Simbrain 3

A Flexible, Visually-Oriented Neural Network Simulator

Zach Tosi^{a,1}, Jeff Yoshimi^b

^a*Indiana University*

^b*University of California, Merced*

Abstract

Simbrain 3.0 is a software package for neural network design and analysis, which emphasizes flexibility (arbitrarily complex networks can be built using a suite of basic components) and a visually rich, intuitive interface. These features support both students and professionals. Students can study all of the major classes of neural networks in a familiar graphical setting, and can easily modify simulations, experimenting with networks and immediately seeing the results of their interventions. With the 3.0 release, Simbrain supports models on the order of thousands of neurons and a million synapses. This allows the same features that support education to support research professionals, who can now use the tool to quickly design, run, and analyze the behavior of large, highly customizable simulations.

1. Introduction:

Simbrain 3.0 (<http://www.simbrain.net/>), a major revision and overhaul of Simbrain 2.0 [47], is an open source neural network tool¹ which can simulate many different network architectures using a visually rich and intuitive interface. This makes it ideal for education and rapid prototyping. Simbrain 3.0 has greatly improved performance: it can handle thousands of model neurons and hundreds of thousands of synaptic connections on a standard desktop PC. Thus Simbrain is increasingly viable as a research tool, particularly in computational neuroscience.

Simbrain's flexibility is based on its modular "tool-kit" structure and scripting abilities. In the last few decades, artificial neural networks (ANNs) have become prominent in many fields, including machine learning, neuroscience, and psychology. The thousands of models that have been developed in these fields are largely based on a common core of basic components and functions. Simbrain supports most of these core components as graphical elements and allows them to be combined in arbitrary ways. Thus many existing models can be developed "out of the box" using Simbrain's graphical user interface (GUI). The intention is to create a sandbox for maximally permissive experimentation. In cases where more custom functions are needed, Simbrain provides a powerful scripting interface. In these ways, just about any modern neural network model at the level of abstraction of point neurons can be implemented in Simbrain.

Simbrain's ease of use is based on its graphical interface, which allows individual neurons and synapses, and groups of neurons and synapses, to be created, edited, and connected together using a familiar point and click interface, as well as a set of keyboard shortcuts that, in our experience, are quickly mastered. One of the goals of Simbrain is to make neural networks as accessible (and manipulatable) as possible, in terms of their behavior, dynamics, analysis, and interactions with an environment. Users can modify parameters of running simulations and observe the effects of these interventions in real-time.

These features make it easy to use Simbrain in a broad range of educational settings, with students of different ages and levels of expertise. In fact, an informal design goal of the team is to make it possible for children and young adults to build and study neural networks. Simbrain has been used at the university level at the University of Sydney (Australia), LMU Munich (Germany), University of Indiana Bloomington (USA), and at the University of California, Merced (USA), among others, and the team has been seeking funding to develop educational modules for use in K-12 settings.

Email addresses: ztosi@umail.iu.edu (Zach Tosi), jyoshimi@ucmerced.edu (Jeff Yoshimi)

¹Using the GNU general public license.

However, Simbrain is not exclusively an educational tool. It is also well-suited to research professionals developing more complex models. It has been extensively used by the second author to study the dynamics of small networks embedded in virtual environments [20, 48], but with the improvements of 3.0 the possibilities have expanded. In as yet unpublished work, the first author has drawn on Simbrain’s enhanced performance capacities and extensibility to study the mechanisms of plasticity that are needed to account for the behavior and structure of cortical microcircuits observed *in vitro* and *in vivo*. In this kind of work, instantaneous visualizations of network data can inspire new hypotheses and lines of research, and make users immediately aware of information about a network which might otherwise go unnoticed, due to the often painstaking process of creating useful data visualizations. In fact, a chance observation of changes in synaptic structure (via Simbrain’s histogram plot) was the impetus for the first author’s line of research.

Thus in a research/professional context this rich user feedback loosely promotes a development cycle involving some or all of the following steps: 1) Come up with an idea for a novel ANN architecture, 2) Implement the idea using the Simbrain GUI (and possibly scripting), 3) Refine rapidly using the GUI’s visual feedback, 4) Fine tune the simulation and potentially develop it without the GUI, using Simbrain as a library.

Simbrain is programmed in Java and thus runs on any machine with a Java Virtual Machine (JRE/JDK 8 or higher required). Neural network graphics are based on the Piccolo library (<http://www.cs.umd.edu/hcil/piccolo/>), a zoomable scene-graph based library built on top of the Java2D API.² The source code is hosted on git-hub (<https://github.com/simbrain/simbrain>) and efforts have been made to produce a clean, readable codebase.

