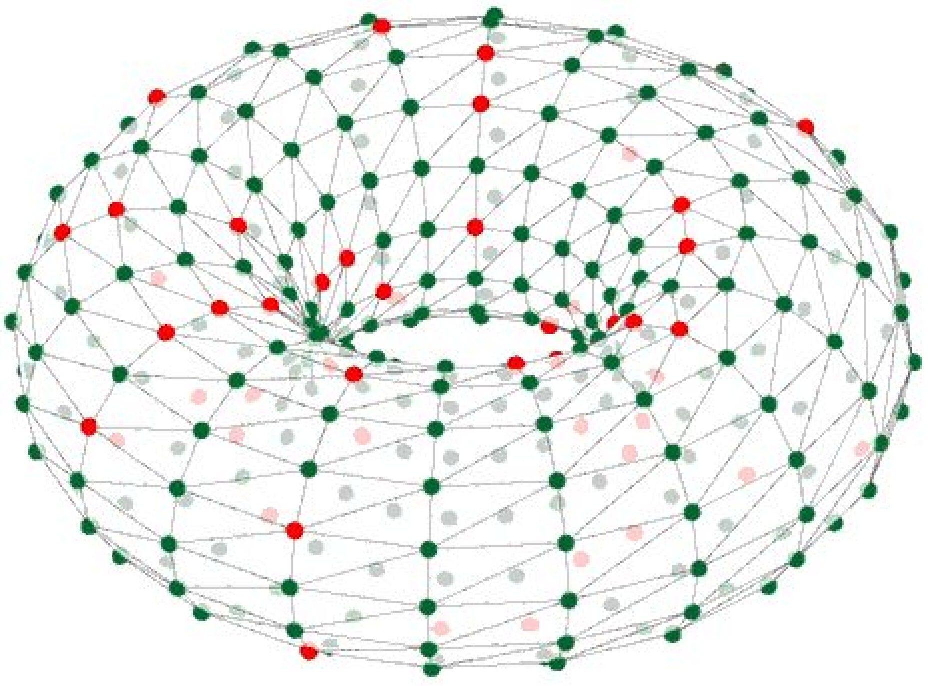
5th SpiNNaker Workshop

Lab Manuals

September
7th- 12th 2015



Manchester, UK

5th SpiNNaker Workshop

Day 1

September 7th 2015

Time	Session	Presenter(s)
12:00	Lunch and Registration	
13:00	Workshop logistics	Simon
13:15	SpiNNaker Hardware & Tools Overview	Simon
14:00	Running PyNN simulations on SpiNNaker	Andrew
15:00	Coffee	
15:30	Lab time	
17:00	Close	

Manchester, UK

Running PyNN Simulations on SpiNNaker

Introduction

This manual will introduce you to the basics of using the PyNN neural network language on SpiNNaker neuromorphic hardware.

Installation

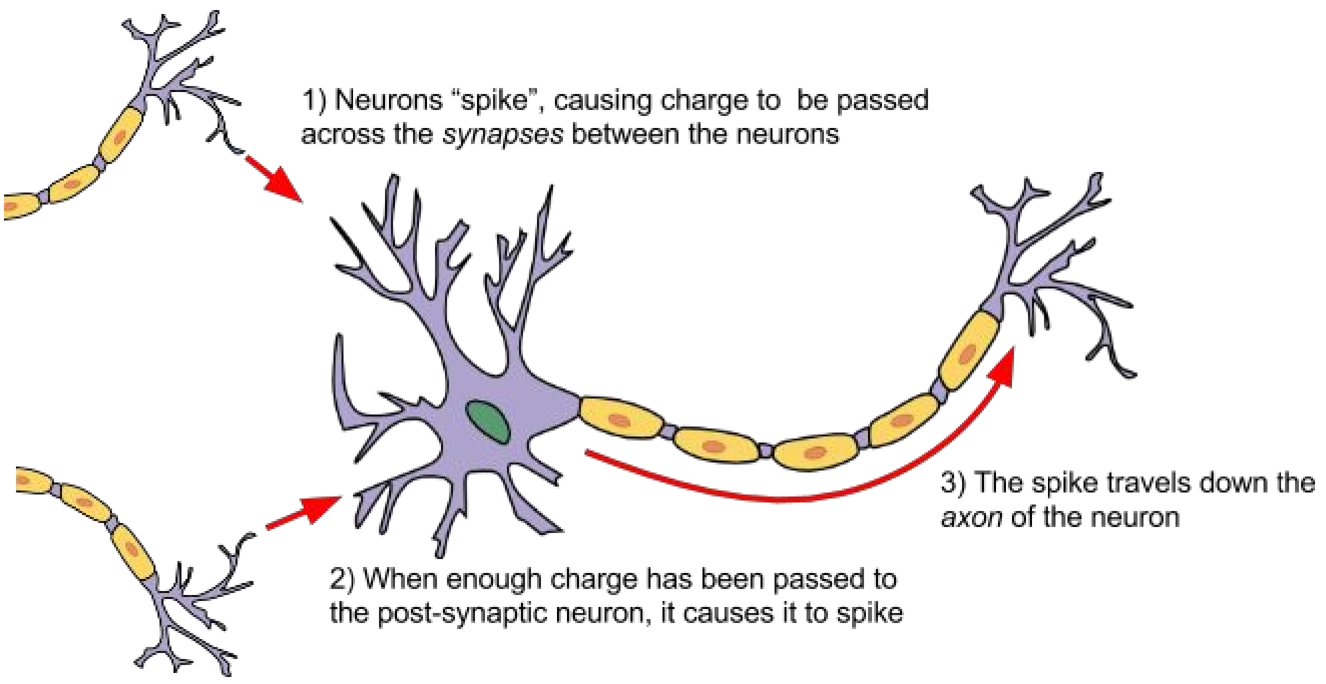
The PyNN toolchain for SpiNNaker (sPyNNaker), can be installed by following the instructions available from here:

http://spinnakermanchester.github.io/2015.005.Arbitrary/PyNNOnSpinnakerInstall.html

Matplotlib is marked as optional, but you will also need to install this dependency to complete some of the forthcoming exercises.

Spiking Neural Networks

Biological neurons have been observed to produce sudden and short increases in voltage, commonly referred to as spikes. The spike causes a charge to be transferred across the synapse between neurons. The charge from all the presynaptic neurons connected to a postsynaptic neuron builds up, until that neuron releases the charge itself in the form of a spike. The spike travels down the axon of the neuron which then arrives after some delay at the synapses of that neuron, causing charge to be passed forward to the next neuron, where the process repeats.


1) Neurons "spike", causing charge to be passed across the *synapses* between the neurons
3) The spike travels down the *axon* of the neuron
2) When enough charge has been passed to the post-synaptic neuron, it causes it to spike

Artificial spiking neural networks tend to model the membrane voltage of the neuron in response to the incoming charge over time. The voltage is described using a differential equation over time, and the solution to this equation is usually computed at fixed time-steps within the simulation. In addition to this, the charge or current flowing across the synapse can also be modelled over time, depending on the model in use.

The charge can result in either an excitatory response, in which the membrane voltage of the postsynaptic neuron increases or an inhibitory response, in which the membrane voltage of the postsynaptic neuron decreases as a result of the spike.

The PyNN Neural Network Description Language

PyNN is a language for building neural network models. PyNN models can then be run on a number of simulators without modification (or with only minor modifications), including SpiNNaker. The basic steps of building a PyNN network are as follows:

1. Setup the simulator
2. Create the neural *populations*
3. Create the *projections* between the populations
4. Setup data recording
5. Run the simulation
6. Retrieve and process the recorded data

An example of this is as follows:

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
        {'spike_times': [[0]]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1), target="excitatory")
pop_1.record()
pop_1.record_v()
p.run(10)

import pylab
time = [i[1] for i in v if i[0] == 0]
membrane_voltage = [i[2] for i in v if i[0] == 0]
pylab.plot(time, membrane_voltage)
pylab.xlabel("Time (ms)")
pylab.ylabel("Membrane Voltage")
pylab.axis([0, 10, -75, -45])
pylab.show()

spike_time = [i[1] for i in spikes]
spike_id = [i[0] for i in spikes]
pylab.plot(spike_time, spike_id, ".")
pylab.xlabel("Time (ms)")
pylab.ylabel("Neuron ID")
pylab.axis([0, 10, -1, 1])
pylab.show()
```

This example runs using a 1.0ms timestep. It creates a single input source (A *SpikeSourceArray*) sending a single spike at time 0, connected to a single neuron (with model *IF_curr_exp*). The connection is weighted, so that a spike in the presynaptic neuron sends a current of 5 nanoamps (nA) to the excitatory synapse of the postsynaptic neuron, with a delay of 1 millisecond. The spikes and the membrane voltage are recorded, and the simulation is then run for 10 milliseconds. Graphs are then created of the membrane voltage and the spikes produced.

PyNN provides a number of standard neuron models. One of the most basic of these is known as the *Leaky Integrate and Fire* (LIF) model, and this is used above (*IF_curr_exp*). This models the neuron as a resistor and capacitor in parallel; as charge is received, this builds up in the capacitor, but then leaks out through the resistor. In addition, a *threshold* voltage is defined; if the voltage reaches this value, a spike is produced. For a time after this, known as the *refractory period*, the neuron is not allowed to spike again.

Once this period has passed, the neuron resumes operation as before. Additionally, the synapses are modelled using an exponential decay of the received current input (5 nA in the above example); the weight of the current is added over a number of timesteps, with the current decaying exponentially between each. A longer decay rate will result in more charge being added overall per spike that crosses the synapse.

In the above example, the default parameters of the *IF_curr_exp* are used. These are:

```
'cm': 1.0,         # The capacitance of the LIF neuron in nano-Farads
'tau_m': 20.0,     # The time-constant of the RC circuit, in milliseconds
'tau_refrac': 2.0, # The refractory period, in milliseconds
'v_reset': -70.0,  # The voltage to set the neuron at immediately after a spike
'v_rest': -65.0,   # The ambient rest voltage of the neuron
'v_thresh': -50.0, # The threshold voltage at which the neuron will spike
'tau_syn_E': 5.0,  # The excitatory input current decay time-constant
'tau_syn_I': 5.0,  # The inhibitory input current decay time-constant
'i_offset': 0.0,   # A base input current to add each timestep
```

PyNN supports both current-based models and conductance-based models. In conductance models, the input is measured in microSiemens, and the effect on the membrane voltage also varies with the current value of the membrane voltage; the higher the membrane voltage, the more input is required to cause a spike. This is modelled as the *reversal potential* of the synapse; when the membrane potential equals the reversal potential, no current will flow across the synapse. A conductance-based version of the LIF model is provided, which, in addition to the above parameters, also supports the following:

```
'e_rev_E': 0.,    # The reversal potential of the exponential synapse
'e_rev_I': -80.0  # The reversal potential of the inhibitory synapse
```

The initial value of the state variables of the neural model can also be set (such as the membrane voltage). This is done via the *initialize* function of the population, which takes the name of the state variable as a string (e.g. "v" for the membrane voltage), and the value to be assigned e.g. to set the voltage to -65.0mV:

```
pop.initialize("v", -65.0)
```

In PyNN, the neurons are declared in terms of a *population* of a number of neurons with similar properties. The *projection* between populations therefore has a *connector*, which describes the connectivity between the individual neurons in the populations. Some common connectors include:

- OneToOneConnector - each presynaptic neuron connects to one postsynaptic neuron (there should be the same number of neurons in each population) with weight *weights* and delay *delays*.
- AllToAllConnector - all presynaptic neurons connect to all postsynaptic neurons with weight *weights* and delay *delays*.
- FixedProbabilityConnector - each presynaptic neuron connects to each postsynaptic neuron with a given fixed probability *p_connect*, with weight *weights* and delay *delays*.
- FromListConnector - the exact connectivity is described by *conn_list*, which is a list of (*pre_synaptic_neuron_id, post_synaptic_neuron_id, weight, delay*)

Commonly, random weights and/or delays are used. To specify this, the value of the *weights* or *delays* of the connector are set to a *RandomDistribution* (note that the FromListConnector requires the specification of explicit weights and delays, and so does not support this; instead the *next()* method of the random distribution can be called to give random values for this connector). This supports several parameters via the *parameters* argument, depending on the value of the *distribution* argument which identifies the distribution type. The supported distributions include a 'uniform' distribution, with parameters of [minimum value, maximum value]; and a 'normal' distribution with parameters of [mean, standard deviation]. A *boundary* can also be specified as [*minimum, maximum*] to constrain the values generated (where an unbounded end can make use of -numpy.inf or numpy.inf); this is often useful for keeping the delays within range allowed by the simulator. The *RandomDistribution* can also be used when specifying neural parameters, or when initialising state variables.

In addition to neuron models, the PyNN language also supports some utility models, which can be used to simulate inputs into the network with defined characteristics. These include:

- SpikeSourceArray - this sends spikes at predetermined intervals defined by *spike_times*. In general, PyNN forces each of the neurons in the population to spike at the same time, and so *spike_times* is an array of times, but sPyNNaker also allows *spike_times* to be an array of arrays, each defining the the times at which each neuron should spike e.g. *spike_times*=[[0], [1]] means that the first neuron will spike at 0ms and the second at 1ms.
- SpikeSourcePoisson - this sends spikes at random times with a mean rate of *rate* spikes per second, starting at time *start* (0.0ms by default) for a duration of *duration* milliseconds (the whole simulation by default).

Using PyNN with SpiNNaker

In addition to the above steps, sPyNNaker requires the additional step of configuration via the .spynnaker.cfg file to indicate which physical SpiNNaker machine is to be used. This file is located in your home directory, and the following properties must be configured:

```
[Machine]
machineName     = None
version         = None
```

The *machineName* refers to the host or IP address of your SpiNNaker board. For a 4-chip board that you have directly connected to your machine, this is *usually* (but not always) set to *192.168.240.253*, and the *version* is set to *3*, indicating a “SpiNN-3” board (often written on the board itself). Most 48-chip boards are given the IP address of *192.168.240.1* with a *version* of *5*.

The range of delays allowed when using sPyNNaker depends upon the timestep of the simulation. The range is 1 to 144 timesteps, so at 1ms timesteps, the range is 1.0ms to 144.0ms, and at 0.1ms, the range is 0.1ms to 14.4ms.

The default number of neurons that can be simulated on each core is 256; larger populations are split up into 256-neuron chunks automatically by the software. Note though that the cores are also used for other things, such as input sources, and delay extensions (which are used when any delay is more than 16 timesteps), reducing the number of cores available for neurons.

Task 1.1: A simple neural network [Easy]

This task will create a very simple network from scratch, using some of the basic features of PyNN and SpiNNaker.

Write a network with a 1.0ms timestep, consisting of two input source neurons connected to two current-based LIF neurons with default parameters, on a one-to-one basis, with a weight of 5.0 nA and a delay of 2ms. Have the first input neuron spike at time 0.0ms and the second spike at time 1.0ms. Run the simulation for 10 milliseconds. Record and plot the spikes received against time.

Task 1.2: Changing parameters [Easy]

This task will look at the parameters of the neurons and how changing the parameters will result in different network behaviour.

Using your previous script, set tau_syn_E to 1.0 in the IF_curr_exp neurons. Record the membrane voltage in addition to the spikes. Print the membrane voltage out after the simulation (you can plot it if you prefer, but you should note that the array returned from get_v() contains a list of [neuron_id, time, voltage] and so you will need to separate out the voltages of the individual neurons).

1. Did any of the neurons spike?
2. What was the peak membrane voltage of any of the neurons, compared to the default threshold voltage of -50mV?

Try increasing the weight of the connection and see what effect this has on the spikes and membrane voltage.

Task 2.1: Synfire Chain [Moderate]

This task will create a network known as a Synfire chain, where a neuron or set of neurons spike and cause activity in an ongoing chain of neurons or populations, which then repeats.

1. Setup the simulation to use 1ms timesteps.
2. Create an input population of 1 source spiking at 0.0ms.
3. Create a synfire population with 100 neurons.
4. With a FromListConnector, connect the input population to the first neuron of the synfire population, with a weight of 5nA and a delay of 1ms.
5. Using another FromListConnector, connect each neuron in the synfire population to the next neuron, with a weight of 5nA and a delay of 5ms.
6. Connect the last neuron in the synfire population to the first.
7. Record the spikes produced from the synfire populations.
8. Run the simulation for 2 seconds, and then retrieve and plot the spikes from the synfire population.

Task 2.2: Random Values [Easy]

Update the network above so that the delays in the connection between the synfire population and itself are generated from a uniform random distribution with values between 1.0 and 15.0. Update the run time to be 5 seconds.

Task 3.1: Balanced Random Cortex-like Network [Hard]

This task will create a network that this similar to part of the Cortex in the brain. This will take some input from outside of the network, representing other surrounding neurons in the form of poisson spike sources. These will then feed into an excitatory and an inhibitory network set up in a balanced random network. This will use distributions of weights and delays as would occur in the brain.

1. Set up the simulation to use 0.1ms timesteps.
2. Choose the number of neurons to be simulated in the network.
3. Create an excitatory population with 80% of the neurons and an inhibitory population with 20% of the neurons.
4. Create excitatory poisson stimulation population with 80% of the neurons and an inhibitory poisson stimulation population with 20% of the neurons, both with a rate of 1000Hz.
5. Create a one-to-one excitatory connection from the excitatory poisson stimulation population to the excitatory population with a weight of 0.1nA and a delay of 1.0ms.
6. Create a similar excitatory connection from the inhibitory poisson stimulation population to the inhibitory population.
7. Create an excitatory connection from the excitatory population to the inhibitory population with a fixed probability of connection of 0.1, and using a normal distribution of weights with a mean of 0.1 and standard deviation of 0.1 (remember to add a boundary to make the weights positive) and a normal distribution of delays with a mean of 1.5 and standard deviation of 0.75 (remember to add a boundary to keep the delays within the allowed range on SpiNNaker).
8. Create a similar connection between the excitatory population and itself.

9. Create an inhibitory connection from the inhibitory population to the excitatory population with a fixed probability of connection of 0.1, and using a normal distribution of weights with a mean of -0.4 and standard deviation of 0.1 (remember to add a boundary to make the weights negative) and a normal distribution of delays with a mean of 0.75 and standard deviation of 0.375 (remember to add a boundary to keep the delays within the allowed range on SpiNNaker).
10. Create a similar connection between the inhibitory population and itself.
11. Initialize the membrane voltages of the excitatory and inhibitory populations to a uniform random number between -65.0 and -55.0.
12. Record the spikes from the excitatory population.
13. Run the simulation for 1 or more seconds.
14. Retrieve and plot the spikes.

The graph should show what is known as Asynchronous Irregular spiking activity - this means that the neurons in the population don't spike very often and when they do, it is not at the same time as other neurons in the population.

Task 3.2: Network Behavior [Moderate]

Note in the above network that the weight of the inputs is the same as the mean weight of the excitatory connections (0.1nA) and that the mean weight of the inhibitory connections is 4 times this value (-0.4nA). Try setting the excitatory connection mean weight and input weights to 0.11nA and the inhibitory mean weight to -0.44nA, and see how this affects the behavior. What other behavior can you get out of the network by adjusting the weights?

5th SpiNNaker Workshop

Day 2

September 8th 2015

	Session	Owner
09:00	Synaptic plasticity using PyNN	Sergio
10:00	Lab time (with coffee at 10:30)	
12:00	Lunch	
13:00	Simple data I/O and visualisation	Alan
14:00	Lab time (with coffee at 15:00)	
17:00	Close	

Manchester, UK

Simple Data Input Output and Visualisation on Spinnaker Lab Manual

1. Introduction

This manual will introduce you to the basics of live retrieval and injection of data (in the form of spikes) for PyNN scripts that are running on SpiNNaker neuromorphic hardware.

2. Installation

The PyNN 0.7 toolchain for SpiNNaker (sPyNNaker 2015.005), can be installed by following the instructions available from here:
http://spinnakermanchester.github.io/2015.005.Arbitrary/PyNNOnSpinnakerInstall.html

Matplotlib is marked as optional, but you will also need to install this dependency to complete some of the forthcoming exercises.

The sPyNNakerExternalDevicesPlugin 2015.009 can be installed by following the instructions available from here:
http://spinnakermanchester.github.io/2015.005.Arbitrary/PyNNOnSpinnakerInstall.html

The Visualiser 2015.002 module can be downloaded from here:
https://github.com/SpiNNakerManchester/Visualiser
and can be compiled by running the command make -f Makefile.<os>

To compile the Visualiser and the c code that can support the live injection and retrieval functionality, you will need Gcc and SQL and OpenGL libraries for c. The instructions to download and install these can be found here:
(link to be filled in)

To refer on how to configure your sPyNNaker installation to use your SpiNNaker machine, please refer to “Using PyNN with SpiNNaker” section of the “Running PyNN Simulations on SpiNNaker” lab manual.

3. PyNN Support
This section discusses the standard support from |PyNN related to spike injection and retrieval.

3.1 Output

The standard support for data output for a platform such as SpiNNaker, through the PyNN language, is to use the methods **record()**, **record_v()**, for declaring the need to record, and **get_Spikes()**, **get_v()**, for retrieval of the specific data.

The issue with the **get** functions are that they are called after **run()** completes, and therefore are no longer live. In the 2015.005 implementation of sPyNNaker, all of the data declared to be recorded via **record()**, **record_v()**, is stored on the SDRAM of the chips that the corresponding populations were placed on. This means that there is a finite amount of recorded data that can be stored before recordings fail.

By writing the data to SDRAM, the data is stored locally and therefore is guaranteed to be read at some point in the future. This memory requirement for recording is considered during the partitioning process, but if the memory that your recording requires, is more than the machines total available space left after other essential memory requirements, then your model will not be able to be ran at all.

Its worth noting that future releases of the sPyNNaker back end should be able to remove this constraint, but it currently is not supported.

3.2 Input

The standard support for data input for a platform such as SpiNNaker, through the PyNN language, is to use the neural models **SpikeSourceArray** and **SpikeSourcePoisson**. The issue with both of these models is that they are either random rate based (the spikeSourcePoisson) or are a playback mode (spikeSourceArray).

In sPyNNaker 2015.005, the playback mode of the SpikeSourceArray does not have the same memory constraint as the record functionality, but by removing the SDRAM limit, it is no longer able to be recorded via the **record()** functionality. Future releases of the sPyNNaker back end should be able to remove this constraint, but it currently is not supported.

4. External Device Plugin Support

As stated previously, the issue with this is that PyNN 0.7 expects its **run()** method to block for the entire time of the run, and therefore it is impossible to set up a real time extraction or retrieval of data via this FrontEnd (sPyNNaker), and has no current support for live retrieval or live injection.

Its worth noting that future releases of PyNN (0.9) may use the MUSIC interface to support live injection and retrieval of data. But the current software version of sPyNNaker (2015.005) only supports PyNN 0.7 and therefore there is no built in support.

To compensate for this, the sPyNNakerExternalDevicesPlugin module was created that contains support for live injection and retrieval of data from a running PyNN 0.7 simulation during the blocking of **run()**.

4.1 Live Output

To activate live retrieval from a given population, the command **activate_live_output_for(**<Population_object>**)** is used. This informs the sPyNNaker backend to add the supporting utility model (Live packet gatherer) into the graph object (which sPyNNaker uses to represent your PyNN neural models) and an edge between your population and the associate LPG for your ports.

Other parameters for the **activate_live_output_for()** function are defined below:

Parameter	**Description**
port	The port number to receive packets from the SpiNNaker machine.
database_notify_host	The hostname for the database notification protocol (described below)
database_notify_port_num	The port number for the database notification protocol (described below)
database_ack_port_num	The port number that the database notification protocol will listen to, to receive the ack packet.

4.2 Live Injection

To activate the live injection functionality, you need to instantiate a new neural model (called a SpikeInjector) which is located in **spynnaker_external_devices_plugin.pyNN.SpikeInjector**

The **SpikeInjector** is considered as any other neural model in PyNN, so you can build a population with a number of neurons etc in the normal way, as shown below:

```
injector_forward = Frontend.Population(
      5, ExternalDevices.SpikeInjector, [‘port’: 12367],
label='spike_injector_forward')
```

The key parameters of the **SpikeInjector** are as follows:

Parameter	**Description**
port	The port that packets are going to be injected in from.
virtual_key	The base routing key that the spike injector is going to use for routing. **This parameter is optional.**

4.3 Python Live reciever

The following block of code creates a live packet receiver:

```
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
    for neuron_id in neuron_ids:
           print “Received spike at time {} from {}-{}”.format(
            time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python live spike connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
    receive_labels=[“receiver”], local_port=19995, send_labels=None)
# register python receiver with live spike connection
```

