Department of Computer Science,
University of Otago

Technical Report OUCS-2013-03

Spinula: software for simulation and analysis of spiking network models

Authors:

Mira Guise, Alistair Knott, Lubica Benuskova

Department of Computer Science, University of Otago, New Zealand

Status:

This is a document in support of a manuscript submitted to ICONIP 2013.

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/research/techreports.php

Spinula: software for the simulation and analysis of spiking network models

Mira Guise, Alistair Knott, Lubica Benuskova

Dept of Computer Science, University of Otago, Dunedin

Abstract

Spinula is a software package for the simulation and analysis of spiking neural network models. It consists of a core library that provides the simulation environment, and additional libraries that support the analysis and visualization of simulation data. This report provides a description of the services implemented by each of these libraries and some examples of how they are used.

Keywords: spiking neural network, software simulation

Spinula is a software package for the simulation of spiking networks based on the Izhikevich neuron (Izhikevich, 2006a). The package allows the simulation of networks of arbitrary size, from single neurons and single synapses to networks containing thousands of neurons and more than one hundred thousand synapses. Simulation experiments based on Spinula are normally interactively constructed as scripts that define the run parameters for the experiment and additional run-time details such as the network architecture and the data collection requirements. Data analysis can also be scripted, making use of a library of functions for loading, transforming and visualizing the collected data. Spinula may also be used to add features to an existing software program. For example, neural network simulation might be added as a feature to an existing program by linking the program to a Spinula code library.

1 Some common terms

A *network engine* generates the simulation environment and there are two of these to choose from that provide a trade-off between performance and flexibility. For example, the higher performing network engine can implement only *grid networks* in which each neuron has a fixed number of connections per neuron, while the less efficient engine can also construct *ad hoc networks* with variable numbers of connections per neuron. Grid networks are created by connecting each pre-synaptic neuron in the network with a fixed

number of randomly selected post-synaptic neurons. Each grid network is therefore unique with respect to the connectivity between neurons, and the polychronous neural groups that it supports.

Independently of the network engine type, the size of a new network is specified in terms of the total number of neurons in the network. Some typical network sizes are 100 neurons (i.e. an *N100 network*) or 1000 neurons (an *N1000 network*). The more flexible network engine also supports experiments on single neurons or small networks of a few neurons. There is also support for introducing both external stimulation and random background stimulation to the network. The external stimulation (or *stimulus*) is the simulated equivalent of an externally applied source of stimulation as might be applied by a microelectrode to a single neuron. It is defined as a *pattern* that is applied to the network repetitively with a constant period (both *Stimulus* and *Pattern* are underlying types in the Spinula core library). The pattern is both spatial (distributed over multiple neurons) and temporal (distributed over time) and is therefore referred to as a spatio-temporal pattern. Random background stimulation is also defined by a spatio-temporal pattern except that the temporal component is generated by a random generator function such as a Poisson Process. Random stimulation is an important contributor to the dynamics of the network but as it is distinct from the externally applied (*foreground*) stimulus it is often referred to in this text as *background* stimulation.

Spatio-temporal firing patterns are composed of neural *firing events* that record the firing of specified neurons at specified times. These firing events can be seen as dots in Panels A and B of Fig. 1. Panel A shows two commonly utilized patterns taken from Izhikevich (2006a) called the *Ascending* pattern and the *Descending pattern*. Panel B shows three different stimuli constructed by repeating the Ascending pattern at one of three different frequencies. Random background stimulation is also composed of firing events and examples of these random firing patterns can be seen in Fig. 2. The complete set of firing events generated by the network during a simulation run is called the network *firing data* and this dataset is one of the primary data sources for Spinula analytical functions. Firing events are captured at one millisecond resolution, a limitation imposed by the one millisecond timestep employed by the simulator. The network can also be saved to a *network state file* at specified intervals allowing the complete state of the simulated network to be reconstructed at a later time. During a simulation run the *internal simulation time* (i.e. the subjective time for the network) may be either faster or slower than real-time depending on the size of the simulated network and the performance of the underlying hardware.

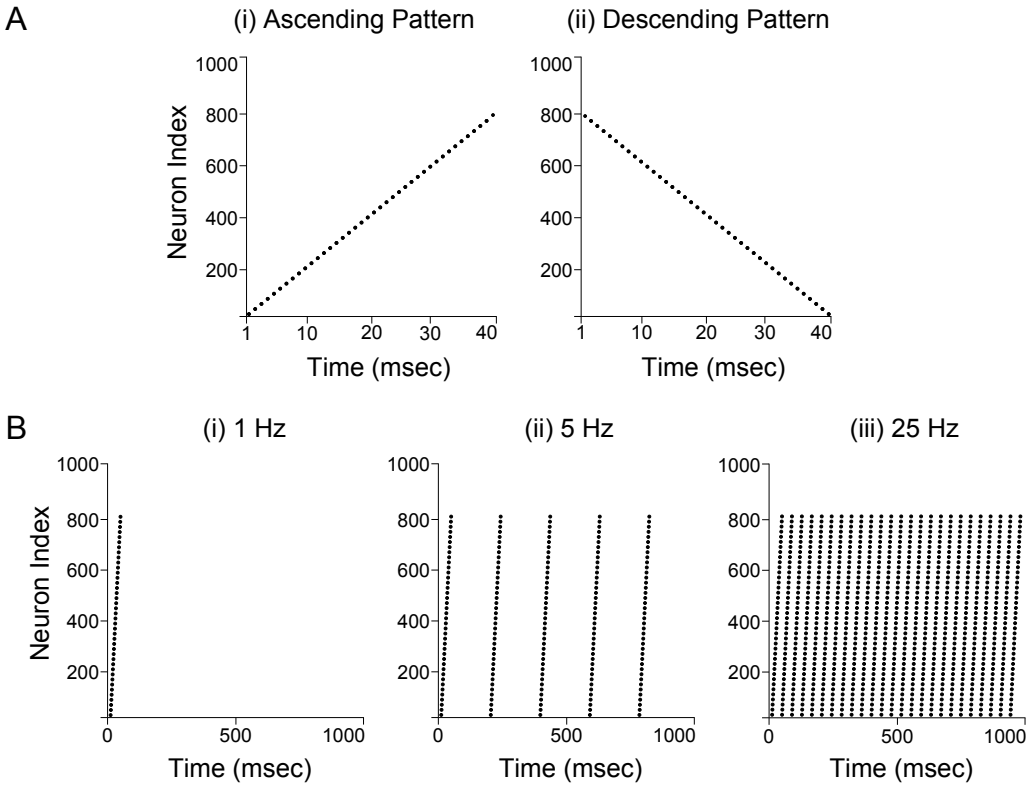


Figure 1: Some example patterns and stimuli. A. The Ascending and Descending patterns are two commonly utilized spatio-temporal patterns. B. Three different stimuli constructed by repeating the Ascending pattern at either 1, 5 or 25 Hz.

2 Scripts

Simulation experiments based on Spinula are normally scripted in a language called F# that in its simplest form displays much of the sparse expressiveness of pseudo-code. Most Spinula scripts are used to define network simulation experiments and therefore contain instructions for constructing the experimental network, starting the simulation run, and gathering network data over the course of the run. Other scripts are written for the purpose of data analysis and involve reading and transforming potentially large volumes of simulation data. Listing 2.1 shows an example of a network simulation script that generates random spiking in a grid network of one thousand neurons.

```
1  // Test the Poisson Process background pattern generator at 1 Hz
2
3  let verbose = true
4  let runSeconds = 5
5  let backgroundFrequency = 1
6  let outputFilePath = ``some file''
7
8  // create a new network with no connections between neurons
9  let network = CrossbarNetwork.CreateAdHocNetwork(
10         CrossbarNetworkSpecifier.N1000_Unconnected_Network, None, verbose)
11
12 // run the network with background stimulation but no stimulus and collect firing data
13 network.Run(runSeconds, None, backgroundFrequency)
14
15 // select the third one second frame of firing data
16 let thirdFrameData = network.OneSecondEventCollector.SelectRange(2000, 3000, false)
17
18 // save the data
19 thirdFrameData.Save(outputFilePath)
20
21 // show the data as a spike raster
22 SpikeVisualisation.ShowSpikeRaster(thirdFrameData.AllEventPairs)
```

Listing 2.1: A sample script that tests the background pattern generator.