Simbrain has been in continuous development since the late 1990s. Its original purpose was to provide a visually oriented framework for studying neural networks (few such programs were available at the time). Since then many more neural network packages have emerged, including several that feature rich visualization capacities [4, 6, 12, 16].³ Two of the most similar, in terms of providing a point-and-click GUI for creating neural networks, are Emergent and Nengo. However, these packages have been designed around specific computational frameworks (the Leabra framework in the case of Emergent; the Neural Engineering Framework in the case of Nengo). Though both frameworks are flexible (e.g. backpropagation is available in Emergent), each focuses on a specific class of neuro-computational models. A primary design goal of Simbrain, by contrast, has been to provide a domain general framework for creating and combining arbitrary neuron, synapse, and network models. Programs like Brian and Nest can produce detailed visualizations, but are not fundamentally GUI-based programs. With the emergence of HTML5, beautiful visualizations are more readily accessible directly in the browser, prompting some to create web-based interactive neural networks.⁴ However, thus far web-based visualizations have been oriented towards single demonstrations rather than general purpose simulation frameworks.

Simbrain’s basic graphical user interface is a desktop (see Fig. 1) with components (neural networks, virtual environments, graphs, etc.) that can be coupled to one another.⁵ Network panels allow users to build networks and watch the changing dynamics of neurons and (in some cases) synapses in real-time, alongside corresponding changes in graphs, tables, and virtual environments. The GUI can be disabled (or other GUI objects hidden) to increase a simulation’s computational speed and efficiency. Simbrain’s logical code is separate from its GUI code and it can be used as a library independently of the GUI.

The shift from Simbrain 2.0 to 3.0 focused on scope and scalability/performance. Improved performance made larger-scale simulations possible, which in turn made it possible to add new network types and training algorithms. 3.0 introduces aggregated groups of neurons and synapses (“neuron groups” and “synapse groups”) which provide a convenient way to manipulate many neurons or synapses at once. Simbrain’s data analysis and visualization software has also been improved. Many of the enhancements have focused on

²Piccolo’s main design features have been incorporated in to Java FX (the latest iteration of Java’s graphics platform), and plans are to gradually port Simbrain to Java FX.

³Also see [26] and the list here: https://grey.colorado.edu/emergent/index.php/Comparison_of_Neural_Network_Simulators.

⁴Some examples: <http://cs.stanford.edu/people/karpathy/convnetjs/>, <http://ncase.me/neurons/>, and <http://scs.ryerson.ca/~aharley/vis/>.

⁵All of the screenshots in this paper can be easily reproduced, using simulations that are included with the latest 3.0.1 release of Simbrain. Figs. 1 and 2 correspond to workspaces *backpropLetters.zip* and *spkExamples.zip*. Figs. 5, 6, 7, 9, and 10 correspond to scripts *CorticalBranching.bsh*, *RealNeuralNetAdEx.bsh*, *bratzenberg.bsh*, *elmanPhonemes.bsh*, and *SORN.bsh*.

Figure 1: The Simbrain Desktop. Displayed here are a network (top-middle), documentation viewer (top-left) input table (bottom-left), and a synapse group editor dialog (right) set to display a histogram of the weight values of a synapse group.

supporting more biologically plausible neural network models, allowing for the simulation of structures like cortical microcircuits in Simbrain.

This article overviews the full scope of Simbrain 3.0’s capabilities and features. We first consider its core elements: neurons, synapses, plasticity rules, synaptic transmission rules, neuron groups, synapse groups, and subnetworks. We then consider its performance features, graphical interface design, worlds, and scripting. Examples of applications are given throughout. We end by discussing future directions for Simbrain.

2. Core Architecture

2.1. Neurons and Synapses

Simbrain supports a wide variety of model neurons (both connectionist and spiking), and rules for synaptic plasticity and transmission. The code was designed with flexibility and extendability in mind. Custom extensions are possible and encouraged. The easiest way to customize the code is via scripts (using beanshell, a type of interpreted Java), though efforts have also been made to make it easy to modify the code directly via the API.⁶ Simbrain takes an object-oriented approach to the representation of common ANN constructs like neurons and synapses, as well as the rules governing their behavior. Most customizations can be implemented as a subclass of a Simbrain core class. This results in an intuitive, easy-to-extend API where the conceptual constructs constituting neural networks have a direct relationship to classes in Simbrain.

⁶Custom neurons, synapses, subnetwork types, neuron group types, update methods, gui widgets, and overall simulation configurations can be created using custom scripts. Over 20 sample scripts illustrating these different types of extension (and serving as easy-to-modify templates) are included. Adding new neuron, synapse, neuron group, and many other object types directly to the Simbrain code is also fairly easy. In the most common case (neuron types), it only involves adding two new files following a standard and easy-to-understand template; the process is described here: see <https://github.com/simbrain/simbrain/wiki>.

