University of Nevada, Reno

A Web Based Application for Model Creation and Output Visualization with the NCS Brain Simulator

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

by

Cameron Jason Rowe

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

May, 2016

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

CAMERON J. ROWE

Entitled

A Web Based Application for Model Creation and Output Visualization with the NCS Brain Simulator

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

MASTER OF SCIENCE

Dr. Frederick C. Harris, Jr., Advisor

Dr. Sergiu Dascalu, Committee Member

Dr. Yantao Shen, Graduate School Representative

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

May, 2016

Abstract

UNR's Brain Computation Lab has developed the NeoCortical Simulator (NCS) to simulate brain activity on relatively large models. Initially, NCS inputs could only be configured using convoluted text files or Python scripts. The NCS Daemon was created to act as a service for accessing NCS from remote applications. This thesis presents the NCS web interface, a web application for creating brain models and simulation parameters, as well as visualizing outputs in realtime as dynamic graphs. It also has the ability to view constructed models in 3D. The web interface has many features to work with NCS directly, including the ability to launch simulations, save working models to a database, or export models as JSON files or Python scripts. The web interface is designed to be the front-end for NCS.

Dedication

For my family.

Acknowledgments

I would like to thank Dr. Frederick C. Harris, Jr., Dr. Sergiu Dascalu, and Dr. Yantao Shen for being on my committee, with special thanks to Dr. Harris for giving me the opportunity to do research in the HPCVIS lab and the Brain lab. I would like to thank Christine Johnson for her work on the NCS daemon as well as Matthew VanCompernolle and Daniel Chavez for their initial work and design of the web interface. I would especially like to thank Daniel for keeping me focused and believing that I could finish on time. Finally I would like to thank my parents for providing me the utmost love and support.

This material is based in part upon work supported by the National Science Foundation under grant number(s) IIA-1329469. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Neuroscience, or the study of the nervous system, has existed since the time of the ancient Egyptians, but has become increasingly popular ever since the 1960s. The critical downfall to traditional neuroscience is that to study the nervous system, you have to perform invasive procedures, such as implanting electrodes in specific areas of an animal's brain. Computational neuroscience aims to solve this problem by simulating brain activity using computers rather than experimenting on live animals. Modern brain simulators are constructed using various types of neuron and synapse models which use neuron spiking functions to build a mathematical relationship between an input stimulus and the probability that a neuron voltage will spike.

Many modern brain simulators exist, including the NeoCortical Brain Simulator (NCS). NCS was developed at the University of Nevada, Reno, and allows for a variety of model and simulation parameters to be set [13, 32] . NCS can view live output from a running simulation, including neuron voltages, neuron firings, synapse currents, and synapse voltages. To interface with NCS, users currently have to create text based input files, and view text based result files. NCS has had many components added to it to make interfacing with external applications and creating input specifications easier. These components include PyNCS, a front-end to NCS written in the Python programming language, which allows users with programming experience to create their models and parameters programmatically, and the NCS Daemon, which allows for remote launching of simulations, and remote viewing of simulation outputs. This paper presents the NCS web interface, a web application that provides a graphical user interface for constructing brain models, setting simulation parameters, viewing simulation outputs, and visualizing 3D representations of constructed models to be used with NCS.

This thesis is structured as follows: Chapter 2 presents the underlying neuroscience used, a background of computational neuroscience, the components of the NeoCortical simulator, and a list of libraries and frameworks used in this work. Chapter 3 discusses the design of the application, the software engineering functional and non functional requirements, and the use case modeling for the application. Chapter 4 provides details of the implementation of the web interface, including the four main components, model builder, simulation builder, reports, and visualizer. Chapter 5 summarizes the work presented and provide ideas for future work on NCS, the NCS daemon, and the NCS web interface.

Chapter 2

Background and Related Work

2.1 Neuroscience

2.1.1 Overview

Neuroscience is the study of the nervous system and the cells in which it is composed of, and consists of the brain, spinal cord, and the nerves throughout the body [34]. By looking specifically at these cells, neuroscientists can learn how the nervous system processes information, as well as how the cells react to received information. This section only discusses the basic functional components of the brain to achieve a basic understanding of the underlying components for a brain simulation. The components discussed include neurons, synapse connections, channels, and stimuli.

2.1.2 Neurons and Channels

The human brain is an extremely complex object, containing upwards of 100 billion neurons. A neuron is the most fundamental cell that composes the brain. Neurons can both receive inputs as well as send outputs to neighboring neurons. Input is achieved by receiving some form of stimulus through dendrites which are connected to the soma, the cell body. Output is sent through the axons to the neighboring cells, as shown in Figure 2.1. Output from one neuron is received as input to another neuron through the excitatory postsynaptic potential [23]. Neurons have gated ionic channels, which can open and close given that special conditions are met, such as when a membrane voltage is a certain value, or certain chemical compositions are present, etc. The gated

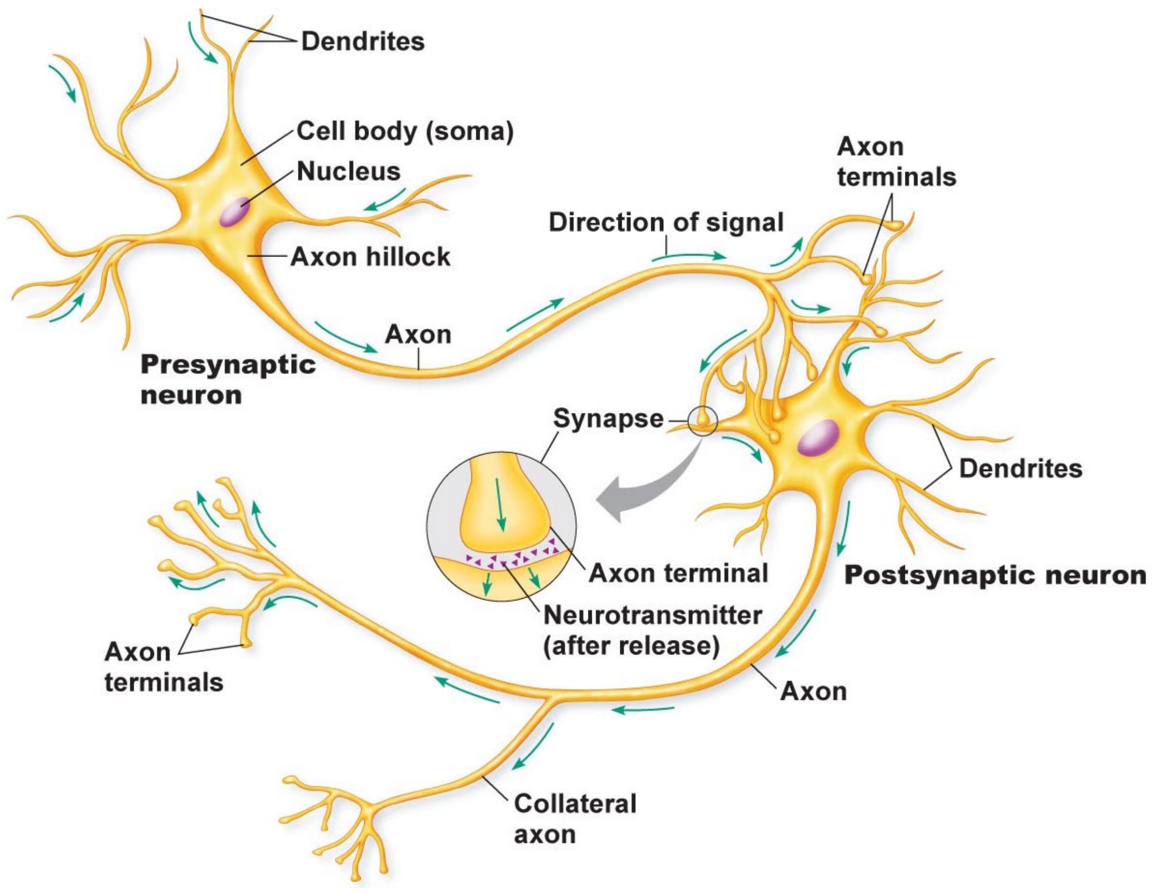


Figure 2.1: The structure of a neuron, from [28].

channels are necessary because the cell membranes of a neuron are impermeable. Ion channels can have different levels of electrical charge both inside and outside the cell membrane, creating a difference is electrical charge known as membrane potential. When the membrane potential exceeds a certain threshold, an action potential signal is transmitted across a synapse to another neuron [14]. When this action potential is triggered, the neuron has been considered to have fired, meaning that a spike in voltage was observed.

2.1.3 Synapses

Given that every neuron in the brain can have up to 10,000 synapses, it has been estimated the human brain has as many as 100 trillion synapses. Synapses are what form the network which allows the brain to store memories and learn new actions. A synapse is a connection between neurons which can transfer the action potential between cells. The brain learns by firing a neuron that is connected to another one repeatedly, leading to long-term potentiation, which is an increase in excitatory postsynaptic potential, which is the strength of the synapse between two neuron over time.

2.1.4 Stimulus

In order to view the behavior of the nervous system, stimuli in the form of an electric current must be applied to target areas of the brain to observe how the system reacts to changes in neuron firing rates and cell membrane potentials. Mathematical functions have been created to describe a relationship between a stimulus and the resulting behavior of a cell. This observation translates to how an electric current will determine the probability of a neuron to spike [23].

2.2 Computational Neuroscience

2.2.1 Overview

Computational neuroscience enables researchers to study brain structures and information-processing properties using neural simulators [27, 29]. Using brain simulators to study neural behaviors has many advantages to traditional methods, in that experiments do not require *in vivo* and *in vitro* methods, often performed on living organisms, as the behaviors can be simulated.

2.2.2 Models

There exist many different types of spiking neuron models used when simulating brain activity. These models often have tradeoffs between biological realism and computational time complexity. The spiking neuron models can be described as mathematical models to determine how action potentials are fired and propagated. This section will discuss the three model types used in the NCS simulator, Izhikevich (IZH) [16], Hodgkin-Huxley (HH) [15], and leaky integrate-and-fire (LIF) [9].

Izhikevich

The IZH neuron model was designed to be as computationally efficient as possible while attempting to retain biological accuracy similar to that of the HH model. IZH has many parameters that can be modified enabling it the simulate fast spiking neuron firing rates, regular spiking rates, and bursting rates. These rates were initially discovered by observing the motor cortex of a rat [18]. The model was proven to be computationally efficient by simulating a mammalian cortex of 10,000 neurons and 1,000,000 synapses [16].

Hodgkin-Huxley

The HH neuron model is one of the most biologically accurate models, however it is also one of the most complex computationally. The model was designed by researchers observing ion concentrations and electrical currents in a giant squid axon in 1952. The HH model describes three currents and how ions flow through the cell membrane via channels. These three currents include a higher potassium current on the outside membrane, a higher sodium current on the inside membrane, and a leak current [15].

Leaky-Integrate-and-Fire

Given that the HH model is so computationally expensive, the LIF neuron model is a more simplistic model that approximates the HH model, focusing primarily on the leak current of neurons. Each channel on a neuron membrane leaks current at a specific rate which is dependent on the membrane capacitance. The model contains a specific threshold, and when the leak current falls below that threshold, the neuron fires and the voltage is reset to the initial value [9].

2.2.3 Simulators

Brain simulators, also referred to as neuron simulators, use neuron models instances to perform biologically realistic simulations on the system using a series of specific time-steps. There are several brain simulators currently used today, including but not limited to BRIAN [10], GENESIS [4], NEST [25], and NEURON [7]. Each simulator listed provides different focus areas and features. Most brain simulators run slowly as computational times for certain neuron models can be computationally expensive as stated previously. Simulators are starting to gain increased performance by using GPUs to aid in computation. GPUs can provide massive increase in performance as they are designed to execute large amounts of instructions in parallel, which is important to brain simulations as the number of cells and synapses can be large [14].

2.3 Neocortical Simulator

2.3.1 Overview

NCS is a brain simulator that can be run on a heterogeneous cluster of machines using both GPUs and CPUs to perform simulation computations. NCS uses NVIDIA

CUDA to run general purpose GPU (GPGPU) code on NVIDIA GPUs, and the message passing interface (MPI) library for distributing computation across machines. Using the distributed cluster along side multiple GPUs, NCS can run large simulations consisting of up to 1,000,000 neurons and 100,000,000 synapses in realtime [13]. NCS runs modified versions of the three neuron models listed above, and can run various different combinations of each. Mathematical descriptions of these models have been done in previous works, so please see [13, 17] for a detailed description.

2.3.2 Simulation Composition

A simulation that is run on NCS uses four core components: neuron models, synapses, stimuli, and reports. Neuron models, synapses, and stimuli have been discussed previously. Reports are specified output values generated from a running simulation. NCS has several optional components that can also be included in a simulation, such as groups, aliases, and calcium dependent channels for LIF neurons and voltage-gated channels for LIF and HH neurons. Each simulation runs with a time-step of one millisecond, where at each time-step the neuron membrane voltage and spike state are reported. Synapses work by checking if a presynaptic neuron triggers an action potential, a current will travel from that neuron through the synapse to a postsynaptic neuron. Stimuli can be applied to groups of cells, and can inject a specified amount of current, or can clamp the membrane voltage at that time-step. When reports are created, they are given a target which can be individual neuron groups. The reports are the only feedback a user would receive from NCS.

2.3.3 PyNCS

NCS is powerful but has had a convoluted interface system which required users to create text files to specify simulation parameters. Attempts have been made to translate inputs from other simulators, but this was not an optimal solution [19]. PyNCS is a front-end to NCS written in Python so users can write simple scripts to configure and launch NCS simulations. PyNCS provides a high level API that allows users to add neurons, synapses, stimuli, neuron groups, neuron aliases, reports, and the ability to launch a simulation. A PyNCS script can be seen in Figure 2.2. More details on the PyNCS API can be seen in [29].

2.3.4 NCS Daemon

The NCS daemon is a centralized service for handling requests made from outside sources to communicate directly with NCS. The daemon is a core component to the web interface as many requests have to be handled by the daemon. The daemon is responsible for many aspects of the web interface, such as handling user accounts, NCS requests, and report streaming. The daemon must store user accounts, which contains user login credentials and stored models. For more information on the daemon capabilities, refer to [18].

2.3.5 Web Components

Previous work has been done to create a web interface to NCS. The interface contained the ability to create brain models, simulation parameters, view output reports, manage a model database, and had a component where a virtual robot could run based on output data from NCS [3, 1, 20]. Most of these components were rough prototypes with limited functionality, and could not work with NCS as no communication was present. There was also a visualization component, however was limited to a desktop application [6].