```
13 live_spikes_connection.add_receive_callback(“receiver”, receive_spikes)
14 p.run(5000)
```

1. Lines 1 to 5 creates a function that takes as its input all the neuron ids that fired at a specific time, from the population with the given label. From here, it generates a print message for each neuron.
2. Lines 6 to 8 imports the python support for live injection/live retrieval. The SpynnakerLiveSpikesConnection handles both live retrieval and live injection.
3. Lines 9 to 11 instantiates the SpynnakerLiveSpikesConnection, and informs the connection that it will receive data under the label “receiver” on port 19996, but will not be injecting data at all.
4. Lines 12 to 13 informs the connection that for any packets being received, needs to be forwarded to the function receive_spikes defined on lines 1 to 5.
5. line 14 executes the model on SpiNNaker.

4.4 Python Live injector

The following block of code creates a live packet injector:

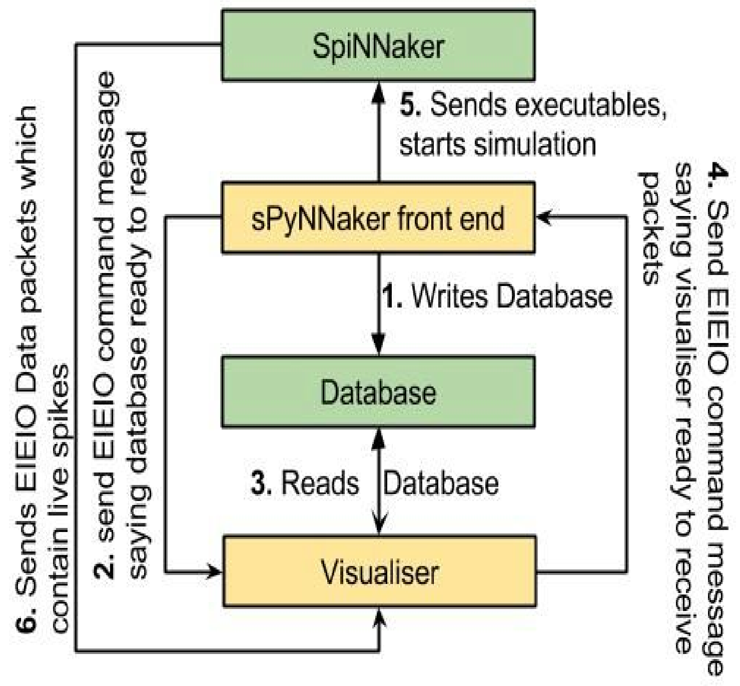
```
1  # create python injector
2  def send_spike(label, sender):
3      sender.send_spike(label, 0, send_full_keys=True)
4  # import python injector connection
5  from spynnaker_external_devices_plugin.pyNN.connections.\
6  spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
7  # set up python injector connection
8  live_spikes_connection = SpynnakerLiveSpikesConnection(
9      receive_labels=None, local_port=19996, send_labels=[“spike_sender”])
10 # register python injector with injector connection
11 live_spikes_connection.add_start_callback(“spike_sender”,  send_spike)
12 p.run(5000)
```

1. Lines 1 to 3 creates a little function that will inject a spike from neuron 0 from the spike injector.
2. Lines 4 to 6 imports the python support for live injection/live retrieval. The SpynnakerLiveSpikesConnection handles both live retrieval and live injection.
3. Lines 7 to 9 instantiates the SpynnakerLiveSpikesConnection, and informs the connection that it will not receive any data, but will inject data via the label spike_sender on port 19996.
4. Lines 10 to 11 informs the connection that when the simulation starts, to call the send_spike function defined on lines 1 to 3.

5. Line 12 executes the model on SpiNNaker.

5. Database Notification protocol

The support built behind all this software is a simple notification protocol on a database that's written during compilation time. The notification protocol is illustrated below:


SpiNNaker
5. Sends executables, starts simulation
sPyNNaker front end
1. Writes Database
Database
3. Reads Database
Visualiser
6. Sends EIEIO Data packets which contain live spikes
2. send EIEIO command message saying database ready to read
4. Send EIEIO command message saying visualiser ready to receive packets

The steps within the notification protocol are defined below:

1. The sPyNNaker front end writes a database that contains all the data objects generated from sPyNNaker during the compilation process.
2. The notification protocol sends a EIEIO command message to all the devices which are listening to hear that the database has been written. This functionality is hidden behind the Visualiser and SpynnakerLiveSpikesConnection software interfaces.
3. These devices then read the database for whatever data they require. These are often to deduce a mapping between received routing keys and neuron ids for the transmitted populations.
4. Once these devices have read the database, they notify the sPyNNaker front end that they are ready for the simulation to start.
5. Once all devices have notified the sPyNNaker front end, the simulation begins.
6. The sPyNNaker front end also notifies the devices when the simulation has begun.

7. The SpiNNaker machine transmits/ receives packets to/from external injections and retrieval devices.

6. Caveats

To use the live injection and retrieval functionality does not come for free. The functionality only supports the use of the ethernet connection, which means that there is a limited bandwidth of a maximum of approx 30 MB/s. This bandwidth is shared between both types of functionality, as well as system support for certain types of neural models, such as the SpikeSourceArray.

Furthermore, this functionality depends upon the lossy communication fabric of the SpiNNaker machine. This means that even though a neuron fires a spike you may not see it via the live retrieval functionality. If you need to ensure you receive every packet that has been transmitted, we recommend using the standard PyNN functionality.

By using this functionality, you are making your script non portable between different simulators. The activate_live_output_for(<pop_object>) and SpikeInjector models are not supported by other PyNN backends (such as Nest, Brian etc).

Finally, by using this functionality, you lose a number of SpiNNaker cores for this functionality. Therefore a model which would just fit onto your SpiNNaker machine before would likely fail to fit on the machine when these functionalities are added in.

7. Visualiser

The visualiser module contains a c based raster plot which is designed to integrate with the notification protocol described previously.

To compile the visualiser code, go into the visualiser module. Then execute:

```
make -f Makefile.<os>
```

The visualiser c code contains 4 different parameters. These are defined below:

Parameter	Description
-colour_map	Path to a file containing the population labels to receive, and their associated colours
-hand_shake_port	optional port which the visualiser will listen to for database hand shaking
-database	optional file path to where the database is located, if needed for manual configuration
-remote_host	optional remote host, which will allow port triggering

7.1 colour_map file format

The colour_map file consists of a collection of lines, where each line contains 4 values separated by tabs. These values, in order are:

1. The population label.
2. The red colour value.
3. The green colour value.
4. The blue colour value.

An example file is shown below:

```
“spike_forward      0      0      255
spike_backwards     0      255    0”
```

8. Tasks

The following tasks when completed will have hopefully taught you how to use the live injection and live retrieval functionality supported by the sPyNNakerExternalDevicesPlugin 2015.009 module.

We have assumed here that you were able to complete the **Running PyNN Simulations on SpiNNaker** tasks. We will be building upon the result from **task 2.1 Synfire Chain [Moderate]** located here:

[insert link here]

Please go back to this lab manual and complete this task before attempting these if you havent already.

Task 1.1: A simple synfire chain with a injected spike via python injector [EASY]

This task will create a synfire chain which is stimulated from a injector spike generated on host and then injected into the simulation.

1. Remove the spike source array population.
2. Replace it with the SpikeInjector population.
3. Build a python injector function.
4. Import and instantiate an SpynnakerLiveSpikesConnection connection.
5. link a start callback to the python injector function.

Task 1.2: A simple synfire chain with live streaming via the python receiver [EASY]

Go back to the original code produced by **task 2.1 Synfire Chain [Moderate]** from **Running PyNN Simulations on SpiNNaker**, and update it to stream the spikes from the If_cur_exp population.

1. remember to call the activate_live_output_for(<pop_object>)
2. Build a python receiver function that prints out the neuron ids for the population.
3. Import and instantiate an SpynnakerLiveSpikesConnection connection.
4. link a receive callback to the python receiver function.

Task 1.3: A simple synfire chain with live injection and live streaming via the python support [Easy]

Take the code from the previous 2 tasks and integrate them together to produce one that injects and streams the packets back to the terminal.

1. Remember that you can use both the recieve_labels and send_labels of the same SpynnakerLiveSpikesConnection.

Task 1.4: A simple synfire chain with live injection via python and live streaming via the c visualiser [Medium]

Take the code from the previous task and remove the python receiver code (or don't if you feel confident) and activate the visualizer to take the packets the original python receiver code processed.

1. Remember to compile the visualiser
2. Remember to generate the correct colour_map
3. Remember to remove the python receiver code (or don't if you're feeling confident).

Task 1.5: 2 Synfire chains which set each other off using python injectors whilst still using the c visualiser [Very Hard]

Take the code from the previous task and modify it so that there are two synfire populations which are tied to one injector population. Modify the receive function so that it contains some logic that fires the second neuron when the last neuron in the first synfire fires, and does the same when the last neuron for the second synfire sets off some neuron id of the first synfire chain.

1. you will need to change the number of neurons the spike injector contains.
2. You will need to change the connector from the spike injector and each synfire population.
3. You will need to modify the receive function, and add a global variable for the SpynnakerLiveSpikesConnection.
4. You'll need at least 2 SpynnakerLiveSpikesConnection and multiple activate_live_output_for(<pop_objevt>) for each population.
5. Remember that each population can only be tied to one LivePacketGatherer, so to visualise and do closed loop systems require more populations.
6. You will need to modify the c visualiser colour_map to take into account the new synfire population.

Task 1.6: 2 Synfire chains which set each other off using python injectors and live retrieval with 2 visualiser instances [Very Hard/Easy]

This task takes everything you've learnt so far and raises the level. Using the code from the previous task. Create two visualiser instances, each of which only processes one synfire population.

1. Remember all the lessons from the previous tasks.
2. Remember to change the ports on the activate_live_output_for(<pop_object>) accordingly.
3. You will need to create at least 2 SpynnakerLiveSpikesConnection's. But it might be worth starting with 3 and reducing it to two once you've got it working.
4. Remember the different colour_maps

Congratulations. Assuming you've successfully completed all the tasks, you should now know how to inject and retrieve packets from a running PyNN script using functionalities supported by the sPyNNakerExternalDevicesPlugin module.

Now if you want to learn how things work under the hood of the SpynnakerLiveSpikesConnection and the activate_live_output_for(<pop_object>), please feel free to try the advanced IO lab manual.

5th SpiNNaker Workshop

Day 3

September 9th 2015

Time	Session	Owner
09:00	Free lab time (with coffee at 10:30)	
12:00	Lunch and close	

Time	Session	Owner
09:00	Registration and coffee	
09:15	SpiNNaker architecture, chip resources and limitations	Steve
10:00	Adding new neuron models	Andrew/Michael
10:30	Coffee	
11:00	Lab time	
12:00	Lunch	
13:00	SpiNNaker system software (SARK)	Steve
13:45	SpiNNaker API	Luis
14:30	Lab time (with coffee at 15:00)	
16:00	Event-driven neural simulation	Alex
16:30	Lab time	
17:00	Close	

Manchester, UK

Creating New Neuron Models for SpiNNaker

Introduction

This manual will guide you in the creation of new neuron models to be run on SpiNNaker. This includes the C code that will be compiled to run on the SpiNNaker hardware, as well as the Python code which interacts with the PyNN script to configure the model.

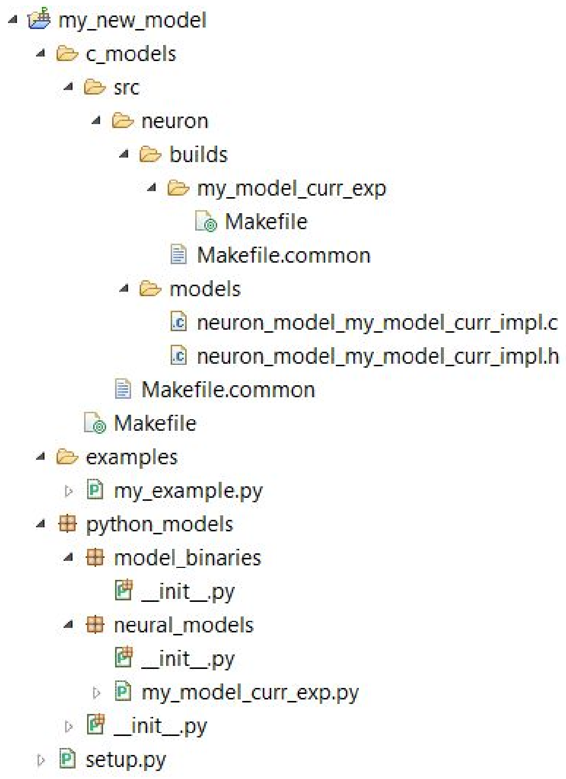
Installation

In order to create new models, you will need to ensure that you have set up a development environment suitable for compiling C code for SpiNNaker. This can be done by following the instructions here:

http://spinnakermanchester.github.io/2015.005.Arbitrary/PyNNOnSpiNNakerExtensions.html

Project Layout

The recommended layout for a new model project is shown below; this example shows a model called “my_model”, with current-based exponential synapses. It is recommended that the C and Python code is kept in the same project to help keep them synchronized.