The script begins by constructing a network, making use of a predefined *network specifier* that describes an N1000 network with no connections. The call to the *Run* method in line 13 begins the network simulation with parameters that specify no stimulus (None) and a random background frequency of 1 Hz. Random background stimulation is generated from a *pattern generator* on each neuron, the current implementation of which is a homogeneous Poisson process where the probability distribution of spiking events generated from the pattern generator is proportional to the time period between events. Firing events are collected automatically throughout the simulation run and line 16 selects firing events generated during the third simulated second between $t = 2000$ and $t = 3000$ msecs. This firing data is then saved to a file in line 19 and displayed in a window in the final line of the script (see Appendix B.3 for a more detailed explanation of this script, and Appendix B for more script examples). Figure 2 shows the resulting output with the

background frequency set to either one hertz (Panel A) or ten hertz (Panel B).

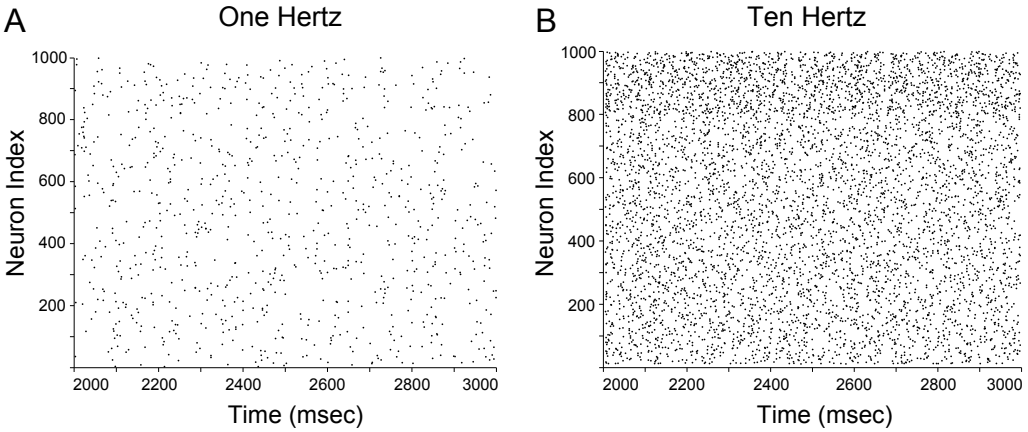


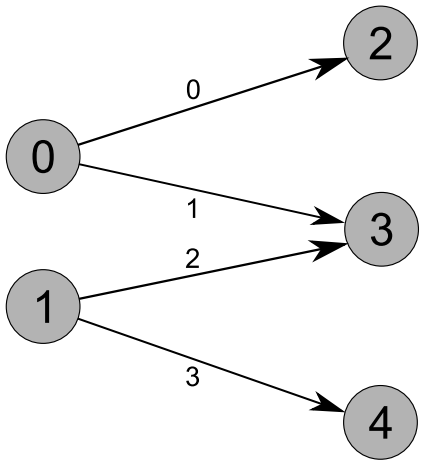
Figure 2: The result of running the sample script: The background frequency was set to either 1 Hz (A) or 10 Hz (B).

3 A small experiment

In this section we will explore some more Spinula features by performing a simple experiment on a small ad hoc network. The network consists of five neurons: two input layer neurons and three output layer neurons, with the structure shown in Figure 3. The network has four synaptic connections arranged in a W-pattern and we can therefore refer to it as a *W-Network*. All connections have the same axonal delay (1 msec) and initial synaptic weight (8.5). Output neurons 2 and 4 of this network are connected to just one input neuron each, while output neuron 3 is connected to both input neurons (0 and 1). The neuron types are excitatory RS type with a firing threshold of about 17 mV so that, even with saturated connection weights, the neurons in the output layer require simultaneous input from two neurons in order to fire. Only neuron 3 receives input from two neurons and therefore only neuron 3 can reach the firing threshold *due to input from neurons 0 and 1 alone*.

The W-network topology has been designed to provide a simple demonstration of the effects of spike-timing-dependent plasticity (STDP) in the presence of background firing. The STDP rule specifies that for each synaptic connection in the network, the degree of synaptic potentiation or depression

is a function of the difference in the firing time of each post-synaptic neuron and the arrival time of spikes at the synapse. There are therefore two necessary conditions in order for the STDP rule to produce changes in synaptic plasticity: both the pre-synaptic neuron and the post-synaptic neuron must fire *and* the pre-synaptic spike must arrive close to a post-synaptic firing event (within a small temporal window). In our experimental network, only output neuron 3 is able to fire in response to the firing of input neurons, and therefore only the connections leading to neuron 3 are plastic *in the absence of background firing.*



(a) Connectivity

Neuron numbering:

Layer	Neuron Indices
input	0, 1
output	2, 3, 4

Connection numbering:

Connection	PreNeuron	PostNeuron
0	0	2
1	0	3
2	1	3
3	1	4

Initial synaptic weight:	8.5

(b) Data

Figure 3: W-Network structure and parameters

A Spinula script must be written to orchestrate the experiment, including definitions for the network architecture, the stimulus and the data collection requirements. The script must then construct the W-network, start the simulation run and display the resulting data (see Listing B.5 of Appendix B for the full script). The experiment requires that the two input neurons are simultaneously stimulated at intervals of one hundred milliseconds throughout the simulation run. We will run the experiment twice with different parameters in order to test the effects of STDP with different levels of background stimulation, firstly in the absence of background firing and secondly in the presence of high frequency background noise. In the latter case the output layer neurons will occasionally be selected to fire by the background pattern generator and these firing events can therefore occur independently of any input from the input layer.

In each run, the network is *trained* by repeated exposure to the stimulus i.e. the STDP rule produces consistent potentiation of synaptic weights for

connections that see coherent pre- before post-synaptic firing, and consistent depression of weights for connections that see post- before pre-synaptic firing. We can therefore expect to see potentiation of connections 1 and 2, because the connection architecture and the stimulus together ensure that neuron 3 will be consistently fired shortly after the firing of neurons 0 and 1. The effect on connections 0 and 3 is less clear, especially if the simulation is run in the presence of high-frequency noise. We will therefore let each simulation run continue for 100 seconds internal simulation time so that any long-term trend in the connection weights (either upwards or downwards) is apparent.

An effective demonstration of the effects of the STDP rule requires that the network data be sampled at a high temporal resolution. We will need to record both the membrane potential of each neuron and the synaptic weight of each connection at the maximum temporal resolution of one millisecond. To achieve this goal we create a *OneMillisecTickDataCollector*, specifying the required number of neuron and connection samples as follows:

```
let hiDataResCollector =
  let numberOfMembraneSamples = 1000
  let numberOfWeightSamples = 100000
  let selectedNeurons = [ 0; 1; 2; 3; 4; ]
  let selectedConnections = [ 0; 1; 2; 3; ]
  let parameters = new OneMillisecTickDataCollectorParameters(selectedNeurons, totalNeurons,
    selectedConnections, totalConnections, numberOfMembraneSamples, numberOfWeightSamples)
  new OneMillisecTickDataCollector(parameters)
```

Collection of weight samples occurs throughout the run (corresponding to 100 seconds $\times$ 1000 msec/sec = 100,000 samples). In contrast, only 1000 samples (i.e. the data produced in one second) are collected for the membrane potential data, as there is little variation in the response to the stimulus over the course of the simulation. Each weight sample includes both the current synaptic weight (w) and the *synaptic derivative* (d), where the derivative is an internal simulator variable that tracks the current magnitude and direction of synaptic change. Membrane potential samples record both the membrane potential (v) and the membrane recovery variable (u), both variables derived from the Izhikevich equations (Izhikevich, 2006a).

