brain studio

User Manual

Version 0.001

Chapter 1 : Background

The basic objective for the mathematical modelling of brain structures using networks of phenomenological spiking neurons (SNNs) is is to conform to the dynamical behaviour of their real counterparts. This can be used as a proof of their biological plausibility, before these models are employed to make new testable predictions. However, SNNs inherently entail some distinct functional attributes that need to be carefully considered and built upon, in order to achieve biologically meaningful simulations.

The rich dynamical behaviour of single cells in the brain can be clustered into distinct categories, all of which can be captured by phenomenological spiking neuron models (Izhikevich 2007), and result in different group dynamics. In addition, this computational framework enables the investigation of known neural phenomena that have been considered important for our quest, such as oscillatory activity, plasticity and neuromodulation. Nonetheless, the great flexibility of these models is accompanied by a vast number of open parameters, often without physiological meaning, that make their employment a difficult challenge and create a need for optimization. The amount of the available computational resources is therefore critical for both the ability to simulate a realistic number of neuronal computational units, as well as for work which is prerequisite for the simulation.

For these reasons, a substantial number of methods and software systems have been recently proposed, that facilitate the design and simulation of SNNs. First, a number of software systems aims to the optimization of single spiking neurons in order to replicate recorded electrical traces of real cells. A review of these methods can be found in Geit et al. 2008. Despite their success, these tools have failed to establish a globally accepted methodology, since there is no single measure of goodness that accounts for all the desired features in the behaviour of a model neuron. For instance, some optimization tools focus on capturing the exact voltage trajectories and spike trains of biological neurons. The successful tuning of ionic-based models through this process, can result in valuable predictions on the structure of these cells. However, this is a difficult, and often impossible, task for simple phenomenological models, due to their limited state space. In contrast, these simple models can be tuned to produce the same type of dynamical behaviour found in their real counterparts, without a significant computational cost.

Second, a wide variety of simulation environments are dedicated to the calculation of the action potential propagation and the membrane potential dynamics of SNNs. These tools are using typical numerical integration methods, with optimized performance for different research applications. One category of tools, a characteristic example of which is the software NEURON (Hines and Carnevale 1997), is specialized on muli-compartmental neuron models and sub-cellular processes. The tool GENESIS (Bower and Beeman 2012) extends this concept to small-scale neural networks. On the other hand, a number of tools, such as NeMo (Fidjeland and Shanahan 2010), Nengo 2.0 (Bekolay et al. 2013), NEST Kayraklioglu et al. 2015 and CARLsim 3.0 (Beyeler et al. 2015), aim at simulations of large-scale networks, by focusing on phenomenological, single-unit models of neurons, and thus reducing the level of biological detail. Finally, the tools Brian (Goodman and Brette 2008) and″Geppetto Simulation Engine″2016 belong to a category where a compromise between flexibility and network size is intended.

From the network design perspective, a number of software simulators are supplemented by embedded graphical environments, that include Nengo, Geppetto and Spikestream (Gamez 2007). However, there is currently no implementation of a simulator-independent graphical user interface (GUI) that exhibits syntactic interoperability. Such an approach would be very beneficial for rapid prototyping, the cross-validation of simulation results and the reuse of existing model components. At a lower level, the vast majority of the above systems provide application programming interfaces (APIs) and can be used as libraries of general-purpose programming languages, such as C/C++ (NeMo and CARLsim 3.0) or Python (Brian, Nengo 2.0, NEST and Geppetto), while others provide their own scripting languages (NEURON and GENESIS). Unlike in the case of GUIs, the issue of compatibility that emerges from the plethora of available APIs can be addressed by the use of simulator-independent languages, such as PyNN (Davison et al. 2008) and neuroML (Cannon et al. 2014).

Despite the major progress of recent years, there is still a considerable room for improvement of the above techniques and introduction of new approaches. Brain Studio aims at addressing this deficiency in the existing software tools, where flexibility, computational performance, and real-time visual monitoring are all crucial factors. As the computational foundation, two well-known simulators, Nemo and Brain, have been selected and are presented below in more detail. These two simulators feature distinct strengths, namely speed and flexibility, which make them to complement one another.

NeMo spiking neural network simulator

NeMo is a high performance spiking neural network simulator that was originally developed in the Computational Neurodynamics Lab at Imperial College London by Fidjeland and Shanahan 2010, and has the form of a C++ library. NeMo takes advantage of the large number of CUDA-based graphics processing units (GPUs) of the NVidia graphics cards to provide a remarkably high memory bandwidth, and run parallel simulations with large number of neurons and synapses in real time. In a recent commercial of-the-shelf desktop computer, this system is able to simulate up to 500.000 Izhikevich neurons in real time, connected with 10.000 synapses each, under biologically plausible conditions that correspond to almost 5 billion spike events per second.

Although NeMo supports the development of custom neuron models, the default numerical integration of the neuronal differential equations is carried out using the Eulers method with a step size of 0.25 milliseconds (ms). Also, the conductance delays of the synapses have a precision of 1 ms, with a supported range from 1 to 64 ms. The default featured models include various forms of the phenomenological integrate-and-fire neurons including the simple model by Izhikevich et al. 2003, as well as the analytical conductance-based model by Hodgkin and Huxley 1952. In addition, learning in NeMo is realized by means of spike-time dependent plasticity (STDP), a form of Hebbian learning, initially introduced by Song et al. 2000. Finally, NeMo can be used directly, as a C++ class library, but it also provides interfaces for the languages Python, Matlab, PyNN and pure C.

The key feature of NeMo is its performance in parallelising the propagation of synaptic events in the network and distributing computation over the available resources (Brette and Goodman 2012). One of the major performance bottlenecks in GPU-based distributed applications is the time required for global memory access. In the case of SNN simulators, fetches of global memory chunks are required for each synaptic event at each timestep in order to calculate the effect of 17 2 Modelling tools this event on the post-synaptic neuron. NeMo addresses this issue by clustering synapses that have the same source and delay, so as potential synaptic events can be fetched simultaneously.

Despite its speed improvements over other systems, NeMo is inadequate for a large number of applications, due to limitations in flexibility and the lack of visualization tools. The default version of NeMo provides a fixed and limited set of neural models and a single type of synapses. Any modification to the equations of these models, or implementation of new ones, constitutes a complicated procedure, which requires adjustments in the C++ source code and re-compilation of the system. In addition, NeMo features a low-level API where neurons and synapses can be only added individually, which can make the design of complicated experiments a laborious task. For these two reasons, the implementation of a simulation with NeMo often becomes more time consuming than using other approaches.

Brian neural simulator

Brian (Goodman and Brette 2008), and its more recent version Brian 2.0 (Stimberg et al. 2014), is a SNN simulator based in python, focusing on high-flexibility and rapid prototyping. To achieve this, its API provides a concise syntax, compared to other python-based simulators, and the ability to write new equations for neurons and synapses in standard mathematical form. In addition, Brian features basic data recording, analysis, and plotting tools, and therefore it can support the complete process of designing and conducting a computational experiment based on SNNs.

The syntactic advantages of Brian can be partly attributed to the dynamic typing capabilities of python, according to which, objects are assigned a type at the runtime. Through this feature, the user can call the same methods to define entities in a network, using different types of attributes, and the system parses any given information transparently. Furthermore, neural populations or synapses with common characteristics can be grouped in single vectorised objects, and processed at once. Hence, scripts written in Brian become more readable, easy to learn and quickly extensible, while the simulated network models can incorporate a great deal of biological detail, compared to similar software systems.

Like NeMo, Brian is an open source project that can be developed and used in almost any platform. It only utilizes three standard python libraries, namely NumPy, Scientific Python (SciPy) and Pylab/Matplotlib, and it does not require local compilation of individual components. Finally, Brian interfaces with the language PyNN, and it can also be employed using its simulator-independent API.

On the negative side, a significant drawback of this framework is the compromise of computational efficiency, in both memory and time. Brian has been shown to be significantly slower in execution time than other approaches, in a comparison using different benchmark models in (Bekolay et al. 2013). Due to the expensive function call overhead, it is not well-suited for interactive, real-time simulations, such as the control of a robotic platform, or the real-time adjustments using a GUI. For instance, the former example would require a network update, in the order of milliseconds, every time that the sensors of the robot receive new stimulus. Additionally, as in the case of NeMo, Brian is designed for simulations of unit neurons, and the support for multi-compartmental models is limited, while currently there is no GUI implementation provided.

Brain Studio

The crucial factors in the design of Brain Studio have been flexibility, computational performance, and real-time visual monitoring. Therefore, the first aim here is to automate and speed-up the process of designing and testing large-scale SNNs, through a suite of new simulation tools. In particular, this suite should provide (1) a framework for the compact representation of modular SNNs, suitable for large-scale brain circuits and cognitive systems, (2) a generic and powerful simulator that allows the fast implementation, visualization and real-time adjustments of SNNs, and (3) a fast optimization technique, which could be used in order to mimic the dynamical behaviour of specific neurons in the brain, using simple phenomenological models. As the computational foundation of these features, Brain Studio intergrates both Nemo and Brian. (note: Brain Studio uses its own bespoke version of NeMo that comes with the Brain Studio download, standard versions of NeMo are not compatible). Nemo and Brain simulators feature distinct strengths, namely speed and flexibility, and thus have little functional overlap which make them to complement one another. The employment of both systems, for the purposes of Brain Studio, establishes a good trade-off between design and simulation performance.

Chapter 2 : Brain Studio Overview

The Brain studio tool has the form of a spiking neural network editor and simulator that focuses on cognitive architectures and large-scale neural systems. Its main aim is to provide the first independent graphical environment that can be used with multiple SNN simulators simultaneously. With this approach, the user can integrate different existing models that might have been developed based on different syntax, or design a new model from scratch.

Brain studio consists of two individual software layers. A GUI can be used to design and monitor an experiment in a user-friendly manner, and a stand-alone back-end executes the required calculations. These two parts communicate over the network via the TCP/IP protocols (**Fig.1**) and can be instantiated independently. The complete system is fully cross-platform and can run in either Windows-based, Unix-based or Macintosh computer systems.

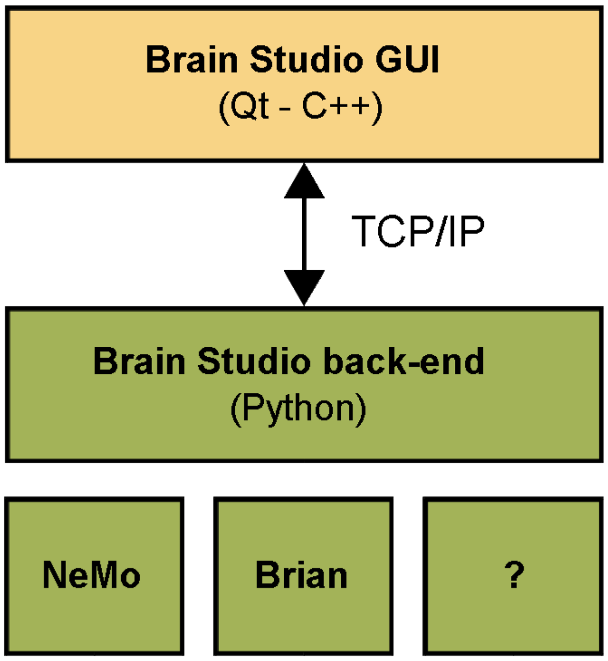


Figure 1. Hierarchical overview of brain studio's architecture. The *front-end* user interface communicates with the selected version of the *back-end* via a TCP/IP connection.

One major advantage of tool is that it allows the live real-time monitoring and adjustments of the designed networks during simulation. The final experiments are represented and archived using a simple XML-based model description format that includes any live adjustments. Hence, saved projects can be loaded directly by the back-end, and run without the need for visualization. In addition, its simulation engine is modular and can be easily extended to account for any low-level neural network simulator that provides a python application program interface (API), as well as new visualization techniques. These features make brain studio an advantageous platform for rapid prototyping and integration of existing network modules, written in different simulators.

In order to demonstrate this extensibility, and evaluate the speed of brain studio, in the alpha version NeMo simulator has been integrated, and gives the ability to simulate large-scale networks in near real-time. Future versions of Brain Studio will also incorporate the Brain neural network simulator. The following sections give details regarding the structure of individual components of brain studio, as well as its resulting simulation performance.

Network topology and simulation structure

Network topologies in brain studio are internally represented using only two unordered lists of entities, that include *nodes* and *edges* (**Fig.2A**). A *node* can be any group of indexed computational units, such as equations of spiking neurons, rate-based neurons or random event generators, which share common characteristics and connectivity. It contains statistical information for the parameters and states of these units, as well as their number and basic graphical properties. Each parameter or state can be defined either as a single number, a random variable that follows a known distribution, or as a function of random variables.

In addition, a group of similar, directed connections between two nodes is described by an *edge*. As in the previous case, edges also contain statistical information that defines the pattern and the parameters of their underlying connections. The definition of a connectivity pattern requires its type and the indexes of the computational units in the source and the target nodes. Finally, the available types of patterns include *all-to-all*, *topographic* (one-to-one), and *sparse* connection groups (see **Fig.2B**).

In order for a complete experiment to be defined, a third list of entities is used that represents *actions* during the network simulation. Each action requires a timestamp and can refer to either the adjustment of a parameter in a single unit or a node in the network, the stimulation of a node, or the termination of the experiment.