2.4 Libraries and Frameworks

2.4.1 Overview

The NCS web interface depends on several libraries and frameworks to function. The libraries range from style aids to controlling the structure of the entire program. The libraries used are listed in this section.

```
#!/usr/bin/python

import os, sys
import ncs

def run(argv):

  sim = ncs.Simulation()

  # Izhikevich neuron
  regular_spiking_params = 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
                 })
  group=sim.addNeuronGroup("group_1",1,regular_spiking_params,None)

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

  # Add stimulus
  stim_param = {
        "amplitude":10
         }

  sim.addStimulus("rectangular_current", stim_param, group, 1, 0.1, 1.0)

  # Report neuron voltage
  report=sim.addReport("group", "neuron", "neuron_voltage", 1, 0.0, 1.0)
  report.toAsciiFile("./regular_spiking_izh.txt")

  # Run the simulation for 1 second
  sim.run(duration=1.0)

  return

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

Figure 2.2: An example of a simple PyNCS Python script from [18].

2.4.2 Python

Python is the programming language used to write the web interface's server. Python is a dynamically typed, interpreted, general-purpose and multi-paradigm programming language which allows for fast prototyping while still providing adequate performance. Python's standard library is extremely large and makes it really simple to add functionality to existing programs. It is for this reason, as well as the fact that several web frameworks written in Python already exist, that Python was chosen for the web interface [26].

2.4.3 Javascript

JavaScript is the primary programming language used for creating dynamic web applications. JavaScript is a dynamically typed, interpreted, multi-paradigm programming language that allows for manipulation of elements displayed on a webpage, as well as providing the primary structure for program logic [8].

2.4.4 HTML

HyperText Markup Language, or HTML, is the standard language used for dictating the structure of webpages. The structure is defined by nesting HTML elements inside one another. HTML elements are denoted using tags, with the ending tag including a slash, such as <input> ... </input>. The structure helps determine how styles, positions, and sizes of elements on the page are relative to each other [8].

2.4.5 CSS

Cascading Style Sheets, or CSS, is the standard language used to provide style to a webpage. CSS allows you to define specific style parameters on a class level, or for an individual object id. These styles are then linked in to the webpage through the use of HTML class and identifier attributes [8].

2.4.6 LESS

LESS is a stylesheet language that is compiled to CSS, but allows for additional functionality. LESS provides several features on top of CSS including variables, functions, operators, and more. All styles for the web interface are written in LESS as the additional functionality makes it easier to write good styles, and increases maintainability of the source code [31].

2.4.7 AngularJS

Overview

AngularJS, often referred to as Angular, is an open-source web application framework for creating single page web applications. It provides a client-side Model-View-Controller (MVC) for dynamic manipulation of webpage components based on specific program logic. Angular allows an application to be split into separate controllers, that each get their own scope where each scope keeps program logic restricted to specific areas of the webpage. This allows for the application to maintain a more modular design for ease of development [11].

MVC

Model-View-Controller is a programming design pattern where data is decoupled from the visualization of that data. The first component, model, is where data and program logic are represented. The next component, view, is what displays the data. Finally, the controller, is what allows interaction between the view and the model data.

Directives and Data Binding

Angular provides methods for directly modifying html elements using directives that are placed inside the HTML. Angular directives have the signature ng–[directive] where [directive] can perform specific operations. These directives include repeating elements, showing elements when a expression is true, and many more. This allows for easily creating dynamic pages that can change based on program logic. Angular also

```
<div ng-controller="Simple">
  <button type="button" ng-click="addRandomNumber()">
    Add Random Number
  </button>

  <div class="row" ng-repeat="number in randomNumbers">
    <h3>Random Number: {{number}}</h3>
  </div>
</div>

<script>
  app.controller("Simple", ["$scope", function($scope) {
    $scope.randomNumbers = [];

    $scope.addRandomNumber = function() {
      $scope.randomNumbers.push(Math.random());
    };
  }]);
</script>
```

Figure 2.3: An example of AngularJS Directives.

allows inputs to be directly bound to a specific model property through the use of data bindings. An example of a few angular directives can be seen in Figure 2.3.

2.4.8 Bootstrap

Bootstrap is a front-end library for improving the look and feel of web applications. Bootsrap allows for responsive layouts, meaning that the look of an application will dynamically change and still work on mobile devices. Bootstrap also improves the look and feel of common elements like buttons as well as adding special styles for components like panel containers [30].

2.4.9 ThreeJS

ThreeJS is library that provides easy access to WebGL and 3D rendering. WebGL is a subset of OpenGL that allows 3D rendering for web applications. ThreeJS contains many common geometric shapes and can be considered a full 3D rendering engine. ThreeJS abstracts away writing custom GLSL shaders, although custom shaders are still supported. In its place, ThreeJS uses material classes for styling objects with things like colors or textures, while mesh classes are used for representing geometric shapes [33].

2.4.10 Highstocks

Highstocks is a JavaScript library for creating dynamic charts and graphs of supplied data. Highstocks is a derivation of the library Highcharts; Hicharts is similar to Highstocks, however it is intended for static data or for viewing small slices of dynamic data. Highstocks builds on this by providing methods for being able to scroll through large data sets, as well as providing the ability to "zoom in" on specific sections of the data. Highstocks as well as Highcharts provide methods for exporting the current view of the graph as an image for download [12].

2.4.11 NodeJS

Overview

NodeJS is a server side runtime environment for handling build tasks geared towards web development. The web interface uses four common components of NodeJS, which are described below [22].

NPM

Node Package Manager (NPM) is a tool for managing server side packages when building a web application. NPM has the ability to install new packages from a configuration file, update installed packages to newer versions, and uninstall packages no longer needed. The three components listed below, Bower, WebPack, and Gulp, were all installed using NPM.

Bower

Bower handles the installation of libraries to be used in the client-side web application. This includes AngularJS, Bootstrap, Highstocks, and other libraries used in this project. It has similar functionality to NPM, in that it installs, maintains, and can remove packages that are being used by the application.

WebPack

WebPack enables multiple JavaScript files to be combined into one large JavaScript file for use on the client. This has several benefits, including increased performance as the browser does not have to make several fetch requests for each file, and makes the code more modular as different sections can be split into different files. WebPack also has the added benefit of applying local scopes to each file, meaning that if one file needs a component of another file, the second file must be included using the `require("file.js")` command. Again this provides more modularity for a more robust code base.

Gulp

Gulp is a build tool that can run subprocesses on triggered events. For instance, Gulp can watch for file changes in a directory, and perform an action when a file changes. This can be used to merge all of the HTML into a single file when any of the HTML files change, compile the LESS code into CSS code when the LESS files are changed, and call WebPack on the JavaScript files when they are changed. Gulp can also copy files to the target build directory for any of these actions. Gulp is also in charge of launching the web interface's Python server on first execution.

2.4.12 Flask

Overview

Flask is a micro webframework written in Python for creating a RESTful web server. Flask defines the route a URL can take, and specific actions can be executed when a route is passed to the server. An example of this can be seen in Figure 2.4 [2].

REST

Representational State Transfer (REST) is a design pattern for communication between a client and a server. For web applications, URLs, or Uniform Resource Locations, are commonly used to identify a REST resource. In the case of the web

```
from flask import Flask
app = Flask(__name__, static_url_path='', static_folder='')

@app.route('/')
def main():
    return app.send_static_file('index.html')