Neurons correspond to objects with general properties, including an activation value and a boolean value indicating a spike or action potential (in effect, a Dirac or Kronecker delta function depending upon the temporal framework) has occurred. During network updates the neuron class typically handles procedures relating to the calculation of weighted incoming sums from synapses and the appropriate transfer of activation states (including spiking behavior if applicable) to and from synchronizing buffers. The procedure for determining the neuron’s state at the next time-step (and thus its overall behavior) is delegated to the neuron’s update rule object. For instance, specific point neuron models (e.g. adaptive-exponential integrate and fire, Izhikevich, etc.), are implemented in Simbrain as different subclasses of an abstract neuron update rule class. Neurons aggregated into groups share a single copy of an update rule object to save memory, unless update rule parameters are heterogeneous among the member neurons of the group. Supported neuron models include:

- Adaptive Exponential Integrate and Fire (AdEx) [8]
- Decaying activation
- Fitzhugh-Nagumo[11]
- Hodgkin-Huxley⁷ [18]
- Interactive activation and competition [43]
- Izhikevich [22]
- Heavyside function
- Leaky (linear) Integrate and Fire (as adapted from [29])
- Linear/identity
- Morris-Lecar [32, 40]
- Naka-Rushton [33, 39]
- Sigmoid-function (logistic, hyperbolic tangent, arctangent; continuous or discrete time)
- Spiking threshold/binary

Sample neurons and their visualizations are shown in Fig. 2.

Synapses are similar to neurons in that they correspond to objects with general properties, but also possess delegated objects which correspond to specific synapse types. In particular, synapses may possess a plasticity rule (in the code a “learning rule”) and/or a post-synaptic efficacy rule (in the code a “spike responder”) depending upon the time type (discrete or continuous) of the simulation.

Plasticity rules modify the strength of a synapse at each update iteration. As with neurons, these objects are updated at every iteration, except for synapse groups which can use a single plasticity rule instance for the entire group. With each iteration, the size and color of the visible synapses can be seen changing, so that synaptic plasticity dynamics can be directly visualized. Simbrain supports the following synaptic plasticity mechanisms:

- Rate-based Hebbian plasticity [17]
- Oja [37]
- Subtractive normalization [31]
- STDP [45]
- Log-STDP ⁷ [13]

⁷ These items cannot be created from a GUI dialog, but can be created (and then viewed in the GUI) through scripting, shell commands, or custom source code.

Efficacy rules govern how a synapse generates a post synaptic response to a spiking event. These rules apply only if the pre-synaptic neuron is a spiking neuron and the post-synaptic neuron is set to accept synaptic inputs. Simbrain supports the following models of synaptic transmission:

- Instantaneous jump and (exponential) decay
- Convolved instantaneous (exponential) jump and decay
- Alpha function [41]
- Instantaneous jump and instantaneous decay (step)
- Probabilistic
- UDF (Use, Depression, Facilitation) ⁷[30]

Simbrain has been designed such that a traditional connectionist neuron model (i.e. a rate coded, activation based, non-spiking neuron) can receive input from neurons governed by spiking models and vice versa (for example, the neurons on the left hand side of the network panel in Fig. 2 are rate coded neurons driving spiking neurons). This allows all the neuron types to interact, and is in keeping with Simbrain’s goal of permissive experimentation.⁸

2.2. Groups and Subnetworks

Simbrain networks are composed of individual neurons and synapses (what we call “loose neurons and synapses”), as well as larger aggregates of neurons and synapses in to “neuron groups” and “synapse groups” (examples are shown in Figs. 1,5,9, and 10). Much of the design effort in 3.0 was put in to making it as easy to interact with these groups as it is to interact with individual neurons and synapses. Any changes which can be made to an individual neuron or synapse can be made to all the neurons or synapses in a group. As network simulations become larger these groups replace loose neurons and synapses as the basic building blocks of a model. Groups can also interact with loose elements.

A number of features are specific to the group level. At the neuron group level users can record activations to a file or set the activations of the neurons in a group from a file.⁹ Input values to a neuron group can be displayed, edited, and fed to the group using an input table (see Fig. 1).

The synapse group dialog (Figs. 3 and 1) displays statistics for synapses in a group, and can be used to edit those values. The ratio of excitatory to inhibitory synapses can be adjusted, and the distribution of weight strengths for each of these populations can be set using values drawn from the following distributions: uniform, normal, log-normal, gamma, exponential, and Pareto. These changes are immediately visible in a histogram. For synapse groups with less than 1000 synapses the dialog has a tab that displays these values in a table arranged as they would be in a weight matrix. Sparse matrices are supported in this table view, and individual weight values can be edited. The weight matrix can also be saved or loaded as a .csv file (or .mat file, see Sect. 2.6).