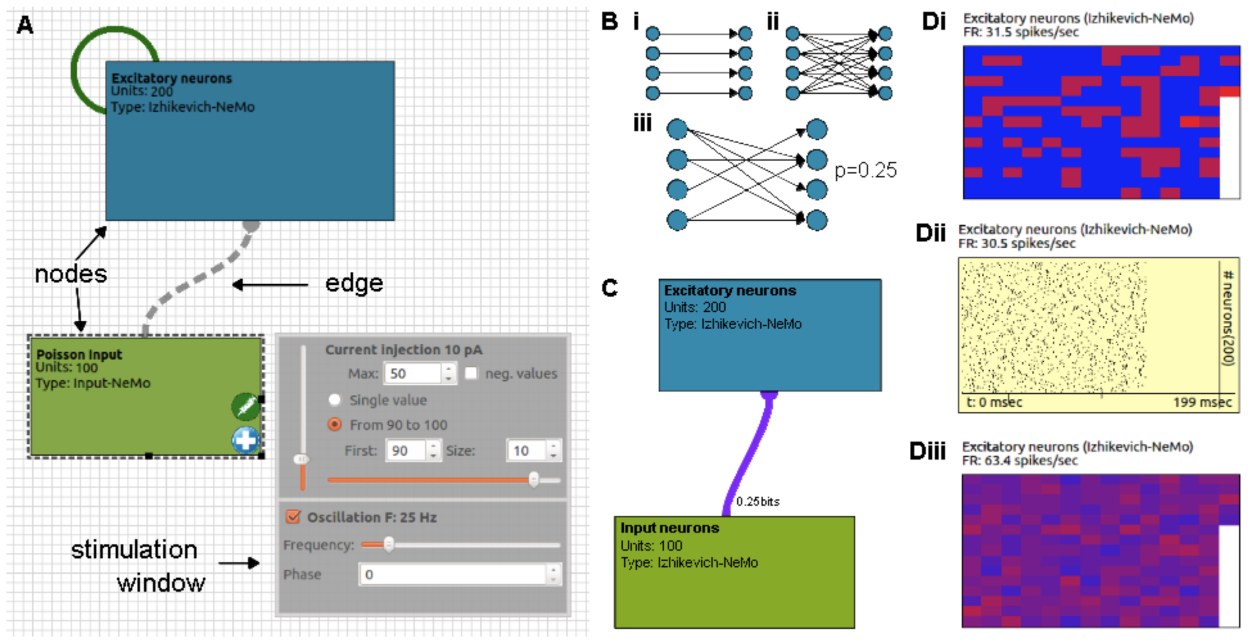


Figure 2. Nodes and edges in brain studio interface. A: Default representation of network topology. Dashed lines represent topographic and green lines plastic connections. **B:** Available connectivity patterns include **(i)** topographic **(ii)** all-to-all and **(iii)** sparse edges. **C:** Visualization of effective connectivity between two nodes. Connection colour: transfer entropy $\in [0, max]$ bits. Connection thickness: average synaptic weight. **D:** Visualizations of nodes: **(i)** Firing rates of individual neurons (using a cold to hot colour scale). **(ii)** Raster plot of neuron spikes over time.

Using this approach, all entities of an experiment can be represented internally using a compact and modular structure that includes only basic parameters and statistics. Brain studio is able to encode and save this structure in text files, using an XML-based format, under the extension `.*brn*'. Therefore, the user is provided with two options for the access and modification of experiments, either via the high-level graphical representation of the front-end, or directly via the automatically generated *brn* files. An example experiment that illustrates the format of these files, for a simple network with one node and one recurrent edge, is shown below.

```
<actions>
    <stimulate t="50" node="0" current="150" frequency="0"\>
    <stimulate t="350" node="0" current="100" frequency="90"\>
    <adjust t="650" node="0" parameter="sigma" value="7.0"\>
    <stop t="1000"\>
</actions>
<nodes>
    <Izhikevich-NeMo id="Excitatory" x="12" y="68" w="302" h="167" c="#357998">
        <version>0.001</version>
        <a>0.02</a>
        <b>0.2</b>
        <c>-65+15*RANDF()**2</c>
        <d>8-6*RANDF()**2</d>
        <eval_for_each>False</eval_for_each>
        <neurons>200</neurons>
        <sigma>1</sigma>
        <u>b*v</u>
        <v>-65</v>
    </Izhikevich-NeMo>
</nodes>
<edges>
    <NeMoSynapticPathway-NeMo id="Excitatory-Excitatory">
        <source>Excitatory</source>
        <target>Excitatory</target>
        <preFirst>0</preFirst>
        <preLast>199</preLast>
        <postFirst>0</postFirst>
        <postLast>199</postLast>
        <version>0.001</version>
        <__connectivity>probability</__connectivity>
        <__probability>0.1</__probability>
        <delay>RANDI(1,60)</delay>
        <plastic>False</plastic>
        <weight>RANDF()</weight>
    </NeMoSynapticPathway-NeMo>
</edges>
```

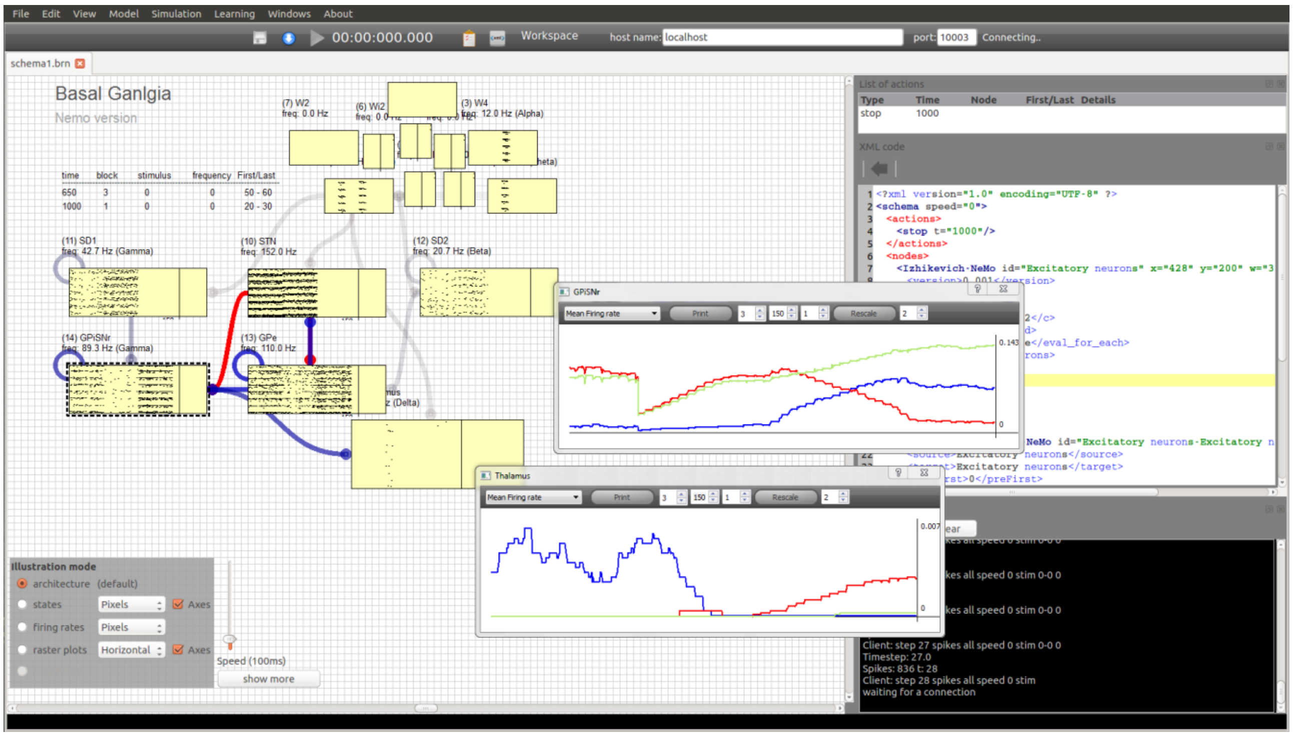
Finally, in case that a more advanced simulator is needed, brain studio has the ability to convert *brn* files either to the simulator-independent languages PyNN by Davison et al. (2008) and NeuroML 2.0 by Cannon et al. (2014), or to python code that is compatible with brian simulator.

The front-end: Designing experiments

The first part of brain studio is a tool that aims to the high-level design and graphical representation of neural networks as well as the remote monitoring of their simulation.

It is written in the programming language C++, based only on the cross-platform application framework Qt and the C++ standard library, providing high consistency and compatibility across systems. A real-time snapshot of this tool is shown in **Fig.3**.

At each point of the design process the user can toggle between the default editor mode and the real-time simulation mode. When the simulation mode is selected, the current experiment is transferred to a connected instantiation of the back-end and a number of editing options are disabled. The back-end is then instructed to initialize the network by generating all the connections and allocating the necessary memory, and to finally start the simulation.



time	block	stimulus	frequency	First/Last
650	3	0	0	50 - 60
1000	1	0	0	20 - 30

Type	Time	Node	First/Last	Details
stop	1000			

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema speed="0">
  <actions>
    <stop t="1000"/>
  </actions>
  <nodes>
    <Izhikevich-NeMo id="Excitatory neurons" x="428" y="200" w="3
```