if __name__ == '__main__':
    app.run(port=8080)
```

Figure 2.4: An example of a simple Flask application.

interface, a simple form of REST is used where URLs correspond to specific actions the server should execute. Data can be passed using specific URLs with GET or POST commands. Data is sent from the server to the client through GET requests, and data can be passed from the client to the server using POST requests.

Sessions

Sessions are a feature of Flask that allows a specific state of the client to be saved server side while the server connection is active. What this means is, client-side data can be saved through the server periodically or when specific events occur. For instance, the web interface saves the state of the current model every 30 seconds, or whenever a critical action is taken, such as a new cell is added. This ensures that even if the user accidentally refreshes the page, their data is not lost.

2.4.13 RabbitMQ

RabbitMQ is a message queue library that allows for storing data in queues, and forwarding that data when a connection is available. This allows a user to login, launch a simulation, and logout, and still expect the report data to be available. RabbitMQ will allow this to happen by filling a data queue even while the user is offline. The the user returns and logs in, the data will be made available [24].

2.4.14 JSON

JavaScript Object Notation (JSON) is a data representation that is human readable, and can be used to send data across a network because JSON objects can be easily serialized. Given that JSON originated from JavaScript, web applications have native support for JSON objects. Many programming languages also have support for JSON built-in, and if they don't, libraries usually exist to work with JSON objects. JSON is the primary data representation for the web interface, and for the NCS Daemon [5].

2.4.15 MongoDB

MongoDB is a No-SQL database where JSON objects are used as the internal data representation. MongoDB uses a binary format of JSON, called BSON, for performance and storage reasons. MongoDB is used for the flask sessions mentioned earlier, and is used extensively for storing data in the NCS Daemon [21].

Chapter 3

Design

3.1 Overview

The web interface acts as a front end to NCS. The interface is split into four core components: model builder, simulation builder, reports, and visualizer. The brain builder can create new models, modify models by adding cells, synapse connections, channels, and columns, as well as the ability to import and export models. The simulation builder can modify simulation parameters, and can create inputs and outputs to be triggered at certain times with certain probabilities. The reports allow for output to be seen in dynamic graphs displaying the requested output, such as neuron voltage, synapse current, etc. The visualizer provides a 3D representation of the model created using WebGL to render neurons as cubes.

3.2 Web Interface Requirements

3.2.1 Functional Requirements

The functional requirements are based on the interface functionality of the web interface. The list of functional requirements are listed in Table 3.1.

3.2.2 Non-functional Requirements

The non-functional requirements are based on requirements of communicating with the NCS daemon and implementation specifics for meeting the desired behavior of the system. The non-functional requirements are listed in Table 3.2.

Table 3.1: The NCS web interface functional requirements.

Number	Description
FR01	The web interface shall add new users.
FR02	The web interface shall login users given valid user credentials.
FR03	The web interface shall retrieve the personal, lab, and global models for a user from the NCS daemon.
FR04	The web interface shall save personal, lab, and global models.
FR05	The web interface shall allow a user to revert a model to a previous save.
FR06	The web interface shall allow a user to modify a brain model.
FR07	The web interface shall import NCS simulation Python scripts and will be reflected in the interface.
FR08	The web interface shall export brain model and simulation parameters to a JSON file or a NCS simulation Python script.
FR09	The web interface shall configure simulation parameters.
FR10	The web interface shall create input stimulus and output reports.
FR11	The web interface shall display live graphs of report data in real time.
FR12	The web interface shall configure report graphs for thresholds and plot lines.
FR13	The web interface shall display a 3D rendering of a created model.

Table 3.2: The NCS web interface non-functional requirements.

Number	Description
NFR01	The web interface will be written in HTML, LESS, JavaScript, and Python.
NFR02	The web interface will use AngularJS to handle connecting data to interface.
NFR03	The web interface will use Bootstrap for layout and styling.
NFR04	The web interface will use Flask for client-server communication.
NFR05	The web interface will use MongoDB for storing session information.
NFR06	The web interface will use JSON for requests and data passing.
NFR07	The web interface will use RabbitMQ to stream report data from the NCS daemon.
NFR08	The web interface will use Highstocks for rendering report data.
NFR09	The web interface will use ThreeJS for rendering 3D models.

3.3 Use Case Modeling

3.3.1 Overview

To better understand the functionality of the web interface, use cases are provided in this section. These use cases show how the user can interact with the web interface. A diagram of the use cases is shown in Figure 3.1 and detailed use cases are discussed below.

3.3.2 Detailed Use Cases

Register

When a user first visits the website, they will have two options, either login if they already have an account, or register a new account. To register a new account a user must supply a valid email address, a first name, a last name, a password, and a lab id. If the daemon is able to create an account, the user will be returned to the login page to login to their account.

Login

In order for the user to have access to the web application, the user must login to the website upon arrival. If the user provides valid credentials, they will be redirected to the model builder tab of the application. If the user does not supply valid credentials, an error will be displayed telling them they input the wrong values.

View Tab

Once the user is logged in, they have the option to view any of the four tabs. The user can select any tab at any time to change their view.

Create New Model

Once a user is logged in, they will initially have an empty model. The user can simply start to add components such as new cell groups, or previously created configurations.

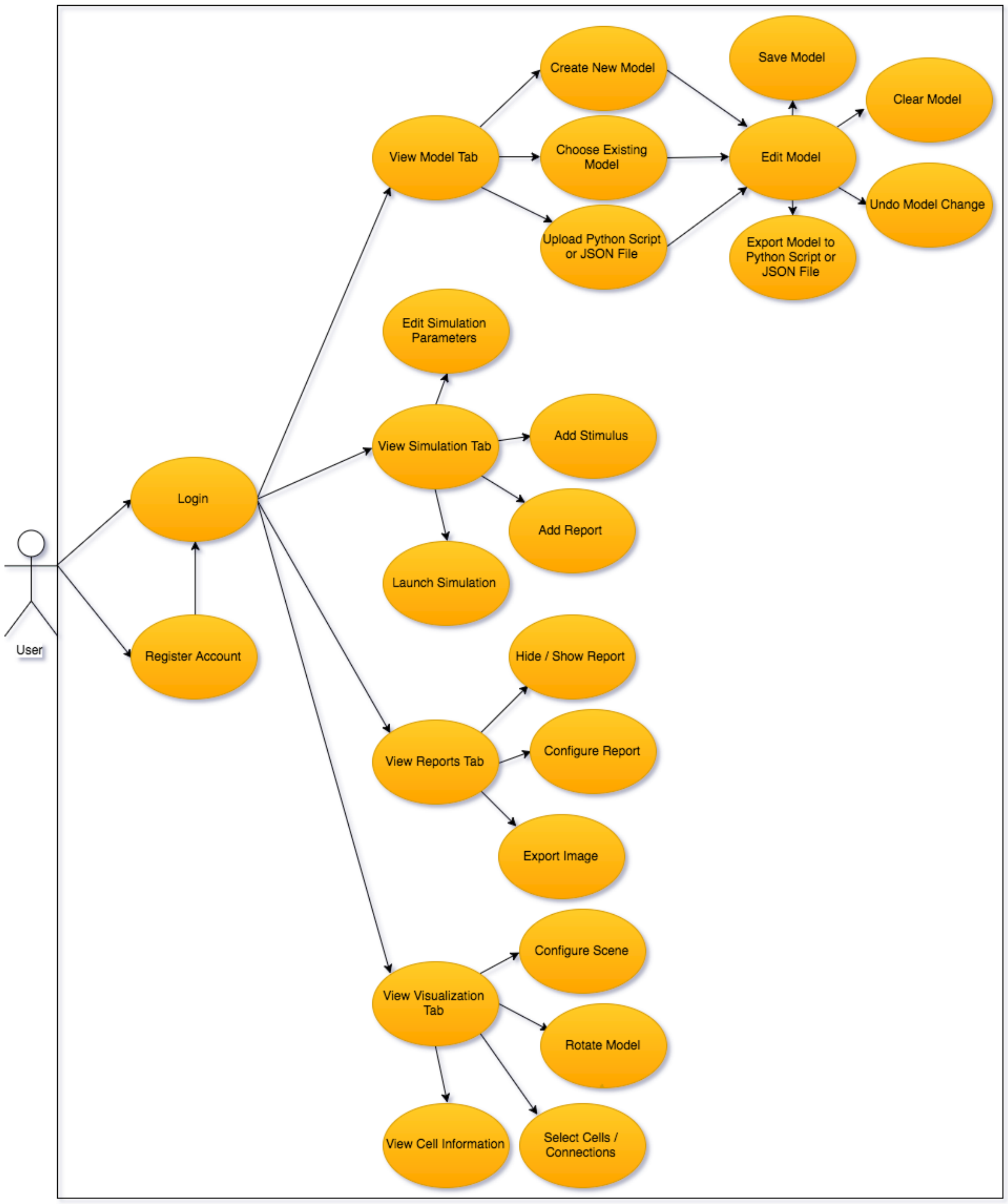


Figure 3.1: A use case diagram of the NCS web interface.

Choose Existing Model

When the user has an empty configuration, they can choose to load an existing model. Once a model is selected, the interface will be updated to display all of the components and parameters for that model.

Upload Python Script or JSON File

The user has the option to upload a JSON file consisting of a previously exported model, or a NCS Python script that has been previously exported or created by hand. Once a file has been uploaded, the interface will be updated to display all of the components and parameters for that model.

Edit Model

Once a user has started a new model or loaded an existing one, they can begin to edit the model. The user can add new components, such as cell groups, cell aliases, or connections. Previously saved components can also be added to the model through the side panel. When a specific component is selected, the user can modify the parameters of that component. For instance if a cell group is selected, the user can modify the number of cells in that group, or modify the parameters for that cell type directly. When at least two cell groups are present, a user can create a connection between them.

Save Model

After a user has made changes to a model, they can choose to save that model. When the user chooses to do so, they will be prompted to name the model, and which database to save it to, personal, lab, or global. Once the model is saved, it can be accessed later through the side panel.

Clear Model

If a user wants to create a new model and discard all current components and parameters, they can choose to clear the model. This will discard the current working model and clear the interface to a state as if the user had just logged in.

Undo Model Change

If a user wants to discard the changes they've made, but not clear out the current model, they can choose to undo the most recent changes. This will revert the current model to the state that it was at when it was last saved.

Export Model to Python Script or JSON File

A user can choose to export the current model either as a JSON file that is the exact representation of the model, or as a Python script that can be executed to instantly launch PyNCS.

Edit Simulation Parameters

When the user views the simulation tab, they can edit several global simulation configuration options. These options include the simulation name and duration.

Add Stimulus

The user can add many stimulus configurations to the simulation. The stimulus consists of a stimulus type, specific parameters for that stimulus type, an input probability, a start and end time, and a list of input targets. Input targets consist of cell groups and synapse connections created in the model builder.

Add Report

The user can add many output reports to the simulation. The report consists of a report type, an output probability, a start time and an end time, as well as a report target and a target type.

Launch Simulation

Once a user is satisfied with the model configuration and simulation parameters, they can launch a simulation on NCS. The user will be prompted if they want to launch the current model or a previously saved model. Once a model is launched the user can view the output in the reports tab.

