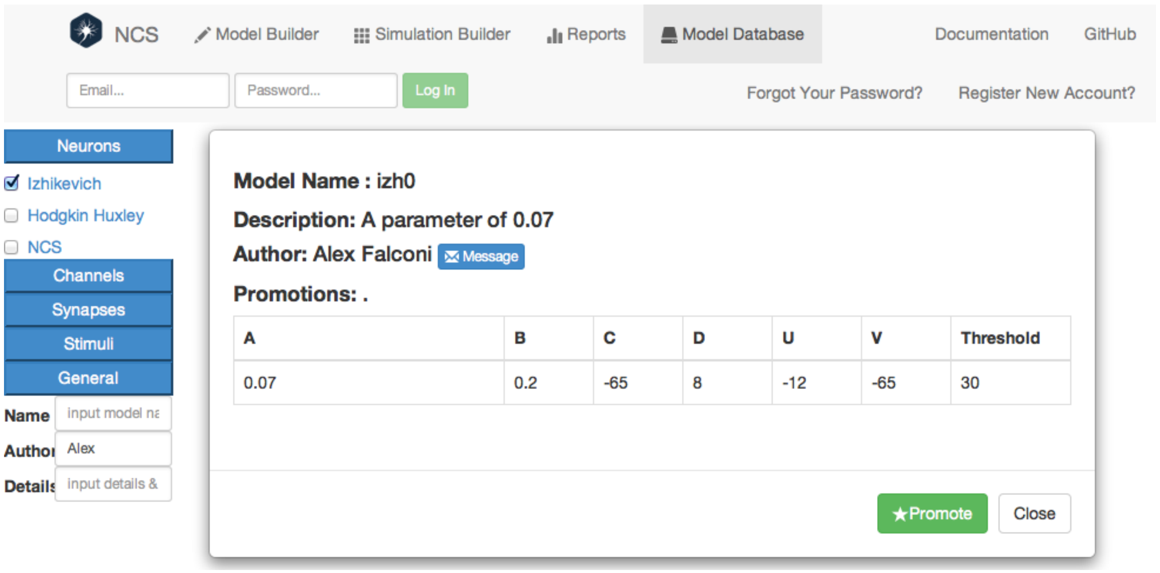
The search panel is located on the left side of the model database tab. The initial panel shows groupings for model types that can be expanded to reveal model type selection boxes. Selecting a model type box adds models of the selected type to the filtered results. If the name of the type is selected, the search panel expands to show the parameter search options. Here, the user enters an exact or range of values into the search box and clicks the search icon to update the model list. Only the models of the selected type that have the specified value appear in the list. A user can specify as many included types and parameter values as desired for a search. The general filter group includes filtering the list based on name, author, description, and scope values. An example of the list filtered by selecting the group Neuron, type Izhikevich, and specifying the author name are shown in Figure 4.5.



ID	Author	Type	Description
izh0	Alex Falconi	neuron	A parameter of 0.07
izh3	Alex Falconi	neuron	Created in response to journal article
izh6	Alex Falconi	neuron	Version 6 of standardized izh neuron
izh9	Alex Falconi	neuron	Created for UNR Brainlab presentation, 2014

Figure 4.5: The models in the list are populated based on the search filter values, located in the left search panel.

By selecting the name of a list item, a user can view the models details. A model view opens on the page and shows the models general information, such as the name, author, and description. The detail view includes a table of parameters and the models parameter values. Each detail view is specific to the selected model type in order to intuitively display the information, as shown in Figure 4.6.



Model Name : izh0

Description: A parameter of 0.07

Author: Alex Falconi

Promotions: .

A	B	C	D	U	V	Threshold
0.07	0.2	-65	8	-12	-65	30

Figure 4.6: Selecting a model name in the list opens the detailed view which shows the parameter values and the promote option..

Use Cases - Repository

UC01 Search Models

A user selects model types to be included in the search results, and then expands the parameter dropdown for each type to enter parameter values. The list is populated based on the filters by type selections and entered parameter values.

UC02 Examine Model

After a user has applied search filters to the model list, the list contains relevant models. The user selects the name of a model in the list and a detail view window opens. The user views the parameter values for the model.

UC03 Upload and Download Model

A user can choose to upload a model from database to model builder and download model from model builder to database.

4.3.2 Reports

Upon entering the reports tab of the application, users will be greeted by a reports control panel. From here, users will be able to control a myriad of functions which allow the user to add graphing instances into the environment, add graphing columns, change settings, and view the reporting status of the simulator. When a user adds a graph instance, it is either in the form of a line graph or a raster plot. The graph is instantiated and placed in the environment. The environment which houses the graphs is comprised of graphing columns. A graphing column is a container that houses and displays graphs vertically (e.g. if a user has one graphing column, it is possible to have one "stack" of instantiated graphs. If a user has two graphing columns, it is possible to have two "stacks" of graphs). A key feature enabled by these graphing columns is the ability to drag graphs from one graphing column to another. This is the main implementation that allows for spatial customization where a user can customize a certain region of the webpage to have a certain set of graphs.

When a graph is instantiated, each graph will have its own set of buttons at the header of the instantiated graph. These buttons grant the user a series of customization tools: add or delete lines, change the color of a line, zoom in or zoom out within a graph, change the vertical dimensions of a graph, pause a graphs' reporting state, resume a graphs' reporting state, record the current state of the graph as an animated GIF or static SVG.

Use Cases - Reports

UC01 Add Line Graph or Raster Plot

A user has the ability to add reporting windows that can abstract the data in the form of a Line Graph or a Raster Plot.

UC02 Zoom In or Zoom Out

A user can zoom into or out of a reporting window to accommodate their viewing preferences. The differences in zoom levels can be viewed in Figure 4.7.