After each simulation run, the collected weight and membrane samples can be retrieved from the *OneMillisecTickDataCollector* and either saved to a file or passed to a Spinula data visualization function as follows:

```
MillisecondResolutionDataVisualisation.ShowCollectedMembraneData(hiResDataCollector)
```

```
MillisecondResolutionDataVisualisation.ShowCollectedWeightData(hiResDataCollector)
```

The results after running the network for 100 seconds can be seen in Fig. 4, with membrane potential data on the left and the corresponding synaptic weight data on the right. With no background stimulation (A, left and right) neuron 3 is the only output neuron to produce regular spiking in response to repeated stimulation of the input neurons 0 and 1. If the firing of neuron 3 occurs just *after* the firing of the input neurons then the consistent correlation between the regular spiking of the input neurons and the spiking of neuron 3 leads to potentiation of connections 1 and 2, as specified by the STDP rule. However, the other two output neurons (2 and 4) receive insufficient input from the input neurons (0 and 1) to reach firing threshold in the absence of background firing. The STDP rule therefore produces no net effect on connections 0 and 3.

If background firing is enabled then all of the output neurons will fire, although perhaps only infrequently. This irregular firing can produce occasional correlations in the firing of output and input layer neurons within the STDP window. In order to study this effect we will repeat the experiment but this time with a background firing rate of 50 Hz. The results in Fig. 4B (left and right) show firing of all neurons, including output neurons 2 and 4. The spiking of output neuron 3 shows both correlated and uncorrelated components although the correlations are sufficient to produce rapid potentiation of afferent connections 1 and 2, despite the additional non-correlated spiking. On occasion, the additional output neuron spiking events produced by the 50 Hz background stimulation occur in the same STDP window as a pre-synaptic spiking event and therefore produce synaptic change. These rare correlated events can occur either before or after input neuron firing and the connections leading to output neurons 2 and 4 therefore display both potentiation and depression. Repeated runs of the experiment show a small overall bias towards depression: if pre- before post-synaptic firing events are equally as likely as post- before pre- events, then this bias is likely due to the default parameters employed for the STDP implementation in the simulation engine: the parameters $A_+ = 0.10, A_- = 0.12$ produce a temporally asymmetric STDP equation with a bias towards depression.

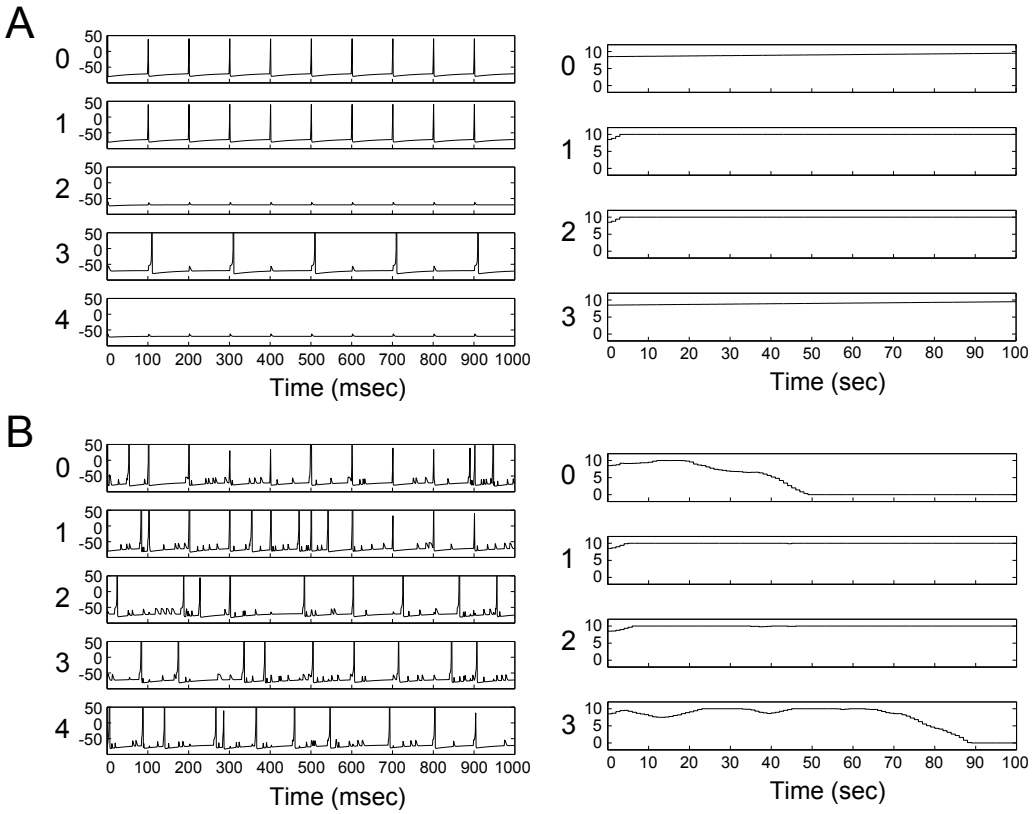


Figure 4: The response of a W-Network following repeated simultaneous stimulation of the two input neurons. Panels on the left show changes in membrane potential (v) for each neuron in the network and panels on the right show synaptic weight (w) changes for each connection. The initial synaptic weight value was 8.5. A. With no background stimulation just one of the three output neurons produces regular spiking (neuron 3). The synaptic weights onto neuron 3 are potentiated, reaching saturation within a few seconds (Connections 1 and 2 in the right-hand panels). B. With a background stimulation frequency of 50 Hz spiking events are frequent for all three output neurons although only neuron 3 firing is strongly correlated with the firing of the two input neurons. The visible stepping in the synaptic weight is an artifact of the weight aggregation process occurring just once each second (each step is exactly one second wide).

A Spinula Libraries

Spinula libraries are implemented using a development framework from Microsoft called the .Net Framework. Although this development choice creates a dependency on the Microsoft platform, it also allows access to a large class library containing many useful data structures and functions. In addition, .Net is built upon a language-neutral platform called the Common Language Infrastructure (CLI) and .Net developers are therefore able to select the most suitable development language for their given task. Although the .Net Framework is designed to run exclusively on recent versions of Microsoft operating systems, alternative implementations of the CLI such as Mono support additional operating systems such as Linux. The Spinula package contains three core libraries: the *SpikingNeuronLib* library provides core network simulation services; the *SpikingAnalyticsLib* library provides data analysis services; and the *SpikingVisualisationLib* library provides data visualization services. These libraries may either be utilized within an interactive scripting environment, or linked into a new or existing program that provides a user interface to the Spinula software.

A.1 The *SpikingNeuronLib* library

The most significant function provided by the SpikingNeuronLib library is the simulation environment that is generated by one of two network engines, either the *IzhikevichNetwork* engine or the *CrossbarNetwork* engine. I created the initial *IzhikevichNetwork* engine as a Python port of the original Matlab implementation from Izhikevich (2006a). The ported code made use of a C matrix library called *numpy* to re-implement the matrix-based manipulations that are central to the original Matlab code. Importantly, both Python and *numpy* are widely utilized in the neuroscience community and many projects have been built using these tools.

At a later date, a C++ version of the demonstration code became available on the author's website (Izhikevich, 2006b) that had substantially better performance than the original Matlab version. This C++ version became the foundation of a new version of the *IzhikevichNetwork* engine, after some code refactoring and bug fixes. My primary aim in refactoring was to repackage the code as a CLR-compliant software library and to provide some additional flexibility in the interface between the core simulation engine and consumers of the simulation data. The refactored network engine code also encompassed the Izhikevich algorithms for finding PNGs (based on the original code from

Izhikevich, 2006b). For performance reasons both the original demonstration code and the refactored *IzhikevichNetwork* engine implement a grid network and no other network topologies are supported. Each neuron therefore has a fixed number of connections allowing for an efficient array-based implementation.

A new more flexible engine was then developed that allows not only grid networks but also more free-form network architectures. The *CrossbarNetwork* engine utilizes the same simulation algorithm as the *IzhikevichNetwork* engine, but with substantial changes to the underlying data structures that allow a more flexible simulation engine and more opportunities for efficient data collection and for instrumentation of the simulation. The *CrossbarNetwork* engine supports the construction of ad hoc networks with arbitrary connectivity such as the W-Network described in Section 3. The increased flexibility has also aided in the development of new network features such as metaplasticity that have been added to the *CrossbarNetwork* engine but not to the less adaptable *IzhikevichNetwork* engine.