Hide / Show Report

When the user is viewing reports, they can choose to hide or show specific reports when the minimize / maximize button is pressed.

Configure Report

Each report has a configuration panel that can be expanded to show configuration options for the specific graph being viewed. Configuration options include setting minimum and maximum thresholds, adding additional thresholds to add gradient values to the graph, and plot lines can be added on the horizontal axis to help identify a reference point for the nearby values.

Export Image

At any point, a user can export the data currently being displayed on the graph. The export formats are: a png image, a jpeg image, a svg image, or a PDF.

Configure Scene

When viewing the 3D model, the user can configure the scene being rendered. The configuration options include a toggle for the rendering of neurons, synapses, and columns. and threshold values for firing neurons.

Rotate Model

The user can choose to rotate the 3D model to see it from different angles. The model can be rotated in both the horizontal and vertical directions. This allows for all sides of the model to be seen.

Select Cells and Connections

The user can click and drag the mouse over the scene to select all cells and synapses within the selection box. This allows for specific areas of the model to be seen more easily compared to the parts not selected.

View Cell Information

The user can hover the mouse over specific cells and synapses to bring up a panel with more information about the object being highlighted. If the object is a neuron, the information displayed is the cell name, the input current, and the voltage of the cell. If the object is a synapse connection, the information displayed is the presynaptic connection and the postsynaptic connection.

3.4 Architecture

The web interface consists of a client-server architecture to achieve the functionality of the use cases provided in the previous section. The web interface acts as a client which connects to its own Flask server. The web interface and its server then communicate with the daemon which acts as a proxy server for the web interface. The daemon then communicates directly with NCS and streams results back to the web interface. The system architecture diagram is shown in Figure 3.2. The web interface is composed of four main components which each correspond to their own tab in the application. These include model builder, simulation builder, reports, and visualization. The arrows in the diagram represent actions that can be taken from each entity.

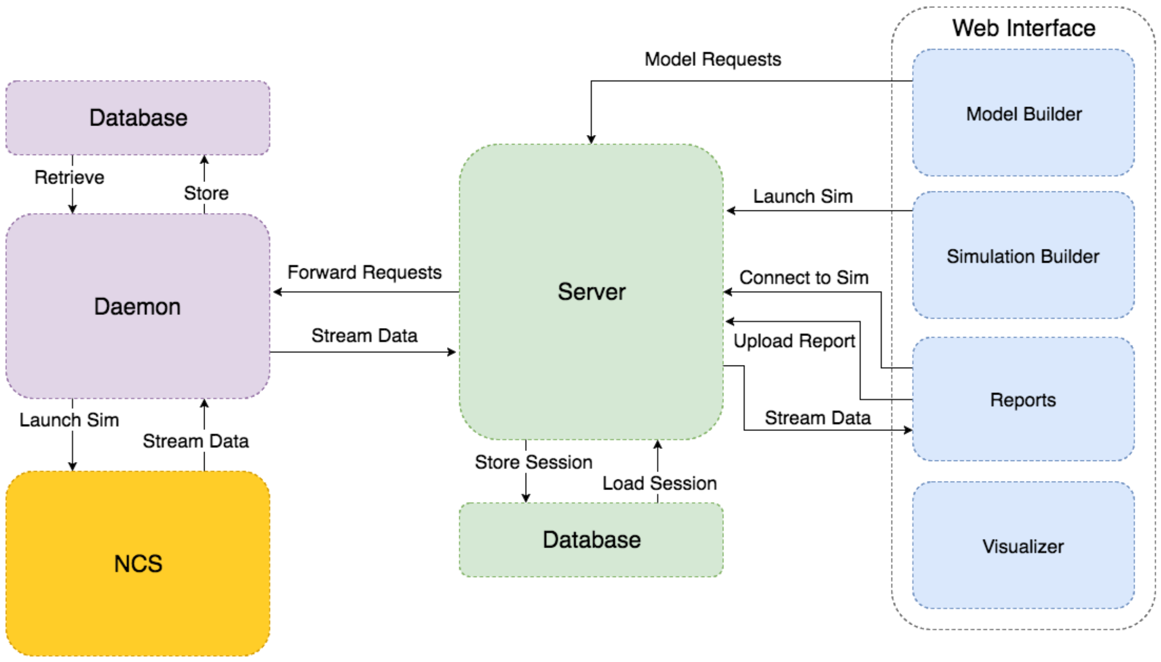


Figure 3.2: An architecture diagram of the NCS web interface.

Chapter 4

Implementation and Results

4.1 Overview

This chapter describes each of the components used in the web interface. It also describes where and how the libraries and frameworks described in Chapter 2 were used to develop the interface.

4.2 Login and User Registration Page

4.2.1 Login

When the user first visits the application webpage, they will be greeted with the login screen. From here the user will be required to input valid credential that were previously registered. When the user enters their credentials, an HTTP request is sent to the server to login. The server then forwards the request to the daemon which will perform an account authentication and return success or failure based on the users information. If the user is authenticated, they will be redirected to the model builder tab and can begin working. If they are rejected, an error message is displayed asking them to try again. If the 'Remember me' option is selected when the user logs in, a cookie will be created that does not expire for four weeks, allowing them to use the site without logging in for that time period. If they do not select this option, the cookie will expire when they close the web browser, prompting them to login next time they return. The login screen can be seen in Figure 4.1.

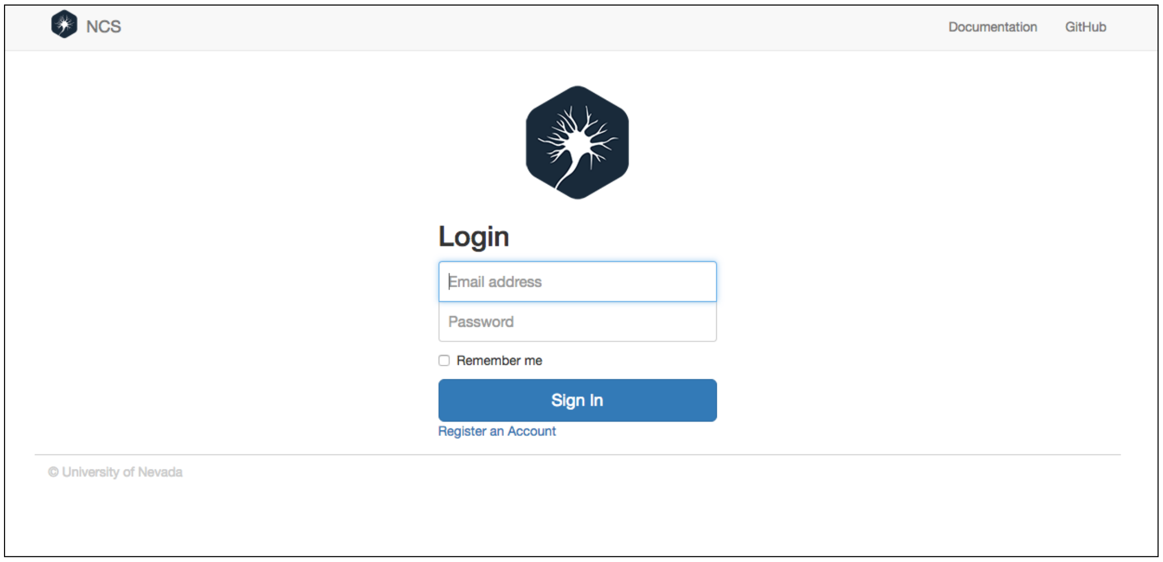


Figure 4.1: The login page to the web application.

4.2.2 User Registration

If the user does not have an account, they can choose to create one by selecting the 'Register an Account' link on the login page. This will display the registration form which has many required components, which can be seen in Figure 4.2. These components include first name, last name, institution, lab id (which is required for storing models in a lab database), email address and password. When the user has entered their information, an HTTP request is made to the server to register the user. The server then forwards this request to the daemon which will add the user to the database.

4.3 Model Builder

4.3.1 Overview

The model builder tab is where users will create and modify their models to be run on NCS. The user can perform many operations on the model, including adding and removing cells, adding and removing synapse connections, adding and removing channels, as well as adding and removing columns. When a specific component is

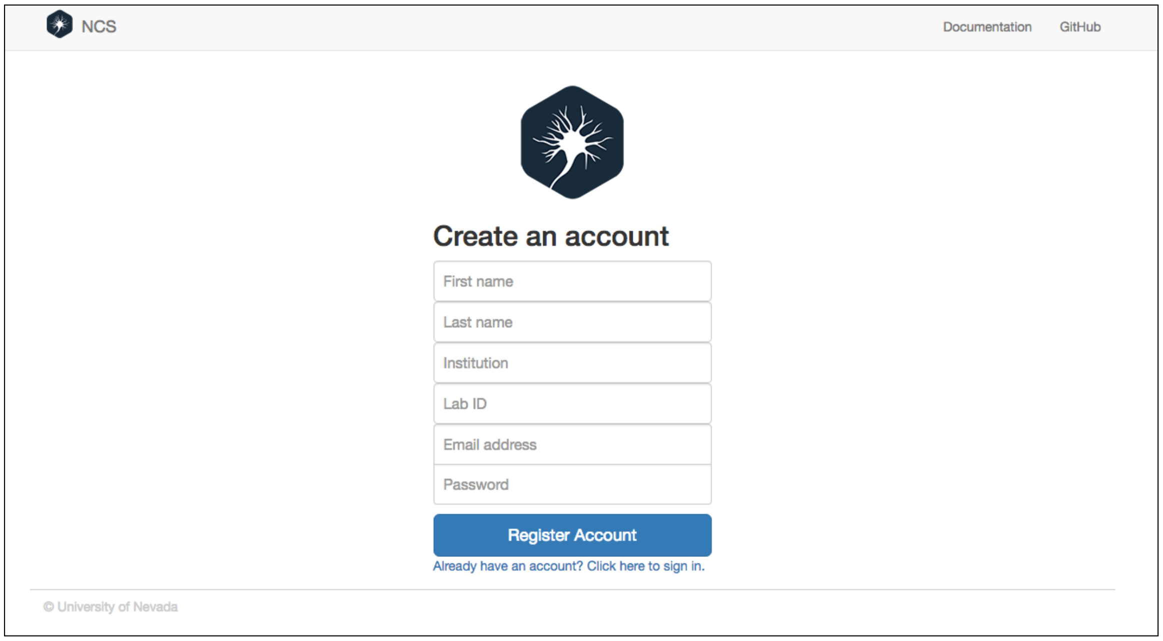


Figure 4.2: The registration page for the web application.

selected, the parameters of the component can be modified.

4.3.2 Layout

The model builder is separated into three main layout components. There is a column in the center of the page that is the current view of the components for the model you are working on. Also, there is a column on the right side of the page that consists of the specific parameters for the selected component. A left drawer can be located on the left side of the screen but is initially hidden until the user toggles it to be viewed. The top of the tab contains several buttons for managing the current model. This includes buttons to clear the current model, undo the current changes, and save the current model. The layout can be seen in Figure 4.3.

4.3.3 Left Drawer

The left drawer is located at the left side of the screen and is hidden until the user decides to toggle it to be viewed, and can be seen in Figure 4.4. The left drawer uses an Angular controller, the *DrawerController* for managing the view. The left

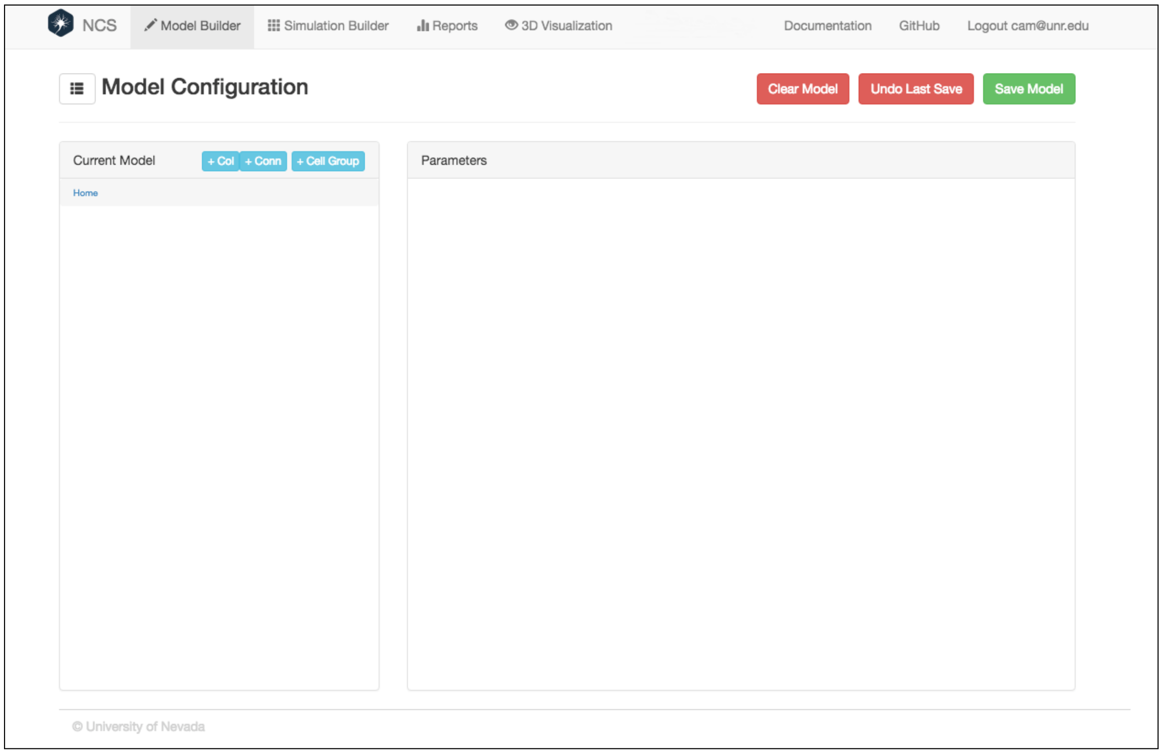


Figure 4.3: The layout of the model builder tab.

drawer contains a filter that can filter the types of models provided. There are three filter options: all, local, and database. The all filter will show all available models, where the local filter will only show example models built-in to the software, and database will only show models loaded from the daemon. Models loaded from the daemon include the users personal collection, their lab collection, and their global model collection. When the page is first loaded, an HTTP request is made to the server to fetch the models from the daemon, and the model list is populated.

Preview Panel

When models are loaded into the drawer, the user has the ability to preview the content of the models that are available. When the preview icon is selected on one of the models, a panel will appear that shows the contents of the model. This preview will behave differently depending on if an individual cell, a cell group, or an entire model is selected. If an individual cell is selected, the specific parameters for that cell

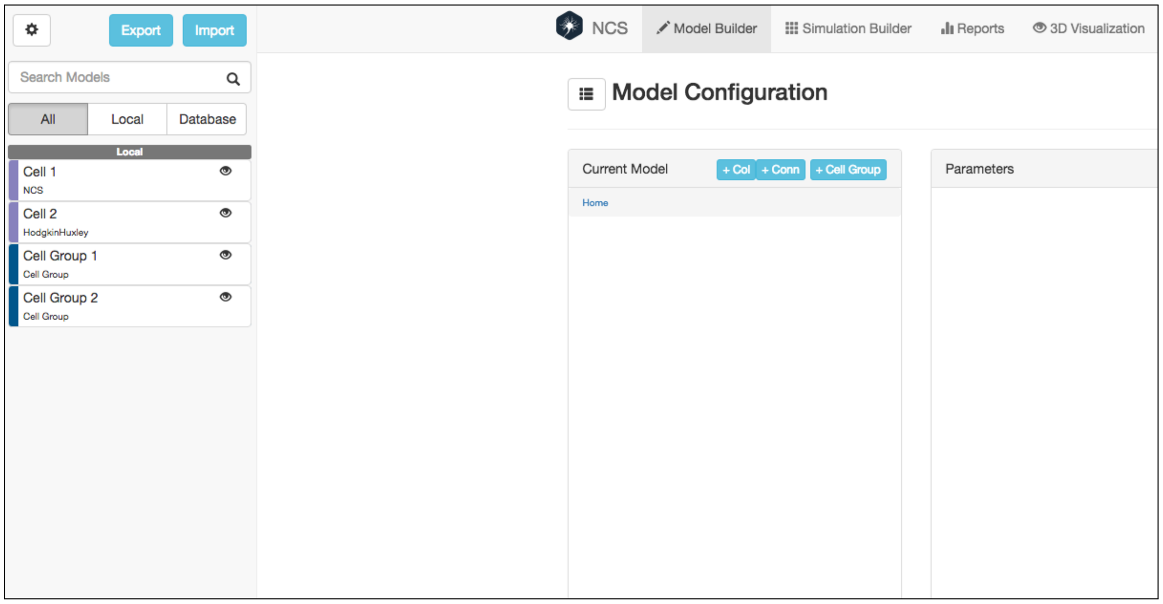
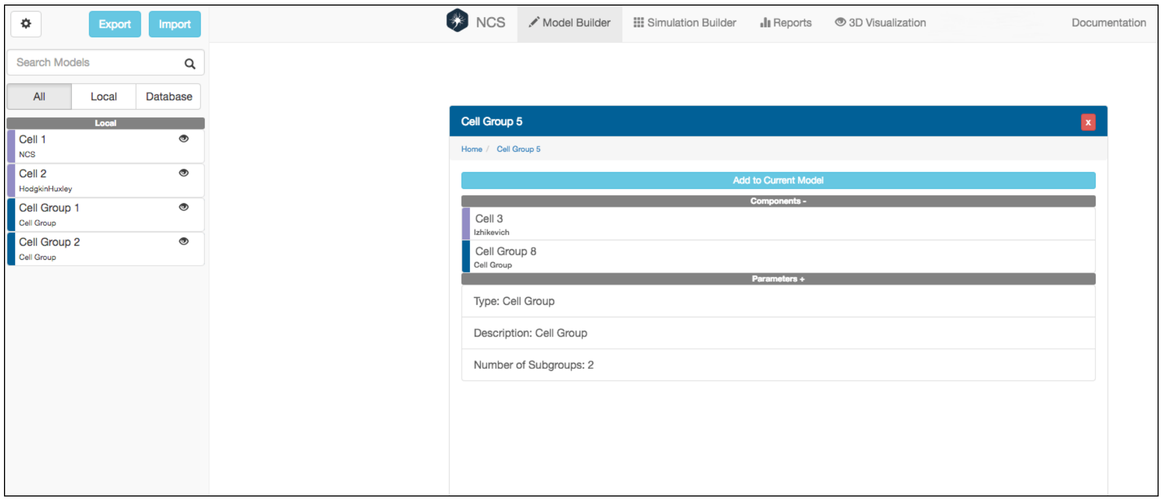


Figure 4.4: The left drawer with built-in models and model populated from database.

are displayed. If a cell group or model is selected, then all components within that group are also shown, with the ability to navigate through and add specific portions of the viewed group, as shown in Figure 4.5. If a user wants to add an item from the drawer to the model, they simple have to double click on it. This will trigger a ng–doubleclick directive which adds the current element into the model.

Import and Export Model

Import Model The web interface provides the ability to upload a pre-existing JSON model or PyNCS script into the browser to be worked on. When the user selects a file to upload, an HTTP request is sent to the server containing the file contents. The server will perform different actions based on the filetype. If the file contains a JSON object, then the JSON is loaded and returned to the client. If the file is a PyNCS script, a request is made to the daemon to convert the PyNCS script to a JSON file. The JSON object is then returned from the daemon and a response is sent to the client containing the JSON object.



Type: Cell Group

Description: Cell Group

Number of Subgroups: 2

Figure 4.5: The preview panel view with the contents of a cell group.

Export Model The model currently being worked on can be exported for later use. The model can either be exported as a JSON file, or as a PyNCS script. An example of an exported model can be seen in Figure 4.6. When an export is triggered, an HTTP request is made to the server containing a JSON object that contains both the current model parameters as well as the current simulation parameters. If JSON was the chosen export format, then the JSON object is simply written to a file and returned to the client as a download. If a PyNCS script is the selected output, a request is sent to the daemon to convert the JSON object to a PyNCS script. The script contents are returned to the server which writes the contents to a file before triggering a download of the script file on the client.

4.3.4 Current Model Panel

The current model panel is the Bootstrap panel that is located on the left of the model builder tab. This panel contains a list of the current contents of the model the user is working on, and is where new components will be added. Cell groups, cell group folders, synapse connections and columns will all be displayed here. New components can be added by selecting any of the '+' buttons located at the top of the panel. All of these buttons will trigger a modal window with specific parameters to be set for

