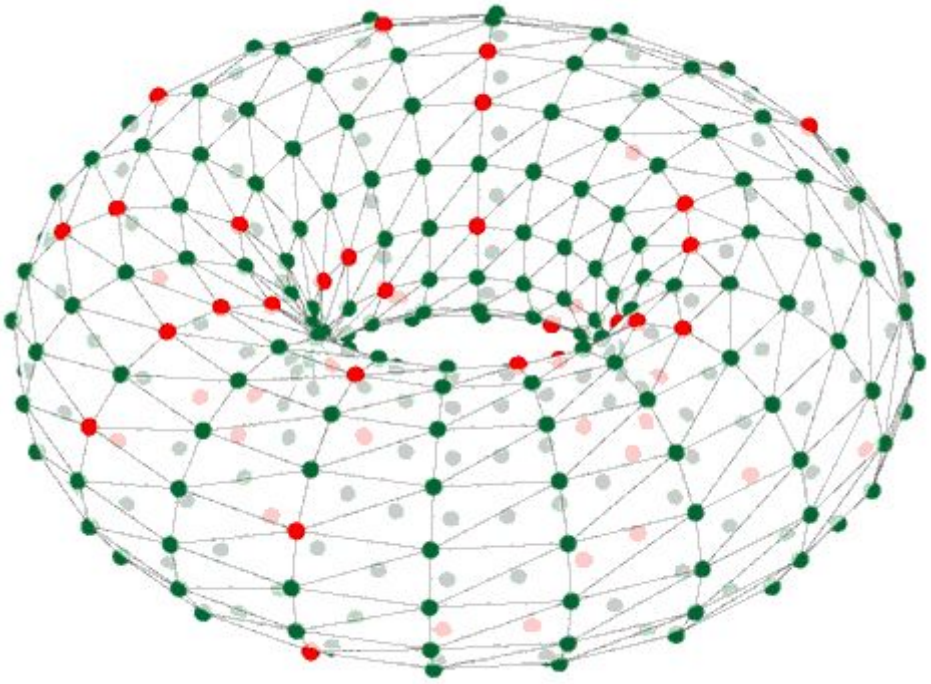
7th SpiNNaker Workshop

Lab Manuals

October
3rd - 6th 2017



Manchester, UK

SpiNNaker

Intro Lab

This lab is meant to expose workshop participants to examples of problems which can be applied to the SpiNNaker architecture.

Installation

Begin by installing the software which is used to control the SpiNNaker board and run Neural Networks on it. The software installation instructions can be found at the following link. Please install the PyNN 0.8 version of the software:

https://spinnakermanchester.github.io/latest/spynnaker_install.html

Please then also install the build tools which will be used later in the workshop. These include the C compiler, spinnaker_tools, spinn_common and the SpiNNFrontEndCommon and sPyNNaker source code:

https://spinnakermanchester.github.io/latest/spynnaker_extensions.html

In addition, the visualisers used in this lab require the following steps to be done:

1. Install the OpenGL libraries (depending on your platform):
    a. On Fedora Linux (replace dnf with yum on older platforms):
    ```
    sudo yum install freeglut3-devel
    ```
    b. On Ubuntu Linux:
    ```
    sudo apt-get install freeglut3-dev
    ```
    c. On Windows 64-bit, from an administrative console run:
    ```
    pip install
    https://github.com/SpiNNakerManchester/SpiNNakerManchest
    er.github.io/releases/download/v1.0-win64/PyOpenGL-3.1.1
    -cp27-cp27m-win_amd64.whl
    ```
    d. On Windows 32-bit from an administrative console run:
    ```
    pip install
    https://github.com/SpiNNakerManchester/SpiNNakerManchest
    er.github.io/releases/download/v1.0-win32/PyOpenGL-3.1.1
    -cp27-cp27m-win32.whl
    ```
    e. On Mac OS X no installation is required
2. Install the visualisers (using sudo if you have done a central installation, or --user if you have done a user-only installation):
    ```
    [sudo] pip install SpyNNakerVisualsers [--user]
    ```

File download

All of these examples can be found here:
https://spinnakermanchester.github.io/latest/intro_lab.html

Please download and open a terminal at the top level of the folder.

Run Applications

Below is a list of applications with the corresponding folders and execution commands, please run each script as it currently stands, and attempt to understand what the application is doing.