In addition to generic groups of neurons and synapses, Simbrain also contains specialized neuron groups and subnetworks, which have custom update rules, connection patterns, and special functions accessible via their dialogs and context menus. For instance, echo state networks (ESNs) require that the weight matrix characterizing the synapses have a spectral radius less than 1[23]. Networks trained with supervised learning, e.g. backpropagation networks, have a special interface for training and validation. Some of the custom neuron groups and subnetworks currently available are:

⁸Note that while Simbrain does support mixed networks with discrete-time and continuous-time neurons, the mixed mode has not been extensively used and is thus based on a fairly simple method: discrete iterations are associated with fixed continuous time-steps. Plans are in place to improve Simbrain’s support for hybrid continuous/discrete networks by allowing users to set the ratio of iterations to time-steps. One benefit of this approach is that it will allow for the complete simulation of liquid-state machines (LSM)[29] without using special update rules or a custom script. LSMs consist of a dynamical reservoir of recurrently connected spiking neurons that typically run in continuous time, which provide input to a linear classifier which runs in discrete time. Translation between the two requires that the discrete-time iterations receive inputs taken over some time interval from the continuous-time reservoir [29, 24, 35].

⁹This works slightly differently for spiking and non-spiking neurons. For spiking neurons, spike times are written to a file, and input values are interpreted as membrane potentials (causing action potentials to occur as they would in any other context). For non-spiking neurons, activation values are recorded as columns in a .csv file, and are read into the group in the same manner.

Figure 2: Four spiking neuron models being injected with different amounts of current (in different units), and time series of the resulting membrane potentials. From top to bottom: Morris-Lecar (red) producing detailed membrane potential dynamics and evenly-spaced spikes; adaptive exponential integrate and fire (blue) producing a chaotic spike train; a linear leaky integrate and fire neuron (green) producing a constant frequency spike train, and Izhikevich neurons (yellow) exhibiting bursting behavior.

Figure 3: The synapse group dialog. The histogram shows excitatory and inhibitory weights separately. The slider bar adjusts the ratio of excitatory to inhibitory synapses. Distributions for the two populations are set using the bottom panels. All changes are immediately visible in the histogram.

Figure 4: Performance in mean iterations per second for recurrently connected neuron groups varying by number of Izhikevich model neurons [22], and connection sparsities ranging from 0.01 to 0.1 Each network was run for 100,000 iterations, with a time-step of 0.1 ms, for a total of 10 simulated seconds, on an Ubuntu 14.04 workstation with an Intel Xeon E5-2620 v3 2.4 GHz 6 core processor and 16 GB of DDR4 ECC SDRAM, using the Oracle Java Development Kit (JDK) version 1.8 update 20.

- Standard Backpropagation [42]
- Competitive Learning [43, 5]
- Hopfield networks [19]
- Winner takes all and k-winner takes all
- Kohonen’s Self-Organizing Map (SOM) [25]
- Elman’s Simple Recurrent Network (SRN) [10]
- Jaeger’s Echo-State Network (ESN) [23]

2.3. Network Update and Performance

Simbrain 3.0 is the first version of the software to be optimized for the study of networks containing on the order of 10,000 neurons and hundreds of thousands of synapses (i.e. models on the order of cortical microcircuits). Models up to this size can now be efficiently run on a moderately high-end personal computer (arbitrarily large networks can be run at a cost in performance; for a sense of how performance scales with size, see Fig. 4). For example, Simbrain can simulate a standard 1000 neuron Izhikevich network [22] in better than real time (0.4 real seconds to each simulated second for time-steps of 0.5 ms on a Mac Book Pro with a 2.2 GHz Intel i7 CPU). An example of such a network running in parallel is shown in Fig. 5. Simbrain 3.0 was not developed with the aim of competing in the high-performance neural computing arena (e.g., so as to be comparable in performance to GeNN), but rather with the aim of making a relatively common scale of simulation in network level computational neuroscience available in a rich GUI framework that allows for rapid prototyping and visual analysis.

The performance improvements are based in part on CPU parallelization for network updates. Some proportion of the neurons in the network and their incoming synapses are assigned to an individual thread, such that the number of threads is equal to the number of processors recognized by the host operating system. Thread execution and synchronization are accomplished using Java’s built-in executor service [2][15]. In each network update, the parallel threads write the updated value for each neuron for the next time-step/iteration

Figure 5: A large simulation that showcases Simbrain 3.0’s increased capacity. The network on the left side of the network panel contains 4096 recurrently connected Izhikevich neurons. A corresponding network of linear neurons on the right side of the network panel displays an exponential convolution of the Izhikevich spike trains. The histogram on the right displays the activation values of the group of linear neurons and updates itself in real-time with the network.

to a buffer. Network-wide temporal synchronization is then accomplished by writing the values in each neuron’s buffer to the variables corresponding to its current state serially. This ends the update for that time-step/iteration, and the parallel threads are invoked again, beginning the update for the next time-step/iteration. The performance improvements associated with concurrent updating are shown in Fig. 4. Currently Simbrain is not configured to use a message passing interface which would allow it to run across *multiple nodes* in a cluster, though it is capable of taking advantage of *all available processor cores on a single node* (regardless of the number of CPUs).