In addition to the network engines, the SpikingNeuronLib library exports a number of additional types such as *FiringEvent*, *Triplet*, *Pattern* and *Stimulus*. There are also more task-specific types such as the *MatchScanner* that provides methods for searching the firing data for spatio-temporal patterns, and the *PatternGenerator* that provides different methods for the generation of random firing patterns.

A.2 The *SpikingAnalyticsLib* library

The SpikingAnalyticsLib library has two main purposes: firstly it extends the services provided by the SpikingNeuronLib library, providing additional functionality that wraps some of the core types exported from this library, particularly the CrossbarNetwork engine, Pattern and Stimulus; secondly, it provides a library of functions for the analysis of network data such as the changes in neuron membrane potentials and synaptic weights that occur as a network responds to a stimulus. Network firing data is the primary source for many analyses and this data is generated at a rate of more than 30 million firing events every hour in a simulated 1000 neuron network.

The extended CrossbarNetwork engine provided by the SpikingAnalyticsLib library simplifies data collection by adding two specialised data collector types: the OneMillisecTickDataCollector (briefly discussed in Section 3 and discussed in more detail in Section B), and the OneSecondTickDataCollec-

tor. The former collects membrane potential or synaptic weight data that quickly change over timescales close to the maximum time resolution of the simulation engine, while the latter collects only firing data which, for performance reasons, is made available just once each second. The large difference in the sampling rate of the two collectors (one millisecond versus one second) produces a difference in the performance penalty of enabling each collector type. For this reason the collection of firing data is enabled by default (as the OneSecondTickDataCollector has minimum overhead), while the collection of higher resolution data is provided as an option due to the performance penalty of enabling the OneMillisecTickDataCollector.

The extended CrossbarNetwork engine provides a number of additional functions including methods for network maturation, for saving compressed versions of the network state and for deducing the structure of an activated PNG. The structure of PNG activation is deduced by mapping the intervals between the firing events collected during a stimulus response to the connection delays between the corresponding neurons in the network. The ability to save compressed network state files is useful in experiments that examine changes in the network state over extended time periods where disk space usage can sometimes be substantial.

The SpikingAnalyticsLib library also defines types related to Response Profiling: ResponseFingerPrintFrame, ResponseFingerPrintFrameSet, FrameScore, FrameProfile, WindowMap, WindowMapSet, MatchedProfilePair, MatchedProfileSet and NaiveBayesClassifier are all types that support this methodology. Of these the most important are the FrameProfile and WindowMap types. A frame profile is a measure of the neural response to a repeated stimulus that is recorded for each neuron in the network. The profile is composed of spike count histograms that are generated for each neuron in a fixed-sized interval following each stimulus presentation. The WindowMap type uses these histograms to define *temporal windows*, peaked regions in the histograms where there is a high probability of post-stimulus spiking. See Guise et al. (2014) for more details.

There are many other types and functions included in the SpikingAnalyticsLib library: for example, extensions to the Pattern and Stimulus types provide many additional methods for defining or modifying firing patterns and stimuli; network specifier types provide a simplified means for defining the structure and topology of a new network and also provide a number of built-in network specifications; the PNGScanner type supports scanning for PNGs in collections of matured networks; and the PNGDescriptor type describes the connected structure of a PNG.

A.3 The *SpikingVisualisationLib* library

The SpikingVisualisationLib library provides convenience functions for plotting PNGs, spike rasters and response histograms. It can be used for the visualization of firing data or changes in neuron membrane potential and synaptic weights etc. It also supports the visualization of polychronous neural groups, using connectivity data from the network or from analytical data generated in the SpikingAnalyticsLib library. The current implementation manipulates plot data using *Deedle* and generates R plotting commands using the R Type Provider (BlueMountain Capital Management LLC, 2012a). It is therefore necessary to install the R Statistical Software (R Core Team, 2014) along with both of these libraries prior to using the SpikingVisualisationLib library.

B A walk-through of some scripts

Spinula scripts are written in a CLI-compliant language called F# (F-Sharp) that was chosen for its flexibility and expressiveness (Microsoft, 2014). The F# language supports two different modes of development: an interactive scripting mode and a compilation mode. Multiple modes allow the scripts in this section to be either compiled and executed as native code, or to be executed in an interactive environment called F-Sharp Interactive (FSI). The compilation option provides substantial performance advantages, while the interactive mode is useful for data manipulation and for rapid iterative development.

Spinula scripts are generally written in a simple procedural style although the F# language allows both functional and object-oriented styles of programming for more complex programming tasks. Although the scripts typically make use of the underlying functionality provided by the Spinula libraries, other libraries may also be referenced. Both Spinula libraries and any additional libraries must be declared in advance by including a reference at the start of each script. Most scripts only indirectly access the network simulation engine, instead utilizing the additional functionality provided by the *SpikingAnalyticsLib* library. Alternatively, Spinula scripts requiring direct access to the simulation data may reference the *SpikingNeuronLib* library.

The following sections present a selection of Spinula scripts in order to provide some additional background on the use of the Spinula libraries. They

are intended to be read primarily by those with a background in software development, and should ideally be read sequentially and together, as each new script expands on the last to provide a more comprehensive description of the scripting process. The first two scripts are composed of a simple sequence of commands, while later scripts employ more advanced techniques and make use of additional libraries features.

B.1 Finding PNGs

This script introduces the general procedures for creating a Spinula script and also the process of referencing the Spinula libraries. Both *network specification* and *network maturation* are discussed, and two Spinula types are introduced: the *IzhikevichNetwork* and *PNGScanner*. The purpose of this script is to find polychronous neural groups in a simulated network running inside the IzhikevichNetwork engine. The script creates a new IzhikevichNetwork and matures it for two hours before scanning for adapted PNGs in the network structure. The network weights are then scrambled and the network is rescanned. The PNG counts, both before and after scrambling the synaptic weights, are then reported. The script reproduces an experiment reported in Izhikevich et al. (2004) that showed a substantial reduction in PNG numbers following synaptic disruption of a matured network.

Izhikevich (2006b) has created two unique algorithms for the discovery of polychronous neural groups, a fast *event-driven* algorithm and a slower more precise *activity-driven* algorithm (see Izhikevich et al., 2004, Izhikevich, 2006a). The Spinula implementation of these algorithms is described in Guise et al. (2013). While both algorithms are intended to find what Martinez and Paugam-Moisy (2009) call *supported* PNGs, the slower algorithm tests the ability of each potential PNG to polychronise and can therefore discover *adapted* PNGs i.e. groups in which the synaptic weights have been adjusted by STDP to be compatible with polychronisation. Neither of these algorithms is suited to finding *activated* PNGs that produce observable changes in the firing data, although many of the PNGs discovered by the slower algorithm may have the potential for activation if presented with a compatible stimulus.

The maturation step is particularly important: networks are usually initialized with either uniform synaptic weights or with small random variations in the weight distribution, neither of which are conducive to the presence of adapted PNGs. Maturation of a network involves running the simulated network with a stimulation protocol that produces random neural firing, usually

at around 1 Hz (i.e. each neuron in the network is fired at a random time such that it fires *on average* once per second). This maturation protocol is continued for sufficient time to ensure that the synaptic weights come to an equilibrium with the dynamics of the network produced by this low level of stimulation. Izhikevich (2006a) typically matured the network for 24 hours (internal simulation time) although just two hours simulation time seems to be sufficient (Guise et al., 2013).

The script demonstrates the use of the slower algorithm for finding PNGs and is shown in Listing B.1. The first few lines of the script specify references to the required Spinula libraries. Within these reference declarations the quoted strings normally specify the full filesystem paths to each dynamic link library (DLL) although only the filename is shown in these script examples. The script interpreter dynamically loads the libraries specified here, allowing it to resolve the type and method names that are referenced throughout the script. Some of the .Net Framework libraries are referenced here by default: an example of one of these implicit references is the *System* library which is *opened* in line 5 even though the underlying library is not explicitly referenced. The *open* statements are used to provide easy access to the specified libraries. In the Common Language Infrastructure, access to the methods in a referenced library normally requires a *namespace qualification*, a long path that is prepended to each method call. However, short-cut access can be provided by *opening* the required libraries as shown in lines 5 - 7.