1. Neural Network Synfire Chain
2. Sudoku Game Through Neural Network
3. Balanced Random Network
4. Simple Learning Network

Neural Network Synfire Chain

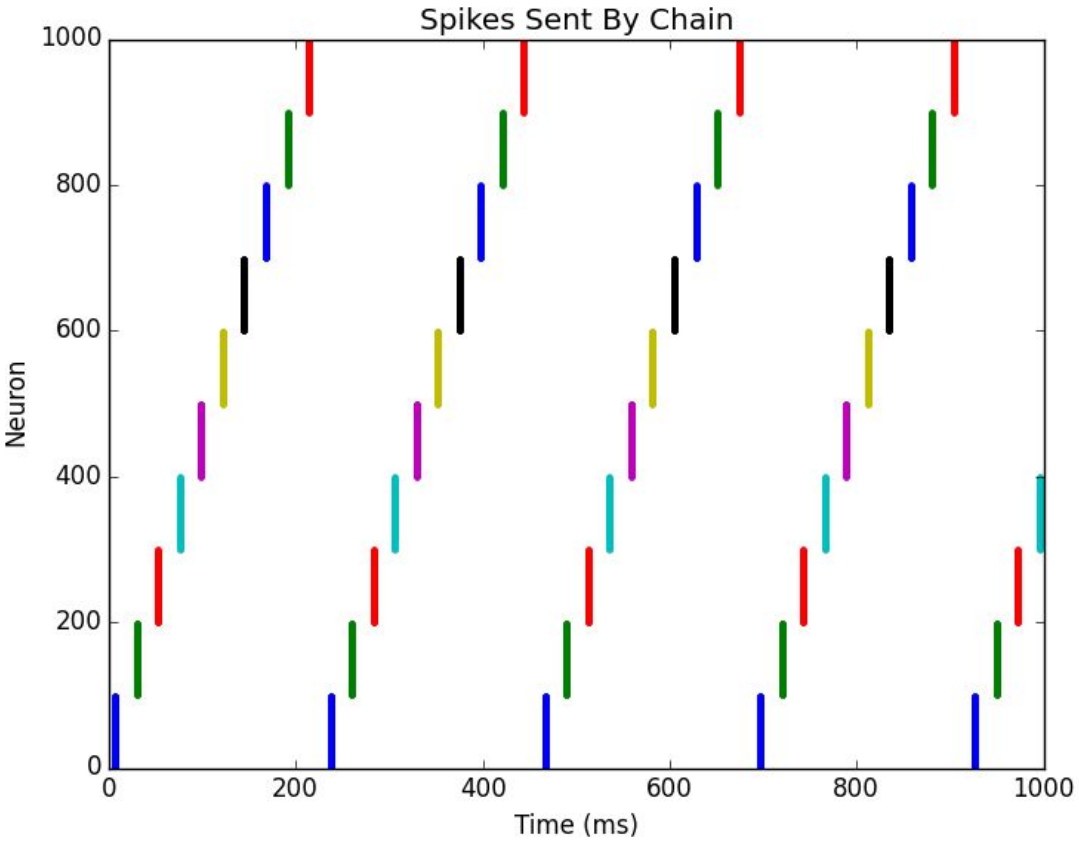


Figure 1: The output from a simple Synfire chain.

To run this example, from the top level of the folder type:

```
cd synfire
python synfire.py
```

A plot like the above should appear.

This example shows a PyNN Neural Network with a chain of 10 populations of 100 neurons each, where 10 neurons from each population excite all the neurons in the next population in the chain. The first population is then stimulated at the start of the simulation to start the chain running.