```
{
    "author": "Jane Doe",
    "cellAliases": [],
    "cellGroups": {
      "cellGroups": [
        {
          "name": "IZH Cells",
          "num": 150,
          "type": "Izhikevich",
          "parameters": {
            "a": {
              "type": "exact",
              "value": 0.2
            },
            "b": {
              "type": "exact",
              "value": 0.2
            },
            "c": {
              "type": "exact",
              "value": −65
            },
            "d": {
              "type": "exact",
              "value": 8
            },
            "threshold": {
              "type": "exact",
              "value": 30
            },
            "u": {
              "maxValue": −11,
              "minValue": −15,
              "type": "uniform"
            },
            "v": {
              "maxValue": −55,
              "minValue": −75,
              "type": "uniform"
            }
          }
        }
      ],
    },
    "description": "Regular Spiking",
    "name": "Test Model",
    "synapses": []
  }
```

Figure 4.6: An example of a simple model JSON from [18].

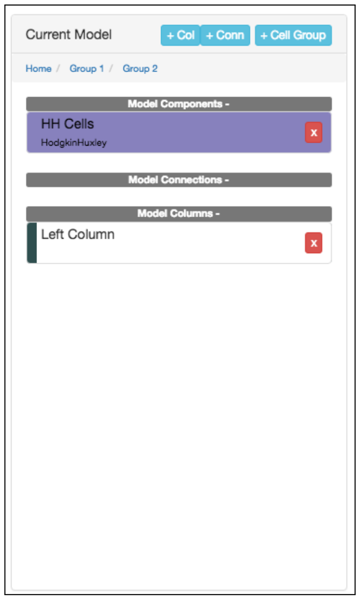


Figure 4.7: The current model panel with breadcrumbs showing components several levels deep.

the type of component to be added. Each model gets its own Angular controller to limit the scope of the *ModelBuilderController* as much as possible. Once components have been added, they can be selected by clicking on them. If the component is a cell group folder, this means that the folder can contain subcomponents that are not visible at the current scope of the model. These folders can be double clicked to trigger a ng−dblclick event that will cause the current list of components to be hidden, allowing the subcomponents to be viewed. When the subcomponents are shown, a breadcrumb trail is added to the top of the panel which shows where you came from. This trail will keep growing the deeper the user traverses into the model, and can be seen in Figure 4.7. At any point the view can be changed to any of the parent views by clicking on the desired breadcrumb. Components can be removed by selecting the remove button attached to each component.

4.3.5 Parameters Panel

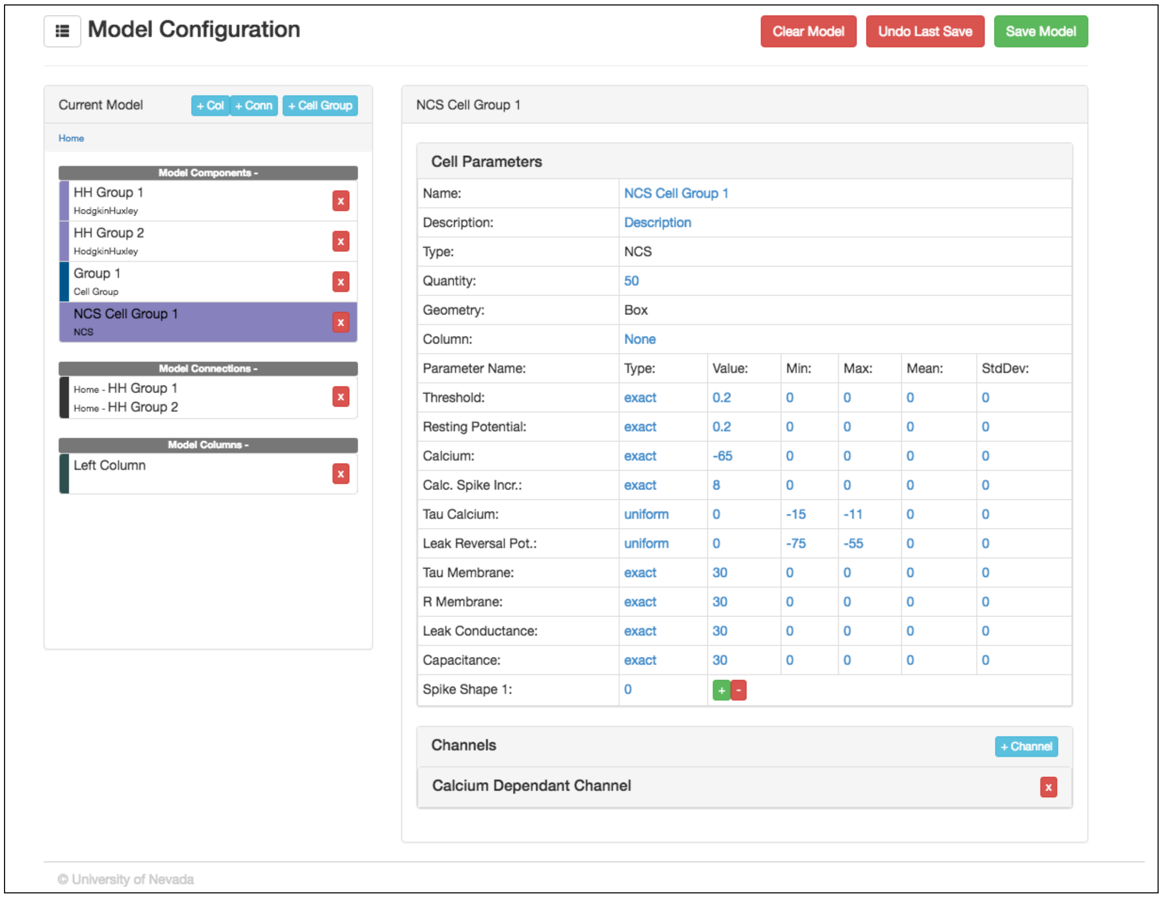
The parameter panel is the panel located on the right hand side of the page and contains all of the specific parameters for the selected component. The parameters are represented in large tables that can list all of the possible options. Each cell type gets its own table, and which table is displayed is dependent on the scope's displayed.classification component as well as the displayed.parameters.type component set in the *ModelBuilderController*. The displayed object is set when a component is selected in the current model panel. These two classifications will determine the parameters shown, as an Izhikevich cell will have different parameters from a NCS cell. Any parameters that can be edited will be highlighted in blue and can be edited by clicking on it. This will trigger an angular-xeditable component that displays the input field inline with the value. Angular-xeditable components are third party Angular directives that allow data values to be edited on the same line they are displayed [35]. Data values are checked before being saved for correct values by triggering an onbeforesave event which validates the data. A complete model example can be seen in Figure 4.8.

If the cell type permits it, a sub-panel representing channels will be displayed at the bottom of the parameters panel. Channels can be added to both Hodgkin Huxley cells as well as NCS cells. When the channel panel is selected, more parameter options are displayed for the selected channel, as seen in Figure 4.9. These can be edited the same way as the cell group parameters, by clicking on them. Multiple channels can be added to a single cell group and can be removed by selecting the remove channel button.

When a synapse connection is selected, the panel displays parameters for the selected synapse similar to the parameters for a cell group. The same is true when the user selects a column to edit. All of the parameters can be edited the same way using a x-editable input component.

4.3.6 Model Operations

There are three global model operations that can be performed on the current model. The model can be cleared, the last save can be deleted and the current changes will be removed and a previous version will be loaded, or the model can be saved. Each



HH Group 1
HodgkinHuxley
HH Group 2
HodgkinHuxley
Group 1
Cell Group
NCS Cell Group 1
NCS

Home - HH Group 1
Home - HH Group 2

Left Column

NCS Cell Group 1

Cell Parameters

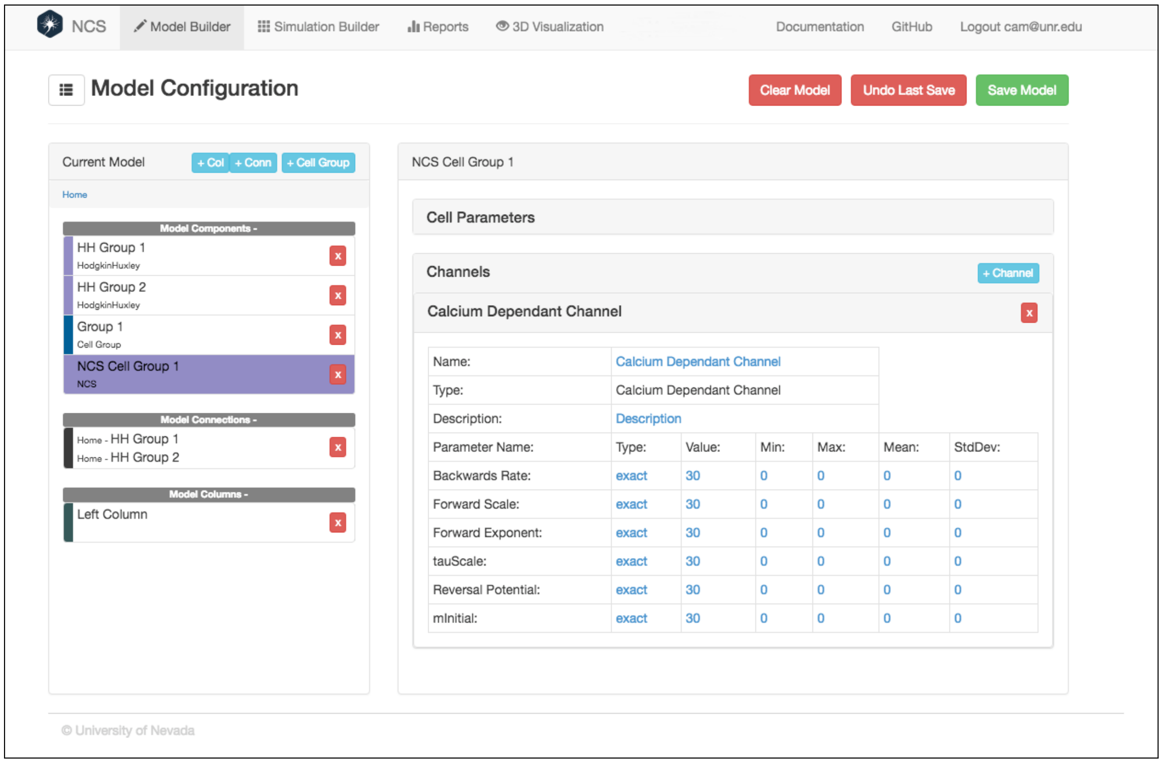
Name:	NCS Cell Group 1
Description:	Description
Type:	NCS
Quantity:	50
Geometry:	Box
Column:	None

Parameter Name:	Type:	Value:	Min:	Max:	Mean:	StdDev:
Threshold:	exact	0.2	0	0	0	0
Resting Potential:	exact	0.2	0	0	0	0
Calcium:	exact	-65	0	0	0	0
Calc. Spike Incr.:	exact	8	0	0	0	0
Tau Calcium:	uniform	0	-15	-11	0	0
Leak Reversal Pot.:	uniform	0	-75	-55	0	0
Tau Membrane:	exact	30	0	0	0	0
R Membrane:	exact	30	0	0	0	0
Leak Conductance:	exact	30	0	0	0	0
Capacitance:	exact	30	0	0	0	0
Spike Shape 1:	0	+ -				

Channels
+ Channel

Calcium Dependant Channel

Figure 4.8: An example of a complete model.



Name:	Calcium Dependant Channel
Type:	Calcium Dependant Channel
Description:	Description

Parameter Name:	Type:	Value:	Min:	Max:	Mean:	StdDev:
Backwards Rate:	exact	30	0	0	0	0
Forward Scale:	exact	30	0	0	0	0
Forward Exponent:	exact	30	0	0	0	0
tauScale:	exact	30	0	0	0	0
Reversal Potential:	exact	30	0	0	0	0
mInitial:	exact	30	0	0	0	0

Figure 4.9: The channels panel for editing channel parameters.

operation has a button at the top of the model builder tab, and each button will spawn a modal window to confirm or prompt for more information. Each modal gets its own controller, again to minimize the use of the *ModelBuilderController* as much as possible. ‘Clear Model’ will simply prompt before clearing the model. When the option to clear the model is selected, the *ClearModelController* will simply tell the *CurrentModelService* to clear the loaded model. Once the model is cleared, the interface will automatically update to display an empty model. ‘Undo Last Save’ will prompt the user for where the last model save is located, either in the personal, lab, or global model database. Once this is selected, and the option to undo the last save is selected, an HTTP request will be sent to the server which will forward the request to the daemon. The daemon will then remove the most recent save and return a previous version of the model. Upon success, the previous model is returned to the client, where the interface is updated to show the previous version. When ‘Save Model’ is selected, a modal is presented prompting for the model name and which database to save to.

Once the data has been entered and selected, the *SaveModalController* will create an HTTP request to save the model. The request will then be forwarded to the daemon and the model will be saved.

4.4 Simulation Builder

4.4.1 Overview

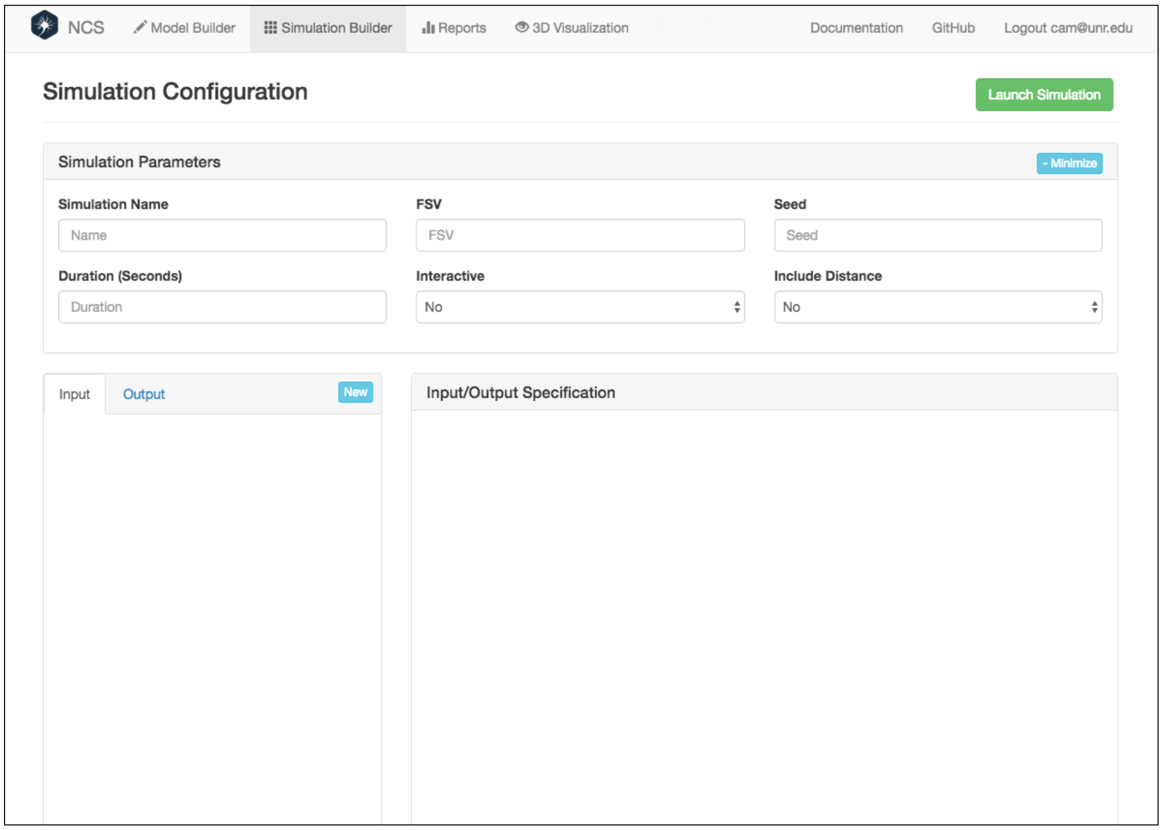
The simulation builder tab is where simulation parameters can be configured, and where simulations can be launched on NCS. Simulation parameters include the simulation name, the duration the simulation will run, as well as simulation inputs and outputs. When a user decides to launch a simulation, they can choose to launch the simulation they currently have loaded, or can choose from a previous saved one.

4.4.2 Layout

The simulation builder tab is split into three main views. There is a top panel which contains the global simulation options such as name and duration. There is a tabbed column on the bottom left which indicates a list of inputs and outputs. The panel in the bottom right includes the specific parameters for the input or output that is selected.

4.4.3 Parameter Panel

The parameters panel simply contains options to be used as global simulation configurations. There are six global options, with two primary options, simulation name and duration, and the other four options are not currently used but are reserved for future use. The simulation name is used to keep track of a specific simulation configuration to be run. The duration specifies how long the simulation should run on NCS before being terminated. The duration is given in seconds. These inputs are directly connected to the underlying data through ng-model directives.



Simulation Configuration