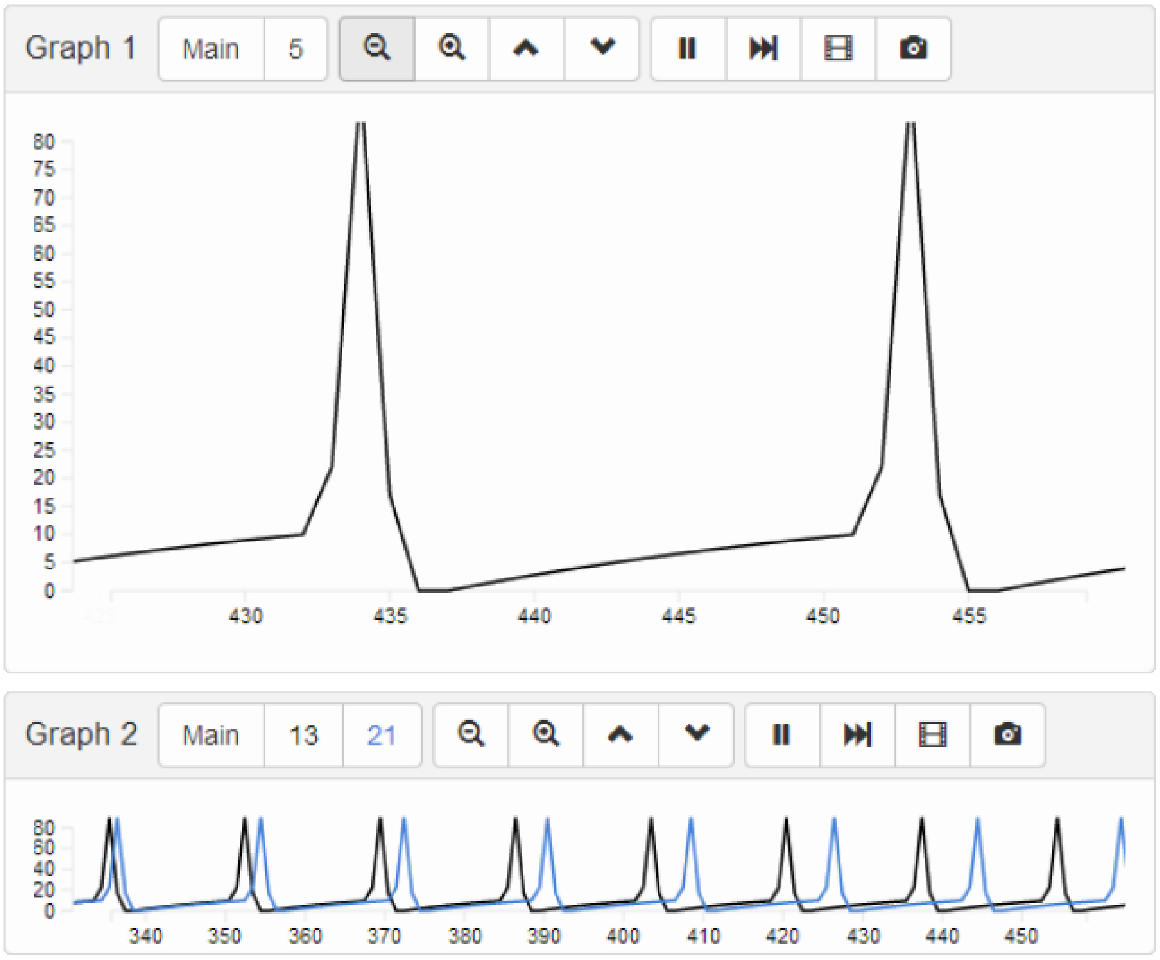


Figure 4.7: An example of multiple zoom levels and window dimensions.

UC03 Change Color

A user can choose to change the color of a line in a line graph to any color using a gradient color picker for easier viewing of a certain cell as seen on Figure 4.8.

UC04 Play or Pause

As a graph is dynamically reporting data, a user can choose to pause the reporting process and the graph will halt at the current data shown. When the user is ready, he or she may continue to view the reports by pressing the play button. Play and pause buttons are viewable in Figure 4.9.

UC05 Position Slider

Should the user wish to view old data that has already been reported, the user may

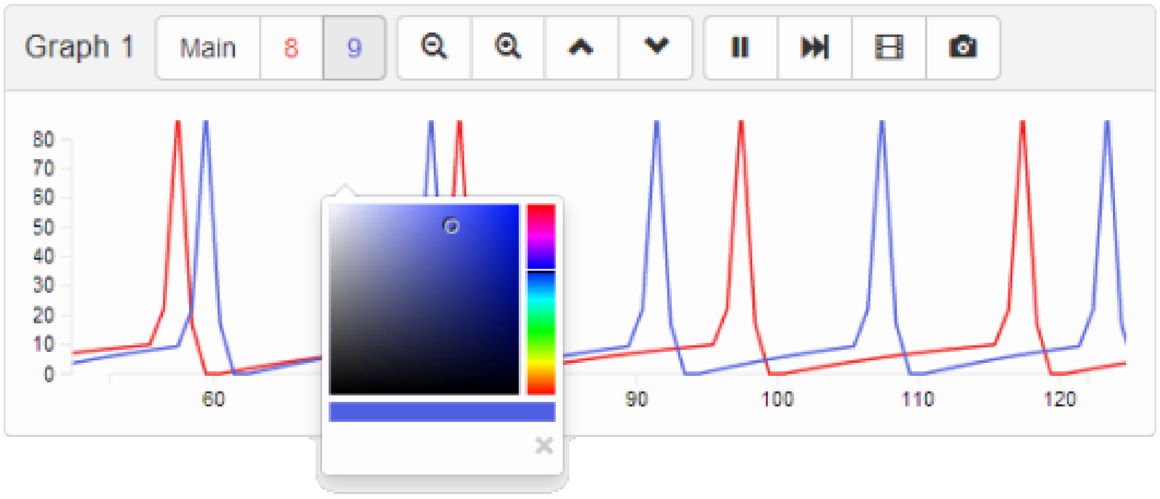


Figure 4.8: An example of multiple cell channels and color changes to facilitate easy interpretation.