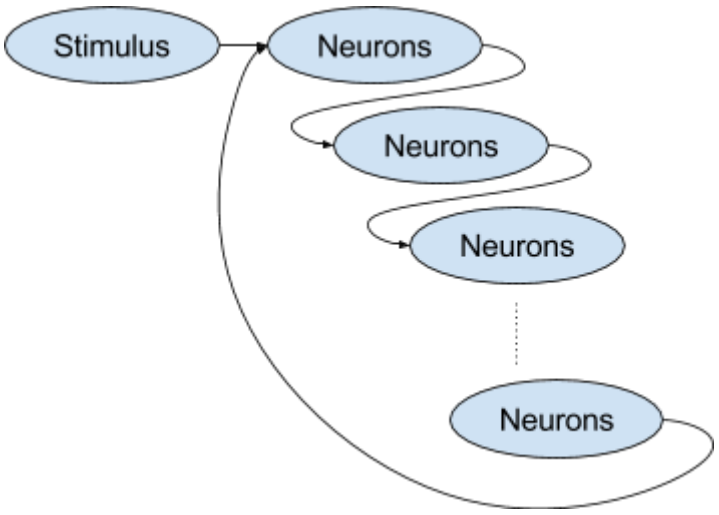


Figure 2: The Synfire Chain of Populations

Sudoku Game Through Neural Network

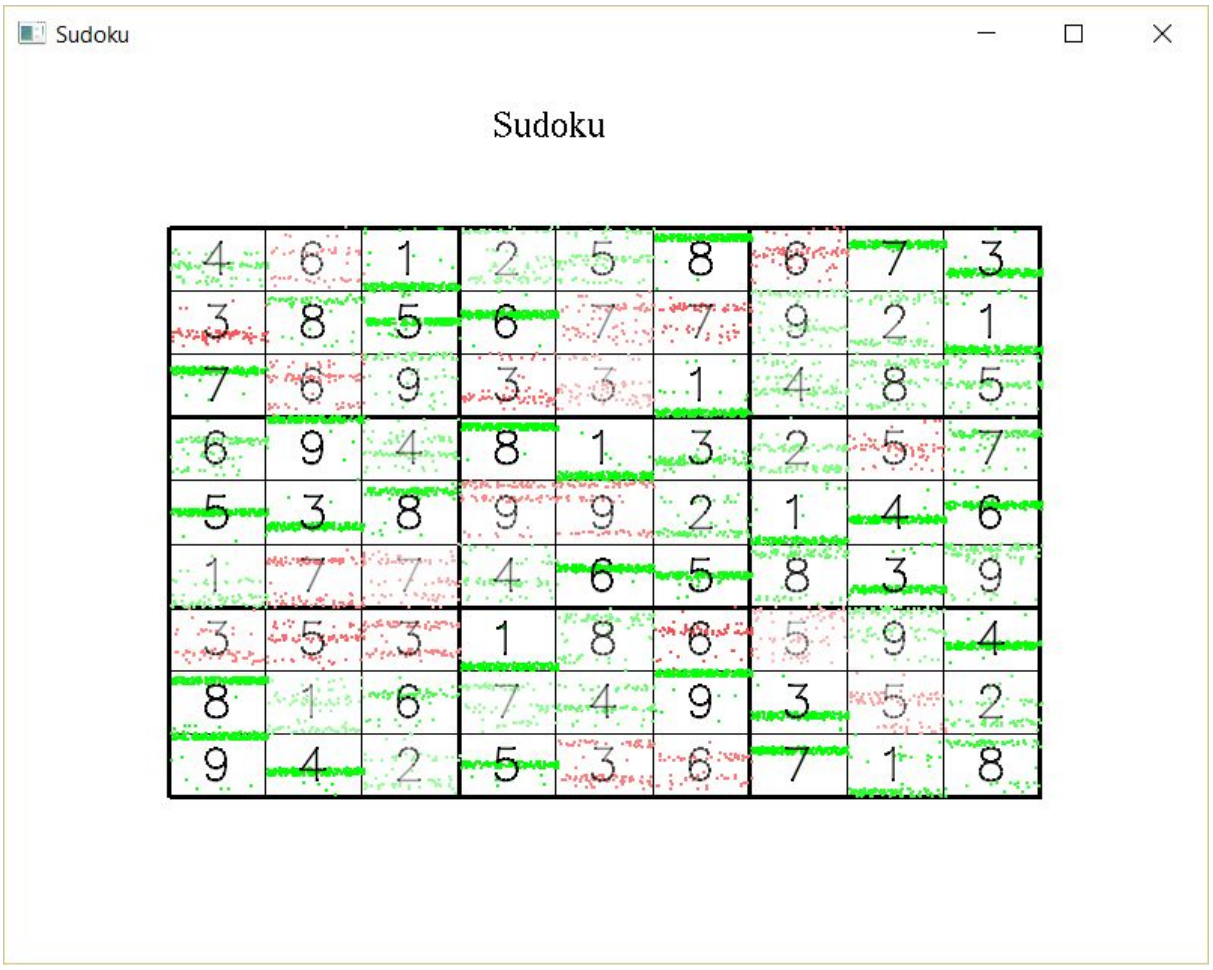


Figure 5: The output from the Sudoku game application

To run this example, from the top level of the folder type:

```
cd sudoku
python sudoku.py
```

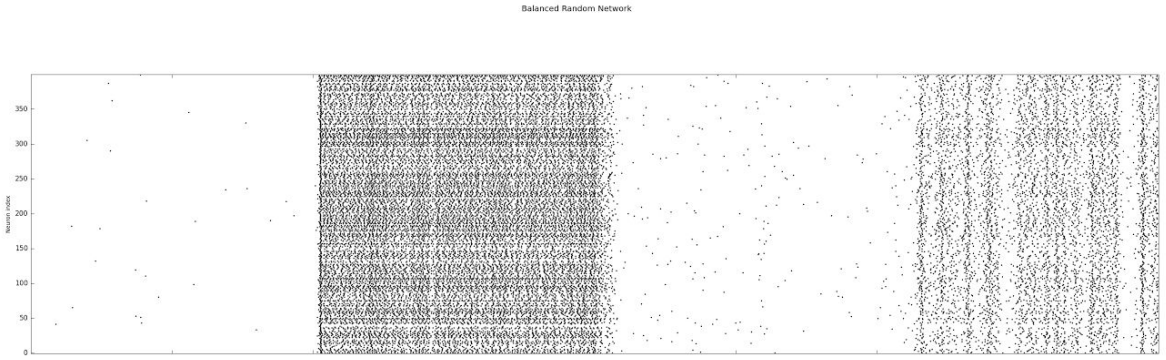
A visualiser will pop up, which is shown in Figure 5.

This example shows a PyNN neural network which describes a neural network for running sudoku problems. The spikes representing each cell are shown behind each number, with green output indicating that the value is valid according to the rules of Sudoku and red output indicating that the value is invalid. The problem to be solved is described near the top of the sudoku.py file, with 0s representing values to be computed. Note that on a small SpiNNaker board, the network is not always successful at solving the problem.