In line 9 the implicit reference to the System library is used to retrieve the default document folder path for the Microsoft Windows platform. Lines 17 to 20 define the parameters of the new Izhikevich network: an N1000 network with 800 excitatory neurons and 200 inhibitory neurons, 100 synapses per neuron, and a maximum synaptic delay of 20 milliseconds. This specification is packaged together into a network specifier in lines 21 and 22 that can be used to reload the network in later steps. Network specifiers are convenience types that allow network parameters to be passed around and saved. There are separate specifiers for each network engine i.e. the IzhikevichNetworkSpecifier (used here), and the CrossbarNetworkSpecifier used in later scripts. The CrossbarNetworkSpecifier has additional attributes that reflect the increased feature set of this more flexible network engine (e.g. optional attributes to specify metaplasticity parameters).

Even without maturation, the new network constructed in lines 25 and 26 will support many structural PNGs, although PNGs with adapted weights are unlikely to occur. Spinula provides a convenience type called the PNGScanner to simplify the scanning process for either supported or adapted PNGs,

```
#r @"SpikingNeuronLib.dll"
#r @"SpikingAnalyticsBaseLib.dll"
#r @"SpikingAnalyticsLib.dll"

open System
open SpikingNeuronLib
open SpikingAnalyticsLib

let outputFolder = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)
let algorithmVersion = FindGroupsVersion.Current        // slower activity-driven PNG search algorithm
let runSeconds = 2 * 3600                               // mature for this many seconds
let backgroundFiringRate = 1                            // with a one hertz random background
let matureNetworkBaseName = "MatureNetwork"
let matureNetworkName = sprintf "%s_%d.txt" matureNetworkBaseName (runSeconds / 60)

// Network definition
let numExcitatoryNeurons = 800
let numInhibitoryNeurons = 200
let numSynapsesPerNeuron = 100
let maxDelay = 20
let networkSpecifier = new IzhikevichNetworkSpecifier(numExcitatoryNeurons, numInhibitoryNeurons,
        numSynapsesPerNeuron, maxDelay)

// Create a new Izhikevich network
let network = new IzhikevichNetwork(numExcitatoryNeurons, numInhibitoryNeurons,
        numSynapsesPerNeuron, maxDelay)

// Mature the network and save the network state to a file
PNGScanner.GenerateMaturationStateFiles(network, None, runSeconds, backgroundFiringRate,
        outputFolder, matureNetworkBaseName)

// Get a count of the PNGs in the network state file
let beforeCount = PNGScanner.FindAllPNGsInStateFile(outputFolder, matureNetworkName, networkSpecifier,
        "before.txt", algorithmVersion)

// Load the same network state file and scramble the network
let scrambledNetwork = PNGScanner.ScrambleNetworkInStateFile(networkSpecifier,
        outputFolder, matureNetworkName)

// Get a count of the PNGs in the scrambled network
let afterCount = PNGScanner.FindAllPNGsInState(scrambledNetwork, outputFolder,
        "after.txt", algorithmVersion)

printfn "%d %d" beforeCount afterCount
```

Listing B.1: A script for finding PNGs in an Izhikevich Network.

and all subsequent steps in the script (maturation, synaptic shuffling and scanning for PNGs) are performed using static methods on this type. The maturation step is performed in lines 29 and 30 and the resulting matured network is then saved to a network state file in the output folder. The saved state file is then loaded in lines 33 and 34 and scanned for PNGs. Lines 37 and 38 load a new instance of the saved state file and shuffle the synaptic weights. The shuffled network is then rescanned in lines 41 and 42, returning the number of adapted PNGs in the network after shuffling. Finally, the PNG count before and after shuffling is reported in line 44 using a standard function for formatting and writing data to the console window.

B.2 Response Fingerprinting

This next script example provides an introduction to *Response Fingerprinting*, particularly the *FrameProfile* and *WindowMap* types. Also new in this script is the use of a *CrossbarNetwork*, the saving of *network state files* and the creation of a stimulus using the *Pattern* and *Stimulus* types. A Response Fingerprint is a probabilistic representation of the firing pattern produced by PNG activation as a trained network responds to an input stimulus (see Guise et al. (2014) for a description of the method). The script demonstrates the effect of stimulus training on the size of the stimulus-specific PNG activation response, using the number of temporal windows in the fingerprint to determine the size of the activation response. The generation of a Response Fingerprint has two steps: in the first step a *frame profile* is created that represents the stimulus response over a defined number of fixed-sized frames; and in the second step *temporal windows* are identified and mapped within the profile data. The SpikingAnalyticsLib library defines two types within the *ResponseFingerPrint* namespace that implement these steps: the FrameProfile type creates a frame profile, and the WindowMap type performs the mapping of temporal windows. These two important types are further described in the following sections.

B.2.1 The FrameProfile type

A *FrameProfile* is a measure of the consistent spiking response of selected neurons as the network responds to a known stimulus. It is generated by repeatedly presenting the stimulus within fixed-sized *response frames* that are sufficiently long to encompass both the stimulus and the subsequent firing response. Histograms of the spike counts at each temporal offset within

the response frame are aggregated across multiple frames, and these *response histograms* are stored for each selected neuron. Creating a frame profile is the most time-consuming step in the fingerprinting process, with the processing time depending on the required sensitivity. Using typical parameters of a 250 millisecond frame size and one hundred seconds of simulation produces 400 response frames of data. Additional response frames can be added to increase the sensitivity of the fingerprint if the neural response to a stimulus is minimal, such as with an unknown stimulus.

B.2.2 The WindowMap type

A *WindowMap* is a data structure for storing the temporal windows that define a Response Fingerprint and consists of a mapping between each selected neuron and the temporal windows that have been identified for that neuron. This mapping is generated from *FrameProfile* data (see Section B.2.3) and is stored in a raw form that supports different views of the temporal windows (see Section B.2.4). The WindowMap type provides window filtering and selection methods and a range of other methods that manipulate the raw window data e.g. for partitioning windows into layers, or for comparing response profiles generated under different network conditions, or with different stimuli.

B.2.3 WindowMap Generation

Unlike the generation of a FrameProfile, WindowMap generation is very fast. Initial peak isolation involves scanning a small fixed-sized window across the response histogram data for each neuron, selecting peaks in the accumulated spike counts. Any peaks that pass an initial threshold are isolated as potential windows by extending the histogram data within and around the peak. The raw WindowMap data for each selected neuron consists of a pair of lists representing the temporal offsets and spike counts for each of the isolated peaks. Initial thresholding is set low by default so that small spike count peaks are not excluded. Support for additional thresholding of the raw window data is provided using a range of different criteria.

The algorithm for determining peaks in the data has three control parameters: the *base threshold*, the *initial consistency threshold*, and the *window size*. Spike counts must be above the base threshold (the base spike count level) in order to be included within a peak. Contiguous temporal offsets

with spike counts above the base threshold must additionally contain a minimum cumulative spike count that is calculated from the initial consistency threshold. The window size determines the minimum number of contiguous offsets that can be tested to meet the initial consistency criterion. For those windows that pass, the isolated contiguous group is then extended at the leading and trailing edges until the spike counts either dip below the base threshold or begin increasing (indicating an adjacent peak).

B.2.4 WindowMap Views

The Response Fingerprinting method can be used in diverse ways such as examining changes in PNG activation with stimulus degradation, or exploring the efficiency of different training methods. These different uses sometimes require different analyses of the temporal window data: changes in the number of temporal windows might be sufficient for a simple analysis of the fingerprint data, but more complex experiments might also require that any changes in temporal window properties be examined e.g. changes in the temporal precision of each peak, or in the temporal offset relative to the stimulus. For this reason the data in a WindowMap is stored in a raw form that allows different views of the underlying window data.