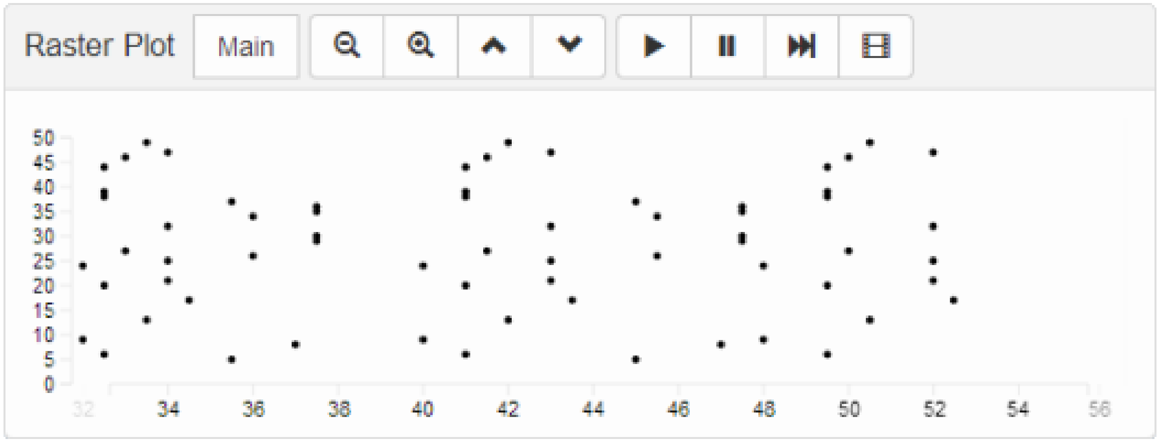


Figure 4.9: An example of the raster plot reporting window.

drag the position slider as necessary to view data as they wish.

UC06 Graph Recording

A user has the ability to record a frame of the graph and save to file. Upon the recording options is an animated GIF, a static GIF, or an SVG.

4.4 Conclusion and Future Work

4.4.1 Conclusion

The NCR project follows a fundamental principle: a fluid harmony between the powerful back end brain simulator and the intuitive front end array of tools is essential to the human paradigm of easy to use, easy to share, and easy to understand. NCR allows users to view available brain models with ease and intuitively view a graphical abstraction of the simulation output. Combined, these various components form a product which encourage an easy workflow of user control when interacting with something as complex as the NCS.

4.4.2 Future Work

Future work for the NCR project includes the addition of non-core features; for example, the implementation of a note taking tool to attach notes or comments to models in the database, or a discussion forum for users so as to expand their collaborative potential with other users across the world. Other ideas include the ability to copy brain models from other existing databases, and the flexibility to accept different types of models that may not currently be compatible with the database.

Other types of future work include the migration onto other platforms such as mobile devices like smart phones or tablets. Adapting the application to table type technology or other touch screen devices may include utilizing the full potential of a touchscreen interface; for example, allowing the user to pinch-zoom a graph within the reports tab, or to swipe through a listing of brain models pulled from the database instead of clicking through a pagination scheme.

Chapter 5

Hodgkin-Huxley Neuron Model, Simulation Building Blocks, and Python Interface

5.1 Hodgkin-Huxley Neuron Model

There is always a trade-off between biological accuracy and execution time when it comes to brain simulations. NCS6 had built-in IZH and LIF neuron models. HH, which is more biologically accurate model, has been added in NCS7 as shown in Figure 5.1. HH model is based on a set of four differential equations. The circuit representation is shown in Figure 5.2. Equations used in NCS are based on the equations provided by Alan Hodgkin and Andrew Huxley [29] from their experiment of the giant axon of a squid. However, channel's current contributions are based on a generic equation [26]. In addition to the three built-in neuron models, users can create their own plugin interface for different neuron models.

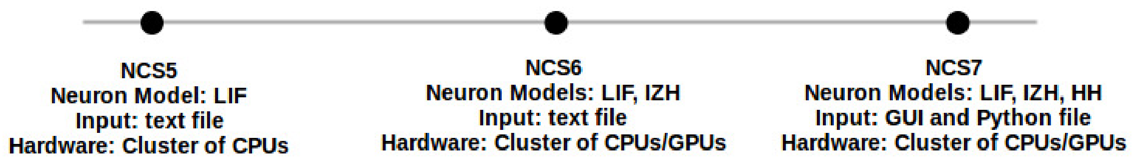

NCS5
Neuron Model: LIF
Input: text file
Hardware: Cluster of CPUs
NCS6
Neuron Models: LIF, IZH
Input: text file
Hardware: Cluster of CPUs/GPUs
NCS7
Neuron Models: LIF, IZH, HH
Input: GUI and Python file
Hardware: Cluster of CPUs/GPUs

Figure 5.1: NCS Timeline

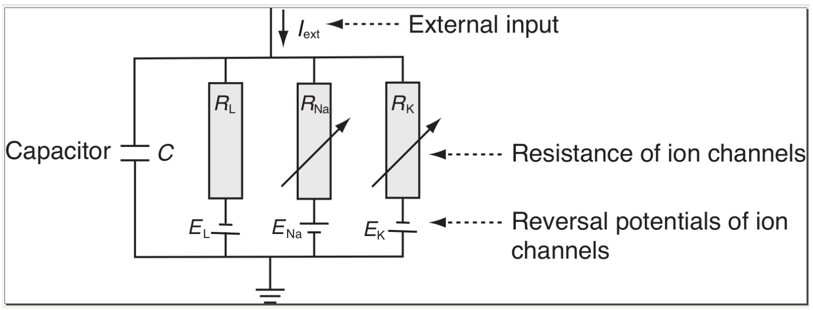


Figure 5.2: Hodgkin-Huxley circuit representation [52]. Membrane potential is measured on capacitor C.

5.2 Simulation Building Blocks

Building blocks for simulation in NCS are shown in Figure 5.3. They are channels, neurons, synapses, groups and aliases, stimulus, report, and simulation info. There are 3 types of built-in channels in NCS: one for HH neuron model and two for LIF neuron model. HH neuron model has generic voltage-gated particle channel that can be used to model leakage channel, sodium channel, and potassium channel. LIF neuron model has two types of channels: generic voltage-gated ion channel and Calcium-dependent ion channel. Calcium-dependent channel is voltage independent. IZH neurons do not have any channels since simulation is done with 6 variables. Also, users can create plugin for additional channel if needed. Each neuron group contains one type of neuron but it can have many neurons. For example, there might be 500 IZH neurons in one group or 300 HH neurons, and so on. If more than one type of neuron is needed in a group, neuron aliases can be used. For example, group3 = sim.addNeuronAlias(“group3”,[group1, group2]). So, if group1 has IZH neurons and group2 had HH neurons, group3 has both IZH and HH neurons. There are 2 types of built-in synapses: flat and NCS (a.k.a LIF). Each connection group includes following information: presynaptic group, postsynaptic group, synapse/connection type, and probability of connection. If more than one connection is needed, connection alias can be used similar to the neuron alias.