Balanced Random Network

To run this example, from the top level folder type:

```
cd balanced_random
python balanced_random.py
```

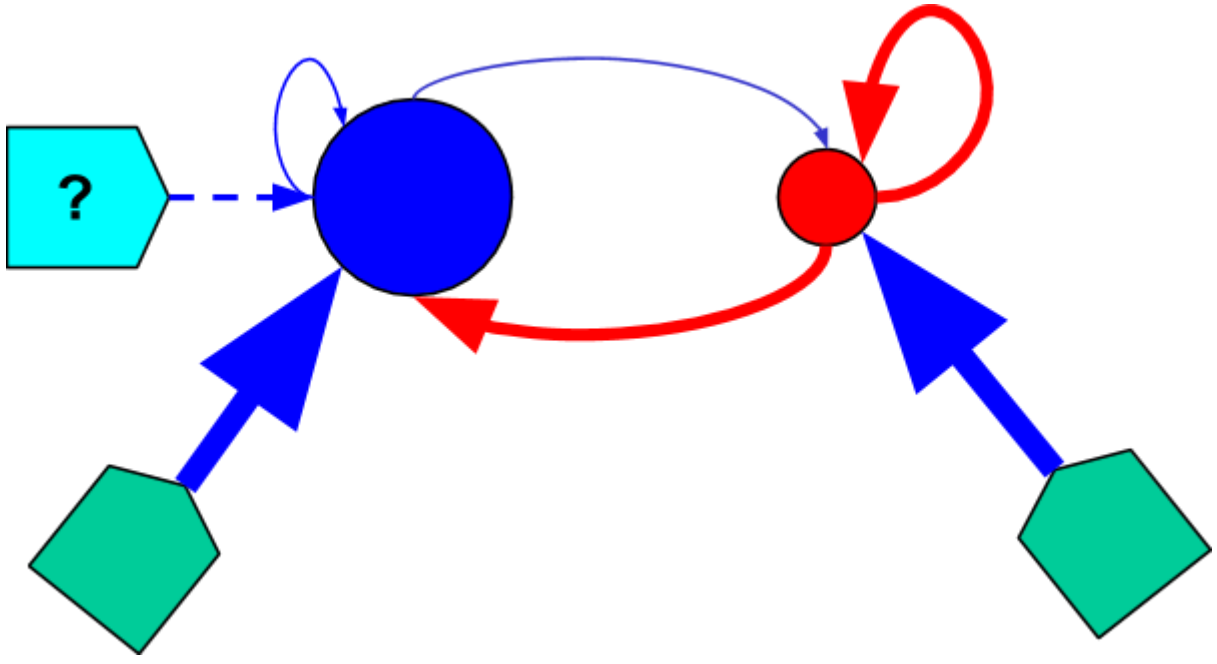


A plot like the above should appear.

This example shows a type of neural network known as a “Balanced Random Network”, which is believed to be representative of some areas of the brain. It is a combination of two populations, E and I, each stimulated to cause some random activity. These are then coupled to each other so that E *increases* the activity in I and I *decreases* the activity in E. Additionally, E *increases* the activity in itself and I *decreases* the activity of itself.

Without any external stimulation, random spikes are generated. If external stimulation is also added to E, it will spike even more. With a high rate of external input, the activity of E is high and the with a low rate, the activity is low, but with an intermediate rate, the network shows a more interesting pattern, with bursts of activity occuring at random times in the simulation.

The rates are changed in between runs of the simulation. You can change the existing rates and see the effect, or add additional runs with different rates.



Simple Learning Network

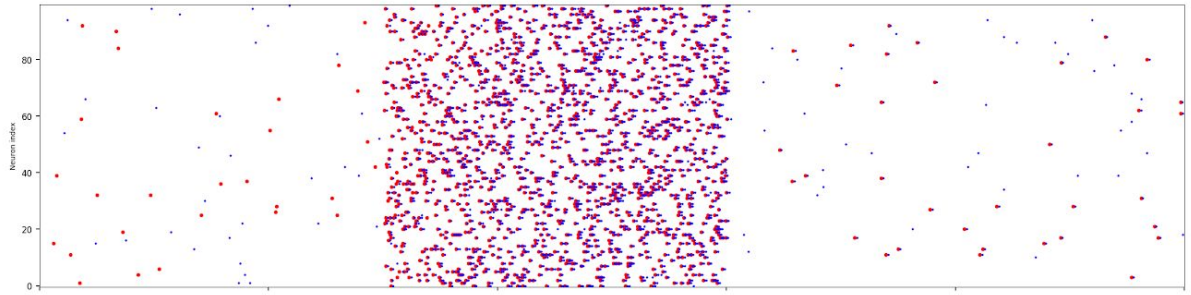


Figure 7: The output from the learning application

To run this example, from the top level folder type:

```
cd learning
python stdp.py
```

A plot like the above should appear.