```
kes all speed 0 stim 0-0 0
kes all speed 0 stim 0-0 0
Client: step 27 spikes all speed 0 stim 0-0 0
Timestep: 27.0
Spikes: 836 t: 28
Client: step 28 spikes all speed 0 stim
waiting for a connection
```

Figure 3. Graphical user interface of brain studio in simulation mode. A snapshot taken during a real-time simulation of a model of the basal ganglia. Left: Graphical representation of the experiment. Right: XML representation of the experiment.

During the simulation mode, different types of two-dimensional plots can be used to visualize the activity within nodes. These include raster plots, local field potential and firing rate graphs, among other visualizations. Examples of these plots are shown in **Fig.2D** and **Fig.3**.

Furthermore, a number of real-time visualizations are also available for the edges, that aim to highlight the dynamical properties of the simulation from the network perspective. The properties that are visualized include structural links, such as the absolute number of connections or connection weights, statistical dependencies (functional connectivity) and causal interactions (effective connectivity).

An example is illustrated in **Fig.2C**, where the thickness of the line between two nodes represents the average weight of all individual connections and its colour represents information flow, calculated by means of transfer entropy.

The use of a programming language with low-level capabilities resulted in a reduced utilization of computer resources. In addition, since the necessary computation for the network simulation can be carried out remotely, on a separate device, brain studio front-end is able to perform analysis of the current state of the network in real-time, and thus enable visualizations of more advanced metrics, such as transfer entropy.

Finally, adjustments in existing nodes and edges are also supported in simulation mode. The system records any real-time changes, such as parameter tuning or stimulation, and updates the network accordingly. These changes can be saved as `actions' with the time of each adjustment, relatively to the simulation, used as a timestamp.

The back-end: Running experiments

The back-end is the core component of brain studio, whose main role is to execute the simulation, independently from the user interface. Written in the programming language python, it is highly extensible and adjustable by the user, and it is designed to operate in both mobile robotic systems with limited resources, as well as in high-end workstations. Its two supported modes of operation include a *slave* mode, where the front-end has the complete control of the simulation, and a *master* mode, where *brn*-experiments are loaded and executed independently. When in the master mode, connection requests by the front-end are also accepted, which can either monitor or take control of the simulation.

The back-end also constitutes the only part of the system that holds information regarding the currently-supported mathematical models for nodes and edges. This information is retrieved by the front-end following the initial network handshake between the two. Hence, when a new neuron model is added to the brain studio, the front-end is updated automatically without the need of a new compilation. The list of the default available computational models that can be clustered in a node contains the following:

- **RateLayer-Python:** Equations for a simple rate-based neuron, implemented in python.
- **Input-NeMo:** An empty neuron unit, without dynamics, that can be forced to spike.
- **PoissonSource-NeMo:** A source that generates spikes according to a Poisson process.
- **Kuramoto-NeMo:** Delay-coupled Kuramoto oscillators, implemented in NeMo.
- **HH-NeMo:** Hodgkin-Huxley equations of neuron's ionic currents implemented in NeMo.
- **IF_curr_exp-NeMo:** Equations of the integrate-and-fire neuron model, with conductance-based synaptic input, implemented in NeMo.
- **QIF-NeMo:** Equations of the Quadratic integrate-and-fire neuron model, implemented in NeMo.
- **Izhikevich-NeMo:** Equations of the Izhikevich model, implemented in NeMo.

- **Izhikevich2007-NeMo:** Equations of the Izhikevich model, in the form where parameters have biophysiological meaning, implemented in NeMo.
- **Izhikevich2007_TCR-NeMo:** Equations of thalamo-cortical relay neurons, based on the Izhikevich model, implemented in NeMo.
- **Izhikevich2007_TI-NeMo:** Equations of thalamic interneurons, based on the Izhikevich model, implemented in NeMo.
- **IzhikevichRS-NeMo:** Equations of regular spiking neurons, based on the Izhikevich model, implemented in NeMo.

In addition, the list of the default instantiatable synaptic models that can be used as edges, along with their parameters, includes:

- **NeMoSynapticPathway-NeMo:** Synapses between groups of spiking neurons implemented in NeMo. Available parameters include weight, delay, and the existence of STDP.
- **RateToSpikeConverter-Python:** Equations that convert the current rate of a neuron to spiking input, with the same firing rate. The only available parameter is the weight of the connection.
- **SpikeToRateConverter-Python:** Equations that convert the current firing rate of a spiking neurons to rate-based input. Parameters include the window of the measured firing rate and a weight.

Finally, the list of the provided models of instantiatable nodes includes a number of extra, special cases that require further discussion. These cases regard the integration of brain-studio with external systems and demonstrate the extensibility of its core architecture. Here, a node represents an interface and the number of units it contains depends on the number of inputs and outputs supported. The currently supported nodes include:

- **Servomotor-output:** This single-unit node controls the speed of a servo motor, via pulse width modulation (PWM) on a digital port of the computational platform used. It can be used only as a connection target, and requires a SpikeToRateConverter. This node has been successfully tested on a raspberry pi computer.
- **Retina-input:** This node handles the raw input of a web camera, which transforms into spikes. The received image is initially down-sampled, and then the difference between frames is calculated for each pixel, and mapped into the firing rate of a corresponding neuron. This node can be used only as a connection source. The number of its units represents the available pixels.
- **Webots-Robot:** This node provides an interface with the 3D simulated environment of Webots, where a differentially-driven mobile robot is controlled. The communication is implemented as a second TCP/IP connection. This node can be used both as a source and a target of an edge, as it combines camera input and motor output.
- **UT2004:** This node provides an interface to control an virtual agent within the 3D environment of the video game Unreal Tournament 2004. It comprises 22 rate neurons that are directly connected with either an input or

an output of this simulated agent. The outputs include 13 neurons that encode exteroception, interoception and proprioception, while the inputs include 9 neurons that encode both motor sequences as well as single motor functions. For more details on the methodology used for this interface see Fountas et al. (2011).

Extending brain studio

Brain studio can be extended in various ways. Three main categories of extensions are currently supported including new models for nodes, new models for edges and new visualizations. The updates in all cases can be carried out entirely by changing parts of the source code of the back-end in python. After an update has finished, the front-end can be automatically notified by the changes during the initial network handshake between the two parts, and the new extensions can be displayed.

Adding new nodes A node can be added to the existing collection, in case that either a new mathematical model, or a new interface with another software system is required. In brief outline, this process involves, first, the creation of a new python class that extends the base class *Node*, the specification of the required `*fields*' which can be used to configure the model, an output array, and finally the implementation of two methods for the model initialization and update.

The required fields include any parameters or states of the simulated neurons that can be edited in the front-end, as well as any group attributes, such as the number of neurons. There is no limitation on the number of fields that can be defined, as long as they belong to one of the following supported types (bool, integer, float, integer list, float list or picklist). Using one of the aforementioned editing options, fields can be assigned a single value, a random variable or a function of random variables.

Since the only point of the interaction between the new class and the rest of the simulated network is through the input and output arrays, a new node can represent numerous types of different network components. In the case of a new mathematical model, the node can be either a group of instances of a specific neuron model with common characteristics, or a complete neural network. The second case is particularly useful when existing SNNs need to be integrated into a larger simulation. In both cases, calculations can be performed directly in python, or using another simulator such as NeMo or Brian.

In addition, the user can build general nodes that correspond to any models supported by a selected simulator. For instance, the class *NeuronGroup*() in Brian 2.0 simulator can support any group of neurons, whose differential equations are written as a python multi-line string using Brian's syntactic rules. In this case, the fields of the node will contain the attributes of the constructor of this class, along with the parameters of the neuron model.

Finally, new nodes can be added when a new interface with an external system is required. Representative examples of this category comprise the special cases of nodes in the end of the previous section.

Adding new edges As in the case of nodes, a new edge can either represent a new type of connection, or an interface with an existing simulator, that can be used to instantiate any supported types of this simulator. The necessary steps consist of the creation of a new python class that extends the provided base class *Edge*, the specification of the required `*fields*' which can be used to configure the connection, and the implementation of two methods for the model initialization and update. However, the update function here can be neglected, if both the source and the target nodes of the new edge are using the same simulator as a back-end. In this case, brain studio can redirect the implementation of this connection to the underlying simulator.

Adding new visualizations This final type of extension concerns the development of new methods to visualize the behaviour of nodes and edges in the network. Initially, the type of plot that would better illustrate the desired network property needs to be defined. The front-end of brain studio features a wide range of supported types, which are implemented through the Qt library QCustomPlot by Eichhammer (2014). Then, the user can write python code that is executed at every timestep, and performs analysis on the state of the desired network component. The resulting information can be visualized by the front-end in real-time.

In addition, the source code of the front-end can be also altered and recompiled if better computational performance is required during the analysis, or the flexibility of the back-end is not enough. The implementation of this functionality of brain studio is still at a preliminary stage.

Chapter 3: Software Installation

Brain Studio can run on all major desktop operating systems, namely Linux, OS X and Windows. The requirements currently include python (with NumPy and SciPy), as well as NeMo simulator. In order to install Brain Studio from source, follow the steps bellow:

Installing the backend

Installation of the back-end is not required as this part of the system is completely based on Python, however there are the following prerequisites:

1. Install the bespoke version of the NeMo simulator that comes with Brain Studio following the instructions that can be found in section 4 of the accompanying NeMo manual.
2. Download and install Python 2.7, NumPy and SciPy. On Windows and OS X the recommend way to install all python-related packages is through the collection Anaconda: https://www.continuum.io/downloads. For Linux systems, Python is pre-installed on the majority of home distributions, so installation of the packages NumPy and SciPy can be performed via the python module easy_install.

```
> sudo easy_install numpy

> sudo easy_install scipy
```

Building Brain Studio from source:

- Initially, download the latest Qt version (for compatible versions >= 5.0) from www.qt.io/download. This is a completely cross-platform C++ library that can be installed on all operating systems compatible with Brain Studio. During the installation, do not untick the option of installing Qt Creator along with the basic libraries.
- Once Qt is installed, the next step is to open the project file "brainstudio.pro", located in the folder "BrainStudio/", with Qt Creator. Following the instructions of Qt Creator, set a folder that it will use to build the application and configure the project.
- Finally, press run to compile.

Installing Brain Studio:

- Installation of Brain Studio is available in selected operating systems. If the current operating is supported, a script with a name "install_on_XXX", will be available in the path:

"BrainStudio/installation_scripts/"

In this case, there is no need of downloading Qt. The user will be asked to specify the path of the installation and all the necessary files and libraries will be installed automatically within this directory.

Chapter 4 : Tutorial

When Brain Studio is successfully installed, it can run by simply typing

```
> brainstudio XXX.brn
```

on a terminal. The argument XXX.brn is optional, and represents a specific file that Brain Studio will load. To run only the back-end of Brain Studio, type