After specifying information about channels, neurons, and connections, initialize

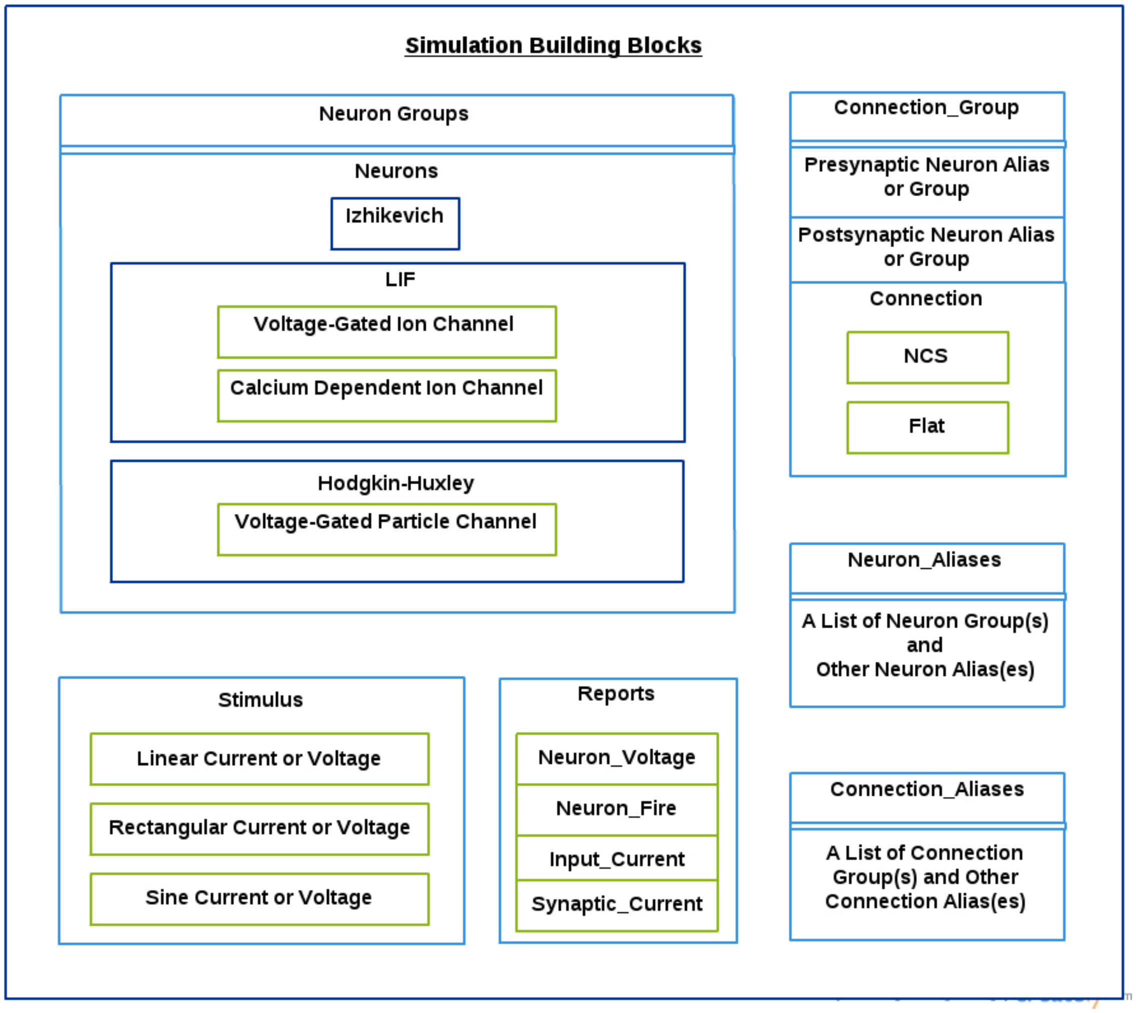

Simulation Building Blocks
Neuron Groups
Neurons
Izhikevich
LIF
Voltage-Gated Ion Channel
Calcium Dependent Ion Channel
Hodgkin-Huxley
Voltage-Gated Particle Channel
Connection_Group
Presynaptic Neuron Alias or Group
Postsynaptic Neuron Alias or Group
Connection
NCS
Flat
Neuron_Aliases
A List of Neuron Group(s) and Other Neuron Alias(es)
Stimulus
Linear Current or Voltage
Rectangular Current or Voltage
Sine Current or Voltage
Reports
Neuron_Voltage
Neuron_Fire
Input_Current
Synaptic_Current
Connection_Aliases
A List of Connection Group(s) and Other Connection Alias(es)

Figure 5.3: Simulation building blocks for models in NCS

the simulation. Once the simulation is initialized, specify stimulus (input), reports (output), and overall simulation information. There are six types of stimuli available: linear current, linear voltage, rectangular current, rectangular voltage, sine current and sine voltage. Also, 4 types of reports are provided: neuron voltage, neuron fire, input current, and synaptic current. Stimulus is given to neuron groups and report data can be collected from neuron group or connection group. Last step is to specify the simulation information such as the duration or number of steps for the simulation.