Simulation Parameters

Simulation Name	FSV	Seed
Name	FSV	Seed
Duration (Seconds)	Interactive	Include Distance
Duration	No	No

Input/Output Specification

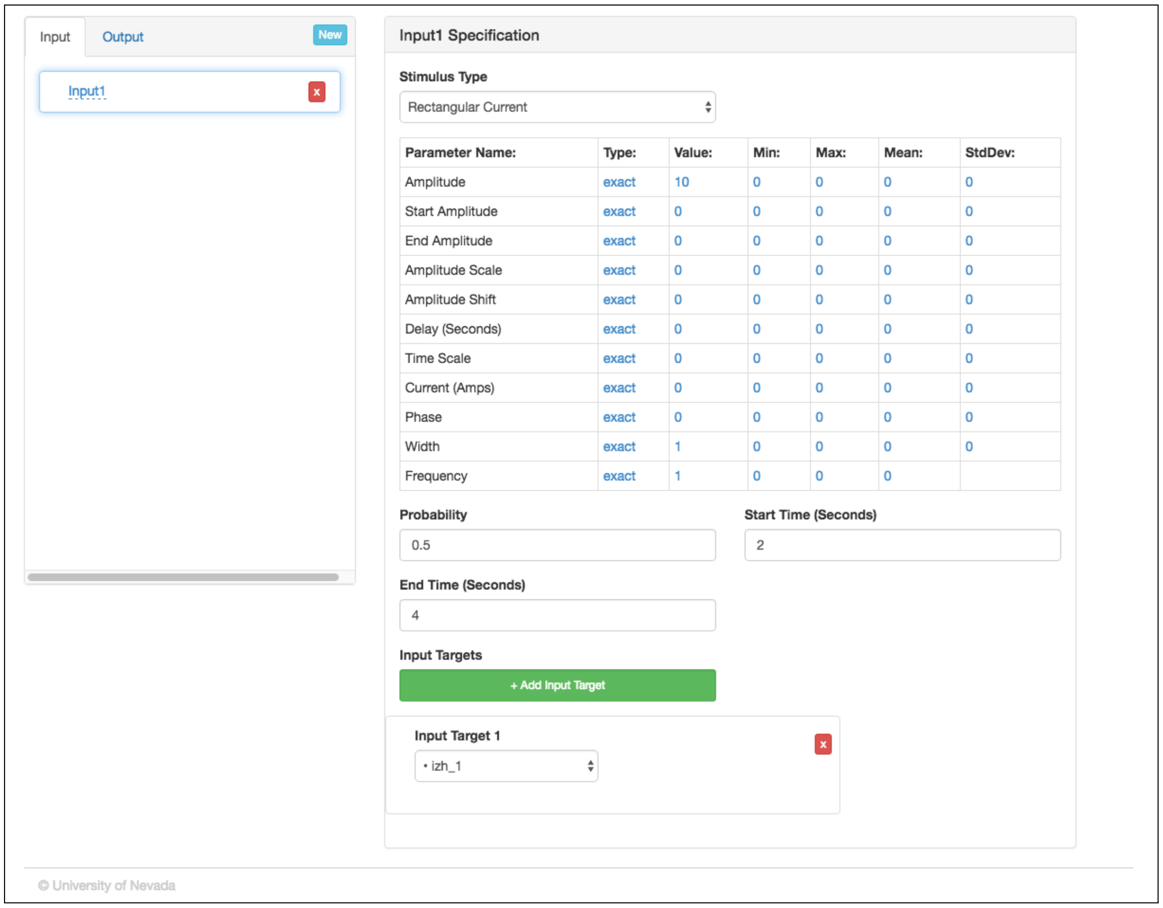
Figure 4.10: The initial view of the simulation builder tab.

4.4.4 Input/Output Panel

The input and output panel consists of two lists which can be toggled by selecting the input or output (IO) tab options. As IO objects are added, theses lists will grow larger with the ability to select specific inputs and outputs. The IO objects are displayed through the use of the ng-repeat directive which repeats the contents for each item in the list. Each list group is shown or hidden based on the currently selected tab. When an IO object is created, an object is created with defaulted parameters. When an IO object is selected, the *SimulationController* sets the selected object to the IO object clicked. IO objects can be removed by selecting the 'x' button next to the object. This will remove the object from either the inputs or outputs list and the list, and if the currently selected object is the one removed, the selected object is updated to a different object in the list. An IO object can also be renamed by clicking on their name in the list. This will open a dynamic text input option to rename the object. This is achieved by using the x-editable library and attaching an editable component to each of the list objects.

4.4.5 Input/Output Specifications Panel

The input and output specification panel is used to specify specific parameters for a specific input or output object. When an item is selected, the specification panel title is updated to that of the IO object's name. Different parameters are then shown based on the class type of the IO object selected, which is either `"simulationInput"` or `"simulationOutput"`. When an input object is selected, the parameters displayed include the stimulus type, a list of specific input parameters, the probability of the input, the start time, the end time, and input targets. The stimulus type is a selection object with the options rectangular current, rectangular voltage, linear current, linear voltage, sine current, and sine voltage. For each specific input parameter, the user can choose a value type of exact, uniform, or normal. If a value type is exact, the exact value is used. If a value type is uniform, the minimum value, maximum value, mean,



Input1 Specification

Stimulus Type: Rectangular Current

Parameter Name:	Type:	Value:	Min:	Max:	Mean:	StdDev:
Amplitude	exact	10	0	0	0	0
Start Amplitude	exact	0	0	0	0	0
End Amplitude	exact	0	0	0	0	0
Amplitude Scale	exact	0	0	0	0	0
Amplitude Shift	exact	0	0	0	0	0
Delay (Seconds)	exact	0	0	0	0	0
Time Scale	exact	0	0	0	0	0
Current (Amps)	exact	0	0	0	0	0
Phase	exact	0	0	0	0	0
Width	exact	1	0	0	0	0
Frequency	exact	1	0	0	0	

Probability: 0.5

Start Time (Seconds): 2

End Time (Seconds): 4

Input Targets

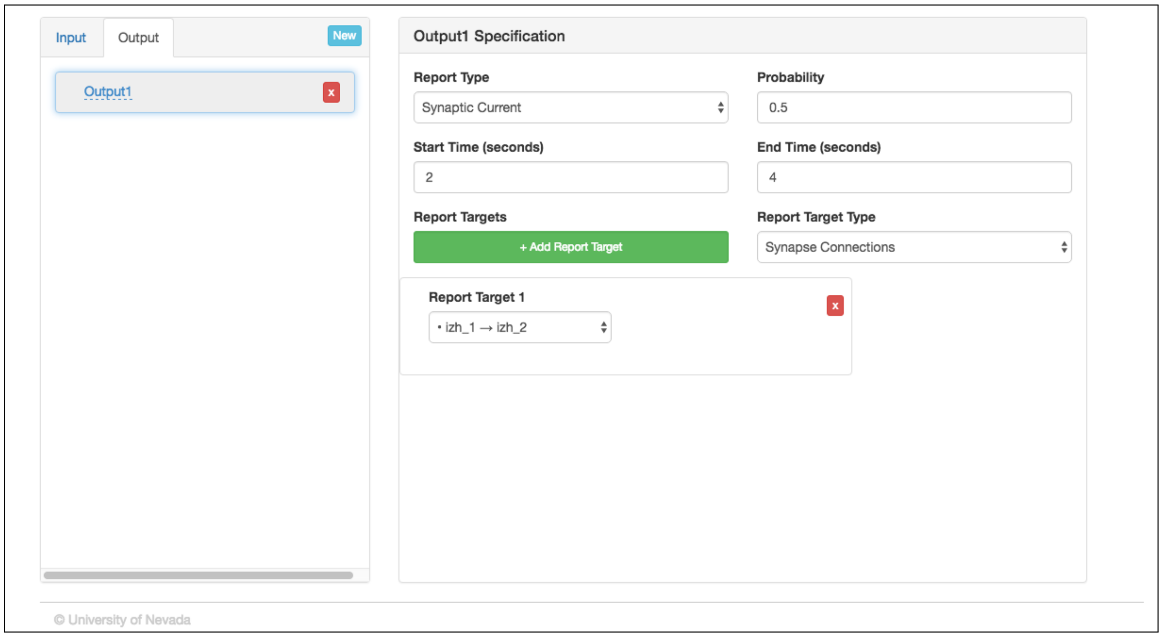
Input Target 1: • izh_1

Figure 4.11: A possible input configuration in the simulation builder tab.

and standard deviation are used with a uniform distribution calculated at runtime in NCS. The same is true if a value type is normal, however a normal distribution is used instead of a uniform distribution. Multiple input targets can be added dynamically given a loaded model. The list of input targets is dynamically updated when the user updates the model. After the user updates the model, when the simulation tab is selected, a `"page-changed"` event is triggered which causes the *SimulationController* to update all possible input and output targets.

4.4.6 Launch Simulation

When the user has successfully configured their simulation, they can choose to launch a simulation. When this happens, a modal window is displayed prompting the user which model they would like to use with their simulation, the current model loaded, or



Output1 Specification

Report Type: Synaptic Current

Probability: 0.5

Start Time (seconds): 2

End Time (seconds): 4

Report Targets

Report Target Type: Synapse Connections

Report Target 1: • izh_1 → izh_2

Figure 4.12: A possible output configuration providing a synaptic current report on a synapse connection.

a previously saved model. When the model is executed, a *LaunchSimulationController* is used to load the possible models which have been loaded from the daemon for the user's account. When the user launches the simulation, if the user selected the current model option, the model is copied from the *CurrentModelService*, otherwise it is set from the list of loaded models. The model and the simulation parameters are then saved as a JSON object and an HTTP request is made to the server to launch the simulation.

Streaming Data

When the launch simulation request is received by the server, the sever looks to see if the user already has running simulations, and if not creates a new *report_ object* to cache the local result data obtained from the daemon. Once the report objects are retrieved or created, the server then spawns a thread for each active report to be tracked from NCS. Each thread establishes a RabbitMQ channel to the daemon message queues for transferring the data from the daemon to the server. When the server is started, a persistent connection is created to the daemon which is later used to create channels that connect to specific message queues. When a channel is created, a specific routing key is created that has the format `"[username]..[report_name]"` which guarantees that each report for each user gets a unique routing key. A channel will stream data until a `"STOP"` message is received. The server then forwards the request to the daemon which handles launching the simulation and streaming the results.

4.5 Reports

4.5.1 Overview

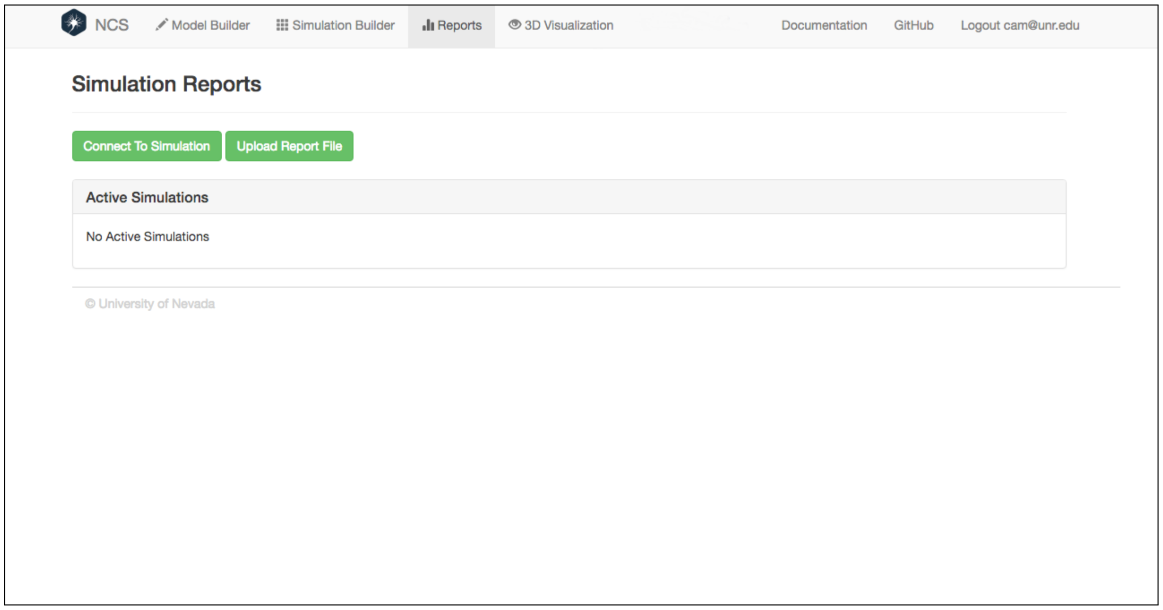
The reports tab is where outputs specified in the simulation builder tab can be viewed as graphs. Graphs can be viewed as live output from NCS or can be viewed by uploading a file of a previously run result. Graphs can then be exported to an image to be used later.

4.5.2 Layout

The layout of the report tab is very simple. When the tab is first viewed, you have the option to connect to a running simulation, or upload a previously generated report file, as seen in Figure 4.13. When one of these options is selected, the active simulations panel will be populated with the corresponding outputs. Several graphs can be rendered at once, as shown in Figure 4.14.

4.5.3 Simulation Selector

When a user connects to a running simulation, it is possible that the user may have several simulations running in parallel. If the user has several simulations running, they can choose which simulation they want to view by selecting one of the possible simulation tabs in the active simulations panel. All of the displayed data is determined by which simulation is currently active. Because of this, when the user selects a simulation tab, the *ReportsController* will set the active simulation to the index of the simulation selected. When the active simulation changes, all of the data in the tab is updated to reflect the change.



Simulation Reports

Active Simulations

No Active Simulations

© University of Nevada

Figure 4.13: The initial view of the reports tab.

4.5.4 Report Panels

When report data is available, all of the reports will be displayed in their own collapsable panel. Each panel will contain the graph of the data being displayed, as well as a panel for configuring the graph. Each output graph panel can be minimized or maximized to enable seeing multiple results easier.

Constructing Graphs

When the user connects to a running simulation, the server is sent an HTTP request to get information about the current configuration being run on NCS. When a simulation is launched, a request is sent to the daemon to launch the simulation on NCS as described in Section 4.4.6. After a simulation has been launched, an HTTP request can be made to retrieve the current report specifications. This request will return the current configuration, including all active simulations, and all reports connected to each active simulation, as well as the data currently stored for each report. Once the configuration has been loaded, a Highstocks *StockChart* is created for each report in the active simulation. The *StockChart* uses defaulted values for gradients and plot

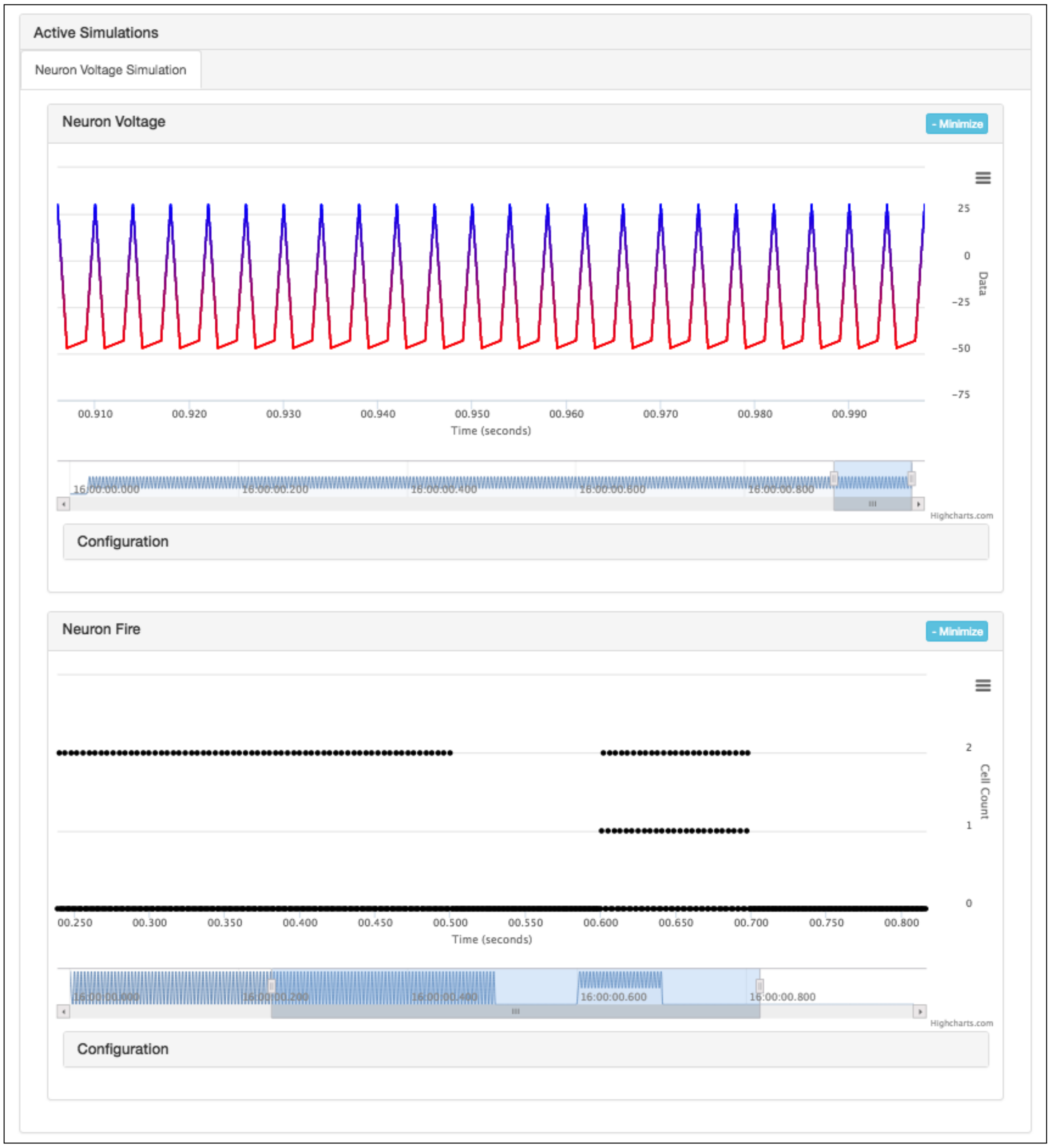


Figure 4.14: The reports tab with two active reports.

lines when the graph is initially created, although these can be customized later.