One such view is the spike count ratio view (see *ExtractRatioData*) that computes values describing the *selectivity* and the *consistency* of the isolated window peak for each temporal window. Selectivity is measured as a ratio of the spike count within the peak relative to the total spike count, a ratio that reflects the proportion of spikes that are captured by the window. Consistency is the ratio of the spike count within the peak relative to the total number of frames and reflects the proportion of trials in which the window captured a spike.

Another view is the mean-variance view (see *ExtractMeanVarianceData*) which computes the mean and variance of each of the selected temporal windows. The variance is particularly useful for filtering the initial window selection on the basis of the temporal precision of each window peak, but can also be used to investigate changes in the temporal precision of windows with training or with changes in network properties (e.g. see Guise et al., submitted). Such comparisons require a mapping between the temporal windows of two WindowMaps (e.g. WindowMaps generated from the same network but with different network conditions). The mean value is primarily used to facilitate these comparative mappings but is also useful for studying shifts in window position under different network conditions.

B.2.5 Scripting the FrameProfile and WindowMap types

The script in Listing B.2 begins by constructing a new CrossbarNetwork and maturing it over two hours (internal simulation time). It then profiles the network's response to a 5 Hz stimulus by analyzing the firing events generated by the network over a one hundred second simulation run. Using the default frame size of 250 msec this produces 400 frames of firing data. The script generates the following files: a *network state file* representing the network state after maturation, another state file representing the network state after training, and two files that store a frame profile of the network (before and after training). The filenames of these files include some metadata that describes either the number of seconds of maturation or the number of seconds of training. For this reason the filenames of the saved network states are constructed on-the-fly by appending the appropriate metadata to the *base names* defined in lines 21 and 22. Likewise, the filenames of the saved network profiles are constructed from the corresponding network state base names using a string formatting function called *sprintf* in lines 23 and 24.

Construction of the stimulus involves defining a pattern and then specifying the pattern repetition frequency. The pattern is created as a linear sequence of 40 firing events, fired at one millisecond intervals and with the neurons selected at uniform intervals across the range of neuron indices (line 27).[1] Other static methods on the Pattern type provide alternative means of defining the stimulus parameters. The Stimulus is then constructed from this pattern at line 28 with a repetition frequency of 5 Hz. The method call in lines 31 and 32 performs both initialization and maturation of a new CrossbarNetwork with a network specifier that defines a 1000 neuron network with randomly assigned connections, and additional arguments that specify the output folder path and the network state base name used to construct the filename for saving the matured network. The matured network is then reloaded in lines 35 and 36 and the first of two profiles is generated (*before training*) in lines 40 and 41. The pattern used to construct the stimulus is passed to the profiling method and the resulting profile (saved to a file in line 44) represents the response of the network to that pattern. At this point of the training procedure the network has had no prior exposure to the pattern and the resulting profile is unlikely to show a strong response.

The script constructs a WindowMap for the untrained network at line 47 so that the number of temporal windows can be counted at a later stage.

[1] This combination of parameters produces the Ascending pattern of Fig. 1

```fsharp
#r @"SpikingNeuronLib.dll"
#r @"SpikingAnalyticsBaseLib.dll"
#r @"SpikingAnalyticsLib.dll"

open System
open System.IO
open SpikingNeuronLib
open SpikingAnalyticsLib
open SpikingAnalyticsLib.ResponseFingerPrint
open SpikingAnalyticsLib.PatternExtensions

let outputFolder = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)
let backgroundFiringRate = 1            // one hertz background for maturation, training and profiling
let patternName = "Ascending"           // pattern used for training and profiling
let includeInhibitoryNeurons = false    // inhibitory neurons included for fingerprinting
let runSeconds = 2 * 3600               // two hour maturation period
let patternStimulationsPerSecond = 5    // training frequency
let trainSeconds = 120                  // training period
let verbose = true                      // profiling feedback

let matureNetworkBaseName = "MatureNetwork"
let trainedNetworkBaseName = "TrainedNetwork"
let profileNameMatureNetwork = sprintf "profile_%s.txt" matureNetworkBaseName
let profileNameTrainedNetwork = sprintf "profile_%s.txt" trainedNetworkBaseName

// Create a linear firing pattern composed of 40 firing events, one each millisecond
let pattern = Pattern.FromLinearSequence(1, 1, 40)
let stimulus = Stimulus.Create(patternStimulationsPerSecond, pattern)

// Create a mature network composed of 1000 randomly connected neurons
CrossbarNetwork.CreateMatureNetwork(runSeconds, backgroundFiringRate,
        CrossbarNetworkSpecifier.N1000Network, outputFolder, matureNetworkBaseName)

// load the matured network
let matureNetwork = CrossbarNetwork.CreateFromFile(
        Path.Combine(outputFolder, sprintf "%s%d.txt" matureNetworkBaseName runSeconds)
        )

// Profile the network's response to the stimulus
let profileBefore = new FrameProfile(matureNetwork, Some(pattern),
        patternName, backgroundFiringRate, verbose)

// Save the profile
profileBefore.Save(Path.Combine(outputFolder, profileNameMatureNetwork))

// Create a response fingerprint (temporal windows)
let windowMapBeforeTraining = new WindowMap(profileBefore, includeInhibitoryNeurons, verbose)

// Train the network on a 5 Hz stimulus for 120 secs
matureNetwork.Train(stimulus, trainSeconds, backgroundFiringRate, outputFolder, trainedNetworkBaseName)

// load the trained network
let trainedNetwork = CrossbarNetwork.CreateFromFile(
        Path.Combine(outputFolder, sprintf "%s%d.txt" trainedNetworkBaseName trainSeconds)
        )

// Re-profile the network's response to the stimulus
let profileAfter = new FrameProfile(trainedNetwork, Some(pattern),
        patternName, backgroundFiringRate, verbose)

// Save the profile
profileAfter.Save(Path.Combine(outputFolder, profileNameTrainedNetwork))

// Create the post-training response fingerprint
let windowMapAfterTraining = new WindowMap(profileAfter, includeInhibitoryNeurons, verbose)

printfn "%d %d" (windowMapBeforeTraining.AllWindows.Count) (windowMapAfterTraining.AllWindows.Count)
```

Listing B.2: A script that performs Response Fingerprinting.

The network is then trained on the stimulus at line 50 with the trained network state being saved to a file at training completion. The saved state is then reloaded in lines 53 and 54 and profiled in lines 58 and 59. The profile saved in line 62 should now show a strong response to the stimulus and the WindowMap generated from the profile at line 65 will reflect the strong response by identifying a substantially larger number of temporal windows relative to the earlier profile (for the untrained network). The count of temporal windows both before and after training is reported in the final line.

B.3 Background pattern generator

The next script is a version of the script shown in Listing 2.1 that has been modified to allow the script to be run with different parameters. References to the required Spinula libraries have also been added. The script has been rewritten as a *function definition with parameters*, and a *function call with matching arguments*. The modified script allows a range of experimental scenarios to be tested by simply varying the values supplied as arguments to the function. The script also demonstrates the use of one of the *data collector* types (*OneSecondTickDataCollector*) and the use of a *SpikeVisualisation* method for visualizing the firing data. The script references the R type provider (lines 4 and 5) and a data manipulation library called *Deedle* (lines 6 and 7) both of which are dependencies of the visualization library (see BlueMountain Capital Management LLC, 2012a,b).