This example shows the spike outputs from two populations of neurons that are connected together with a learning connection. At the start, both of the populations spikes regularly but the output is uncorrelated. In the middle, some learning is done; a training signal is applied to both populations and so both populations spike at a higher rate. At the end, whenever there is a red spike, there is a blue spike; the blue spikes still occur by themselves as well.

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

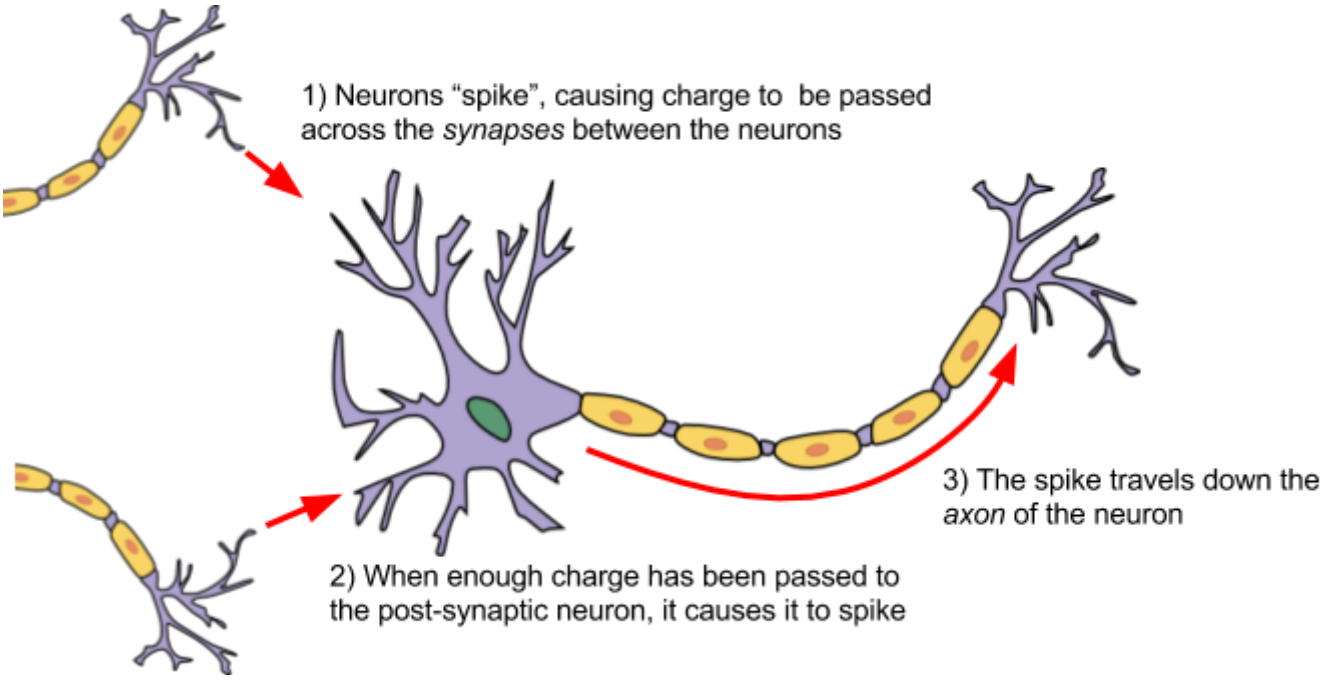
The PyNN toolchain for SpiNNaker (sPyNNaker), can be installed by following the instructions available from here:

http://spinnakermanchester.github.io/latest/spynnaker_install.html

Please install for PyNN version 0.8. Although PyNN version 0.7 is supported, this is only for backwards compatibility with existing scripts. New users are advised to learn the newer syntax of PyNN 0.8.

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.



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 spiking 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 sim
import pyNN.utility.plotting as plot
import matplotlib.pyplot as plt

sim.setup(timestep=1.0)
sim.set_number_of_neurons_per_core(sim.IF_curr_exp, 100)

pop_1 = sim.Population(1, sim.IF_curr_exp(), label="pop_1")
input = sim.Population(1, sim.SpikeSourceArray(spike_times=[0]), label="input")
input_proj = sim.Projection(input, pop_1, sim.OneToOneConnector(),
                            synapse_type=sim.StaticSynapse(weight=5, delay=1))
pop_1.record(["spikes", "v"])
simtime = 10
sim.run(simtime)

neo = pop_1.get_data(variables=["spikes", "v"])
spikes = neo.segments[0].spiketrains
print spikes
v = neo.segments[0].filter(name='v')[0]
print v
sim.end()

plot.Figure(
    # plot voltage for first ([0]) neuron
    plot.Panel(v, ylabel="Membrane potential (mV)",
               data_labels=[pop_1.label], yticks=True, xlim=(0, simtime)),
    # plot spikes (or in this case spike)
    plot.Panel(spikes, yticks=True, markersize=5, xlim=(0, simtime)),
    title="Simple Example",
    annotations="Simulated with {}".format(sim.name())
)
plt.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 fixed (or Static) 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.