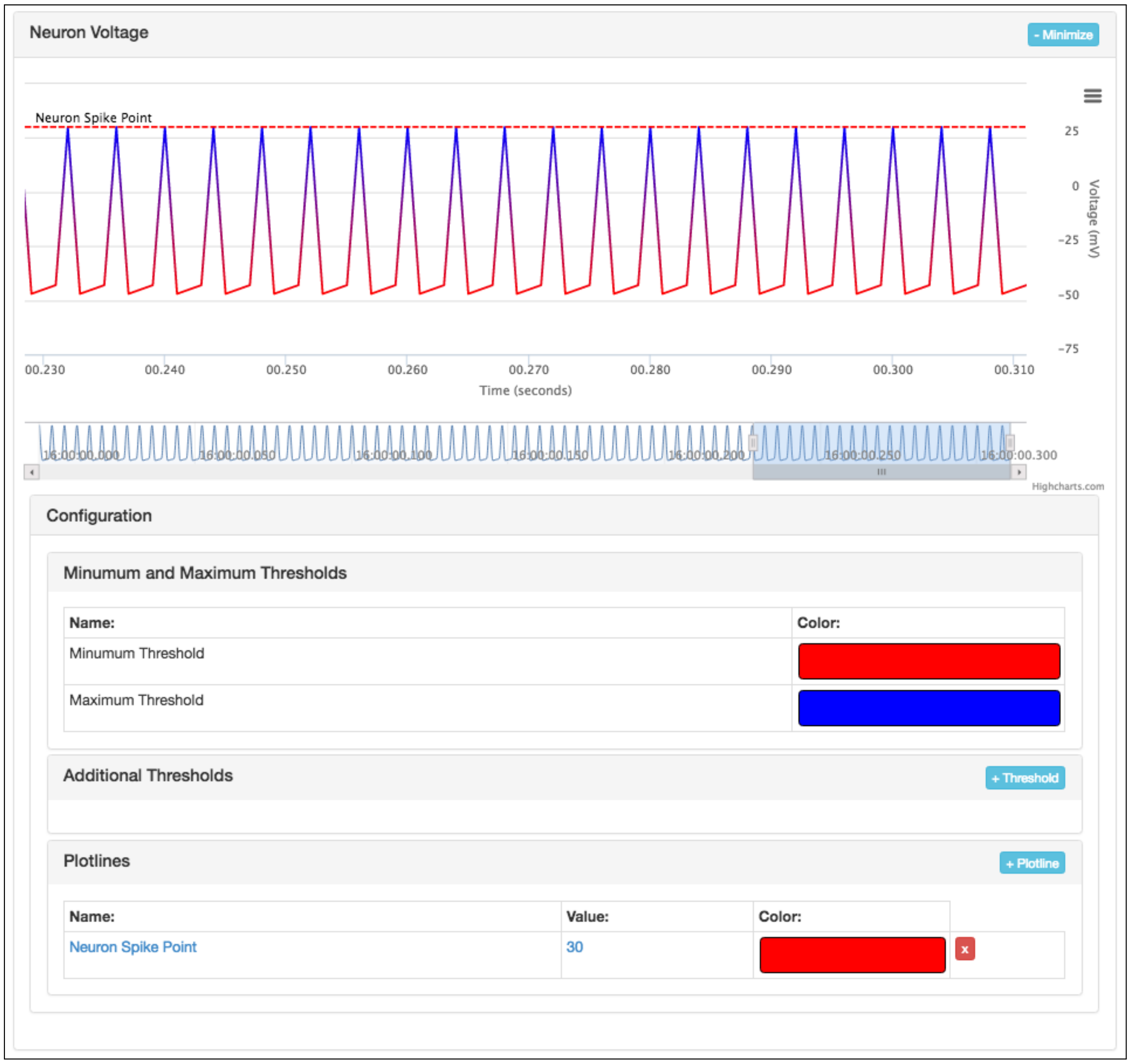
Streaming Data

Once the charts have been created, an HTTP request is made to the server every 250 milliseconds to update the data. This time was chosen because the amount of data being passed can be large if a simulation has a lot of cells and a high output probability. NCS has a time-step of 1 millisecond in which all report data will be updated [29]. Due to this limitation, data is grabbed in bulk to ease the number of requests the web interface has to make to the server. Once data is received, each report is updated with its corresponding data and the graphs are redrawn.

Configuration Panel

Each graph has the ability to be customized through the configuration panel attached to each graph. Each graph has a collapsable configuration panel that can modify the gradients used in the graph, add additional gradients, and add plot lines to act as a reference point when viewing data. The configuration panel is implemented as a Bootstrap panel−collapse object that is initially set to a collapsed state. The configuration panel can be seen in Figure 4.15.

Thresholds To represent the threshold colors, a form−group is used where the background color of the HTML element corresponds to the current color being used. When selected a colorpicker angular directive is used to provide a popup that is bound to the gradient data from the current report object. This allows colors to be changed. The *ReportsController* has several watchers that watch for when the gradients are changed, and when they are trigger the graph to be updated and redrawn with the new look. A watcher is an Angular function that will check if specified data is changed every time the scope is updated. This gives users realtime feedback as they customize their graphs. Additional thresholds can be added and can be set to specific areas in the graph. What this means is when a user adds a new threshold, they can control a slider which acts as a range between the smallest and largest values in the graph.



Configuration

Minumum and Maximum Thresholds

Name:	Color:
Minumum Threshold	🟥
Maximum Threshold	🟦

Additional Thresholds

+ Threshold

Plotlines

+ Plotline

Name:	Value:	Color:	
Neuron Spike Point	30	🟥	x

Figure 4.15: The reports tab with one report and its configuration panel.

This causes the threshold to be set at the specified value in the range of smallest to largest.

Plot Lines Plot lines can be added which are lines that can be placed on the horizontal axis to act as a reference point for data being displayed. When a plot line is added, the *ReportsController* will create a new object with the configuration settings of the plot line and add it to the plotlines array. The *ReportsController* has a watcher on the plotlines array and when it changes, either by adding a new plot line, by changing the configuration of a previously existing plot line, or by removing a plot line, the current graph is updated and redrawn to show the change.

Graph View

The graph generated has several sections for making viewing of the data easier. By default, the graph will display all of the data that is available for the specific report. The graph is structured in a way that the x-axis represents the time the data was received, and the y-axis represents the specific value for that data point. The y-axis can be a synaptic current, neuron voltage, number of cells fired, and an input current. The y-axis will dynamically change based on the type of graph presented. If no type is presented, such as when the data is uploaded from a file, the axis will simply be labeled `"Data"`. The graph has the ability to zoom to specific parts of the data, and scroll throughout the data once it has been zoomed. This is achieved through the *HighStock* chart built-in scrolling functionality. If the zoomed view is placed at the far right, only the new data will be displayed as it is received by the *ReportsController*.

Export Image

HighStock charts have the ability to be exported as an image of a current graph. When the user chooses to export an image, they can export the image as either a .png, .jpeg, .svg, or .pdf. When the export occurs, the data currently rendered, including thresholds, plot lines, and the current zoom level are what is rendered to the image.

4.6 Visualizer

4.6.1 Overview

The visualizer tab provides a 3D rendered WebGL view of the current model loaded. This tab is considered to be in a prototype stage as the functionality is currently limited to simply rendering a constructed model. The visualizer tab can be seen in Figure 4.16.

4.6.2 Layout

The visualizer tab has two components, the configuration panel at the top of the tab, and the WebGL renderer below it.

4.6.3 Configuration Panel

The configuration tab is a panel that contains options for configuring the WebGL renderer. It contains options for toggling the rendering of neurons, connections, and columns. It also has the ability to set a minimum and maximum threshold, as well as the threshold color which exists as a gradient between the two colors. These thresholds exist to help render a running simulation which is not currently implemented.

4.6.4 WebGL Renderer

Scene Initialization

The WebGL render is a HTML canvas object that has a ThreeJS WebGL renderer attached to it. When the *VisualizationController* is initialized, the ThreeJS WebGL renderer is created and attached to the HTML canvas that represents where the scene will be visualized. The renderer uses two cameras to achieve the current look, a ThreeJS PerspectiveCamera and a ThreeJS OrthographicCamera. The PerspectiveCamera is used for rendering the 3D scene. The OrthographicCamera is used for rendering 2D elements, namely the selection box which is discussed later. A ThreeJS Scene object is created to handle the rendering of the scene which will contain all of the objects to be rendered. A second Scene object is created to hold all 2D objects such as the selection box described below. This 2D Scene object is rendered using the Orthographic camera. An OrbitControls object is added to the PerspectiveCamera to enable mouse input that can rotate the scene. Event handlers are then added for mouse input to enable interaction with the scene.

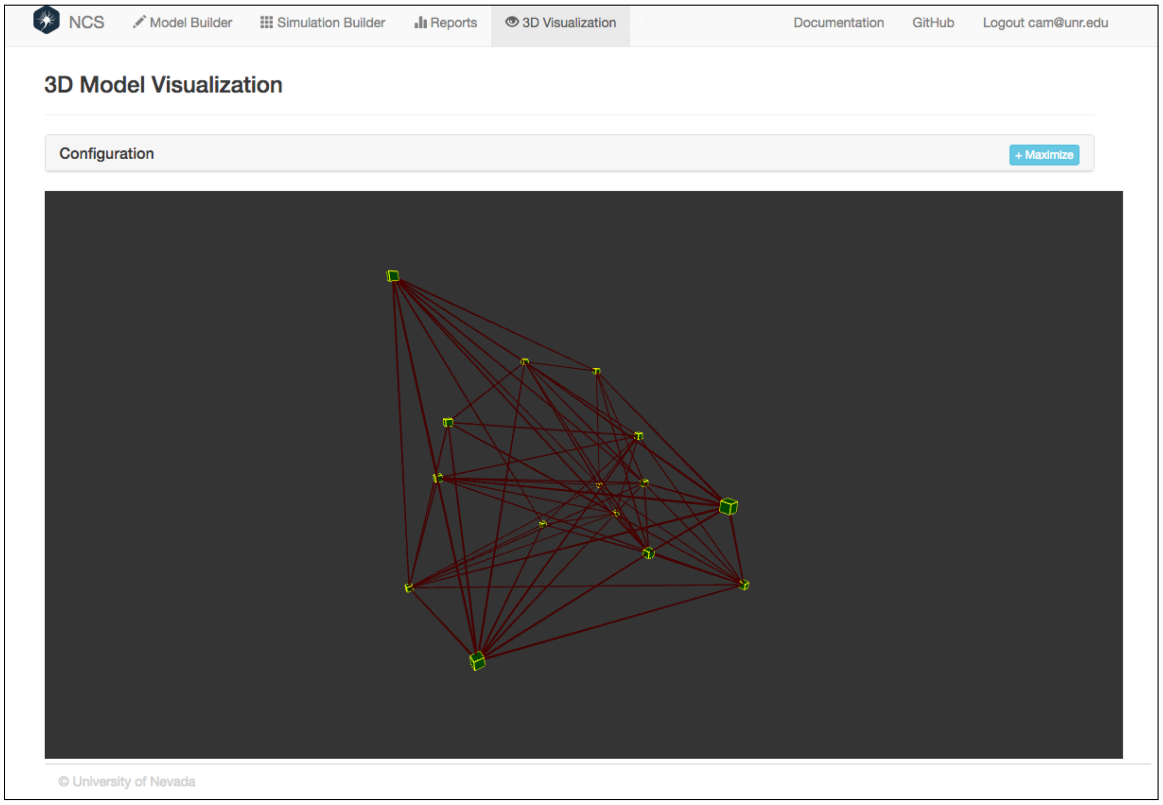


Figure 4.16: The visualization tab with a model consisting of 15 cells, with a connection probability of 40% and a spread of 30.

3D Model Construction

Once the scene is setup, the 3D models can be constructed either from the current model, or from a randomly generated model for testing purposes. Cubes are used to represent neurons, cylinders are used to represent synapse connections, and transparent cubes are used to represent columns. Since no positional information is given when constructing models other than the positions of columns, objects need to be constructed using random locations. If neurons have an associated column, they will be given uniformly distributed locations within the volume of the column. If no column is present, the neurons will be uniformly distributed throughout the entire scene with a given spread. Columns are constructed using the presynaptic and postsynaptic connection information. Since all neuron objects are created first, a lookup is performed when constructing the synapse objects to find the location of both the pre and post synaptic connection objects to attain their positions. Once the positions are known, a cylinder is created, rotated, and placed as shown in Figure 4.17. Finally the column objects are created using the information supplied when the columns are added in the model builder tab. Columns are represented as bounding boxes which will contain neurons inside of them.

3D Selection

Models and connections can be selected by clicking and dragging the mouse to highlight specific object. 3D group selections was achieved by creating a *SelectionBox* class that is initialized with a start position and dynamically updates as the user drags the mouse. The selection box when updating can be seen in Figure 4.18. When the user releases the mouse, first the selection boundaries are computed, then for every object in the scene, the objects coordinates must be converted to screen coordinates. This is done by getting the objects position from the matrixWorld property which is a matrix representing the global transform of the object, then projecting the camera transform onto the position vector. The x and y positions are then updated to reflect the size of the canvas being rendered. Once the object's position is represented