```
> bs_backend
```

When Brain Studio runs for the first time, a welcome screen is presented to the user, and some initial configuration settings are requested. These include the specification of (1) the 'workspace' folder, i.e. the default folder that the Brain Studio will use to load and save new networks, and (2) the default IP and PORT that will be used for the communication with the back-end. Once these settings are submitted, the main window of Brain Studio is shown to the user. If the specified workspace folder includes a file called schemaX.brn, where X is a number between 1-9, then this file will automatically open. In the opposite case, a new network will be initialized.

Creating a new network

A network in Brain Studio consists of nodes, edges and actions. In this part of the tutorial, a new network will be designed that consists of 3 populations of neurons (nodes), connected with 5 different edges. This architecture is illustrated in **Fig.4**.

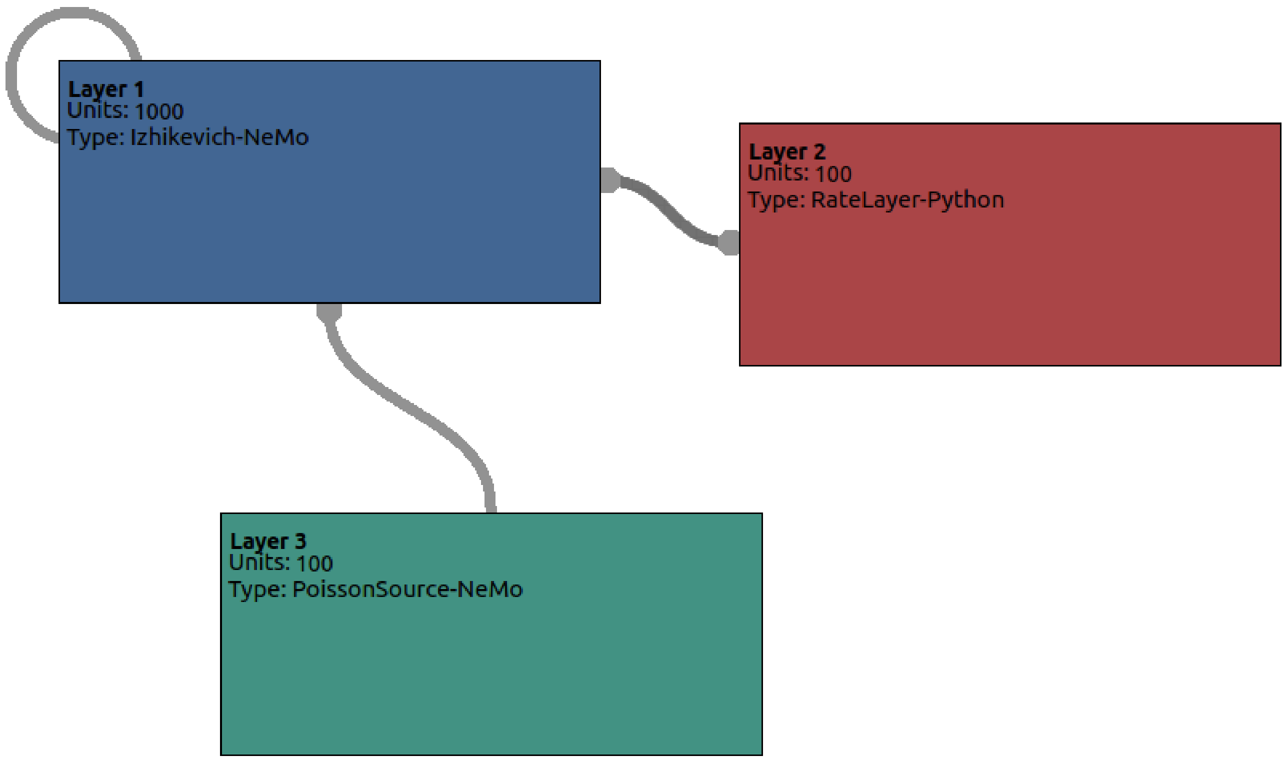


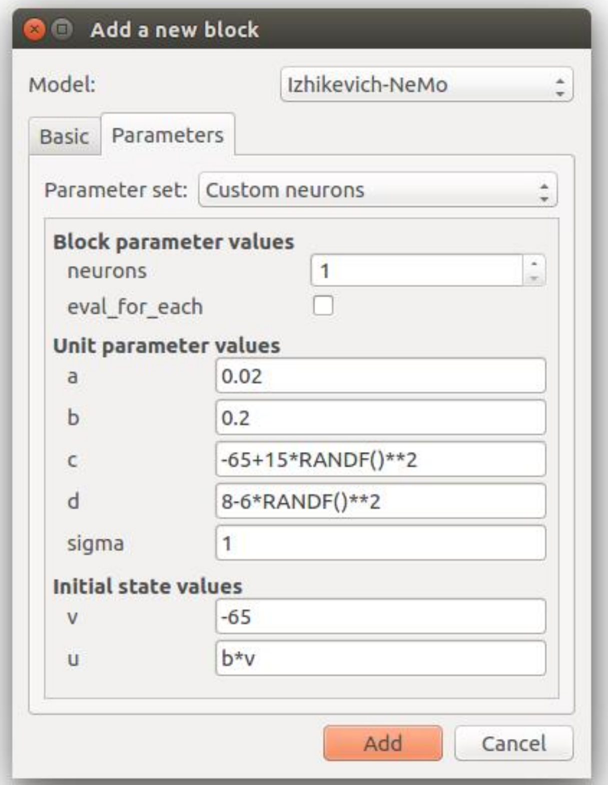
Figure 4. The architecture of this example.

To create the first node, either double-click on the area of the network schema or use *right-click → new block,* to open the "**Add a new block**" window. This window can be used to specify all the features of this block and the underlying population of neurons. Initially, the model of the neural units can be selected from the top drop-down list. Select *Izhikevich-NeMo* to model a number of spiking neurons that are governed by the Izhikevich simple model, implemented in NeMo.

On the "**Basic**" tab, the user can set all the necessary parameters that define the node as a visual entity, namely name, colour and size. Give the name "Layer 1" on this block and keep the default colour and size. The size can be easily changed later, directly from the schema.

Warning: Names of nodes should be unique, as Brain Studio is using them for indexing.

In the second tab, called "**Parameters**", the user can define all parameters that can be adjusted in the particular model used. In this case, the adjustable parameters are the number of neurons, the Izhikevich parameters a, b, c, d and sigma, as well as the initial values of the model variables v and u. All these parameters are given a default value and thus can be left as are. As shown in this example of neuron, the parameters and initial states can be either single values, or mathematic formulas. In the adjacent example parameter c is a formula. Formulas are written in standard python mathematical syntax. There are also numerous functions for mathematical, logical and random number generators from probability distributions available for use in formulas. A full list can be seen in appendix B. In the adjacent example c=-65+15*RANDF()**2, which is a random sample from a uniform distribution between 0 and 1 that has been raised to the power of 2 then multiplied by 15 and then added to -65. State variable u in the adjacent example is b*v which is the resulting value of parameter b multiplied by the resulting value of state variable v in this case this would be 0.2 multiplied by -65.



Model: Izhikevich-NeMo

Parameter set: Custom neurons

Block parameter values

neurons	1
eval_for_each	☐

Unit parameter values

a	0.02
b	0.2
c	-65+15*RANDF()**2
d	8-6*RANDF()**2
sigma	1

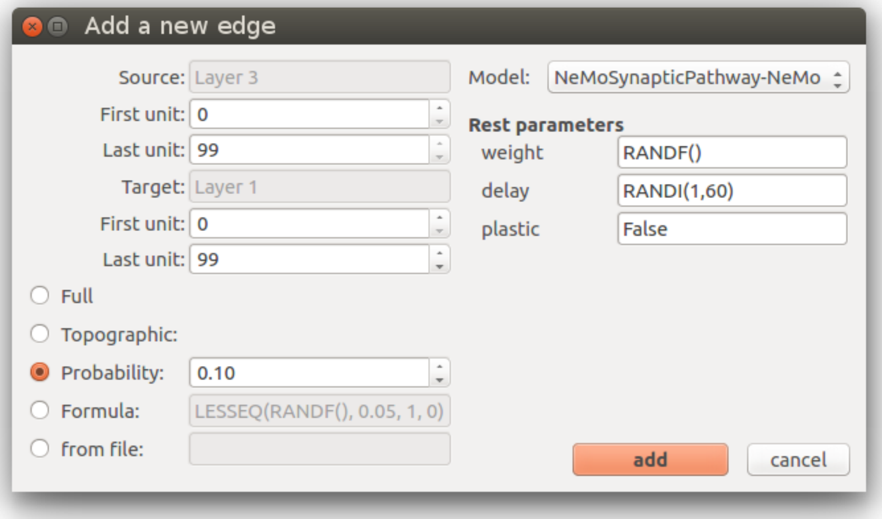
Initial state values

v	-65
u	b*v

After editing the node details press the button “Add”. This new node can be moved, resized, renamed and reassigned a new number of computational units, directly from the main schema. Change the number of Izhikevich neurons to 1000. If the user intents to view or change any other parameter, they can open a window with all settings by right-clicking on top of the node, and then selecting “properties”.

Following the same procedure, add two more nodes, one called “Layer 2” and one called “Layer 3”. Layer 2 should contain 100 rate based neurons, using the model called *RateLayer-Python*. Layer 3 should also contain 100 units, but this time each unit will be a homogeneous Poisson process with parameter λ. This model can be found under the name *PoissonSource-NeMo*. Layer 3 will serve as the input of this system.

The next step is to create edges. Select Layer 3 and drag the plus symbol showed on the lower-right corner towards Layer 1. Release the mouse while hovering above Layer 1 and the window for a new edge will appear. As in the case of nodes, this window contains a drop-down menu from which the user is able to select the model of this edge. Brain Studio automatically will detect the edges that are compatible with the particular source and target nodes, and it will filter out the rest. Select *NeMoSynapticPathway-NeMo*. To create all-to-some type of connectivity, where every neuron from Layer 3 is connected to 10% of the neurons of Layer 1, select connection by "probability" and set the value to be 0.1. In addition, set this connection to project only to the first 100 neurons of Layer 1 by changing the number in the corresponding field called "last unit". Press the "add" button.


Add a new edge

Source: Layer 3
First unit: 0
Last unit: 99
Target: Layer 1
First unit: 0
Last unit: 99

Model: NeMoSynapticPathway-NeMo

Rest parameters

weight	RANDF()
delay	RANDI(1,60)
plastic	False

Full
Topographic:
[✓ Probability:] 0.10
Formula: LESSEQ(RANDF(), 0.05, 1, 0)
From file:

Following the same procedure, set the first 100 spiking neurons of Layer 1 to be connected to all rate-based neurons, with probability 10%, and using the python-based spike-to-rate converter with window 50 ms. Next, connect all rate-based neurons to spiking neurons with indexes between 300 and 400, and again probability 10%. Finally, create a recurrent edge in Layer 1, that connects neurons with indexes between 300-400, to 700-800. The network is now ready. Press the save button.

Simulating the network

The back-end of Brain Studio can be viewed as a TCP/IP server. When the front-end is executed, it automatically searches for back-end connections, given the IP and PORT that the user has selected. In case that this connection cannot be established, Brain Studio executes an instance of the back-end internally.

After following all steps of the previous section, press the download button located next to the save button. This commands the Brain Studio to try to establish a connection with the back-end and send all the necessary data, in order to initialize the simulation. The back-end will then initialize the simulation and return a confirmation signal. The effect of this signal will be to enable the play button.

Press the play button. This will activate the simulation which at this point can be only visible via the time counter. The next step is to change the illustration mode. The available modes can be accessed from the control box which is located in the lower-left part of the schema. Select "raster plots" to visualize all spiking events of Layers 1 and 3. Since Layer 2 does not contain a model that can produce spikes, its current visualization will automatically turn to 2D plots of the state variable of each neuron within this group. If all the above steps were performed successfully, the schema in Brain Studio should be similar to **Fig.5**.

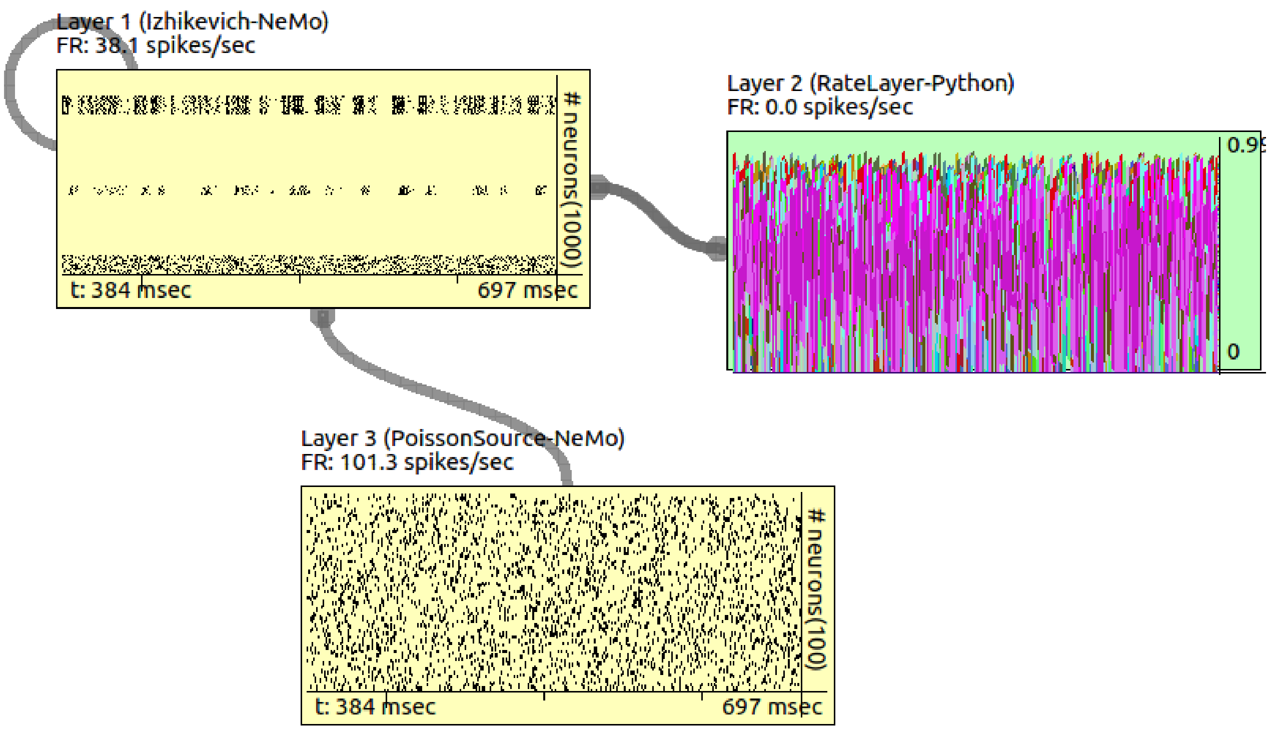


Figure 5. The network of this example being simulated.

□□□
□□□
□□□
□□□
□□□□□

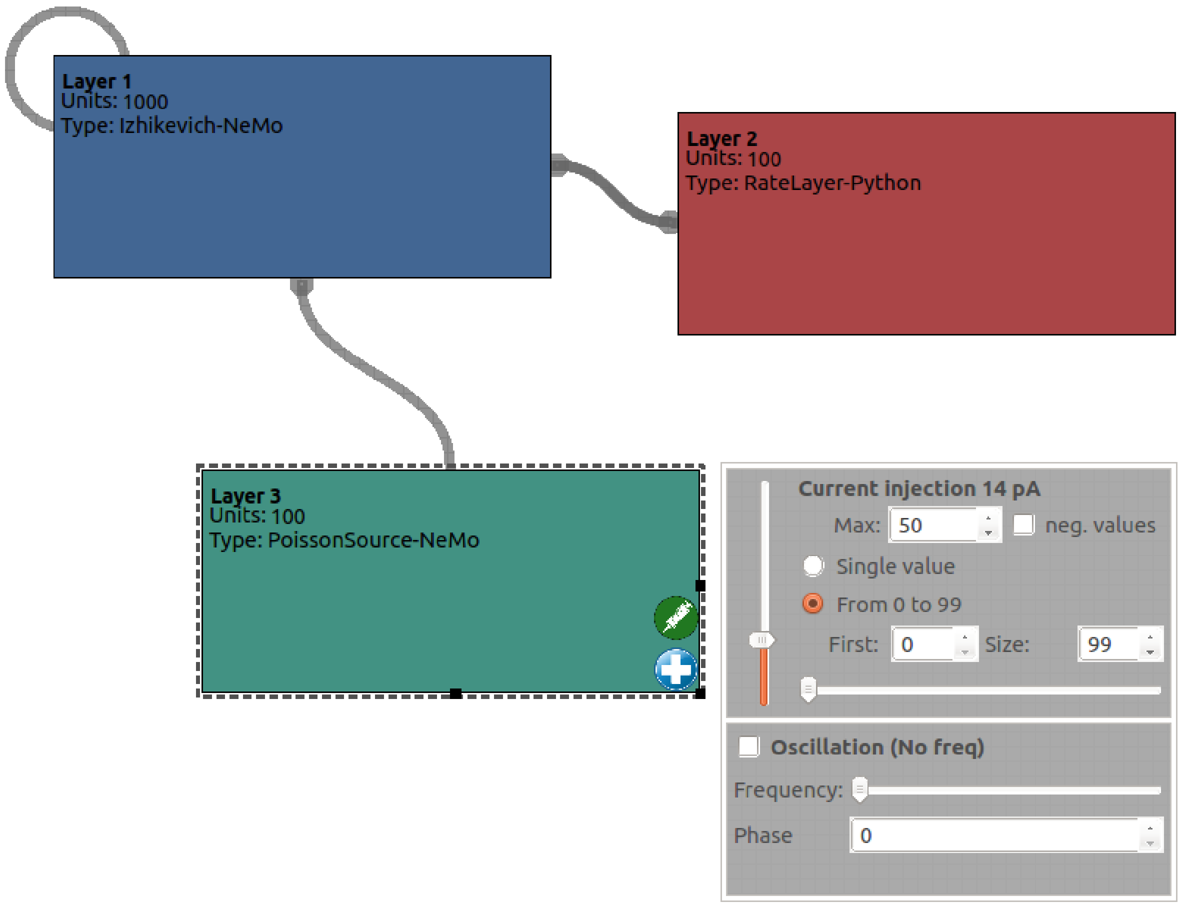
```
> bs_backend -nogui XXXX.brn
```

To simulate the last network that was loaded using the front-end, simply type