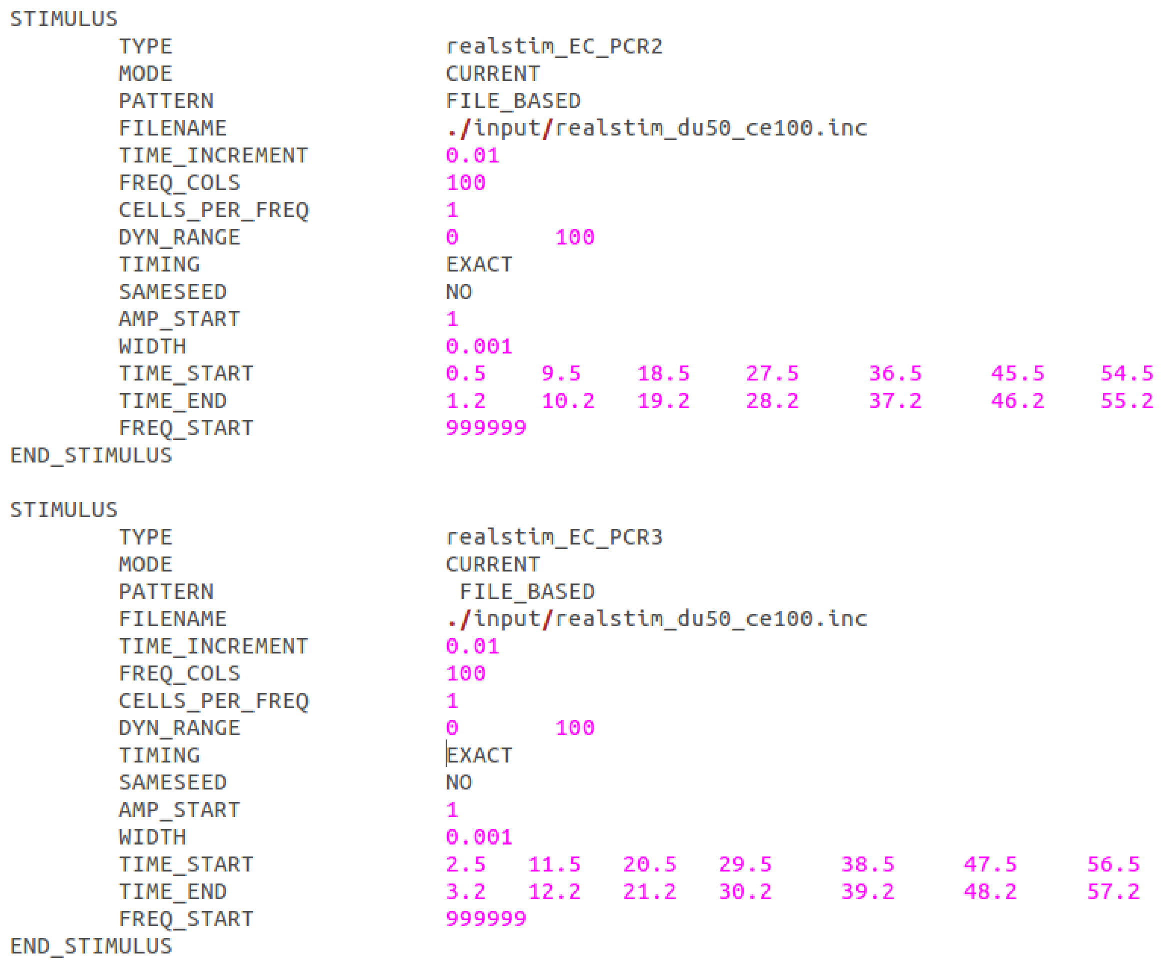
5.3 Python Interface

The backend of NCS is written using CUDA and C++11. Prior to NCS7, text files were used as an input to the simulator as shown in Figure 5.4. The advantage of the text file was that it required minimum programming skills. However, they didn't allow loops and tended to be large. Due to this, it was inconvenient to write, debug, and maintain code. NCS7 introduced both a Graphical User Interface [3] and a Python interface so scientists with varying programming skills can utilize NCS. Python was chosen because it is relatively easy to learn and it has been widely used in the field by various simulators such as NEURON [14], NEST [16], and BRIAN [11].

Following functions are provided for Python interface in NCS7. For information about function parameters and their descriptions, please refer to ncs.io/docs/.

- `addNeuron`
- `addNeuronGroup`
- `addNeuronAlias`
- `addSynapse`
- `addSynapseGroup`
- `addSynapseAlias`
- `addStimulus`

- `addReport`
- `init`
- `run`



```
STIMULUS
        TYPE                        realstim_EC_PCR2
        MODE                        CURRENT
        PATTERN                     FILE_BASED
        FILENAME                    ./input/realstim_du50_ce100.inc
        TIME_INCREMENT              0.01
        FREQ_COLS                   100
        CELLS_PER_FREQ              1
        DYN_RANGE                   0       100
        TIMING                      EXACT
        SAMESEED                    NO
        AMP_START                   1
        WIDTH                       0.001
        TIME_START                  0.5    9.5    18.5    27.5     36.5     45.5    54.5
        TIME_END                    1.2    10.2   19.2    28.2     37.2     46.2    55.2
        FREQ_START                  999999
END_STIMULUS

STIMULUS
        TYPE                        realstim_EC_PCR3
        MODE                        CURRENT
        PATTERN                      FILE_BASED
        FILENAME                    ./input/realstim_du50_ce100.inc
        TIME_INCREMENT              0.01
        FREQ_COLS                   100
        CELLS_PER_FREQ              1
        DYN_RANGE                   0       100
        TIMING                      EXACT
        SAMESEED                    NO
        AMP_START                   1
        WIDTH                       0.001
        TIME_START                  2.5   11.5   20.5   29.5     38.5     47.5     56.5
        TIME_END                    3.2   12.2   21.2   30.2     39.2     48.2     57.2
        FREQ_START                  999999
END_STIMULUS
```