Performance improvements are also based on enhancements made to the Java Virtual Machine in the last 20 years, including just-in-time (JIT) compilation[3] and HotSpot [38]. This allows Simbrain to run faster than interpreted languages like Matlab and Python. JIT compiled Java code has been shown to approach and in some cases surpass optimized C code [28][36]. So, although C/C++ still outperforms Java generally speaking, the gap between Java and natively compiled languages has been diminishing [34] [46].

2.4. Graphical User Interface and the Simbrain Workspace

The GUI follows a “least surprise” design principle: make things operate as a modern computer user would expect them too, so that most operations can be done in an intuitive way, often without consulting documentation. Neurons, synapses, groups and subnetworks can be selected by lassoing or single clicking. Mouse-wheel and track pad gestures allow for quick zooming and panning. “Semantic zooming” reveals more details of a network as the level of zoom is increased (for example, neuron activation values are only displayed when zoomed in, but are hidden when viewing large networks). An effort has been made to allow for deep, fine-grained editing of all parameters, while masking most of this from the main interface. Double-clicking any object brings up a dialog which can be used to view and edit specific parameters. These dialogs often have multiple tabs each of which supports numerous interventions. An extensive right-click menu system is in place. Keyboard shortcuts can be used to quickly perform such tasks as copy-paste, connecting neurons, randomization, and activation clearing. Many of these operations are now demonstrated via on-line video demonstrations (see <https://www.youtube.com/channel/UCuvRAuIyNiuRIrVnEztDr-Q>).

Simbrain opens to a blank desktop or “workspace.” Networks and other components are added using the tool-bar, menus, or key commands. Each component contains its own system of menus, key commands, and visual components. Simulations are run by pressing a play button in the workspace tool bar. Networks and other components are connected together using “couplings” (links between components in a simulation), which can be edited via a special coupling management dialog, or by using special context menus which appear when right-clicking on any network component. For example, all the neurons in a neuron group can

Figure 6: A recreation of a living neural network, using data recorded from a 512 multielectrode array. The original neural network was prepared as an organotypic culture of neurons taken from rat somatosensory cortex. Transfer entropy was used to find which neurons had the highest likelihood of being connected by a synapse [44, 21]. The resulting matrix was imported in to the network above and used as its synaptic weight matrix. A spike raster plot is on the right (time on the horizontal axis, one row for each of the neurons on the vertical axis, and dots for spikes). Spiking neurons and their “axons” are shown in black, or yellow if the pre-synaptic cell is producing a spike.

be coupled to a high dimensional projection plot by right clicking on the neuron group’s interaction box and navigating a menu displaying all the “consuming” components that can accept a vector of values as inputs (see Fig. 9). In a similar way, couplings can be used to link neurons to the sensors and effectors of a virtual agent (see Fig. 7). Couplings (in conjunction with scripts) have also been used by the second author to link networks to external devices (e.g. the sensors and effectors of a robot), and to other programs running outside of Simbrain, though no such applications or demos are currently included with the Software (plans are in place to provide a GUI interface to external sensors and devices in a future release).

Neurons and synapses in Simbrain are color coded to make as much information available at a glance as possible. For continuously valued neuron rules, red represents positive or excitatory activity, blue represents negative or inhibitory activity, and both fade to white as the activation approaches zero. An action potential is represented by a burst of yellow in the neuron and all efferent synapses (see Fig. 6). For synapses all inhibitory connections (weight values less than 0) are blue, while all excitatory connections are represented as red. The size of a synapse represents the absolute value of its strength.

Synapses are not always visible. When a network contains too many visible synapses they become an unhelpful tangle and a performance drain. In such cases synapse groups are used. The tangle of individual synapses is replaced by a single green translucent arrow that represents the synapse group as a whole. Users can then interact with the synapse group using context menus and the synapse group dialog. Examples of visible synapses are in Figs. 2 and 7. Invisible synapses in synapse groups are shown in Figs. 1 and 5. Examples showing how both styles can be mixed are Figs. 6, 9, and 10. As noted above, the ability to interact at the high level of neuron groups and synapse groups supports a visual language where large groups of neurons and synapses can be treated the basic building blocks of a simulation.

2.5. Worlds

Most neural network simulations involve some kind of interaction with an environment. Often the “environment” is a simple table of values containing, for example, sample inputs in a classification task.¹⁰ In some cases a richer environment is used, e.g. a simulated physical environment in which an organism’s sensors, effectors, and other properties are modeled. For example, some models of reinforcement learning involve neural networks embedded in agents which learn to obtain optimal reward in a virtual environment.