```
> bs_backend -nogui
```

Interacting with the simulation

During a simulation, the user is able to modify a number of parameters in the network and observe the result of this adjustment in real-time. This is one of the best advantages of this system since it can accelerate the process of tuning, as well as the investigation a dynamical phenomenon. The main adjustable parameters include any parameters of the mathematical models of nodes and edges, either in a whole group or individually. In addition, the user can inject stimulus to the neurons of a selected node. This can be done by clicking the icon with a syringe, in the lower-right part of a selected node. The provided stimulus can be either flat, or oscillatory and it can be applied to a particular subset of the selected group.



Current injection 14 pA
Max: 50 ☐ neg. values
○ Single value
● From 0 to 99
First: 0 Size: 99

☐ Oscillation (No freq)
Frequency:
Phase 0

Figure 6. The network of this example, when extra stimulation is applied to Layer 3.

Saving data

Simulation data can be saved in various forms. To save all spike events of the network on a new text file, click *file → save spikes.*

Chapter 5 : Back End Interface

In addition to being able to run and control simulations from the GUI via TCP/IP, and run simulations in the back end independently of the GUI, it is also possible to run and control the back end and its brain simulation via a python interface. One may wish to use this facility if for example a brain simulation is being used to control a robot. In such a situation the robot software may pass information to the brain simulation in the back end via the python interface, then ask the simulation to update one time step, and then retrieve information from the brain simulation via the back end interface after the update. By repeatedly performing these three actions the brain simulation can be used as a robot controller with the input/output interfacing being performed by the brain studio back end python interface. When controlling the back end via the python interface it is still possible to connect the GUI via TCP/IP and monitor the simulation as well as pause it.

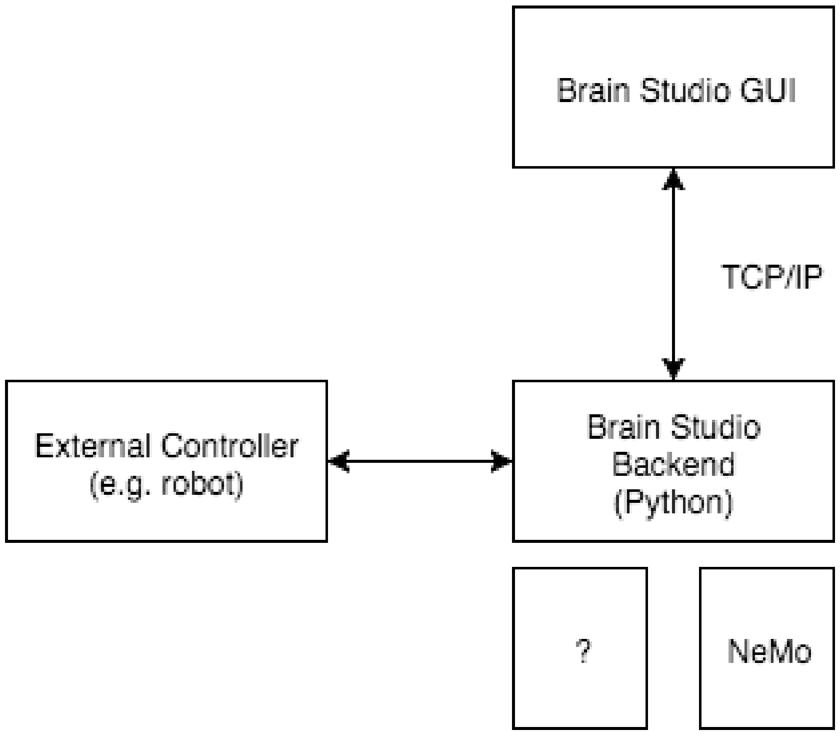

Brain Studio GUI
TCP/IP
External Controller (e.g. robot)
Brain Studio Backend (Python)
?
NeMo

Figure 7. Schematic of communication with the back end.

Creating an Interface

The back end interface is instantiated in python by importing the module BrainStudioInterface from the script BrainStdInterface.py and creating an interface object as follows:

```
> from BrainStdInterface import BrainStudioInterface
> bsInterface = BrainStudioInterface()
> bsInterface.initialize('localhost', 10000)
```

This operation will insatiate the back end simulator allowing for TCP/IP connections from the GUI. The initialize function takes two parameters which are the ip address and port number for the TCP/IP connection. In the above example the ip address is 'localhost' and the port number is 10000.

Loading a Simulation

In order to load a brn file to simulate, use the command load_file. The function takes 2 parameters. The first is the name of the brn file to load and the second is a time to wait in milliseconds for the process to complete. If the timeout is 0 the interface will wait indefinitely for the process to complete. In the following example the file test.brn is loaded and a timeout of 1000 milliseconds is waited for the process to complete.

```
> bsInterface.load_file("test.brn", 1000)
```

The function returns the following values to indicate success or failure:

1	file was loaded successfully
0	operation timed out
-1	could not load file as a GUI client was in control of the back end
-2	could not load file as the back end was destroyed

Clearing a Simulation

To clear a loaded simulation use the clear_sim function. The function takes two parameters, the time to wait for the operation to complete and a boolean value specifying whether to respawn the back end simulator. You may not wish for a new simulator to be spawed if you are in the process of exiting your application, else you will want to respawn. If the timeout is 0 the interface will wait indefinitely for the process to complete. In the following example the interface waits indefinitely for the process to complete and does not respawn a simulator:

```
> bsInterface.clear_sim(0, False)
```

The function returns the following values to indicate success or failure:

1	simulation was cleared
1	operation timed out
-1	could not clear simulation as a GUI client was in control of the back end
-2	could not clear simulation as the back end was destroyed

Updating the Simulation

To update the simulation use the step_sim function. The function takes three parameters, a float that specifies the simulation speed, a list of tuples that specify neurons and stimulus values, and a time to wait for the process to complete. If the timeout is 0 the interface will wait indefinitely for the process to complete. In the following example the speed is set to 1 and neurons 100 and 110 receive a stimulus of 10:

```
> bsInterface.step_sim(1, [(100,10),(110,10)], 0)
```

The function returns the following values to indicate success or failure:

1	simulation was updated
2	operation timed out
-1	could not update simulation as a GUI client was in control of the back end
-2	could not update simulation as the back end was destroyed

Inputting Data to the Simulation

To input data to nodes in the simulation use the set_node_data function. The function takes two parameters, a string that specifies the name of the node interfacing to, and a dictionary specifying neurons and stimulus values. The dictionary consists of 'first_neuron' which is a number specifying the first neuron to stimulate, 'last' which is a number specifying the last neuron to stimulate, and 'input' which specified the stimulus values for all neurons from 'first_neuron' to 'last'. In the following example the neurons 210 to 219 in node 'Layer 1' are stimulated:

```
> inputs = np.array([150, 150, 150, 150, 150, 150, 150, 150, 150, 150])
> args = {'first_neuron': 210, "last":219, "inputs":inputs}
> bsInterface.set_node_data("Layer 1", args)
```

Fetching Data from the Simulation

To fetch data to nodes in the simulation use the get_node_data function. The function takes one parameter, a string that specifies the name of the node interfacing to. If the node contains spiking neuron then a list of neuron numbers that spiked on the last update will be returned. If the node contains rate neurons then the rate values of all the neurons in the node will be returned. In the following example the values of all neurons in node 'Layer 1' are fetched:

```
> data = bsInterface.get_node_data("Layer 1")
```

Chapter 6 : Extending the Back End

In line with graph representations, in Brain Studio a population of neurons whatever the type, be it Izhikevich-NeMo, RateLayer-Python or what ever are considered nodes. All synaptic pathways whatever the type are considered edges. This is illustrated in figure 8.

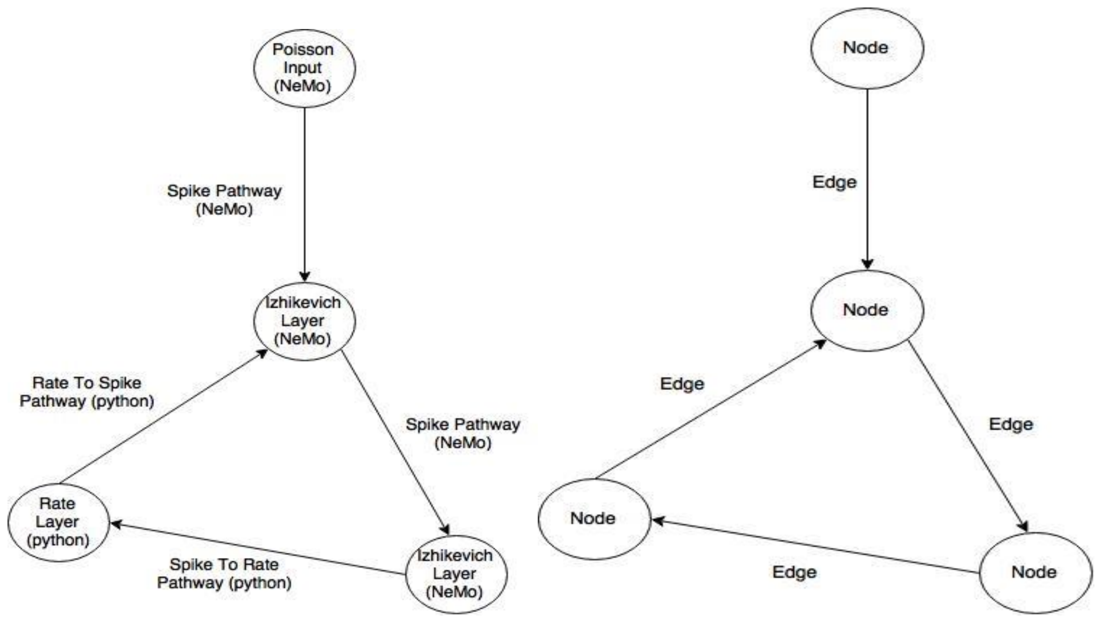


Figure 8. Networks of neuron populations and synaptic pathways are networks of nodes and edges.

This generic model is implemented in the back end architecture. All specific neuron population types, be it Izhikevich-NeMo, RateLayer-Python or what ever are derived from a Node class. All synaptic pathways types are derived from an Edge class. This is shown in figure 9. New types of nodes and edges can be added to the back end by deriving from the respective Node or Edge class and implementing the specific details of the model you wish to have. The node and edge classes that are derived must specify the fields, parameters and state variables that describe the setup of the node or edge. As this is done in a standard way the front end GUI does not need to know anything about the newly added nodes or edges except the field, parameter and state variable configuration and what type of model it is (i.e rate or spike). In order for the back end to instantiate the node or edge for simulation a few specific functions must also be implemented for the derived class.

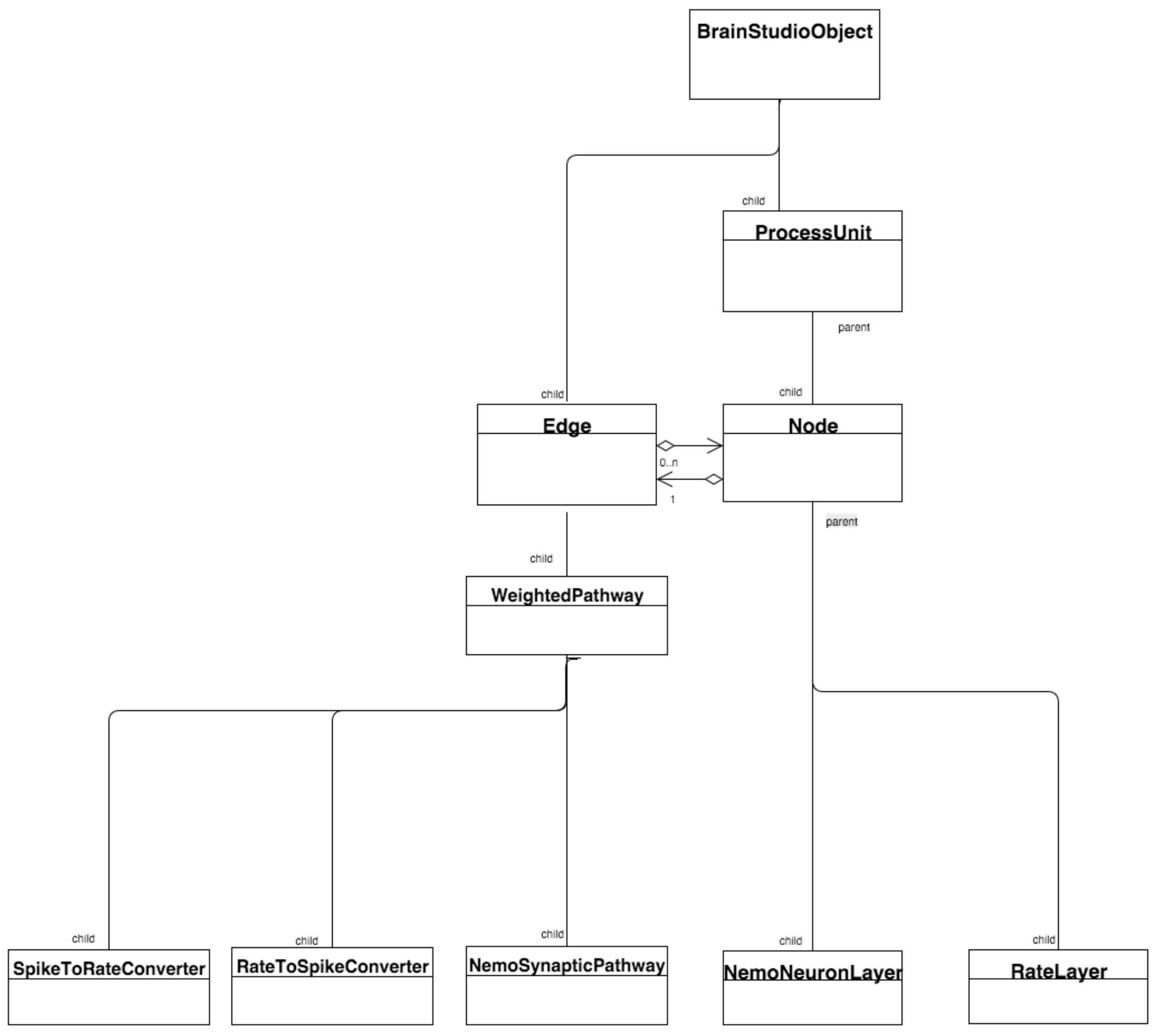


Figure 8. Back end class hierarchy of nodes and edges.

Fields

Field are lists of list that specify variable names and their possible values. Each inner list specifies a field to be edited in the GUI and stored in the brn file. When instantiating a node or edge from a brn file the field value may be used to configure the node or edge. Each inner list specifies a field, its type and possible value ranges. An inner list specifying a field is added to the list of fields by appending to ‘self.fields’. In the following example a field called neurons is added that is an integer with a minimum value of 1.