Populations and Neuron Models

In PyNN, the neurons are declared in terms of a *population* of a number of neurons with similar properties. 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': 0.1, # The refractory period, in milliseconds
'v_reset': -65.0,  # 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 (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 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).

Projections and Connectors

Populations of neurons are joined together using a *Projection*. This is a directed connection where spikes are sent from the source, or pre-population and the target, or post-population. The projection between populations of neurons 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).
- AllToAllConnector - all presynaptic neurons connect to all postsynaptic neurons.
- FixedProbabilityConnector - each presynaptic neuron connects to each postsynaptic neuron with a given fixed probability *p_connect*.
- FromListConnector - the exact connectivity is described by *conn_list*, which is a list of tuples (*pre_synaptic_neuron_id, post_synaptic_neuron_id, weight, delay*) or just (*pre_synaptic_neuron_id, post_synaptic_neuron_id*). Note: All tuples must be the same length. If *weight, delay* are included the ones supplied through the synapse_type parameter of the Projection are ignored.
- FixedTotalNumberConnector - an exact number of connections *n_synapses* are made, drawn at random from the possible connections, with replacement. Note that this means that connections can be repeated.

As well as a connector the Projection must also have a synapse_type which determines how the synapse behaves when spikes are received. For example a StaticSynapse which has fixed weights and delays is specified as follows:

```
synapse_type=sim.StaticSynapse(weight=0.75, delay=1.0)
```

Random Parameters

Commonly, random weights and/or delays are used. To specify this, the value of the *weight* or *delay* of the synapse type are set to a *RandomDistribution*; note that the FromListConnector should then be specified with tuples of only (*pre_synaptic_neuron_id, post_synaptic_neuron_id*). The RandomDistribution 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 *low* (the minimum value) and *high* (the maximum value); and a ‘normal’ distribution with parameters of *mu* (the mean) and *sigma* (the standard deviation); as well as a ‘normal_clipped’ distribution, which takes the same parameters as ‘normal’ but with the addition of boundary parameters of *low* and *high* - 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.

Recording Data

All the Populations in a simulation can be recorded; the data which can be recorded is dependent on the simulation model. In general, all of the neuron models in PyNN allow the recording of the times at which each neuron spikes, *spikes*, and the membrane potential, *v*. In contrast, the input models (i.e. SpikeSourceArray and SpikeSourcePoisson) only allow the recording of spikes. On SpiNNaker, our neuron models additionally allow the recording of the neuron input using *gsyn*; technically, PyNN reserves this for the recording of synaptic conductance in models which support this (e.g. IF_cond_exp) but we also allow the recording of the synaptic currents in models such as IF_curr_exp.

Running the Simulation

Once the network has been described and the data to be recorded has been selected, the simulation can be started by calling *run* with the duration that the simulation is to be executed for. The *run* method can be called multiple times in sequence to run for further durations. In between each run, it is possible to change parameters of the network; at present SpiNNaker simulations only support the changing of the parameters

of the populations, such as changing the *i_offset* to adjust the input to the neurons. It is also possible to retrieve recorded data (see below) in between runs.

If you want to reset the simulation back to time 0 this can also be done using the *reset* call. At this point, it is now possible to make further changes in SpiNNaker simulations, such as adding Populations and Projections; note that these changes will result in a full remapping which takes longer than changes to the parameters.

Retrieving and Plotting Data

Once the simulation has been run, the Population `get_data` method can be used to retrieve the recorded data in the form of a Neo object (see http://neuralensemble.org/neo/). Each Neo object has a list of segments, one per reset-run cycle (so there will only be one if you never call reset). The content of each of the segments depends on the data recorded and requested.

Spike data is accessible via the *.spiketrains* property; there is one SpikeTrain for each neuron in the population. Each SpikeTrain can be treated as a numpy array of the times during the simulation at which the neuron spiked.

Other data is accessible via the *.filter(name=<signal_name>)* method, where *<signal_name>* is the name of the data item to retrieve (i.e. *v* for the membrane voltage). This returns an array of AnalogSignalArray objects; in the case of SpiNNaker there will only be one element in this array as all data is gathered together into a single array, thus the 0th element can always be used (e.g. *.filter(name='v')[0]*). The AnalogSignalArray in turn contains a list of AnalogSignalArray objects, one for each neuron. Each of these sub-arrays contains the list of values of the signal, one per time-step. Both SpikeTrain and AnalogSingnalArray objects extend Quanties arrays; this means that they come with the unit of the values as well. The SpikeTrain values are all in milliseconds, and the membrane voltages are in millivolts. These objects also hold additional metadata.

The results of Neo.segements[0].spiketrains and Neo.segements[0]..filter(name=)[0] can be passed to the *pyNN.utility.plotting.Panel* as shown in the example above. The module *spynnaker8.spynakker_plotting* contains a *SpynakkerPanel* object can also be used in the same way for slightly faster spike plots and to display heatmap-style plots for analog signal data.

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 255; larger populations are split up into 255-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.