Simbrain currently contains several virtual environments, with plans for more on the way. A neural network can be linked to multiple virtual environments, each of which can be edited in the GUI or customized with scripts, to create complex multi-component, embodied neural network simulations.

The “odor world” is a 2d virtual environment featuring agents and objects that can have sensors, effectors, behaviors, and other features attached to them. Though it can be used to model arbitrary sensory stimuli, it was originally developed to support a smell simulation, and is easiest to think of in terms of dispersing olfactory stimuli. Stimulus vectors are attached to objects, which emit a radius of “smell” which decays with distance, according to a user-selected decay function. Agents can be fitted with smell sensors that respond to these smell gradients. Location sensors are also available, as is a simple speech model. Effectors can be used to turn and move agents.

The odor world component can be used to model simple Braitenberg vehicles [7], which generally pursue or avoid one another, and are easy to create and modify using a simple neural network coupled to an Odor World (see Fig. 7). Users can grab and move the objects or agents in the world, for example, to watch them pursue or avoid one another, and can manipulate their neurons and weights to alter their behavior, e.g. to control how vigorously they pursue one another.

2d Braitenberg simulations have been used in educational settings (see, for example, the workspaces *chasing-part1.zip* and *chasing-part2.zip*, which come pre-packaged with Simbrain), and in research settings. In ongoing work between the second author and Scott Hotton, Simbrain has been used to facilitate a detailed bifurcation analysis of pairs of interacting Braitenberg vehicles. The system is six dimensional (3 neural activations determine the heading and speed for each agent), and the parameter space is 4 dimensional (2 weights per agent). Saddle-node-like and Hopf-like bifurcations have been observed in the system, and part of the parameter space produces “meandering paths”, which can have arbitrarily many loops (Fig. 8). The simulation shown in Fig. 7 was used to confirm some of the mathematical analyses, and to further explore the parameter space.

¹⁰Simbrain has a “data world” for handling tabular data like this, but it will not be described here.

Figure 7: Braitenberg simulation, suitable both for introduction of basic concepts and more advanced exploration of parameter space.

Figure 8: Sample meandering paths of the two agents, identified with help from Simbrain. Blue is the physical path followed by one of the two agents; orange is path followed by the other. The black curves show a single “periodic arc.” Different classes of meandering paths are characterized by how many periodic arcs they contain. Paths containing arbitrarily many periodic arcs are possible depending on how the weights are set.

Simbrain has two kinds of simulated text environment, or “text world.” The first is a reader world that “reads” text, i.e. that converts text to numeric outputs that can be sent to networks. The second is a display world that converts numeric inputs to text that is displayed. These components are primarily used to model language. An example of a simulation that uses these components is shown in Fig. 9. It is a replication of Elman’s classic 1990 model [10]. A simple recurrent network (SRN) is trained on a sequence of phonemes (forming three “words”, *ba*, *dii*, and *guruu*). The current letter in the reader world is converted into a pattern of activations in the input layer of the SRN, the output activations are converted back in to a phoneme, and the output is compared with the desired output to produce an error value. The network learns to predict the

Figure 9: Elman Simulation. A phoneme stream in the “reader world” (upper left) is converted in to 6-component vectors in the input layer of the simple recurrent network (center). The output vector of the network is converted back in to a phoneme and is shown in the “display world” (lower left) as the predicted next phoneme (the actual next phoneme and an error are also displayed). A real-time display of the hidden layer’s activation is projected into a 2D space using principal components analysis (right). As the network learns, the clusters representing more common phoneme sequences becomes more distinct.

next phoneme in a training sequence, and is then tested on novel sequence. The predicted next phoneme, actual next phoneme, and error are shown in the display world. Consistently with [10], error rises between words (when it cannot predict the next phoneme), and is low within words.

2.6. Scripting and Interfacing with Other Software

Simbrain scripts contain Java commands in a form that can be executed dynamically. Script files placed in a special directory appear automatically as menu items in the main desktop menu after the program is restarted. Simbrain comes with a variety of script files that can be used as templates for creating new scripts. Simbrain’s scripting allows advanced users to dynamically interact with Simbrain’s source code, creating customized GUI components, neurons, networks, etc. Examples of customized simulations using scripts are in Figs. 5, 6, 7, and 10.

Workspaces are stored as .zip files. These files contain human-readable and editable .xml representations of all components and couplings in a simulation. Large matrices are represented as compressed byte streams in the xml.

Stored workspace files can be used in conjunction with scripts. In such cases, the workspace file can be incrementally modified using Simbrain’s point and click methods, and specific custom features of a simulation can be modified via the code in the script. This creates a relatively fast work-flow for building custom simulations.