```
self.fields.append(['neurons', 'Neurons','integer', '1', ''])
```

What follows is a complete list of field types and how to define them:

Strings are specified as follows:

[field_name, destription, field_type, default_value]

field_name	: is a string containing the name of the variable
descritions	: is a string containing the text to be displayed by the field to be input in the GUI
field type	: is a string specifying the type in this case 'string'
default_value	: is a string containing the default value of the field entry in the GUI

Bools are specified as follows:

[field_name, destription, field_type, default_value]

field_name	: is a string containing the name of the variable
descritions	: is a string containing the text to be displayed by the field to be input in the GUI
field type	: is a string specifying the type in this case 'bool'
default_value	: is a string containing the default value of the field entry in the GUI

Integers are specified as follows:

[field_name, destription, field_type, min_value, max_value]

field_name	: is a string containing the name of the variable
descritions	: is a string containing the text to be displayed by the field to be input in the GUI
field type	: is a string specifying the type in this case 'integer'
min_value	: is a string containing the minimum value allowed for the field entry in the GUI. If the string is empty the is no restriction
max_value	: is a string containing the maximum value allowed for the field entry in the GUI. If the string is empty the is no restriction

Floats are specified as follows:

[field_name, destription, field_type, min_value, max_value]

field_name	: is a string containing the name of the variable
descritions	: is a string containing the text to be displayed by the field to be input in the GUI
field type	: is a string specifying the type in this case 'float'
min_value	: is a string containing the minimum value allowed for the field entry in the GUI. If the string is empty the is no restriction
max_value	: is a string containing the maximum value allowed for the field entry in the GUI. If the string is empty the is no restriction

Lists of integers are specified as follows:

[field_name, destription, field_type, min_value, max_value]

field_name	: is a string containing the name of the variable
descritions	: is a string containing the text to be displayed by the field to be input in the GUI
field type	: is a string specifying the type in this case 'intlist'
min_value	: is a string containing the minimum value allowed for the field entry in the GUI. If the string is empty the is no restriction
max_value	: is a string containing the maximum value allowed for the field entry in the GUI. If the string is empty the is no restriction

Note: returned values should be a string containing integers separated by commas.

Lists of floats are specified as follows:

[field_name, destription, field_type, min_value, max_value]

field_name	: is a string containing the name of the variable
descritions	: is a string containing the text to be displayed by the field to be input in the GUI
field type	: is a string specifying the type in this case 'floatlist'
min_value	: is a string containing the minimum value allowed for the field entry in the GUI. If the string is empty the is no restriction
max_value	: is a string containing the maximum value allowed for the field entry in the GUI. If the string is empty the is no restriction

Note: returned values should be a string containing floats separated by commas.

Picklists of floats are specified as follows:

[field_name, destription, field_type, list, default_value]

field_name	: is a string containing the name of the variable
descritions	: is a string containing the text to be displayed by the field to be input in the GUI
field type	: is a string specifying the type in this case 'picklist'
list	: is a lists of strings to be put in the pick list
default_value	: is a string containing the index of the default selection

Configurations

Configurations are dictionaries containing the parameters and state variable names and default values for these. Unlike fields parameters and state variables can be formulas that are evaluated when the node of edge is instantiated for simulation. The parameters names, state names, default parameters and default states are all lists of strings. In order to allow one derived node or edge class to support multiple configurations the configuration dictionary is indexed on neuron configuration name and each of these contains a tuple of the parameter names, state names, default parameters and default states for that particular configuration.

Warning: all parameter and state names MUST be in lower case

A configuration is added to the dictionary of configurations adding to the member variable ‘self.configurations’. What follows is an example of adding parameters and states to a configuration:

```
parameter_names = [‘a’,’b’,’c’,’d’,’sigma’]
state_names = [‘u’, ‘v’]
default_parameters = [‘0.002’,’0.2’, ‘-65+15*RANDF()**2’,’8-6*randf90**2’.’1’]
default_states = [‘-65’,’b*v’]
self.configurations[‘Izhikevich’] = (parameter_names, state_names,
default_parameters, default_states)
```

Adding New Nodes

Defining the Class

New nodes are created by writing a ‘.py’ file that implements a class derived from the Node class. The new file must be placed in the nodes directory ‘BrainStdBE/BrainStdBECore/Nodes/’. The file must specify two variables __version__ and __abstract__. __version__ is a number that specifies the version number of the derived node file. This is so the back end can check for conflicts between the current implementation of this class in the back end and any loaded brn files that require a newer version of the class. The __abstract__ variable is a Boolean that specifies if this class can be instantiated or not. If __abstract__ is True this class cannot be instantiated in a simulation but only used to derive from in order to make other classes that may themselves be instatiatable.

The class name of the new derived node must always be called BrainStdBEClass. Below is an example of how to define a new node class in a py script:

```
__version__ = 0.001
__abstract__ = False

from BrainStdBECore.Node import Node as Node
```

```
class BrainStdBEClass(Node) :
      # member functions follow here
```

The Class init Funciton

The init function should specify the implementation details required for the front end and the node. These are as follows:

self.architecture	technology used to implement the models in the class. Current values are ‘NeMo’ or ‘Python’
self.model_type	type of neuron model, either ‘spike’ or ‘rate’
self.input_field	name of the field in the list of fields that specified the number of inputs allowed
self.output_field	name of the field in the list of fields that specified the number of outputs allowed
self.units_field	name of the field in the list of fields that specified the total number of neurons
self.fields	structure that specifies the fields to be edited in the front end and used to configure the node (see fields section above)
self.configurations	structure that specifies the parameters and state variables to be edited in the front end and used to configure the node (see configurations section above)

Below is and example of an init function:

```
def __init__(self) :
      super(BrainStdBEClass,self).__init__()
      self.architecture = 'NeMo'
      self.model_type ='spike'
      self.input_field = 'neurons'
      self.output_field = 'neurons'
      self.units_field = 'neurons'
      self.fields.append(['neurons', 'Neurons','integer', '1', ''])
      self.fields.append(['eval_for_each', 'Evaluate formula for each neuron',
      'bool', 'False'])
      self.configurations[‘Izhikevich’] = ([‘a’,’b’,’c’,’d’,’sigma’] [‘u’, ‘v’],
      [‘0.002’,’0.2’, ‘-65+15*RANDF()**2’,’8-6*randf90**2’.’1’], [‘-65’,’b*v’])
```

The init function MUST first call the init function of the node super class. self.architecture is used in the front end to identify to the user what technology is implementing the model in the back end. self.model_type allows brain studio to know what nodes can connect to what edges, i.e the output of a spike node will only be able to connect to and edge that allows spike inputs.

In the above example a field called neurons is added that is in integer value that must be 1 or greater. self.input_field specified that the field ‘neuron’ is used to specify the number of inputs. . self.output_field specified that the field ‘neuron’ is used to specify the number of outputs. . self.units_field specified that the field ‘neuron’ is used to specify the number of neurons in the node. In this example the field ‘neuron’ is used for all three settings, however in some circumstances one may wish to specify a number of units and have the number of inputs and outputs less that the total number of units. It depends how you wish to set up your node.

Instantiating a Node

To insatiate a node an initialize member function is required. In this function you get the node description and use that to create whatever the node requires, usually neurons. The function should first call the super class as follows:

```
super(BrainStdBEClass,self).initialize(brain, node, args)
```

The initialize function should return the number of neurons created and takes three arguments:

brain	brain is the object that the node belongs to
node	node is the xml data from the brn file that defines the node
args	args are further arguments given by the simulator and is currently unused

The information in the node parameter is the fields and configurations data that was defined in the class init function and for which the user has used the GUI to provide specific values. There is a helper function called self.get_safely that is used to extract the fields data and throws errors in there is a problem. The function takes three arguments; the XML node, the name of the field you wish to fetch, and the type of the field. Possible field types are: 'string', 'int', 'float', 'bool', 'intlist', and 'floatlist'. All the field types correspond those that are allowed when defining self.fields in the class init function. Note that picklist results are extracted using ‘string’. The following example extracts the value of a field called ‘neurons’ that is an integer:

self.size = self.safely_get(node, 'neurons', 'int')

To calculate the results of formulas of parameters and state values you must first compile them using the construct_formulas function as follows passing in the XML node:

```
formulas = self.construct_formulas(node)
```

To get the results of formulas use the evaluate_formulas_batch function. The function takes six arguments:

formulas	the compiled formulas
node	the xml node
attributes	a list of state and parameter variables to evaluate, ie self.paramters, self.states or self.attributes which is a combination of the both.
paras	a dictionary of existing parameter values, normally this is empty at this point so pass in dict()
states	a dictionary of existing state values, normally this is empty at this point so pass in dict()

num	the number of evaluations. E.g the number of neurons if you wish to evaluate for each neuron

Below is an example of evaluating for self.size number of evaluations (in the previous example self.size was extracted from the field 'neurons'):

```
args = self.evaluate_formulas_batch(formulas, node, self.attributes, dict(),
dict(), self.size)
```

What is returned is a list of dictionaries from which you should extract the first dictionary as follows:

```
eval_results = args[0]
```

Each dictionary key will return a list of numbers of the size that was evaluated (in this case self.size). So to get the results of a state variable 'v' do the following:

```
v=eval_results['v']
```

Size Functions

It is required to provide two functions that can be queried to find out the number of input and output neurons. In the following example the inputs and output are the same size and also equal to the number of neurons in the node:

```
def get_input_size(self):
      return self.size

def get_output_size(self):
      return self.size
```

Getting and Setting Data

It is required to provide three functions to allow for the simulator to get and set data in a node. The first of these functions called get_all_data should get the output values for all neurons. In the following example the output data is stored internally as a numpy array called sel.outputs and so is converted to a list:

```
def get_all_data(self):
      return list(self.outputs)
```

The second function get data for specific neurons as such it accepts a dictionary that specified the fist and last neuron you wish to get data for. Here is anexample of how to do this:

```
def get_data(self, args):
      first_neuron = args['first_neuron']
      last = args['last']
      outputs = self.outputs[first_neuron: last+1]
      return outputs
```

The last function sets input data in a node and so also takes a dictionary specifying the first and last neuron for which you are setting data as well as the input values for all of the neurons as list. What follows is an example of this that sets a member variable called self.inputs that may be used on update later:

```
def set_data(self, args):
      inputs = np.array(args['inputs'])
      first_neuron = args['first_neuron']
      last = args['last']
      for n in range(first_neuron, last+1):
      self.inputs[n] += inputs[n-first_neuron]
```

Updating a Node

So that a node can be updated an update function must be provided. This function should first call update on the super class. Once this has been done the node internal update can be performed, for example as follows:

```
def update(self):
      super(BrainStdBEClass,self).update()
      self.outputs = np.tanh(self.inputs)
      self.inputs = np.zeros(self.size)
```

Adding New Edges

Defining the Class

New edges are created by writing a '.py' file that implements a class derived from the Edge class. The new file must be placed in the nodes directory 'BrainStdBE/BrainStdBECore/Edges/'. The file must specify two variables __version__ and __abstract__. __version__ is a number that specifies the version number of the derived edge file. This is so the back end can check for conflicts between the current implementation of this class in the back end and any loaded brn files that require a newer version of the class. The __abstract__ variable is a Boolean that specifies if this class can be instantiated or not. If __abstract__ is True this class cannot be instantiated in a simulation but only used to derive from in order to make other classes that may themselves be instatiatable.

The class name of the new derived edge must always be called BrainStdBEClass. Below is an example of how to define a new node class in a py script:

```
__version__ = 0.001
__abstract__ = False

from BrainStdBECore.Edge import Edge as Edge

class BrainStdBEClass(Edge) :
      # member functions follow here
```

The Class init Funciton

The init function should specify the implementation details required for the front end and the node. These are as follows:

self.architecture	technology used to implement the models in the class. Current values are 'NeMo' or 'Python'
self.input_model	type of neuron signal model, either 'spike' or 'rate', that can input into the edge
self.output_model	type of neuron signal model, either 'spike' or 'rate', that can output from the edge
self.fields	structure that specifies the fields to be edited in the front end and used to configure the edge (see fields section above)
self.configurations	structure that specifies the parameters and state variables to be edited in the front end and used to configure the edge (see configurations section above)

Below is and example of an init function that has no fields or configurations. As usual the super class must first be called:

```
def __init__(self) :
      super(BrainStdBEClass,self).__init__()
      self.architecture = 'NeMo'
      self.input_model = 'spike'
      self.output_model = 'spike'
```

Instantiating an Edge

To insatiate an edge an initialize member function is required. In this function you get the edge description and use that to create whatever the edge requires, usually synapses. The function should first call the super class as follows:

```
super(BrainStdBEClass,self).initialize(brain, node, args)
```

The initialize function should return the number of synapses created and takes three arguments:

brain	brain is the object that the node belongs to
node	node is the xml data from the brn file that defines the node
args	args are further arguments given by the simulator and is currently unused

The information in the node parameter is the fields and configurations data that was defined in the class init function and for which the user has used the GUI to provide specific values. Fields can be fetch using safely_get in the same way as for nodes. Formulas can also be evaluated in the same way as for nodes (see section Instantiating a Node).

Updating Edges

To update an edge you must have a get_data function. The function should ask its source (self.source) for their output data, perform whatever operation on these and then set the result in its target (self.target). See get_data and set_data functions in Node to see how is done. Below is an example in which preFirst and preLast specified the first and last neuron numbers in the source that are connected to and postFirst and postLast does the same for the target. In addition the edge has a weight matrix to multiply its sources by before giving them to the target.