Figure 5.4: Input file for NCS6 and prior versions

The following example is a model of regular spiking IZH neurons with synapse. It is written using Python interface and its output is shown in Figure 5.5

```
#!/usr/bin/python
import os, sys
ncs_lib_path = ('../../../') #Path to ncs.py
sys.path.append(ncs_lib_path)
import ncs

def run(argv):
    #ncs.simulation() is required
    sim = ncs.Simulation()

    #start writing model - biology information
    #addNeuron function
    #Parameters for addNeuron function:
    #     1.  A neuron name (string)
    #     2.  A neuron type (string)
    #         izhikevich, ncs, or hh
    #     3.  A map for parameter names to their values(Generators)
    #         Generators can be exact, Normal, uniform
    #         exact example :  "a":  0.02
    #         uniform example:  "a":  ncs.Uniform(min, max)
    #         normal example:  "a":  ncs.Normal(mean, standard deviation)
    # Example of addNeuron with regular_spiking izhikevich neuron
    regular_spiking_parameters = sim.addNeuron("regular_spiking","izhikevich",
                                                    {
                                                    "a":  0.02,
                                                    "b":  0.2,
                                                    "c":  -65.0,
                                                    "d":  8.0,
                                                    "u":  -12.0,
                                                    "v":  -60.0,
                                                    "threshold":  30,
                                                    })
    #addSynapse function
    #Parameters for addSynapse function
    #     1.  A synapse name (string)
    #     2.  A synapse type (string)
    #         ncs or flat
    #     3.  A map for parameter names to their values (Generators)
    ncs_synapse_parameters = sim.addSynapse("flat_synapse","ncs",
                                    {
                                    "utilization":  ncs.Normal(0.5,0.05),
                                    "redistribution":  1.0,
                                "last_prefire_time":  0.0,
                                "last_postfire_time":  0.0,
                                "tau_facilitation":  0.001,
                                "tau_depression":  0.001,
                                "tau_ltp":  0.015,
                                "tau_ltd":  0.03,
                                "A_ltp_minimum":  0.003,
                                "A_ltd_minimum":  0.003,
                                "max_conductance":  0.3,
                                "reversal_potential":0.0,
                                "tau_postsynaptic_conductance":  0.02,
                                "psg_waveform_duration":  0.05,
                                "delay":  1,
                                })

#addNeuronGroup function
#Parameters for addNeuronGroup function:
#     1.  A name of the group (string)
#     2.  Number of cells (integer)
#     3.  Neuron parameters
#     4.  Geometry generator (optional)
group_1=sim.addNeuronGroup("group_1",1,regular_spiking_parameters,None)
group_2=sim.addNeuronGroup("group_2",1,regular_spiking_parameters,None)

#addSynapseGroup function
#Parameters for addSynapseGroup function:
#     1.  A name of the connection
#     2.  Presynaptic NeuronAlias or NeuronGroup
#     3.  Postsynaptic NeuronAlias or NeuronGroup
#     4.  Probability of connection
#     5.  Parameters for synapse
connection1=sim.addSynapseGroup("connection1","group_1",
                                "group_2",1,"flat_synapse")

#initialize simulation
if not sim.init(argv):
    print "failed to initialize simulation."
    return

#addStimulus function
#parameters for addStimulus function:
#     1.  A stimulus type (string)
#         rectangular_current, rectangular_voltage, linear_current,
#         linear_voltage, sine_current, or sine_voltage
#     2.  A map from parameter names (strings) to their values(Generators)
#         Parameter names are amplitude, starting_amplitude,
#         ending_amplitude, delay, current, amplitude_scale,
    #         time_scale, phase, amplitude_shift, etc.  based on the
    #         stimulus type
    #     3.  A set of target neuron groups
    #     4.  probability of a neuron receiving input
    #     5.  start time for stimulus (seconds)
    #     6.  end time for stimulus (seconds)
    sim.addStimulus("rectangular_current","amplitude":10,group_1,1,0.01,1.0)

    #addReport function
    #Parameters for addReport function:
    #     1.  A set of neuron group or a set of synapse group to report on
    #     2.  A target type:  "neuron" or "synapses"
    #     3.  Type of report:  synaptic_current, neuron_voltage, neuron_fire,
    #         input_current, etc.
    #     4.  Probability (the percentage of elements to report on)
    voltage_report_1=sim.addReport([group_1,group_2],"neuron",
                                "neuron_voltage",1.0,0.0,1.0)
    #An example of a report to file
    voltage_report_1.toAsciiFile("reg_voltage_report.txt")

    #duration (in seconds) - each time step is 1 ms
    sim.run(duration=1.0)
    return

if __name__ == "__main__":
    run(sys.argv)
```

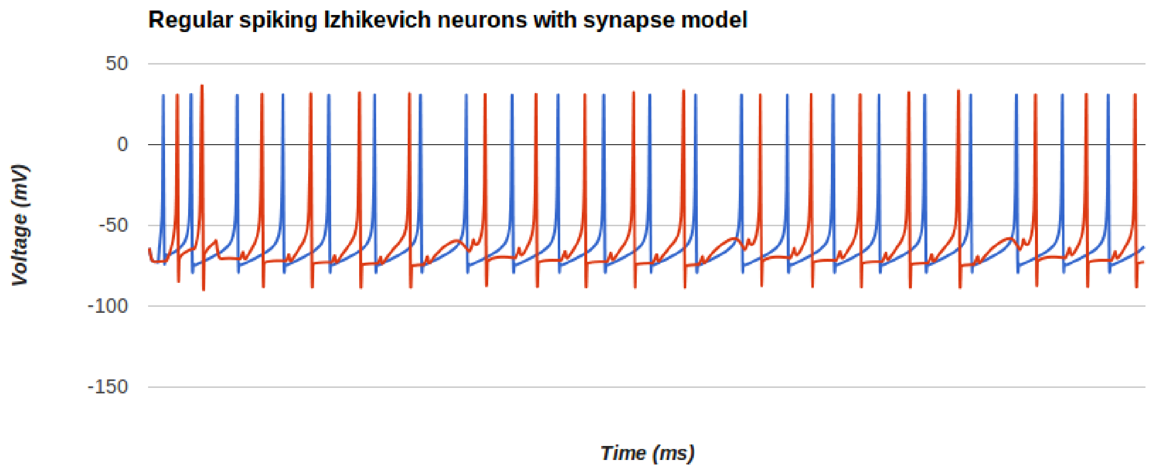


Figure 5.5: 1 second (1000 steps) simulation of regular spiking Izhikevich neurons with synapse. Blue line is group1 and red line is group2.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

NCS is designed to run on the heterogeneous cluster of CPUs and NVIDIA GPUs for large scale simulation. Prior to NCS6, NCS only ran on clusters of CPUs and had one in-built neuron model, Leaky Integrate-and-Fire. Currently, NCS runs on CPUs and GPUs, and it has three in-built neuron models: Izhikevich, Leaky Integrate-and-Fire, and Hodgkin-Huxley that are widely used in the field. Also, users have an option to create their own plugin interface for different neuron models. In addition to the neuron models, a Python interface and a Graphical User Interface has been added to NCS to encourage researchers with varying programming skills to utilize NCS. Furthermore, the Report interface will allow users to view output by the simulator in real time and the MongoDB database will allow researchers to share their models and collaborate with others in the community. To download software and to view documentation, please visit NCS's website: ncs.io.