Simbrain can also be used in conjunction with other programs and languages. The activation values and weight matrices corresponding to a Simbrain network can be exported directly to Matlab .mat files using a menu command. The Simbrain.jar file can be referenced from a Matlab script, so that Simbrain objects (e.g. model neurons or networks) can be created and used in Matlab. In a similar way, the Simbrain.jar file can be used in real-time by programs written in C, C++, and other languages using the Java Native Interface (similarly for Python using Py4J), though we have not created any examples of this. Of course, R,

Figure 10: The self-organizing recurrent network model (SORN) [27] is a network architecture that uses three forms of plasticity—STDP, homeostatic plasticity (where neurons act to maintain a preferred firing rate), and synaptic normalization—to produce patterns of activity that are similar to those observed in neocortex. The script used to produce the simulation is shown on the left.

Python, and other frameworks can be used to prepare data for a Simbrain network, and data resulting from a Simbrain simulation can be imported in to these frameworks for analysis.

3. Future Directions

A major goal for Simbrain 4.0 will be the introduction of parallel GPU computing (compare [9]), most likely though JOCL (Java Bindings for Open CL; a framework for open source parallel computing)[1]. Simbrain’s current performance benchmarks make it ideal for use by neural modelers studying network-level phenomena. However the jump to GPU computing will make larger scale networks possible in Simbrain on desktop PCs.

Other plans include more advanced machine learning algorithms (e.g. deep learning and convolutional neural networks); an immersive 3d environment with customizable visual sensors (which is in active development); audio (both in terms of responding to sound waves as input, and producing sound waves as output); an expanded suite of visualization tools; and an ability to link Simbrain to other applications (e.g. video games) and, as noted above, to the sensors and effectors of physical devices like robots. There are also plans to enhance the neural network GUI, allowing for networks in 3 dimensions, some dendritic and axonal structures, interactions with neuro-modulators, and an ability to read NeuroML files [14]. Finally, there are plans in place to deploy Simbrain simulations in educational settings as more advanced self-guided tutorials, with embedded videos, quizzes, and knowledge-check questions.

4. Conclusion

Simbrain 3.0 is the most recent 10-year iteration of a development process which began over 15 years ago. Design goals have always focused on an intuitive interface and visualization, and an ability to develop arbitrary neural models. It was thus natural in 3.0 to expand the available network, neuron, and

synapse models, and to allow for larger scale simulations. Doing this in a way that maintained Simbrain’s versatility and intuitive interface required a series of complex design choices and software engineering cycles. The large amount of work put into this interface benefit both newcomers and professionals, defining a basic visual language for representing complex networks. Simbrain 3.0 is supported by detailed documentation (http://simbrain.net/Documentation/docs_main.html) and a small but growing open source development community.

5. Acknowledgements

This work was supported in part by funding from the National Science Foundation’s Integrative Graduate Education and Research Trainee (IGERT) program, at the University of Indiana Bloomington.

- [1] Java bindings for opencl. <http://www.jocl.org/>. Accessed: 2015-10-5.
- [2] Executorservice (java platform se 8) - oracle help center, 2016.
- [3] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M Parikh, and James M Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *ACM SIGPLAN Notices*, volume 33, pages 280–290. ACM, 1998.
- [4] Brad Aisa, Brian Mingus, and Randy O’Reilly. The emergent neural modeling system. *Neural networks*, 21(8):1146–1152, 2008.
- [5] Pablo Alvarez and Larry R Squire. Memory consolidation and the medial temporal lobe: a simple network model. *Proceedings of the National Academy of Sciences*, 91(15):7041–7045, 1994.
- [6] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C Stewart, Daniel Rasmussen, Xuan Choo, Aaron Russell Voelker, and Chris Eliasmith. Nengo: a python tool for building large-scale functional brain models. *Frontiers in neuroinformatics*, 7, 2013.
- [7] Valentino Braitenberg. *Vehicles: Experiments in synthetic psychology*. MIT press, 1986.
- [8] Romain Brette and Wulfram Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of neurophysiology*, 94(5):3637–3642, 2005.
- [9] Romain Brette and Dan FM Goodman. Simulating spiking neural networks on gpu. *Network: Computation in Neural Systems*, 2012.
- [10] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
- [11] Richard FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical journal*, 1(6):445, 1961.
- [12] Marc-Oliver Gewaltig and Markus Diesmann. NEST (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.
- [13] Matthieu Gilson and Tomoki Fukai. Stability versus neuronal specialization for STDP: long-tail weight distributions solve the dilemma. *PLoS One*, 6(10):1–18, 2011.
- [14] Pdraig Gleeson, Volker Steuber, R Angus Silver, and Sharon Crook. NeuroML. In *Computational Systems Neurobiology*, pages 489–517. Springer, 2012.
- [15] Brian Goetz and Tim Peierls. *Java concurrency in practice*. Pearson Education, 2006.
- [16] Dan FM Goodman and Romain Brette. The Brian simulator. *Frontiers in neuroscience*, 3:26, 2008.
- [17] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, 1949.