```
def get_data(self, args):

      # fetch data from source

      args = {'first_neuron' : self.preFirst, 'last' : self.preLast }
      inputs = self.source.get_data(args)

      # multiply data by weight matrix

      outputs = np.matrix(inputs)*self.weights
      outputs = outputs.tolist()
      outputs = outputs[0]

      # set data in target

      args = {'inputs' : outputs, 'first_neuron' : self.postFirst, 'last' :
      self.postLast }

      self.target.set_data(args)
```

Error Handling

You can throw errors that will be caught and then displayed in an error box in the GUI as follows. Import the exception class:

```
from BrainStdBECore.BSException import BSException as BSException
```

In one of you member functions raise an error as follows:

```
raise BSException(self.get_where(), ‘YOUR MESSAGEW HERE’)
```

Appendix A : NeMo Neuron Models

Izhikevich Neurons

Parameters	a, b, c, d, σ
State variables	u, v
Dynamics	$dv/dt= 0.04v^2 + 5v + 140 - u + I + N(0, \sigma^2)$
	$du/dt= a(bv - u)$
Fire	$v \geq 30$
Reset	$v \leftarrow c$
$u \leftarrow u+d$	
Numerical integration	Euler with step size of 0.25ms

The Izhikevich neuron model consists of a two-dimensional system of ordinary differential equations defined by:

$$v = 0.04v^2 + 5v + 140 - u + I \qquad (1)$$

$$u = a(bv - u) \qquad (2)$$

with an after-spike resetting

$$\text{if } v \geq 30 \text{ mV, then } v \leftarrow c \text{ , } u \leftarrow u+d \qquad (3)$$

where v represents the membrane potential and u the membrane recovery variable, accounting for the activation of K^+ and the inactivation of Na^+ providing post-potential negative feedback to v. The parameter a describes the time scale of the recovery variable, b describes its sensitivity to sub-threshold fluctuations, c gives the after-spike reset value of the membrane potential, and d describes the after-spike reset of the recovery variable. The variables a-d can be set so as to reproduce the behaviour of different types of neurons. The term I in Equation (2) represents the combined current from spike arrivals from all presynaptic neurons, which are summed every simulation cycle.

In addition to the basic model parameters a-d and state variables u and v, the user can specify a random input current to each neuron. The input current is drawn from $N\ (0, \sigma^2)$, where σ is set separately for each neuron. If σ is set to zero, no input current is generated.

IF curr exp Neurons

Parameters	$v_{rest}, v_{reset}, c_m, \tau_m, \tau_{refrac}, \tau_{synE}, \tau_{synI}, v_{thresh}, I_{offset}$
State variables	v, I_E, I_I
Dynamics	$dv/dt= (I_E + I_I + I_{offset})/c_m + (v_{rest} - v)/\tau_m$
	$dI_E/dt= -I_E\ /\tau_{synE}$
	$dI_I/dt= -I_E\ /\tau_{synI}$
Fire	$v \geq v_{thresh}$
Reset	$v \leftarrow v_{reset}$
Refractory period	τ_{refrac}
Numerical integration	Euler

This integrate-and-fire neuron model with exponential decay implements the standard neuron model in PyNN with the same name. Time-related paramaters are expressed in terms of time steps (by default 1 ms). I_E and I_I are the incoming currents arising from exctatory and inhibitory PSPs resepectively. During the refractory period the voltage stays constant.

HH Neurons

Parameters	none
State variables	v, n, m, h, dir
Dynamics	$dv/dt =(-(g_k n^4(v-E)-g_{Na}m^3h(v-E_{Na})-g_L(u-E_L)) +I)/C$
Fire	v changes direction to decline when previously -45mv
Reset	N/A
Numerical integration	Euler with step size of 0.001ms

The Hodgkin-Huxley model is widely considered as the benchmark standard for neural models. It is based upon experiments on the giant axon of the squid. Hodgkin and Huxley found three different types of ion current: sodium (Na^+), potassium (K^+), and a leak current that consists mainly of chloride (Cl^-) ions. Different voltage-dependent ion channels control the flow of ions through the cell membrane. From their experiments, Hodgkin and Huxley formulated the following equation defining the time evolution of the model:

$$C\frac{dV}{dt} = g_K n^4 (V - E_K) - g_{Na} m^3 h (V - E_{Na}) - g_L (u - E_L)$$ □□□

$$\frac{dn}{dt} = a_n(V)(1 - n) - b_n(V)n$$ □□□

$$\frac{dm}{dt} = a_m(V)(1 - m) - b_m(V)m$$ □□□

$$\frac{dh}{dt} = a_h(V)(1 - h) - b_h(V)h$$ □□□

C is the capacitance and *n*, *m* and *h* describe the voltage dependence opening and closing dynamics of the ion channels. The maximum conductance of each channel are: g_k=120, g_{Na}=36 and g_L =0.3. The reversal potentials are set so that that E_k=-12, E_{Na}=115 and E_L=10.6. The rate functions for each channel are:

$$a_n(V) = \frac{(0.1 - 0.01\text{v})}{\exp(1.0 - 0.1\text{v}) - 1.0}$$ □□□

$$\beta_n(V) = 0.125\exp\left(\frac{-v}{80.0}\right)$$ □□□

$$a_m(V) = \frac{2.5 - 0.1\text{v}}{\exp(2.5 - 0.1\text{v}) - 1.0}$$ □□□□

$$\beta_m(V) = 4.0\exp\left(\frac{-v}{18.0}\right)$$ □□□□

$$\alpha_h(V) = 0.07\exp\left(\frac{-v}{20.0}\right)$$ □□□□

$$b_h(V) = \frac{1.0}{\exp(3.0 - 0.1\text{v}) + 1.0}$$ □□□□

The HH model uses conductance synapses, and so uses reversal potentials to further scale incoming spikes. The latter model is as follows:

$$I_j(t) = \sum_i (Rev - V_j) w_{ij} \sum_k^n \delta(t - d_{ij} - t_{i,k}) \qquad (14)$$

where $I_j(t)$ is the input to neuron j and time t. w_{ij} is the synaptic weight from neuron i to neuron j, and d_{ij} is the synaptic delay from neuron i to neuron j. A list of the all n spikes produced from neuron i during a simulation are denoted by their spike times $t_{i,k}$, where $k=1,2.....n$. δ is a delta function applied to $t-d_{ij}-t_{i,k}$, such that adjusting the current time t by the synaptic delay d_{ij} identifies the spike production time at neuron i for which a spike is due to arrive at neuron j at time t. If $t_{i,k}$ matches this spike time then the delta function produces an output value 1. *Rev* which is the reversal potential, and V_j, which is the voltage of the target neuron. The reversal potentials for the model are set to the same values in all experiments. For excitatory inputs the reversal potential is set to 0mV, and for inhibitory inputs the reversal potential is -70mV.

QIF Neurons

Parameters	a
State variables	v
Dynamics	dv/dt= a×v×(v-1)+I
Fire	v ≥ 1
Reset	v←0
Numerical integration	Range Cutter step size of 0.1ms

The Quadratic Itergrate and Fire model displays Type I neuron dynamics. The time evolution of the neuron membrane potential is given by:

$$\frac{dV}{dt} = \frac{1}{\tau}(V - V_r)(V - V_t) + \frac{I}{C} \qquad □15□$$

where V is the membrane potential, with Vr and Vt being the resting and threshold values respectively. C is the capacitance of the cell membrane. τ is the membrane time constant such that $\tau = RC$ with R being the resistance. I represents a depolarizing input current to the neuron.

An action potential occurs when V reaches a value V_{peak} at which point it is reset to value V_{reset}. The Nemo uses values $V_r = V_{reset} = 0$ and $V_t = V_{peak} = 1$, which reduces equation (15) to:

$$\frac{dV}{dt} = aV(V - 1) + \frac{I}{C} \qquad (16)$$

Here $a = \frac{1}{t}$ and by default is set to the value 2.

Poisson spike source

`Parameters`	λ
`State variables`	`none`
`Dynamics`	`none`
`Fire`	`urand < λ`
`Reset`	`N/A`

A Poisson spike source generates spikes according to a Poisson process with parameter λ. During a single simulation cycle a Poisson spike source generates either zero or one spike (with probability λ). Inter-spike intervals are thus never smaller than the time step, and λ must be set taking into account the size of the time step.

Input neuron

`Parameters`	`none`
`State variables`	`none`
`Dynamics`	`none`
`Fire`	`user-specified`
`Reset`	`N/A`

An input neuron has no internal dynamics, but can be forced to fire (via the step function). It can thus be used for neurons providing input to the network, e.g. from a sensor.

Appendix B : Functions For Use In Formulas

For the purpose of writing formulas for parameter and state variables, the following functions have been provided:

Logical functions:

```
EQUAL,
GREATER,
GREATEREQ,
LESS,
LESSEQ,
NOTEQUAL
```

Mathematical functions:

```
CEIL,
EXP,
FLOOR,
LOG,
ROUND
```

Probability distribution random number generators:

```
BINOMIAL,
CAUCHY,
CHISQUARE,
DIRICHLET,
EXPONENTIAL,
GAMMA,
GEOMETRIC,
GUMBEL,
LAPLACE,
LOGISTIC,
LOGNORMAL,
RANDF,
RANDN
RANDI,
NORMAL,
PARETO,
POISSON,
POWER,
RAYLEIGH,
STUDENTT,
UNIFORM,
VONMISES,
WALD,
WEIBULL,
ZIPF
```

The functions are defined in alphabetical order on the following pages.

BINOMIAL(n, p)

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

Parameters: **n** : float (but truncated to an integer)

parameter, >= 0.

p : float

parameter, >= 0 and <=1.

Returns: **samples** : scalar

where the values are all integers in [0, n].

Notes

The probability density for the binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where *n* is the number of trials, *P* is the probability of success, and *N* is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product *p*n <=5*, where *p* = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

CAUCHY()

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

Parameters: None

Returns: **samples** : scalar

The drawn sample.

Notes

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi\gamma\left[1 + \left(\frac{x-x_0}{\gamma}\right)^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma=0$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

`CEIL`(*x*)

Return the ceiling of the input, element-wise.

The ceil of the scalar x is the smallest integer i, such that $i >= x$. It is often denoted as [x].

Parameters: **x** : scalar

Input data.

Returns: **y** : scalar

The ceiling of x.

`CHISQUARE`(*df*)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

Parameters: **df** : int

Number of degrees of freedom.

Returns: **output** : scalar

Sample drawn from the distribution.

Notes

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

`DIRICHLET`(*alpha*)

Draw samples from the Dirichlet distribution.

Draw *size* samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. Dirichlet pdf is the conjugate prior of a multinomial in Bayesian inference.

Parameters: alpha : array

Parameter of the distribution (k dimension for sample of dimension k).

Returns: **samples** : scalar

The drawn sample, of shape (size, alpha.ndim).

<u>Notes</u>

$$X \approx \prod_{i=1}^{k} x_i^{\alpha_i - 1}$$

Uses the following property for computation: for each dimension, draw a random sample y_i from a standard gamma generator of shape *alpha_i*, then $X = \frac{1}{\sum_{i=1}^{k} y_i}(y_1, \ldots, y_n)$ is Dirichlet distributed.

`EXP`(x)

Calculate the exponential of all elements in the input array.

Parameters: **x** : scalar

Input values.

Returns: **out** : scalar

Output array, element-wise exponential of x.

<u>Notes</u>

The irrational number `e` is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, `ln` (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, EXP `(x)` is always positive.

For complex arguments, `x = a + ib`, we can write $e^x = e^a e^{ib}$. The first term, e^a, is already known (it is the real argument, described above). The second term, e^{ib}, is $\cos b + i \sin b$, a function with magnitude 1 and a periodic phase.

`EXPONENTIAL`(*scale*)

Draw samples from an exponential distribution.

Parameters: scale : float

The scale parameter, $\beta = 1/\lambda$.

Returns: **samples :** scalar

The drawn sample.

Notes

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta}\exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. *β* is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution .

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms, or the time between page requests to Wikipedia.

EQUAL(a,b,c,d)

Tests if a==b.

Parameters: **a** : scalar

b : scalar

c : scalar

d : scalar

Input values.

Returns: **out** : scalar

if a==b returns c

else returns d

`FLOOR`(x)

Return the floor of the input, element-wise.

The floor of the scalar x is the largest integer i, such that $i <= x$. It is often denoted as $[x]$.

Parameters:	**x** : array_like Input data.
Returns:	**y** : scalar The floor of each element in x.

`GAMMA`(*shape, scale*)

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated “k”) and *scale* (sometimes designated “theta”), where both parameters are > 0.

Parameters: **shape** : scalar > 0

The shape of the gamma distribution.

scale : scalar > 0, optional

The scale of the gamma distribution. Default is equal to 1.

Returns: **out** : float

Returns drawn sample.

<u>Notes</u>

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where *K* is the shape and *θ* the scale, and *Γ* is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

`GEOMETRIC`(*p*)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1-p)^{k-1}p$$

where *p* is the probability of success of an individual trial.

Parameters: **p** : float

The probability of success of an individual trial.

Returns: **out** : scalar

Sample from the geometric distribution*ze*.

GREATER(a,b,c,d)

Tests if a>b.

Parameters: **a** : scalar

b : scalar

c : scalar

d : scalar

Input values.

Returns: **out** : scalar

if a>b returns c

else returns d

GREATEREQ(a,b,c,d)

Test if a>=b.

Parameters:	**a** : scalar
	b : scalar
	c : scalar
	d : scalar
	Input values.
Returns:	**out** : scalar
	if a>=b returns c
	else returns d

`GUMBEL`(*loc*, *scale*)

Draw samples from a Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

Parameters: **loc** : float

The location of the mode of the distribution.

scale : float

The scale parameter of the distribution.

Returns: **samples** : scalar

Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with “exponential-like” tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a “fat-tailed” distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

`LAPLACE`(*loc*, *scale*)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

Parameters: loc : float, optional

The position, μ, of the distribution peak.

scale : float, optional

λ, the exponential decay.

Returns: **samples** : scalar

Notes

It has the probability density function

$$f(x;\mu,\lambda) = \frac{1}{2\lambda}\exp\left(-\frac{|x-\mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in economics and health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

LESS(a,b,c,d)

Tests if a<b.

Parameters:	**a** : scalar
	b : scalar
	c : scalar
	d : scalar
	Input values.
Returns:	**out** : scalar
	if a<b returns c
	else returns d

LESSEQ(a,b,c,d)

Tests if a<=b.

Parameters: **a** : scalar

b : scalar

c : scalar

d : scalar

Input values.

Returns: **out** : scalar

if a<=b returns c

else returns d

`LOG`(*x*)

Natural logarithm, element-wise.

The natural logarithm `LOG` is the inverse of the exponential function, so that *log(exp(x)) = x*. The natural logarithm is logarithm in base `e`.

Parameters:	**x** : scalar Input value.
Returns:	**y** : scalar The natural logarithm of *x*, element-wise.

Notes

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *exp(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, `LOG` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `LOG` is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. `LOG` handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

`LOGISTIC`(*loc*, *scale*)

Draw samples from a logistic distribution.

Samples are drawn from a logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

Parameters: **loc** : float

scale : float > 0.

Returns: **samples** : scalar

where the values are all integers in [0, n].

<u>Notes</u>

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where μ = location and S = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

`LOGNORMAL`(*mean*, *sigma*)

Draw samples from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

Parameters: **mean** : float

Mean value of the underlying normal distribution

sigma : float, > 0.

Standard deviation of the underlying normal distribution

Returns: **samples** : float

The desired sample.

Notes

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x\sqrt{2\pi}} e^{\left(-\frac{(\ln(x)-\mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

RANDF()

Return random floats in the half-open interval [0.0, 1.0).

Results are from the “continuous uniform” distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of RANDF by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

Parameters: None

Returns: **out** : scalar

Random float.

`RANDN`()

Return a sample from the “standard normal” distribution.

If positive, int_like or int-convertible arguments are provided, R`ANDN` generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate “normal” (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

Parameters: None

Returns: **Z** : float

A floating-point sample from the standard normal distribution, or a single such float if no parameters were supplied.

`RANDI`(*low*, *high*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the “discrete uniform” distribution in the “half-open” interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Parameters: **low** : int

Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high : int, optional

If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

Returns: **out** : int

A single random int in range low to high.

`ROUND`(a, d)

Round an array to the given number of decimals.

Parameters: **a** : scalar to round

d : int

decimal places to round to

Returns: **out** : scalar

`NORMAL`(*loc*, *scale*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [R250], is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution .

Parameters: **loc** : float

Mean ("centre") of the distribution.

scale : float

Standard deviation (spread or "width") of the distribution.

Returns: **samples :** scalar

The drawn sample.

Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x+\sigma$ and $x-\sigma$).

NOTEQUAL(a,b,c,d)

Tests if a!=b.

Parameters: **a** : scalar

b : scalar

c : scalar

d : scalar

Input values.

Returns: **out** : scalar

if a!=b returns c

else returns d

`PARETO`(a)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the “80-20 rule”. In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

Parameters: shape : float, > 0.

Shape of the distribution.

Returns: samples : scalar

The drawn sample.

Notes

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where *a* is the shape and *m* the scale.

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge. It is one of the so-called “fat-tailed” distributions.

`POISSON`(*lam*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the binomial distribution for large N.

Parameters: **lam** : float or sequence of float

Expectation of interval, should be >= 0. A sequence of expectation intervals must be broadcastable over the requested size.

Returns: **samples** : scalar

The drawn sample.

<u>Notes</u>

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution *f(k;λ)* describes the probability of *k* events occurring within the observed interval λ.

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

POWER(*a*)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

Parameters: **a** : float

parameter, > 0

Returns: **sample** : scalar

The returned samples lie in [0, 1].

Raises: **ValueError**

If a < 1.

<u>Notes</u>

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

RAYLEIGH(*scale*)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

Parameters: **scale** : scalar

Scale, also equals the mode. Should be >= 0.

Returns: **samples :** scalar

The drawn sample.

Notes

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

`STUDENTT`(*df*)

Draw samples from a standard Student's t distribution with *df* degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution NORMAL.

Parameters: df : int

Degrees of freedom, should be > 0.

.

Returns: **samples** : scalar

Drawn sample.

Notes

The probability density function for the t distribution is

$$P(x,df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\,\Gamma(\frac{df}{2})}\left(1+\frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

`UNIFORM`(*low*, *high*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by `UNIFORM`.

Parameters: **low** : float, optional

Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high : float

Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

Returns: **out** : scalar

Drawn sample, with shape *size*.

<u>Notes</u>

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

`VONMISES`(*mu*, *kappa*,)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

Parameters: **mu** : float

Mode ("center") of the distribution.

kappa : float

Dispersion of the distribution, has to be >=0.

Returns: **samples** : scalar

The returned sample, which is in the interval [-pi, pi].

Notes

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and K the dispersion, and $I_o(K)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

`WALD`(*mean*, *scale*)

Draw samples from a Wald, or inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian. Some references claim that the Wald is an inverse Gaussian with mean equal to 1, but this is by no means universal.

The inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

Parameters: **mean** : scalar

Distribution mean, should be > 0.

scale : scalar

Scale parameter, should be >= 0.

Returns: **samples** : scalar

Drawn sample, all greater than zero.

<u>Notes</u>

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 \cdot x}}$$

As noted above the inverse Gaussian distribution first arise from attempts to model Brownian motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

`WEIBULL`(*a*)

Draw samples from a Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter *a*.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

Parameters: **a** : float

Shape of the distribution.

Returns: **samples** : scalar

Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where *a* is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

`ZIPF`(a)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter $a > 1$.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

Parameters: **a** : float > 1

Distribution parameter.

Returns: **samples** : scalar

The returned samples are greater than or equal to one.

Notes

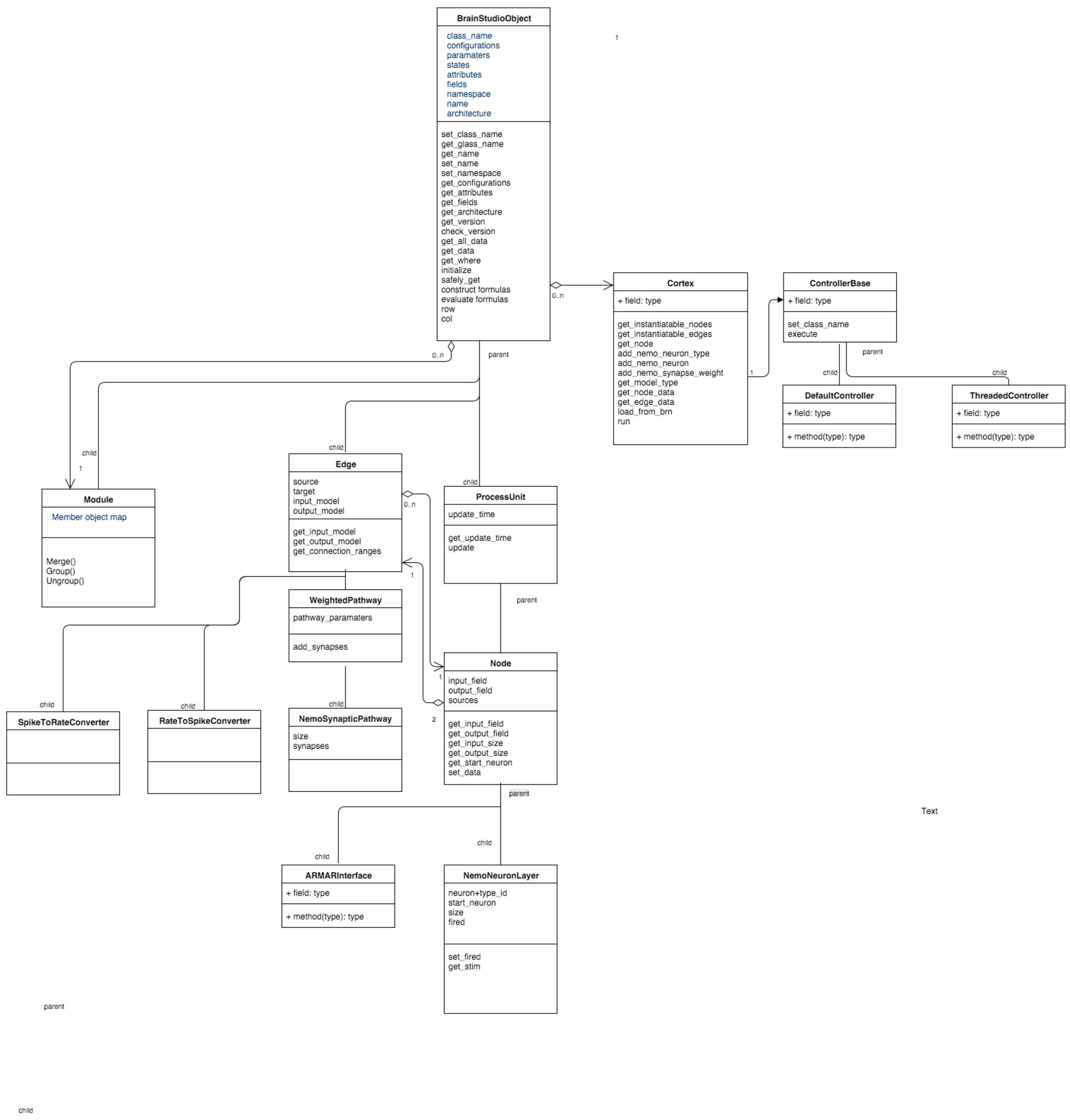
The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\varsigma(a)},$$

where ς is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Appendix C : Backend Architecture



References

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., Choo, X., Voelker, A. R., and Eliasmith, C. 2013. “Nengo: a Python tool for building large-scale functional brain models.” Frontiers in neuroinformatics 7.

Bower, J. M. and Beeman, D. 2012. The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System. Springer Science & Business Media.

Brette, R. and Goodman, D. F. 2012. “Simulating spiking neural networks on GPU.” Network: Computation in Neural Systems 23 (4): 167–182.

Cannon, R. C., Gleeson, P., Crook, S., Ganapathy, G., Marin, B., Piasini, E., and Silver, R. A. 2014. “LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2.” Frontiers in neuroinformatics 8.

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. 2008. “PyNN: a common interface for neuronal network simulators.” Frontiers in neuroinformatics 2.

Eichhammer, E. 2014. “QCustomPlot.” http://www.qcustomplot.com/.

Fidjeland, A. K. and Shanahan, M. P. 2010. “Accelerated simulation of spiking neural networksusing GPUs.” In Neural Networks (IJCNN), The 2010 International Joint Conference on, 1–8. IEEE.

Fountas, Z. 2011. “Spiking neural networks for human-like avatar control in a simulated environment.” Master’s thesis, MSc thesis, Imperial College London.

Gamez, D. 2007. “Spikestream: a fast and flexible simulator of spiking neural networks.” In Artificial Neural Networks–ICANN 2007, 360–369. Springer.

Geit, W. V., Schutter, E. D., and Achard, P. 2008. “Automated neuron model optimization techniques: a review.” Biological cybernetics 99 (4-5): 241–251.

Goodman, D. and Brette, R. 2008. “Brian: a simulator for spiking neural networks in Python.” Frontiers in neuroinformatics 2.

Hines, M. L. and Carnevale, N. T. 1997. “The NEURON simulation environment.” Neural computation 9 (6): 1179–1209.

Izhikevich, E. M. 2007. Dynamical systems in neuroscience. MIT Press, Cambridge, MA.

Song, S., Miller, K. D., and Abbott, L. F. 2000. “Competitive Hebbian learning through spike-timing-dependent synaptic plasticity.” Nature neuroscience 3 (9): 919–926.

Stimberg, M., Goodman, D. F., Benichoux, V., and Brette, R. 2014. “Equation-oriented specification of neural models for simulations.” Frontiers in neuroinformatics 8.