The function definition in Listing B.3 begins at line 16 and continues through to line 37. The function call is on the final line of the script at line 41 and passes the value ten as the background frequency, and a string value computed at lines 39 and 40 as the output file path. Within the function definition, the sequence of script commands is largely identical to the original script in Listing 2.1. A new network is created using a built-in network specifier that creates a 1000 neuron network with no connections. The network is then run with just random background stimulation at the frequency specified by the first function parameter, and the third frame of firing data is saved to the file specified by the second function parameter. Selection of the third frame at lines 30 and 31 uses the *OneSecondEventCollector* property of the CrossbarNetwork type to retrieve the data collector (of type *OneSecondTickDataCollector*) that was used behind the scenes to collect firing events. The selected firing events are then displayed at line 37 using a static method on the *SpikeVisualisation* type.

```
#r @"SpikingNeuronLib.dll"
#r @"SpikingAnalyticsBaseLib.dll"
#r @"SpikingAnalyticsLib.dll"
#I @"RProvider.1.0.12"
#load "RProvider.fsx"
#I @"Deedle.1.0.0"
#load "Deedle.fsx"
#r @"SpikingVisualisationRLib.dll"    // references Deedle
#r @"SpikingAnalyticsFrameLib.dll"    // references Deedle

open System
open System.IO
open SpikingAnalyticsLib
open SpikingVisualisationRLib

let TestBackgroundPatternGenerator (backgroundFrequency:int) outputFilePath =

    let verbose = true
    let runSeconds = 5
    let showOnlyForegroundEvents = false

    // create a new network with no connections between neurons
    let network = CrossbarNetwork.CreateAdHocNetwork(
                      CrossbarNetworkSpecifier.N1000_Unconnected_Network, None, verbose)

    // run the network with background stimulation but no stimulus and collect firing data
    network.Run(runSeconds, None, backgroundFrequency)

    // select the third one second frame of firing data (include background firing events)
    let thirdFrameData = network.OneSecondEventCollector.SelectRange(
                      2000, 3000, showOnlyForegroundEvents)

    // save the data
    thirdFrameData.Save(outputFilePath)

    // show the data as a spike raster
    SpikeVisualisation.ShowSpikeRaster(thirdFrameData.AllEventPairs)

let outputFolder = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)
let outputFilePath = Path.Combine(outputFolder, "temp.txt")
TestBackgroundPatternGenerator 10 outputFilePath
```

Listing B.3: A script that tests the Poisson Process background pattern generator.

B.4 The effects of weights and delays on the membrane potential

The next script shown in Listing B.4 demonstrates the importance of the afferent connection weights and delays in determining the dynamics of the neuron membrane potential. It defines two functions, an outer *helper* function that generates the stimulus for an inner function that runs a network simulation while collecting membrane potential data from a single output neuron. The script demonstrates the effect of varying the arguments to a function, explains the importance of *indenting*, and introduces the use two new types: the *OneMillisecTickDataCollector* type, used to collect synaptic weight and membrane potential data, and the *OneMillisecTickDataCollectorParameters* type used to specify the data collection parameters.

The script specifies a network with a single output neuron and two or more input neurons, and defines the number of inputs as a parameter to each of the two functions. The helper function uses the number of inputs to construct a stimulus that will fire all input neurons simultaneously. The idea is that spikes will propagate along each of the connections between the input neurons and the single output neuron, with the connection lengths determining the propagation delays and hence the degree of spike convergence on the output neuron. Firing of the output neuron is therefore a function of both the connection weights and the connection delays, with the delays determining the degree of spike convergence. The first function (*GenerateMembraneData*) sets up and runs the network simulation and collects the membrane potential data from the output neuron. The second function (*GenerateMembraneDataHelper*) generates a stimulus appropriate to the number of inputs, calls the first function and then displays the collected data.

The previous script used the OneSecondTickDataCollector type to collect firing data over the course of the simulation. The collection of firing data is enabled by default in a CrossbarNetwork and the OneSecondTickDataCollector type is therefore implicitly created during network construction. In contrast, the collection of synaptic weight or membrane potential data requires the explicit construction of the OneMillisecTickDataCollector type which is then passed to the network constructor method. Lines 19 to 40 of the script show an example of this explicit construction method. The indenting throughout the script is critical as it determines the lines in the script which should be grouped together. Lines 20 to 40 define the construction of the network and all have (at least) the same level of indenting, forming a single functional group within the script. Within this group there are

```
#r @"SpikingNeuronLib.dll"
#r @"SpikingAnalyticsBaseLib.dll"
#r @"SpikingAnalyticsLib.dll"
#I @"RProvider.1.0.12"
#load "RProvider.fsx"
#I @"Deedle.1.0.0"
#load "Deedle.fsx"
#r @"SpikingVisualisationRLib.dll"    // references Deedle
#r @"SpikingAnalyticsFrameLib.dll"    // references Deedle

open System
open SpikingNeuronLib
open SpikingAnalyticsLib
open SpikingVisualisationRLib

// Generate membrane voltage data for a network with a single output neuron
let GenerateMembraneData numberOfInputs delays (weights:float list) stimulus =

    let network =
        let numberOfSamples = 5000
        let specifier =
            let postNeuron = numberOfInputs
            let connections =
                // zip three lists together: the presynaptic neurons, the delays and the weights
                Seq.zip3 [ for i in 0..numberOfInputs-1 -> i ] delays weights
                |> Seq.map (fun (preNeuron, delay, weight) ->
                  new Connection(preNeuron, postNeuron, delay, weight))
                |> Seq.toList
            new CrossbarNetworkSpecifier(numberOfInputs + 1, 0, 20, connections)
        let hiResCollector =
            let selectedNeurons = [ specifier.TotalNeurons-1 ]   // just the output neuron
            let selectedConnections = []
            let totalNeurons = specifier.TotalNeurons
            let totalConnections = specifier.Connections.Value.Count
            let numberOfMembraneSamples = numberOfSamples
            let numberOfWeightSamples = 0
            let parameters = new OneMillisecTickDataCollectorParameters(selectedNeurons, totalNeurons,
              selectedConnections, totalConnections, numberOfMembraneSamples, numberOfWeightSamples)
            new OneMillisecTickDataCollector(parameters)
        CrossbarNetwork.CreateAdHocNetwork(specifier, Some(hiResCollector), false)

    network.Run(10, Some(stimulus :> IStimulus), 0, false)
    network.OneMillisecondEventCollector.Value

// Generate and show membrane data
let GenerateMembraneDataHelper numberOfInputs delays weights =

    let stimulus =
        let firingEvents =
            let times = [ for i in 0..numberOfInputs-1 -> 0 ]
            let neurons = [ for i in 0..numberOfInputs-1 -> i ]
            Seq.zip times neurons
            |> Seq.map (fun (time, nindex) -> new FiringEvent(time, nindex, EventLabel.Foreground))
            |> Seq.toArray
        let patternStimulationsPerSecond = 1
        Stimulus.Create(patternStimulationsPerSecond, firingEvents)

    let dataCollector = GenerateMembraneData numberOfInputs delays weights stimulus
    MillisecondResolutionDataVisualisation.ShowCollectedMembraneData(dataCollector, 1000, 1100)

GenerateMembraneDataHelper 2 [ 5; 5; ] [ 10.0; 10.0; ]
GenerateMembraneDataHelper 2 [ 5; 5; ] [ 9.0; 8.0; ]
GenerateMembraneDataHelper 2 [ 5; 5; ] [ 8.0; 8.0; ]

GenerateMembraneDataHelper 2 [ 6; 5; ] [ 10.0; 10.0; ]
GenerateMembraneDataHelper 2 [ 8; 5; ] [ 10.0; 10.0; ]
GenerateMembraneDataHelper 2 [ 12; 5; ] [ 10.0; 10.0; ]
```

Listing B.4: A script that demonstrates the effect of varying afferent connection weights and delays on the neuron membrane potential.

nested groups that define the network specifier (lines 22 to 29) and the data collection parameters (lines 31 to 39).

The data collector can sample both neuron data and connection data by default. Data collection for each type is enabled by passing a non-empty list to either the *selected neurons* or *selected connections* parameter of the OneMillisecTickDataCollectorParameters constructor. For this example we are sampling only neuron data i.e. the membrane potential and the membrane recovery variable. Line 31 specifies data collection on the neuron with the highest index (i.e. the output neuron) while line 32 specifies an empty list for the selected connections, thereby disabling the sampling of connection data. The OneMillisecTickDataCollector is then constructed at line 39 and the simulation is run for ten seconds with an appropriately sized stimulus at line 42. The data collector is used not only to collect membrane data but also to pass the stored data between the two functions. The final line (line 43) of the GenerateMembraneData function determines the function value that is passed back to the calling function at line 58. The stored membrane data is then displayed at line 59 using a static method on the MillisecondResolutionDataVisualisation type. Lines 61 to 67 call the outer (helper) function six times, with different arguments for the connection delays and weights for each function call. The resulting membrane data plots show that both insufficient connection weights, or delays that cause too much divergence in the spike arrival times have a similar effect in preventing the output neuron from firing.