```
my_new_model
    c_models
        src
            neuron
                builds
                    my_model_curr_exp
                        Makefile
                    Makefile.common
                models
                    neuron_model_my_model_curr_impl.c
                    neuron_model_my_model_curr_impl.h
                Makefile.common
            Makefile
    examples
        my_example.py
    python_models
        model_binaries
            __init__.py
        neural_models
            __init__.py
            my_model_curr_exp.py
        __init__.py
    setup.py
```

This template structure can be downloaded from:
http://spinnakermanchester.github.io/2015.005.Arbitrary/template_new_model.zip

C code header file

The C header file defines:

- The neuron data structure `neuron_t`. This includes the parameters and state for each neuron to be executed on a core. This commonly includes the membrane voltage of the neuron, as well as an offset input current.

- The global parameters data structure `global_neuron_params_t`. This includes parameters that are shared across all neurons within a population. This might include such things as the time step of the simulation.

- A definition of a function `input_t neuron_model_convert_input(input_t input)`. This can be used to perform any scaling of input between the input buffers and the neuron model. This allows the values in the input buffers to maintain a higher level of precision for computation, but then returns the value to the expected scale for the input of the neuron. This is currently done for conductance-based models, where input conductances are usually in fractions of micro-siemens; in standard s16.15 fixed-point format, these fractional values waste a lot of the precision, since the top 16-bits will not be used. Multiplying every value by 1024 increases the resolution of these values, but the neuron model expects the values to be in microsiemens; this function divides the input values by 1024 when conductances are used (done using a right-shift by 10 for efficiency).

See neuron_model_my_model_curr_exp.h in the template for an example of a header file. Comments show where the file should be updated to create your own model.

C code file

The C code file defines the functions that make up the interface of the neuron API. Note that pointer types are automatically created for the data structures defined in the header as follows:

```
             neuron_t * → neuron_pointer_t
global_neuron_params_t * → global_neuron_params_pointer_t
```

The neuron interface requires the following functions to be implemented:

-
  ```
  void neuron_model_set_global_neuron_params(
        global_neuron_params_pointer_t params)
  ```
  This function is used to set the global parameters after they have been read by the initialization function. This would often be used to store the parameters in a static variable for later use.

- `state_t neuron_model_get_membrane_voltage(neuron_pointer_t neuron)`
  This function should return the membrane voltage of the neuron from the given neuron structure. This may simply return the value of a variable in the structure, or it might perform a more complex calculation to obtain the membrane voltage. The value returned is used for the recording of the membrane voltage in the simulation.

- `void neuron_model_print(restrict neuron_pointer_t neuron)`
  This function is only used when the neuron model is compiled in “debug” mode (see later). It should use the “log_debug” function to print each of the state variables and parameters of the neuron that might be useful in debugging.

-
  ```
  bool neuron_model_state_update(input_t exc_input, input_t inh_input,
                                 input_t external_bias, neuron_pointer_t neuron)
  ```
  This function takes the excitatory and inhibitory input; any external bias input (used in some plasticity models); and a neuron data structure; and uses these to compute the new state of the given neuron at this timestep. This function is where any differential equation solving should be implemented. After the state update, the function should return whether the neuron is considered to have spiked as a boolean (true if the neuron has spiked, false otherwise). Note that the input does

not specify current or conductance; no conversion of the weights are done before this function is called, other than any scaling performed in `neuron_model_convert_input`.

See neuron_model_my_model_curr_exp.c in the template for an example of an implementation of the neuron interface.

A number of other modules are available for use for performing mathematical functions as part of the neuron state update. The `spinn_common` library provides a number of efficient fixed-point implementations of common functions. This includes `random.h`, which provides random number generation, `normal.h`, which provides normal distributions, `exp.h`, which provides an exp function and `log.h` which provides a log function.

Makefiles

There are a number of Makefiles and Makefile.common files in the template. These have mostly been configured for you. The Makefile.common files set up directory structures, and include the appropriate Makefiles from the SpyNNaker module. The only Makefile that should need to be edited for your model is the one in the `build` folder for your executable; in the template, this is:

```
c_models/src/neuron/builds/my_model_curr_exp/Makefile
```

The Makefile builds the application that will run on SpiNNaker. This is made up of the neuron model that you have created, as well as the rest of the common neuron implementation. This includes the synapse dynamics that are supported by the build, as well as the synapse and spike processing code. The example provided is for a build with support for static synapses only. The inclusion of plastic synapses is covered elsewhere.

The important parts of this Makefile that need to be updated are:

- `MODEL_OBJS` - This includes a list of object files to be produced; note that the path of the object file is the same as the path of the C file, although the actual build process will push this into a build folder. In addition to the object file of the C file described above, this also includes the synapse dynamics type of the build (static in the template).

- `NEURON_MODEL_H` - This points at the neuron model header file.

- `SYNAPSE_TYPE_H` - This provides the synapse type for the build. The synapse type controls the shaping of the synapses in response to the input weights. Within sPyNNaker, support so far includes exponential synapses (with one excitatory and one inhibitory synapse per neuron which decay exponentially with a configured time-constant) and dual-excitatory exponential synapses (with 2 separate excitatory synapses and one inhibitory synapse per neuron, decaying as per the previous type).

Note that the combination of the `NEURON_MODEL_H` and `SYNAPSE_TYPE_H` will determine the overall model - thus `my_model_curr_exp` specifies the inclusion of `neuron_model_my_model_curr.h` as the model and `synapse_type_exponential_impl.h` as the synapse type.

Once the Makefile has been created, you can build the binary by simply typing:

```
make
```

As the build relies on header files that are not explicitly specified in the Makefile, some of the changes that you make may require you to clean the build before building it, by running

```
make clean
```

Finally, you can also build the application in debug mode by typing:

```
make DEBUG=DEBUG
```

This will enable the log_debug statements in the code, which print out information to the iobuf buffers on the SpiNNaker machine; to read this there is a simple application in the example folder that can read iobuf buffers, which can be invoked as follows:

```
python iobuf.py <machine_name> <app_name>
```

where `<machine_name>` is the hostname or IP address of the SpiNNaker board, and `<app_name>` is the name of application (without the .aplx extension). This will retrieve the iobuf buffers for every instance of the application that is found to be running.

Python PyNN Model

Once the C code has been constructed, the PyNN model must be created in Python to translate the PyNN parameters into a form that the C code can understand. In PyNN, populations can be made up of an arbitrary number of neurons, however to maintain real-time operation the number of neurons that are simulated on each core must be limited. The PACMAN module is used by sPyNNaker to partition the populations into subpopulations, based on the specified maximum number of atoms per core of the model, as well as the resources required by the synaptic matrix. The DataSpecification module is then used to write the data for each subpopulation. This is then loaded on to the machine, along with binary executable, using SpiNNMan.

As with the C code, there are number of components that can be re-used, so that only properties relevant to the new model itself need to be defined. This is done by extending the classes that define these properties. For example, the exponential synapse type described previously can be included by extending:

```
spynnaker.pyNN.models.abstract_models.abstract_model_components\
    .abstract_exp_population_vertex.AbstractExponentialPopulationVertex
```

To make use of the common codebase for all populations, all models must extend:

```
spynnaker.pyNN.models.abstract_models.abstract_population_vertex\
  .AbstractPopulationVertex
```

Once the class has been defined, a number of other properties and functions need to be defined:

- `_model_based_max_atoms_per_core` - This keeps track of the maximum neurons per core for this model. This is a user-configurable parameter, but the initial value is defined here as an absolute maximum. The absolute maximum supported by the data structures elsewhere in the C code is 255, so if you are unsure, you can use this value. If your model is particularly complex, you should set this to a lower value, as more processing time will be required per neuron.

-
  ```
  __init__(self, n_neurons, machine_time_step, timescale_factor,
              spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
              ...)
  ```
  This is the initializer of the model class. The parameters of this initializer must include the following variables to match the sPyNNaker interface. These values will be passed down from sPyNNaker, and are mostly passed on to superclass initializers, to it is not critical that you understand what they all do here; however a short description is given below:
    - the number of neurons (`n_neurons`),
    - the machine timestep (`machine_time_step`) in microseconds,

    - the timescale factor (`timescale_factor`),
    - the maximum expected spikes per second (`spikes_per_second`),
    - the number of sigmas in the ring buffer distribution to allow for (`ring_buffer_sigma`),
    - any additional constraints on the vertex (`constraints`), and
    - The vertex label (`label`)

  In addition to these parameters, you should include any parameters required by the other classes that are extended. You can then also add your own parameters as required by your model. All the parameters must be given default values, including those required by the superclasses (not including those listed above, which are given values automatically); this allows the user to only specify those values that they want to change. The `machine_time_step` parameter can be used to convert any values that are in milliseconds to values that are in numbers of time steps (e.g. 10 milliseconds at a timestep of 100 microseconds is 10 / (100 / 1000.0) = 100 machine timesteps).

  Within the init method, the following must be done:

    - Call the initializer of `AbstractPopulationVertex`. This should pass on the parameters listed above, and should also pass the following parameters:
        - `n_params` - The total number of parameters and state variables defined in the `neuron_t` data structure in the C code. This may be more than the number of parameters passed to the initializer, as there may be variables that are not user-configurable.
        - `n_global_params` - The total number of parameters defined in the `neuron_global_params_t` data structure in the C code.
        - `binary` - This is the name of the executable that is produced when the Makefile runs. This is defined in the Makefile. This is the simple name of the file rather than the full path to the binary.
        - `weight_scale` - If any weight scaling is done in `neuron_model_convert_input`, the dividing factor must be specified here, so that the weights can be pre-multiplied by this amount. This value defaults to 1.0, so it doesn't need to be specified if it isn't used.
        - `max_atoms_per_core` - You can pass the `_model_based_max_atoms_per_core` variable defined above for this parameter. This will allow the user to configure this in a script if desired.
    - Call the initializers of other included components. This will pass on the parameters required by these components.
    - Store any of your own parameters. These are usually defined as:
      `self._<param> = utility_calls.convert_param_to_numpy(<param>, n_neurons)`
      The user can specify parameters using a single value; a list of values (with one per neuron); or a `RandomDistribution` object. The `convert_param_to_numpy` function normalises these into a numpy array.

- `initialize_<var>` - This is a set of functions, one for each of the state variables (i.e. those that change during the simulation). This allows the user to call the PyNN initialize() function. Note that the initial values are usually also available via the class constructor (usually postfixed with _init), but that PyNN doesn't usually expose state variables in this way. These functions should also make use of the `utility_calls.convert_param_to_numpy` function to normalise the parameter values.

- Property and setter for each parameter (i.e. those variables that don't change during the simulation). This allows the user to call the PyNN set() function for these variables. As with the above the parameter values should be normalised.
- `model_name(self)` - This property (use the `@property` decorator) simply returns the model name.
- `set_model_max_atoms_per_core(new_value)` - This static method (use the `@staticmethod` decorator) sets the value of `_model_based_max_atoms_per_core`, to allow the user to override the default (with a smaller value).
- `get_cpu_usage_for_atoms(self, vertex_slice, graph)` - This method returns an estimate of the number of cpu clock cycles required per timer tick to run the model for the number of neurons given by `vertex_slice.n_atoms`. This value is unlikely to be critical at this point, so using the function as it is specified in the template should work correctly.
- `get_parameters(self)` - This method returns an array of `NeuronParameter` instances to match the parameters and state variables defined in the `neuron_t` data structure in the C header file. The order of the array must match the order of the parameters and state variables as defined in the `neuron_t` data structure. Along with each parameter, the data type must also be given. This is done using the `DataType` enum; the most commonly used values are `DataType.S1615`, which corresponds to a C fixed-point `accum` data type (or `REAL` in most of our code); and `DataType.UINT32` and `DataType.INT32` which correspond to unsigned and signed 32-bit integers respectively (`uint32_t` and `int32_t`).
- `get_global_parameters(self)` - This method returns an array of `NeuronParameter` instances. This must return the values of the global parameters exactly as defined in the `global_neuron_params_t` data structure in the C header file, including the order of the parameters.
- `is_population_vertex(self)` - This method needs to be defined, but just needs to return `True`. This is used to keep track of the types of the vertices.
- Any additional abstract methods defined by the extended classes. For example, `AbstractExponentialPopulationVertex` requires the `is_exponential(self)` method to return `True`.

Python __init__.py files

Most of the __init__.py files in the template do not contain any code. The one within `python_models` is the exception; this file adds the `model_binaries` module to the executable paths, allowing sPyNNaker to search this folder for your compiled binary. You can also import your module here to make it easy to use in other scripts.

Python setup.py file

This file enables you to install the new module. This is set up to install all the modules in the template; if you add any modules, these also need to be added to this file (it is not recursive; each module has to be added separately). To add the module to your python environment in such a way that you can still edit it, you can run:

```
[sudo] python setup.py develop [--user]
```

You need to use `sudo` if you are installing centrally on Linux or Mac OS X; on windows you need to be in an Administrative console. Add `--user` instead if you want to install only for your username (you shouldn't mix these two options, or you will end up installing it only for the root user).

Using your module

In order to use the new module, you need to import your module in addition to PyNN e.g. for the template module, you can do the following:

```
import pyNN.spiNNaker as p
import python_model as new_models
pop = p.Population(1, new_models.MyModelCurrExp)
```

A more detailed example is shown in the template in `examples/my_example.py`.

Task 1: A Simple Neural Model [Easy]

This task will create a simple neural model using the template, and execute it on SpiNNaker.

Change the template by adding two parameters, one representing a decay (default value of 0.1) and one representing a rest voltage (default value of -65.0). The parameters should be `REAL` values (`DataType.S1615`). Change the model to subtract the difference between the current voltage and the rest voltage multiplied by the decay from the membrane voltage, before adding the total input i.e.