```
function createCylinderFromPoints(p1, p2, color) {
    if(color === undefined) {
        color = 0xff0000;
    }

    var direction = new THREE.Vector3().subVectors(p2, p1).normalize();
    var length = p1.distanceTo(p2);

    var mat = new THREE.MeshBasicMaterial({color : color});
    var geo = new THREE.CylinderGeometry(0.1, 0.1, length, 8);
    var cylinder = new THREE.Mesh(geo, mat);

    var pos = new THREE.Vector3().addVectors(p1,p2).divideScalar(2);
    cylinder.position.copy(pos);

    var q = new THREE.Quaternion();
    if(direction.y > 0.99999) {
        q.set(0,0,0,1);
    }
    else if(direction.y < -0.99999) {
        q.set(1,0,0,0);
    }
    else {
        var axis = new THREE.Vector3(direction.z, 0, -direction.x).
            normalize();
        var angle = Math.acos(direction.y);
        q.setFromAxisAngle(axis, angle);
    }

    cylinder.quaternion.copy(q);
    return cylinder;
}
```

Figure 4.17: JavaScript code for creating a synapse connection between two points.

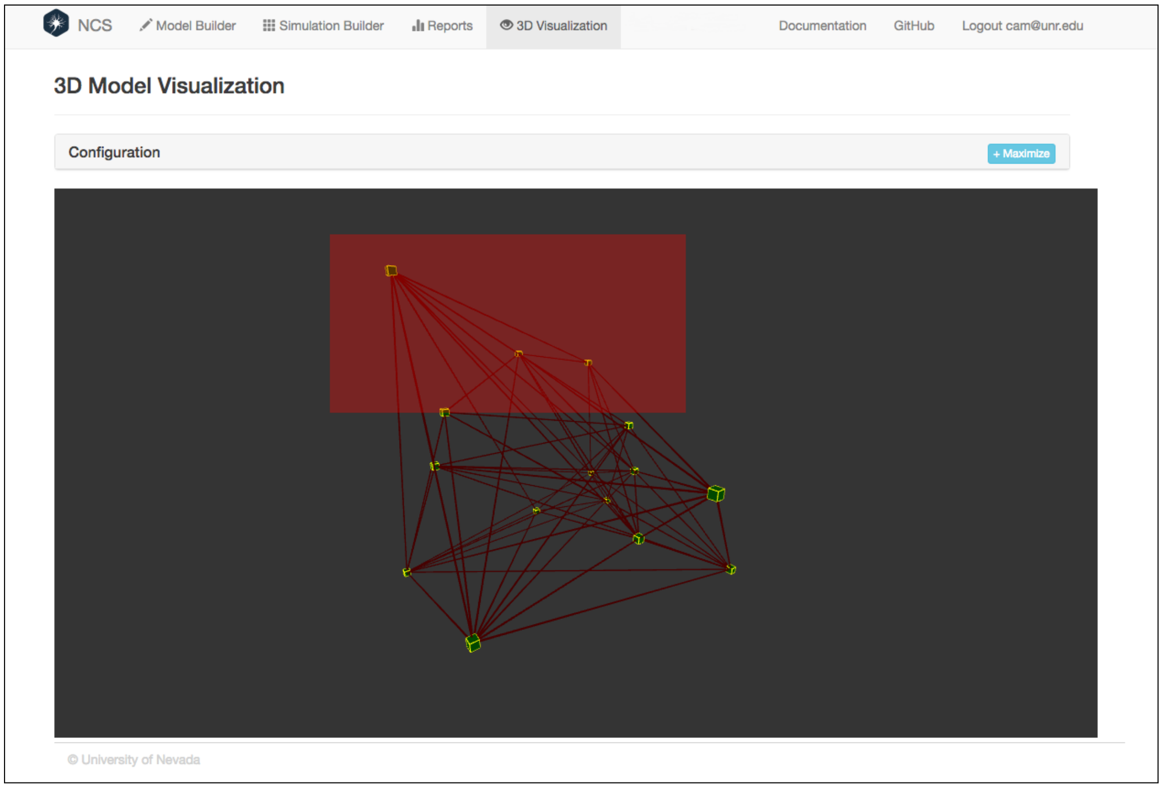


Figure 4.18: The visualization tab when the user is selecting cells.

in screen coordinates, we can simply check if that screen coordinate falls within the boundaries of the selection box. For all objects within the selection, their colors are updated to a brighter version to show that they have been selected. An example of selected cells can be seen in Figure 4.19.

4.7 Summary of Software

The NCS web interface was developed using command-line tools and a text editor. The main development tool is NodeJS which handles building all of the components of the application. This includes combining all HTML files into a single file, combining all JavaScript files into a single file, and compiling the LESS code into a single CSS file. NodeJS also handles launching the Python server, and the Flask framework will restart the server if any changes are made to the code. The NCS web interface is an open source application, and can be found at

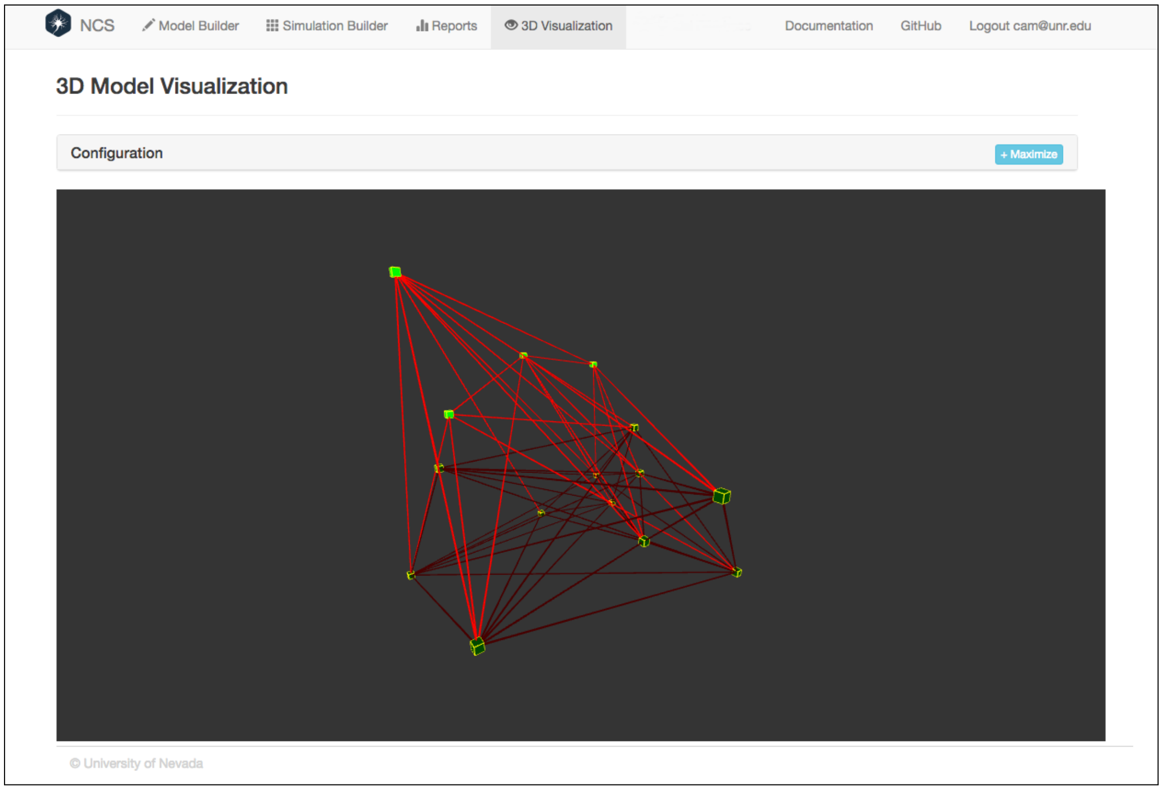


Figure 4.19: The visualization tab with cells selected.

`https://github.com/BrainComputationLab/ncb`. For perspective, the web interface contains approximately 11,000 lines of code, and uses a total of 49 classes, consisting of Python classes, JavaScript classes, AngularJS controllers, and AngularJS services.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Computational neuroscience is a new growing field that allows researchers to study neuroscience like never before. Instead of performing invasive procedures on living creatures, neurological structures and behaviors can be studied by simulating their activity on computers. Many brain simulators have been created, including the Neo-Cortical Simulator (NCS) developed at the University of Nevada, Reno. NCS is a brain simulator that can be distributed across a homogeneous cluster, which allows for large simulations to be run in real time. The problem with the current approach is that developing models and simulations to run on the simulator can be complicated and requires programming knowledge. The NCS web interface was developed to solve this problem by providing a web application to construct brain models and simulation parameters as well as visualizing output data and a 3D representation of the created model. The web interface provides intuitive methods for developing models, and makes it easy to launch simulations on NCS, while making results easy to understand and learn from.

5.2 Future Work

5.2.1 Interface Enhancements

The web interface provides a lot of great features, however there are many improvements to be made. One major improvement that could be made would be to remove the web interface server completely and have the client communicate directly with the daemon. The current problem is that the server simply acts as a proxy to the daemon, where most requests are simply forwarded to have the daemon perform the actual operation. This makes the sever act as a middleman to the middleman, meaning that it could be removed and performance improvements could be made.

Another area for improvement would be streaming data for the reports tab. With the current implementation, the report tab polls the server at a specific interval to retrieve new data that has been output from NCS. This could be improved by adding the ability to directly stream the data through a technology like WebSockets that can keep a persistent connection open between the client and the daemon. This would improve performance as every piece of data that comes in would be instantly streamed to the client.

One interesting addition to the web interface would be to make a native mobile application for constructing models. The benefit would be to provide a better look and feel when working on mobile devices, and performance could be improved by writing natively compiled code that could outperform the code written in JavaScript. The application could be made more robust and secure by handling local data on client devices.

The visualization tab has many improvements that could be made. First, all the tab can do currently is visualize the models that have been created. The ability to visualize neurons firing corresponding to a running simulation could be added to visualize the behavior in three dimensions. Also, the visualizer assumes that all cell groups only contain one cell; the ability to render cell groups of arbitrary sizes would improve the functionality of the visualizer greatly.

Lastly, the implementation of many components could be greatly simplified by fully exploiting all of the features AngularJS has to offer. Given that Angular was being learned as development was underway, certain components were brute forced where Angular could have been used to simplify the amount of HTML used in the application. This would require full rewrites of many components, and thus is a low priority.

5.2.2 Daemon Enhancements

As stated previously, the daemon and web interface server could be merged to have a single service access both NCS and provide a REST interface for the web interface. On top of this, the daemon could add features such as adding inputs and outputs to a simulation after it has been launched. This would allow a long running simulation to behave more dynamically as more information could be obtained by adding more inputs or outputs after after the simulation has already launched.

Bibliography

[1] E. Almachar, A. Falconi, K. Gilgen, D. Tanna, N. Jordan, R. Hoang, S. Dascalu, L. Jayet Bray, and F. Harris. NeoCortical Repository and Reports: Database and Repository for NCS. *Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE-2014)*, 2014.

[2] Armin Ronacher. Flask (A Python Microframework). `http://flask.pocoo.org/`, 2016. [Online; accessed 1-May-2016].

[3] J. Berlinski, C. Rowe, M. D. Chavez, N. Jordan, D. Tanna, R. Hoang, S. Dascalu, L. Jayet Bray, and F. Harris. NeoCortical Builder: A Web Based Front End for NCS. *Proceedings of the 27th International Conference on Computer Applications in Industry and Engineering (CAINE-2014)*, 2014.

[4] J. Bower and D. Beeman. *The Book of GENESIS*. 2003.

[5] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. 2014.

[6] 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)*, pages 37–43, 2013.

[7] N. Carnevale, M. Hines, and J. Moore. NEURON: for empirically-based simulations of neurons and networks of neurons. `http://www.neuron.yale.edu/neuron`, 2013. [Online; accessed 1-May-2016].

[8] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

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

[10] D. Goodman and R. Brette. The Brian Simulator. *Frontiers in Neuroscience*, 3(2):192–197, 2009.

[11] Google. AngularJS - Superheroic JavaScript MVW Framework. `https://angularjs.org/`, 2016. [Online; accessed 1-May-2016].

[12] Highcharts. Interactive JavaScript charts for your webpage | Highcharts. `http://www.highcharts.com/`, 2016. [Online; accessed 1-May-2016].

[13] R. Hoang, D. Tanna, L. Jayet Bray, S. Dascalu, and F. Harris. A Novel CPU/GPU Simulation Environment for Large-Scale Biologically Realistic Neural Modeling. *Frontiers in Neuroinformatics*, 7:19, 2013.

[14] Roger Viet Hoang. *An extensible component-based approach to simulation systems on heterogeneous clusters*. PhD thesis, University of Nevada, Reno, 2014.

[15] A. Hodgkin and A. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952.

[16] E. Izhikevich. Simple Model of Spiking Neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, 2003.

[17] L. Jayet Bray. *A Circuit-Level Model of Hippocampal, Entorhinal and Prefrontal Dynamics Underlying Rodent Maze Navigational Learning*. PhD thesis, University of Nevada, Reno, 2010.

[18] C. M. Johnson. A Centralized Service for Accessing the NCS Brain Simulator Through a Web Interface. Master's thesis, University of Nevada, Reno, 2015.

[19] Nathan M Jordan, Kimberly B 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*, pages 13–19, 2012.

[20] T. Kelly. Neocortical Virtual Robot: A framework to allow simulated brains to interact with a virtual reality environment. Master's thesis, University of Nevada, Reno, 2015.

[21] MongoDB, Inc. MongoDB. `https://www.mongodb.com/`, 2015. [Online; accessed 1-May-2016].

[22] Node.js Foundation. Node.js. `https://nodejs.org/en/`, 2016. [Online; accessed 1-May-2016].

[23] M. A. Paradiso, M. F. Bear, and Connors B. W. *Neuroscience: Exploring the Brain*. Lippincott Williams & Wilkins, 2007.

[24] Pivotal Software, Inc. RabbitMQ. `https://www.rabbitmq.com`, 2015. [Online; accessed 1-May-2016].

[25] H. E. Plesser, M. Diesmann, M. O. Gewaltig, and A. Morrison. NEST: The Neural Simulation Tool. *Encyclopedia of Computational Neuroscience*, pages 1849–1852, 2015.

[26] Python Software Foundation. Applications for Python. `https://www.python.org/about/apps/`, 2016. [Online; accessed 1-May-2016].

[27] E. Schwartz. *Computational Neuroscience*. Mit Press, 1993.

[28] C. L. Stanfield, W. J. Germann, M. J. Niles, and J. G. Cannon. *Principles of Human Physiology*. Pearson/Benjamin Cummings, 2011.

[29] D. Tanna. NCS: Neuron Models, User Interface, and Modeling. Master's thesis, University of Nevada, Reno, 2014.

[30] The core Bootstrap Team. Bootstrap The world's most popular mobile-first and responsive front-end framework. `http://getbootstrap.com/`, 2016. [Online; accessed 1-May-2016].

[31] The core LESS Team. Getting started | Less.js. `http://lesscss.org/`, 2016. [Online; accessed 1-May-2016].

[32] Corey M Thibeault, Roger V Hoang, and Frederick C Harris Jr. A novel multi-gpu neural simulator. In *BICoB*, pages 146–151, 2011.

[33] three.js. three.js - Javascript 3D library. `http://threejs.org/`, 2016. [Online; accessed 1-May-2016].

[34] H. Tuckwell. *Introduction to Theoretical Neurobiology: Volume 2, Nonlinear and Stochastic Theories*. Cambridge University Press, 2005.

[35] Vitaliy Potapov. Angular-xeditable :: Edit in place for AngularJS. `https://vitalets.github.io/angular-xeditable/`, 2013. [Online; accessed 1-May-2016].