B.5 Train a W-Network

The final script simulates the network described in Section 3 referred to as a *W-Network*. Like the previous script, the script shown in Listing B.5 uses a *OneMillisecTickDataCollector* to collect neuron membrane potential data, but as the W-Network experiment requires a comparison between the neuron and connection data the synaptic weight data is also collected. New features introduced in this script are *type-casting*, the use of *options*, *saving* the collected data, and the display of both membrane potential and synaptic weight data using methods on the *MillisecondResolutionDataVisualisation* type.

The script defines a single function (*TrainWNetwork*) with parameters that specify the output folder path (for saving the collected data) and the background stimulation frequency (see the description in Section 3 for more details of the experimental setup). A built-in network specifier is selected at line

```fsharp
#r @"SpikingNeuronLib.dll"
#r @"SpikingAnalyticsBaseLib.dll"
#r @"SpikingAnalyticsLib.dll"
#I @"RProvider.1.0.12"
#load "RProvider.fsx"
#I @"Deedle.1.0.0"
#load "Deedle.fsx"
#r @"SpikingVisualisationRLib.dll"    // references Deedle
#r @"SpikingAnalyticsFrameLib.dll"    // references Deedle

open System
open System.IO
open SpikingNeuronLib
open SpikingAnalyticsLib
open SpikingAnalyticsLib.PatternExtensions
open SpikingVisualisationRLib

let TrainWNetwork pathToOutputFolder backgroundFrequency =

    let verbose = true
    let runSeconds = 100
    let stimulusFrequency = 10

    let specifier = CrossbarNetworkSpecifier.W_Network

    let hiDataResCollector =
        // membrane data: record the first 1000 samples from the run (one sample per msec)
        let numberOfMembraneSamples = 1000
        // weight data: record the entire run (one sample per msec)
        let numberOfWeightSamples = runSeconds * 1000
        let totalNeurons = specifier.TotalNeurons
        let totalConnections = specifier.Connections.Value.Count
        // collect membrane data for neurons: 0, 1, 2, 3, 4
        let selectedNeurons = [ for i in 0..totalNeurons-1 -> i ]
        // collect synaptic weight data for connections: 0, 1, 2, 3
        let selectedConnections = [ for i in 0..totalConnections-1 -> i ]
        let parameters = new OneMillisecTickDataCollectorParameters(selectedNeurons, totalNeurons,
            selectedConnections, totalConnections, numberOfMembraneSamples, numberOfWeightSamples)
        new OneMillisecTickDataCollector(parameters)

    let network = CrossbarNetwork.CreateAdHocNetwork(specifier, Some(hiDataResCollector), verbose)

    // create an input pattern: repeated firing of both input layer neurons simultaneously (at time 0)
    let stimulus =
        let pattern =
            let times = [| 0; 0; |]
            let neurons = [| 0; 1; |]
            Pattern.FromFiringSequence(times, neurons)
        Stimulus.Create(stimulusFrequency, pattern)

    // run the network with this pattern and collect membrane and weight data
    network.Run(runSeconds, Some(stimulus :> IStimulus), backgroundFrequency)
    let hiResDataCollector = network.OneMillisecondEventCollector.Value

    // save the membrane potential data (V and U)
    let membraneDataPath = Path.Combine(pathToOutputFolder, "membraneData.txt")
    hiResDataCollector.SaveMembraneData(membraneDataPath)

    // save the synaptic weight data (weight and derivative)
    let weightDataPath = Path.Combine(pathToOutputFolder, "weightData.txt")
    hiResDataCollector.SaveConnectionData(weightDataPath)

    // show the membrane potential plots
    MillisecondResolutionDataVisualisation.ShowCollectedMembraneData(hiResDataCollector)

    // Create a new plot window
    VisualisationUtilities.NewWindow()

    // show the synaptic weight plots
    MillisecondResolutionDataVisualisation.ShowCollectedWeightData(hiResDataCollector)
```

Listing B.5: A script that shows the changes in synaptic weights in a W-Network during training.

24 and the data collection parameters are defined in lines 26 to 39. The construction method at line 41 uses an *option* type for the second parameter: passing *Some(hiDataResCollector)* for this parameter will include this optional data collector type in CrossbarNetwork construction, while passing *None* for the second parameter will prevent the collection of neuron and connection data (although firing data will still be collected using the implicit OneSecondTickDataCollector). The stimulus is created between lines 44 and 49, first defining a firing pattern that fires both input layer neurons simultaneously at $t = 0$, and then constructing a *Stimulus* by repeating the firing pattern every 100 milliseconds (i.e. 10 Hz). The script uses an array of firing times and an array of input neuron indices to specify the required *Pattern* type.

The simulation begins at line 52 and runs for one hundred seconds with the background frequency specified by the second function parameter. The second parameter on the *Run* method (of the CrossbarNetwork type) is also an option type, allowing an optional stimulus of type *IStimulus* to be specified. Passing either *Some(IStimulus)* or *None* for this parameter will run the simulation either with or without a stimulus respectively. IStimulus is an *interface* type of which there are currently two implementations: *Stimulus* and *MultiStimulus*. The stimulus in this script is of type Stimulus and therefore must be type-cast (using the upcast operator :>) to the expected (IStimulus) type.

After the simulation completes, the data collector contains one thousand membrane potential samples and 100,000 weight samples (line 53). Membrane potential data is collected only in the first second of the simulation as the network responds to the repeated stimulus in a similar way over the course of the simulation. In contrast, the synaptic weight data is expected to change continuously as the network is trained on the stimulus, and therefore the weight data is collected throughout the run. The collected membrane data is saved in lines 56 and 57 and the corresponding weight data in lines 60 and 61. The final few lines display first the membrane data (line 64) and then the synaptic weight data (line 70). In order to prevent the first plot being overwritten by the second plot line 67 creates a new window in readiness for the second data visualization method call.

References

BlueMountain Capital Management LLC, 2012a. F# R Type Provider.

URL http://bluemountaincapital.github.io/FSharpRProvider/

BlueMountain Capital Management LLC, 2012b. Deedle: Exploratory data library for .NET.
URL http://bluemountaincapital.github.io/Deedle/

Guise, M., Knott, A., Benuskova, L., 2013. Experiments on the effect of synaptic disruption on polychronous group formation: detailed methods and results (OUCS-2013-02). Tech. rep., Dept of Computer Science, University of Otago, Dunedin, see http://www.cs.otago.ac.nz/research/techreports.php.

Guise, M., Knott, A., Benuskova, L., 2014. A bayesian model of polychronicity. Neural Computation 26 (9), 2052–2073.

Guise, M., Knott, A., Benuskova, L., submitted. Enhanced polychronisation in a spiking network with metaplasticity. Frontiers in Computational Neuroscience.

Izhikevich, E. M., Feb. 2006a. Polychronization: computation with spikes. Neural computation 18 (2), 245–82.

Izhikevich, E. M., 2006b. Reference software implementation for the Izhikevich model: Minimal spiking network that can polychronize.
URL http://www.izhikevich.org/publications/spnet.htm

Izhikevich, E. M., Gally, J. A., Edelman, G. M., Aug. 2004. Spike-timing dynamics of neuronal groups. Cerebral cortex (New York, N.Y. : 1991) 14 (8), 933–44.

Martinez, R., Paugam-Moisy, H., 2009. Algorithms for structural and dynamical polychronous groups detection. In: Alippi, C., Polycarpou, M., Panayiotou, C., Ellinas, G. (Eds.), Artificial Neural Networks ICANN 2009. Vol. 5769 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 75–84.
URL http://dx.doi.org/10.1007/978-3-642-04277-5_8

Microsoft, 2014. Microsoft Developer Network - Visual F# Development Portal.
URL http://msdn.microsoft.com/en-us/library/ff730280.aspx

R Core Team, 2014. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org