```
v_membrane = v_membrane - ((v_membrane - v_rest) * decay) + input
```

Run the example script and see what happens.

Task 2: A Spiking Neuron Model [Moderate]

This task will look at adding a threshold at which the neuron spikes.

Add further parameters to the model created previously for the threshold voltage of the neuron (`REAL`, default value -60.0), the reset voltage (`REAL`, default value -70.0) and another parameter which is the refractory period (`uint32_t`, default value 2.0), in milliseconds. You will also need a state variable to keep a refractory timer (`int32_t`). Change the model C code to spike (return `true`) when the neuron voltage is greater than or equal to the threshold voltage after the update. If the neuron spikes, the voltage should then be set to the reset voltage, and the refractory timer should be set to the refractory period. Add a condition so that the neuron membrane voltage is only updated while the refractory timer is less than or equal to 0. If it is greater than 0, the refractory timer should be reduced by one.

Update the python code to match the C code. Note that the python code will need to convert the refractory period in milliseconds to the number of machine time steps.

Update the example script to record and plot the spikes, and run it again.

Task 3: A Stochastic Threshold Model [Hard]

This task will look at more complex model using some of the provided functions in the spinn_common library. Note that the models are automatically compiled with this library, so no additions to the Makefile are necessary.

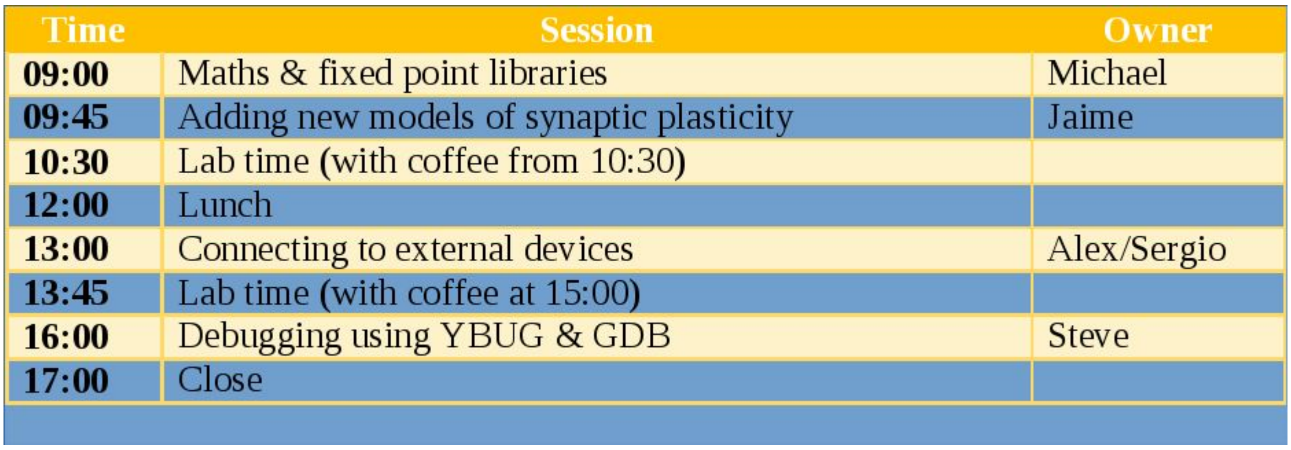
Take the neuron model created in the previous task, and add a parameter representing the probability of the neuron firing if it is over the threshold value. This parameter will be between 0 and 1 in Python (default of 0.5), but as the random number generator generates an integer value, this should be converted into a `uint32_t` value between 0 and 0x7FFFFFFF. Add a global parameter which is the seed of the random number generator. This is an array of 4 `uint32_t` values for the simplest random number generator in `normal.h`. Validate the Marsaglia KISS 64 RNG seed during initialisation of the global parameters (`validate_mars_kiss64_seed(mars_kiss64_seed_t seed)`). When the neural model is over the threshold voltage, call the RNG with the seed (`mars_kiss64_seed(mars_kiss64_seed_t seed)`). The neuron should only spike if the value is greater than than the probability.

Rerun the example script and see how the number of spikes differs for different settings of the spike probability.

5th SpiNNaker Workshop

Day 4

September 10th 2015



Time	Session	Owner
09:00	Maths & fixed point libraries	Michael
09:45	Adding new models of synaptic plasticity	Jaime
10:30	Lab time (with coffee from 10:30)	
12:00	Lunch	
13:00	Connecting to external devices	Alex/Sergio
13:45	Lab time (with coffee at 15:00)	
16:00	Debugging using YBUG & GDB	Steve
17:00	Close	

Manchester, UK

Adding new models of synaptic plasticity

August 28, 2015

Contents of package

examples/stdp_triplet.py PyNN script that reproduces experimental protocol developed by Sjöström et al. [2].

neural_modelling/src/neuron/Makefile Makefile which lists all the neuron models defined in this module.

neural_modelling/src/neuron/builds/Makefile.common Makefile which lists new STDP components defined by this module.

neural_modelling/src/neuron/builds/IF_curr_exp_stdp_mad_pair_additive/Makefile Makefile to build SpiNNaker executable with spike-pair STDP rule.

neural_modelling/src/neuron/builds/IF_curr_exp_stdp_mad_triplet_additive/Makefile Makefile to build SpiNNaker executable with Pfister and Gerstner [1] spike-triplet STDP rule.

neural_modelling/src/neuron/plasticity/stdp/timing_dependence/timing_pair_impl.c C source file containing setup code for spike-pair STDP timing dependence.

neural_modelling/src/neuron/plasticity/stdp/timing_dependence/timing_pair_impl.h C header file containing implementation of spike-pair STDP timing dependence discussed in presentation.

neural_modelling/src/neuron/plasticity/stdp/timing_dependence/timing_triplet_impl.c C source file containing setup code for spike-triplet STDP timing dependence.

neural_modelling/src/neuron/plasticity/stdp/timing_dependence/timing_triplet_impl.h C header file containing implementation of spike-triplet STDP rule discussed in presentation.

workshop_2015_adding_synaptic_plasticity/__init__.py Python module entry point containing code to hook module into sPyNNaker and import timing dependences sub-module.

workshop_2015_adding_synaptic_plasticity/spike_pair_time_dependency.py
Python class to instantiate and configure spike-pair timing dependence from PyNN.

workshop_2015_adding_synaptic_plasticity/spike_triplet_time_dependency.py
Python class to instantiate and configure spike-triplet timing dependence from PyNN.

Additional code changes

My presentation covered the code changes that are required to implement the behaviour spike-triplet rule on SpiNNaker. However there are some other, less interesting changes that are also required to build a functioning learning rule. Remaining changes to Python and C are discussed in comments at the following URL `http://tinyurl.com/ouk2gj2`.

Exercises

These are all more suggestions than anything else, I'd be interested to help with any triplet-rule based experimentation.

Exercise 1

As mentioned in the presentation, the SpiNNaker package already comes with an implementation of the full spike-triplet rule developed by Pfister and Gerstner [1]. This is more computationally expensive than the version developed in this workshop session, but the extra parameters may potentially allow it to better fit experimental data. Try switching the stdp_triplet.py example in the package to use this rule, configured with the parameters fitted by Pfister and Gerstner:

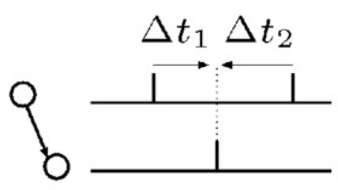
```
timing_dependence = sim.PfisterSpikeTripletRule(
          tau_plus=16.8, tau_minus=33.7,
          tau_x=101, tau_y=114)