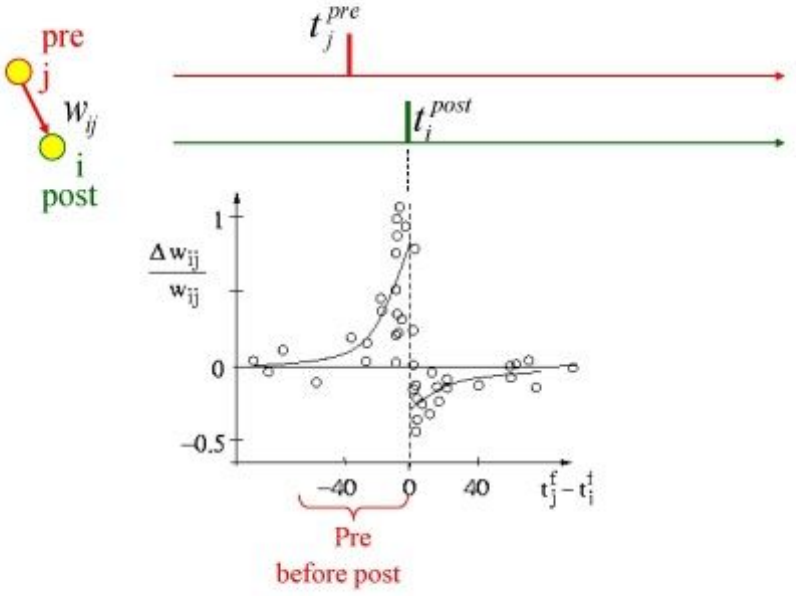
Spike-Time-Dependent Plasticity

STDP plasticity is a form of learning that depends upon the timing between the spikes of two neurons connected by a synapse. It is believed to be the basis of learning and information storage in the human brain.

In the case where a presynaptic spike is followed closely by a postsynaptic spike, then it is presumed that the presynaptic neuron caused the spike in the postsynaptic neuron, and so the weight of the synapse between the neurons is increased. This is known as potentiation.

If a postsynaptic spike is emitted shortly before a presynaptic spike is emitted, then the presynaptic spike cannot have caused the postsynaptic spike, and so the weight of the synapse between the neurons is reduced. This is known as depression.

The size of the weight change depends on the relative timing of the presynaptic and postsynaptic spikes; in general, the change in weight drops off exponentially as the time between the spikes gets larger, as shown in the following figure [Sjöström and Gerstner (2010), Scholarpedia]. However, different experiments have highlighted different behaviours depending on the conditions (e.g. [Graupner and Brunel (2012), PNAS]). Other authors have also suggested a correlation between triplets and quadruplets of presynaptic and postsynaptic spikes to trigger synaptic potentiation or depression.



STDP in PyNN

The steps for creating a network using STDP are much the same as previously described, with the main difference being that some of the projections use a `STDPMechanism` to describe the plasticity. Here is an example of the creation of a projection with STDP:

```
timing_rule = sim.SpikePairRule(tau_plus=20.0, tau_minus=20.0,
                                 A_plus=0.5, A_minus=0.5)
weight_rule = sim.AdditiveWeightDependence(w_max=5.0, w_min=0.0)

stdp_model = sim.STDPMechanism(timing_dependence=timing_rule,
                               weight_dependence=weight_rule,
                               weight=0.0, delay=5.0)

stdp_projection = sim.Projection(pre_pop, post_pop, sim.OneToOneConnector(),
                                 synapse_type=stdp_model)
```

In this example, firstly the timing rule is created. In this case, it is a *SpikePairRule*, which means that the relative timing of the spikes that will be used to update the weights will be based on pairs of presynaptic and postsynaptic spikes. This rule has four parameters. The parameters *tau_plus* and *tau_minus* describe the respective exponential decay of the size of the weight update with the time between presynaptic and postsynaptic spikes. Note that the decay can be different for potentiation (defined by *tau_plus*) and depression (defined by *tau_minus*). The parameters *A_plus* and *A_minus* which define the maximum weight to respectively add during potentiation or subtract during depression.

The next thing defined is the weight update rule. In this case it is a *AdditiveWeightDependence*, which means that the weight will be updated by simply adding to the current weight. This rule requires the parameters *w_max* and *w_min*, which define the maximum and minimum weight of the synapse respectively, Note that the actual amount added or subtracted will depend additionally on the timing of the spikes, as determined by the timing rule.

In addition, there is also a *MultiplicativeWeightDependence* supported, which means that the weight change depends on the difference between the current weight and w_max for potentiation, and w_min for depression. The value of *A_plus* and *A_minus* are then respectively multiplied by this difference to give the maximum weight change; again the actual value depends on the timing rule and the time between the spikes.

The timing and weight rules are combined with *weight* and *delay* into a single *STDPMechanism* object which describes the overall desired mechanism. Note that the projection still requires the specification of a *connector.* This connector is still used to describe the overall connectivity between the neurons of the pre- and post-populations. It is preferable that the initial weights fall between *w_min* and *w_max*; it is not an error if they do not, but when the first update is performed, the weight will be changed to fall within this range.