6.2 Future Work

NCS is the first simulator to propose and implement virtual neurorobotics framework [20]. In the past, simulator interacted with Webots environment [22]. Brainlab would like to have a WebGL based robotic environment that can be used by researchers for free to create and test their models.

In future, we would like to add 3D geometry information for each neuron model types. Geometry information would allow us to take distance into account for synapses rather than delay only. Additionally, that information can be used to visualize the model and neuron firing in real-time with WebGL application that is currently in development.

Bibliography

[1] Arash Ahmadi and Hamid Soleimani. A GPU-based simulation of multilayer spiking neural networks. In *Proceedings of the 19th Iranian conference on Electrical Engineering (ICEE)*, pages 1–5, Tehran, Iran, May 2011.

[2] Edson Almachar, Alexander Falconi, Katie Gilgen, Devyani Tanna, Nathan M. Jordan, Roger V. Hoang, Sergiu M. Dascalu, Laurence C. Jayet Bray, and Jr. Frederick C. Harris. NeoCortical Reporsitory and Reports: Database and Reporsitory for NCS. In *Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE-2014)*, 2014. Submitted.

[3] Jakub Berlinski, Cameron Rowe, Daniel M. Chavez, Nathan M. Jordan, Devyani Tanna, Roger V. Hoang, Sergiu M. Dascalu, Laurence C. Jayet Bray, and Frederick C. Harris, Jr. NeoCortical Builder: A Web Based Front End for NCS. In *Proceedings of the 27th International Conference on Computer Applications in Industry and Engineering (CAINE-2014)*, 2014. Submitted.

[4] Fabrice Bernhard and Renaud Keriven. Spiking neurons on GPUs. In *Computational Science–ICCS 2006*, pages 236–243. Springer, 2006.

[5] Mohammad A. Bhuiyan, Vivek K. Pallipuram, and Melissa C. Smith. Acceleration of spiking neural networks in emerging multi-core and GPU architectures. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8., Atlanta, Georgia, April 2010.

[6] Mike Bostock. D3.js: Data-driven documents. `http://d3js.org/`, 2014. (Retrieved May 19, 2014).

[7] James M. Bower and David Beeman. *The Book of GENESIS*. Second edition, 2003. Chapter 21: Large-Scale Simulation Using Parallel GENESIS.

[8] James M. Bower and David Beeman. GENESIS 2.3. Available at `http://www.genesis-sim.org/GENESIS/`, 2012. (Retrieved September 24, 2012).

[9] James M. Bower and David Beeman. GENESIS 3. Available at `http://www.genesis-sim.org/`, 2012. (Retrieved September 24, 2012).

[10] Laurence C Jayet Bray, Gareth B Ferneyhough, Emily R Barker, Corey M Thibeault, and Frederick C Harris Jr. Reward-based learning for virtual neurorobotics through emotional speech processing. *Frontiers in neurorobotics*, 7, 2013.

[11] Romain Brette and Dan F.M. Goodman. Brian: The Brian spiking neural network simulator. Available at `http://briansimulator.org/`, 2012. (Retrieved October 12, 2012).

[12] Romain Brette and Dan F.M. Goodman. Simulating spiking neural networks on gpu. *Network: Computation in Neural Systems*, 23(4):167–182, 2012.

[13] Justin E. Cardoza, Alexander K. Jones, Denver J. Liu, Roger V. Hoang, Devyani Tanna, Laurence C. Jayet Bray, Sergiu M. Dascalu, and Frederick C. Harris, Jr. Design and Implementation of a Graphical Visualization Tool for NCS. In *Proceedings of The 2013 International Conference on Software Engineering and Data Engieering (SEDE 2013)*, 2013.

[14] Nicholas T. Carnevale, Michael L. Hines, and John W. MOORE. NEURON for empirically-based simulations of neurons and networks of neurons. Available at `http://www.neuron.yale.edu/neuron/`, 2012. (Retrieved September 24, 2012).

[15] Diego Contreras. Electrophysiological classes of neocortical neurons. *Neural Networks*, 17(5–6):633–646., 2004.

[16] Markus Diesmann and Jochen M. Eppler. NEST initiative. Available at `http://www.nest-initiative.org/`, 2012. (Retrieved September 24, 2012).

[17] Andres Fernandez, Ruben San Martin, Enric Farguell, and Giovanni Egidio Pazienza. Cellular neural networks simulation on a parallel graphics processing unit. In *Proceedings of the 11th International Workshop on Cellular Neural Networks and Theire Applications (CNNA)*, pages 208–212., Santiago de Compostela, Spain, July 2008.

[18] Andreas K. Fidjeland, Etienne B. Roesch, Murray P. Shanahan, and Wayne Luk. Nemo: a platform for neural modelling of spiking neurons using GPUs. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 137–144., Boston, MA, July 2009.

[19] Andreas K. Fidjeland and Murray P. Shanahan. Accelerated simulation of spiking neural networks using GPUs. In *Proceedings of the 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8., Barcelona, Spain, July 2010.

[20] Philip H. Goodman, Sermsak Buntha, Quan Zou, and Sergiu M. Dascalu. Virtual neurorobotics (VNR) to accelerate development of plausible neuromorphic brain architectures. *Frontiers in Neurorobotics*, 1(November):1–7., 2007.

[21] Philip H. Goodman, Roger V. Hoang, Laurence C. Jayet Bray, and Frederick C. Harris, Jr. The Neocortical Simulator NCS. Available at `http://www.cse.unr.edu/brain/ncs/`, 2012. (Retrieved September 24, 2012).

[22] Philip H. Goodman, Quan Zou, and Sergiu M. Dascalu. Framework and implications of virtual neurorobotics. *Frontiers in Neuroscience*, 2(1):123–128., 2008.

[23] Google. AngularJS. `https://angularjs.org/`, 2014. (Retrieved May 19, 2014).

[24] Bing Han and Tarek M. Taha. Acceleration of spiking neural network based pattern recognition on nvidia graphics processors. *Applied Optics*, 49(10):B83–B91., July 2010.