- [18] Allan L Hodgkin and Andrew F Huxley. Currents carried by sodium and potassium ions through the membrane of the giant axon of loligo. *The Journal of physiology*, 116(4):449–472, 1952.
- [19] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.
- [20] Scott Hotton and Jeff Yoshimi. Extending dynamical systems theory to model embodied cognition. *Cognitive Science*, 35(3):444–479, 2011.
- [21] Shinya Ito, Michael E Hansen, Randy Heiland, Andrew Lumsdaine, Alan M Litke, and John M Beggs. Extending transfer entropy improves identification of effective connectivity in a spiking cortical network model. *PloS one*, 6(11):e27431, 2011.
- [22] Eugene M Izhikevich et al. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.
- [23] Herbert Jaeger. The echo state approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148:34, 2001.
- [24] Herbert Jaeger, Wolfgang Maass, and Jose Principe. Special issue on echo state networks and liquid state machines. *Neural Networks*, 20(3):287–289, 2007.
- [25] Teuvo Kohonen. The self-organizing map. *Neurocomputing*, 21(1):1–6, 1998.
- [26] Jiri Krenek, Kamil Kuca, Ondrej Krejcar, Petra Maresova, V Sobeslav, and Pavel Blazek. Artificial neural network tools for computerised data modeling and processing. In *Computational Intelligence and Informatics (CINTI), 2014 IEEE 15th International Symposium on*, pages 255–260. IEEE, 2014.
- [27] Andreea Lazar, Gordon Pipa, and Jochen Triesch. SORN: a self-organizing recurrent neural network. *Frontiers in computational neuroscience*, 3, 2009.
- [28] JP Lewis and Ulrich Neumann. Performance of java versus c++. university of southern california 2003.
- [29] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.
- [30] Henry Markram, Yun Wang, and Misha Tsodyks. Differential signaling via the same axon of neocortical pyramidal neurons. *Proceedings of the National Academy of Sciences*, 95(9):5323–5328, 1998.
- [31] Kenneth D Miller and David JC MacKay. The role of constraints in hebbian learning. *Neural Computation*, 6(1):100–126, 1994.
- [32] Catherine Morris and Harold Lecar. Voltage oscillations in the barnacle giant muscle fiber. *Biophysical journal*, 35(1):193, 1981.
- [33] KI Naka and WAH Rushton. S-potentials from colour units in the retina of fish (cyprinidae). *The Journal of Physiology*, 185(3):536–555, 1966.
- [34] Sun Developer Network. The java hotspot performance engine architecture. *Sun Microsystem*, 2007.
- [35] David Norton and Dan Ventura. Improving liquid state machines through iterative refinement of the reservoir. *Neurocomputing*, 73(16):2893–2904, 2010.
- [36] Bogdan Oancea, Ion Gh Rosca, Tudorel Andrei, and Andreea Iluzia Iacob. Evaluating java performance for linear algebra numerical computations. *Procedia Computer Science*, 3:474–478, 2011.
- [37] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267–273, 1982.

- [38] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 1–1. USENIX Association, 2001.
- [39] Jonathan W Peirce. The potential importance of saturating and supersaturating contrast response functions in visual cortex. *Journal of vision*, 7(6):13, 2007.
- [40] John Rinzel and G Bart Ermentrout. Analysis of neural excitability and oscillations. *Methods in neuronal modeling*, 2:251–292, 1998.
- [41] A Roth and MCW van Rossum. Computational modeling methods for neuroscientists, 2009.
- [42] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.
- [43] David E Rumelhart, James L McClelland, PDP Research Group, et al. *Parallel distributed processing*, volume 1. IEEE, 1988.
- [44] Thomas Schreiber. Measuring information transfer. *Physical review letters*, 85(2):461, 2000.
- [45] J. Sjöström and W. Gerstner. Spike-timing dependent plasticity. *Scholarpedia*, 5(2):1362, 2010. revision 142314.
- [46] Guillermo L Taboada, Sabela Ramos, Roberto R Expósito, Juan Touriño, and Ramón Doallo. Java in the high performance computing arena: Research, practice and experience. *Science of Computer Programming*, 78(5):425–444, 2013.
- [47] Jeff Yoshimi. Simbrain: A visual framework for neural network analysis and education. *Brains, Minds, and Media*, 3, 2008.
- [48] Jeff Yoshimi. Narrowing the explanatory gap with bridge metaphors. 2014.