weight_dependence = sim.AdditiveWeightDependence(
          w_min=0.0, w_max=max_weight,
          A_plus=5E-10 * start_w, A_minus=7E-3 * start_w,
          A3_plus=6.2e-3 * start_w, A3_minus=2.3E-4 * start_w)
```

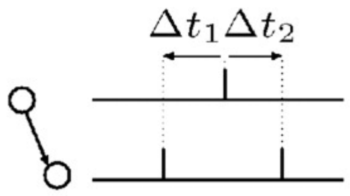
Does this actually reduce the error compared to the version developed in this workshop? Why might this be? The talk this morning on 'Maths & fixed point libraries' may give you some clues!

Exercise 2

Pfister and Gerstner [1] also fitted their model to some experimental data by Wang et al. [3]. These follow the spike-triplet protocol shown in figure 1 which



(a) Pre-post-pre triplet



(b) Post-pre-post triplet

Figure 1: Wang et al. [3] triplet protocol. Each experiment consists of 60 triplets of spikes, one second apart.

Δw	Δt_1	Δt_2
-0.01 ± 0.04	5	-5
0.03 ± 0.04	10	-10
0.01 ± 0.03	15	-5
0.24 ± 0.06	5	-15

(a) Pre-post-pre triplets

Δw	Δt_1	Δt_2
0.33 ± 0.04	-5	5
0.34 ± 0.04	-10	10
0.22 ± 0.08	-15	-5
0.29 ± 0.05	-5	15

(b) Post-pre-post triple

Table 1: Weight changes induced by Wang et al. [3] triplet protocol.

resulted in the weight changes shown in table 1. Can you make a version of stdp_triplet.py that reproduces this protocol?

References

[1] Jean-Pascal Pfister and Wulfram Gerstner. Triplets of spikes in a model of spike timing-dependent plasticity. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 26(38):9673–82, September 2006. ISSN 1529-2401. doi: 10.1523/JNEUROSCI.1425-06.2006. URL `http://www.ncbi.nlm.nih.gov/pubmed/16988038`.

[2] P J Sjöström, G G Turrigiano, and S B Nelson. Rate, timing, and cooperativity jointly determine cortical synaptic plasticity. *Neuron*, 32(6):1149–64, December 2001. ISSN 0896-6273. URL `http://www.ncbi.nlm.nih.gov/pubmed/11754844`.

[3] Huai-Xing Wang, Richard C Gerkin, David W Nauen, and Guo-Qiang Bi. Coactivation and timing-dependent integration of synaptic potentiation and depression. *Nature neuroscience*, 8(2):187–93, February 2005. ISSN 1097-6256. doi: 10.1038/nn1387. URL `http://www.ncbi.nlm.nih.gov/pubmed/15657596`.

Data Specification for AER over Ethernet "AEtheRnet"

Proposal version 0.3, 9 December 2014

Alex Rast, David Lester

1 Introduction

This specification addresses the data format for spikes to be sent over Ethernet between compatible AER-generating and receiving devices. It is expected that the interface can be bidirectional: a device can both issue and receive spikes. However, it is not *required* that a given device must be able to do both; a device could be either a blind issuer of spikes or a passive receiver.

2 Definitions

Throughout this document several terms will be used repeatedly as defined below.
AER: Address-event representation: an event coded as a binary number indicating the event source.
AEtheRnet: The protocol this document describes.
Key: A valid address in the AER address space.
Packet: The basic bundling unit of the AEtheRnet protocol, including any transport-related framing.
Block: An amount of data, bundled into an AEtheRnet packet, not including any headers, all in the same format as specified in the AEtheRnet header.
Device: A hardware device or software process, treated as a black box, that can send and/or receive AEtheRnet packets.
Stream: An internal channel on a device, responding to AER packets in some particular expected way, capable of receiving and/or sending AER packets in their native internal format. A device could in principle support several different streams with possibly different interfaces.

Machine word descriptions: The text indicates the internal format of machine words using a graphical format as below. The highest bit-position indicates the expected word size. Optional bits are assumed to be detected using a flag which if not set causes the bits to be ignored.

Reserved bits (leave as 0)	Optional bits	Field	B	Field
31 16	15 7	6 3	2	1 0

3 Supported Content Formats

The protocol supports 3 different types of content:
A. Packed AER addresses; these may be either 16- or 32-bit.
B. Packed AER addresses with payloads; either 16+16 or 32+32-bit.
C. Device-specific internal commands with internal format determined by the device.

4 General Comments on Data Handling

Transport: This specification does not specify the transport. In principle, any Ethernet-compatible transport could be used. Designers of interfaces should consider the facilities provided by the transport when developing AEtheRnet interfaces. Note that some transports will normally result in data words possibly misaligned according to machine word boundaries. The specification provides a mechanism to address this.

Packet Type Handling: It is not required that all devices be able to service all packet types. For example, a device that had only 16 bit internal registers would not be required to service 32-bit packets. However, the interface must be designed such that receipt of a packet of unsupported type not cause the device to go into an invalid state; i.e. if it receives an unsupported type it should discard the packet and continue to service supported packets. Implementers must specify clearly in interface documentation which packet types are supported.

Endianness: The assumption is that in normal use, packets will be transported over the wire in big-endian format, but represented on the device in little-endian format. All machine words are described as in the device representation, i.e. little-endian.

Errors: AEtheRnet does not attempt to detect or correct errors. If an error does occur, the behaviour is UNDEFINED. Designers of interfaces may use transport facilities (e.g. CRC) to recover from errors if desired, but this is not required. In particular there is no provision for what processing occurs if an invalid AEtheRnet header is received.

Acknowledgements: No devices acknowledge receipt of AEtheRnet packets; similar to hardware AER, the protocol is fire-and-forget. No command sent using a command packet may expect a reply packet of any type.

Command Processing: The interpretation of data in a command is entirely device-specific. AEtheRnet does not impose any restrictions on the format of data in this field, including endianness and error handling, so the comments related to errors and endianness above may not apply in the case of a command.

Latency: For real-time traffic, latency before a packet is sent will be a concern. A device which buffers AER spikes before forwarding them in either direction must ensure that no spike is forwarded later than the time when it should have been received by the receiving target. Therefore devices must send partially-filled AEtheRnet packets as soon as the earliest latency period of any of its members is exceeded. Once a packet is full (contains 256 items or is as large as the device's internal buffer size), it must be sent immediately. Communicating devices can use Command packets, with a suitable protocol, if necessary, to inform each other of their latency requirements and buffering limits.

Packet Ordering: Packets must be issued in sequential time order. If packet payloads contain timestamps then the value of the timestamp must be strictly increasing with packet sequence number. Receiving devices must discard any spikes whose timestamps are received out of strict sequential order.

5 Packet Format

AEtheRnet packets consist of a header and data, packed into possibly variable-length blocks of up to 2K bytes. Each packet contains a single data type (with the exception of command packets). There are 5 different possible headers. The most fundamental is the Basic data header, a 16-bit halfword followed directly by the data. The Prefixed data header consists of 2

16-bit halfwords, the first with the same format as the Basic header, the second being a prefix to OR with all AER keys in the block (position-dependent) to assemble the compete key, and the Command header is a 16-bit halfword followed immediately by device-specific command information. Data (as opposed to command) AEtheRnet packets may also have a fixed payload base, to be ORed with sent payloads. If the packet does not send payloads then the fixed payload base is understood to be a constant payload with each AER key. The underlying word size of the packet determines the size of this payload base. A complete AEtheRnet packet therefore can have one of the following 9 structures:

Header	Command Information
15 0	...

Header	Key	...	Key
15 0	31/15 0	...	31/15 0

Header	Key	Payload	...	Key	Payload
15 0	31/15 0	31/15 0	...	31/15 0	31/15 0

Header	Prefix	Key	...	Key
15 0	15 0	31/15 0	...	31/15 0

Header	Prefix	Key	Payload	...	Key	Payload
15 0	15 0	31/15 0	31/15 0	...	31/15 0	31/15 0

Header	Fixed Payload	Key	...	Key
15 0	31/15 0	31/15 0	...	31/15 0

Header	Payload Base	Key	Payload	...	Key	Payload
15 0	31/15 0	31/15 0	31/15 0	...	31/15 0	31/15 0

Header	Prefix	Fixed Payload	Key	...	Key
15 0	15 0	31/15 0	31/15 0	...	31/15 0

Header	Prefix	Payload Base	Key	Payload	...	Key	Payload
15 0	15 0	31/15 0	31/15 0	31/15 0	...	31/15 0	31/15 0

All headers have as their 2 MSBs a pair of bits "P" (for "Prefixed") and "F" (for "Format") that identify the header type. Prefixed headers have either 10 or 11 depending on whether the prefix is to be ORed with the lower or upper halfword of the key respectively. If the protocol is sending 16-bit keys then a 1 in the F bit for Prefixed headers will extend the internal format of the keys to 32-bit. Likewise if the protocol is sending 32-bit keys a 0 in the F bit for Prefixed headers will truncate the key to 16-bit. Command headers always have 10 for their MSBs and Basic headers have 00 in their MSBs.

The rest of the Command header is the device-specific command code. For Basic and Prefixed headers, a flag"D" (for "Data") allow a fixed pattern to be ORed with payloads. If the packet has no payload the fixed pattern is used as a fixed payload. The data type of the packet (16 or 32 bit) sets the size of the expected field, which should always be the last header item before the start of keys. A second flag, "T" allows payloads to be interpreted as timestamps. If this bit is set, the receiver should interpret each spike as being sent at the time indicated by its payload. (If D and T are both set with no payload in the packet type, the system can efficiently send a block of spikes at the same time). The next 2 bits indicate the datatype in the packet, followed by 2 Tag bits that identify the stream, and finally 8 Count bits that give the number of data items, where an item is either a key or a key/payload pair.

The decode of the datatype flags is as follows:

<u>P - Bit 15</u>
 0 = No key prefix
 1 = With key prefix
<u>F - Bit 14</u>
 0 = Basic packet; OR prefix with lower halfword
 1 = Command packet; OR prefix with upper halfword
<u>D - Bit 13</u>
 0 = No payload prefix
 1 = With payload prefix
<u>T - Bit 12</u>
 0 = Payloads are not timestamps
 1 = Payloads are timestamps
<u>Type - Bits 11:10</u>
 00 = 16-bit key
 01 = 16-bit keys and 16-bit payloads, alternating.
 10 = 32-bit key
 11 = 32-bit keys and 32-bit payloads, alternating.

Thus the header formats have the following structure:

<u>Formats of a data header:</u>

P	F	D	T	Type	Tag	Count	Key Prefix (if P) or Data (not P)
15	14	13	12	11 10	9 8	7 0	15 0

0	0	1	T	Type	Tag	Count	Payload Prefix
15	14	13	12	11 10	9 8	7 0	15 0

1	F	1	T	Type	Tag	Count	Key Prefix	Payload Prefix
15	14	13	12	11 10	9 8	7 0	15 0	31/15 0

<u>Format of a command header:</u>

0	1	Command	(Device-specific)
15	14	13 0	...