[25] Bing Han and Tarek M. Taha. Neuromorphic models on a GPGPU cluster. In *Proceedings of the 2010 International Joint Conference for Neural Networks (IJCNN)*, pages 1–8., Barcelona, Spain, July 2010.

[26] Roger V. Hoang. *An Extensible Component-based Approach to Simulation Systems on Heterogeneous Clusters.* PhD thesis, University of Nevada, Reno, 2014. `http://www.cse.unr.edu/~fredh/papers/thesis/PHD-010-Roger-Hoang/dissertation.pdf` (Retrieved July 24, 2014).

[27] Roger V. Hoang, Devyani Tanna, Laurence C. Jayet Bray, Sergiu M. Dascalu, and Frederick C. Harris, Jr. A Novel CPU/GPU Simulation Environment for Large-Scale Neural Modeling. *Frontiers in Neuroinformatics*, 7, 2013.

[28] Roger V. Hoang, Devyani Tanna, Laurence C. Jayet Bray, Sergiu M. Dascalu, and Frederick C. Harris, Jr. NCS: A novel CPU/GPU Simulation Environment for Large-Scale Biologically-Realistic Neuron Modeling. In *Neural Information Processing Systems (NIPS 2013)*, December 2013.

[29] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.

[30] Jorn Hoffmann, Karim El-Laithy, Frank Gttler, and Martin Bogdan. Simulating biological-inspired spiking neural networks with openCL. In *Proceedings of the 20th international conference on Artificial neural networks: Part I (ICANN)*, Thessaloniki, Greece., September 2010.

[31] Jun Igarashi, Osamu Shouno, Tomoki Fukai, and Hiroshi Tsujino. Real-time simulation of a spiking neural network model of the basal ganglia circuitry using general purpose computing on graphics processing units. *Neural Networks*, 24(9):950–960., November 2011.

[32] Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572., 2003.

[33] Laurence C. Jayet Bray, Devyani Tanna, and Frederick C. Harris, Jr. NCS: A Large-Scale Brain Simulator. In *Neural Information Processing Systems (NIPS 2012)*, December 2012.

[34] Nathan M. Jordan, Kim Perry, Nitish Narala, Laurence C. Jayet Bray, Sergiu M. Dascalu, and Frederick C. Harris, Jr. Design and implementation of an NCS-NeuroML translator. In *Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE 2012)*, Los Angeles, CA., June 2012.

[35] Michele Migliore, C. Cannia, William W. Lytton, Henry Markram, and Michael L. Hines. Parallel network simulations with NEURON. *Journal of Computational Neuroscience*, 21:119–129., 2006.

[36] MongoDB Inc. MongoDB. `http://www.mongodb.com/`, 2014. (Retrieved May 19, 2014).

[37] Jayram M. Nageswaran, Nikil Dutt, Jeffrey L. Krichmar, Alex Nicolau, and Alex Veidenbaum. Efficient simulation of large-scale spiking neural networks using CUDA graphics processors. In *Proceedings of the 2009 International Joint Conference on Neural Networks (IJCNN)*, pages 2145–2152, Atlanta, Georgia., June 2009. IEEE.

[38] Jayram M. Nageswaran, Nikil Dutt, Jeffrey L. Krichmar, Alex Nicolau, and Alex V. Veidenbaum. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks*, 22(5):791–800., August 2009.

[39] Namlook. MongoKit. `http://namlook.github.io/mongokit/`, 2014. (Retrieved May 19, 2014).

[40] Mozilla Developer Network. CSS. `https://developer.mozilla.org/en-US/docs/Web/HTML`, 2014. (Retrieved May 19, 2014).

[41] Mozilla Developer Network. HTML5. `https://developer.mozilla.org/en-US/docs/Web/HTML`, 2014. (Retrieved May 19, 2014).

[42] Mozilla Developer Network. JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript`, 2014. (Retrieved May 19, 2014).

[43] Thomas Nowotny. Flexible neuronal network simulation framework using code generation from NVidia CUDA. *BMC Neuroscience*, 12(Suppl 1.), 2011.

[44] NVIDIA. CUDA 5. Available at `http://www.nvidia.com/object/cuda_home_new.html/`, 2013. (Retrieved August 5, 2013).

[45] Tomaso Poggio, Ulf Knoblich, and Jim Mutch. CNS: a gpu-based framework for simulating cortically-organized networks. Technical report, Massachusetts Institute of Technology, Cambridge, MA., 2010.

[46] Vitaliy Potapov. X-editable. `http://vitalets.github.io/x-editable/`, 2012. (Retrieved May 19, 2014).

[47] Armin Ronacher. Flask. `http://flask.pocoo.org/`, 2014. (Retrieved May 19, 2014).

[48] Ruggero Scorciono. GPGPU implementation of a synaptically optimized, anatomically accurate spiking network simulator. In *Proceedings of the Biomedical Sciences and Engineering Conference (BSEC)*, Oak Ridge, TN., May 2010.

[49] The jQuery Foundation. jQueryUI. `http://jqueryui.com/`, 2014. (Retrieved May 19, 2014).

[50] Corey M. Thibeault, Roger V. Hoang, and Frederick CḢarris, Jr. A novel multi-GPU neural simulator. In *Proceedings of the 3rd Conference on Bioinformatics and Computational Biology (BICoB 2011)*, pages 146–151., New Orleans, LA, March 2011.

[51] Jan-Phillip Tiesel and Anthony S. Maida. Using parallel GPU architecture for simulation of planar i/f networks. In *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, pages 754–759., Atlanta, GA, June 2009.

[52] Thomas P. Trappenberg. *Fundamentals of Computational Neuroscience.* Oxford University Press, USA., Second edition, 2010.

[53] Twitter. Bootstrap. `http://getbootstrap.com/`, 2014. (Retrieved May 19, 2014).

[54] Mingchao Wang, Boyuan Yan, Jingzhen Hu, and Peng Li. Simulation of large neuronal networks with biophysically accurate models on graphics processors. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, San Jose, CA, August 2011.

[55] Dmitri Yudanov, Muhammad Shaaban, Roy Melton, and Leon Reznik. GPU-based simulation of spiking neural networks with real-time performance and high accuracy. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, Barcelona, Spain, July 2010.