Note that on SpiNNaker, although multiple projections to the same target population can be specified with STDP, the restrictions on the current software are that all those projections must use the same rules with the exact same parameters. This is due to the restrictions of the local memory available on each core, reducing the amount of data that can be held for the parameters.

Note: In the implementation of STDP on SpiNNaker, the plasticity mechanism is only activated when the second presynaptic spike is received at the postsynaptic neuron. Thus at least two presynaptic spikes are required for the mechanism to be activated.

Getting Synaptic Data

The weights and delays assigned to a projection can be retrieved using the Projection's *get* method, specifying the data items to get, including 'weight', 'delay' and the parameters of the STDP Mechanism, and the format they are retrieved using. The data formats supported are 'list' format, where the return value consists of a list of tuples of the selected values; and 'array' where each value is returned in a two-dimensional matrix indexed by the source neurons in the pre-population, and the target neurons in the post-populations. In the 'list' result, each tuple additionally contains the source and target neuron ids as the 0th and 1st values in the tuple. In the 'array' result, missing connections are represented as 'NaN' (not a number) and positions where there are multiple connections have their values summed.

Note that on SpiNNaker, it is possible to retrieve the projection data before calling the PyNN run function, but that this data cannot be examined until after run has been called. This is because the individual connectivity data is not generated until the run function is called.

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 time step, 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).

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. Choose the number of neurons to be simulated in the network.

2. Set up the simulation to use 0.1ms timesteps.
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?

Task 4.1: STDP Network [Easy]

This task will create a simple network involving STDP learning rules.

Write a network with a 1.0ms time step consisting of two single-neuron populations connected with an STDP synapse using a spike pair rule and additive weight dependency, and initial weights of 0. Stimulate each of the neurons with a spike source array with times of your choice, with the times for stimulating the first neuron being slightly before the times stimulating the second neuron (e.g. 2ms or more), ensuring the times are far enough apart not to cause depression (compare the spacing in time with the tau_plus and tau_minus settings); note that a weight of 5.0 should be enough to force an IF_curr_exp neuron to fire with the default parameters. Add a few extra times at the end of the run for stimulating the first neuron. Run the network for a number of milliseconds and extract the spike times of the neurons and the weights.

You should be able to see that the weights have changed from the starting values, and that by the end of the simulation, the second neuron should spike shortly after the first neuron.

Task 4.2: STDP Parameters [Easy]

Alter the parameters of the STDP connection, and the relative timing of the spikes. Try starting with a large initial weight and see if you can get the weight to reduce using the relative timing of the spikes.

Task 5: STDP Curve [Hard]

This task will attempt to plot an STDP curve, showing how the weight change varies with timing between spikes.

1. Set up the simulation to use a 1ms time step.
2. Create a population of 100 presynaptic neurons.
3. Create a spike source array population of 100 sources connected to the presynaptic population. Set the spikes in the arrays so that each spikes twice 200ms apart, and that the first spike for each is 1ms after the first spike of the last e.g. [[0, 200], [1, 201], …] (hint: you can do this with a list comprehension).
4. Create a population of 100 postsynaptic neurons.
5. Create a spike source array connected to the postsynaptic neurons all spiking at 50ms.
6. Connect the presynaptic population to the postsynaptic population with an STDP projection with an initial weight of 0.5 and a maximum of 1 and minimum of 0.
7. Record the presynaptic and postsynaptic populations.
8. Run the simulation for long enough for all spikes to occur, and get the weights from the STDP projection.
9. Draw a graph of the weight changes from the initial weight value against the difference in presynaptic and postsynaptic neurons (hint: the presynaptic neurons should spike twice but the postsynaptic should only spike once; you are looking for the first spike from each presynaptic neuron).

Simple Data Input and Output with Spinnaker - Lab Manual

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.

PyNN Support

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

Output

The standard support for data output for a platform such as SpiNNaker, through the PyNN language, is to use the *record* method to declare the need to record, and the *get_data* method, for retrieval of the specific data. In the current implementation of sPyNNaker, all of the data declared to be recorded via record() is stored on the SDRAM of the chips that the corresponding populations were placed on. By writing the data to SDRAM, the data is stored locally and therefore is guaranteed to be read at some point in the future. In the current implementation, if the memory requirements for recording cannot be met, the model will be run for less time, paused whilst the data is extracted, and then resumed. This may be repeated a number of times until the whole simulation has completed.

The issue with the get functions are that they are called after run() completes, and therefore are not live, and so not able to interact with an external device running in real-time. When used with an external simulation, it is possible to call run a number times, extracting the data between each run and passing it to an external simulation. This mode of operation will not work if the external device or simulation cannot also be paused.

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 have to be supplied in advance with all the spikes to be sent (SpikeSourceArray). As with the output of spikes, it is possible to change the input spikes of a SpikeSourceArray, or the rate of the SpikeSourcePoisson between successive calls to run(). Again, this will only work if the external device or simulation can be paused.

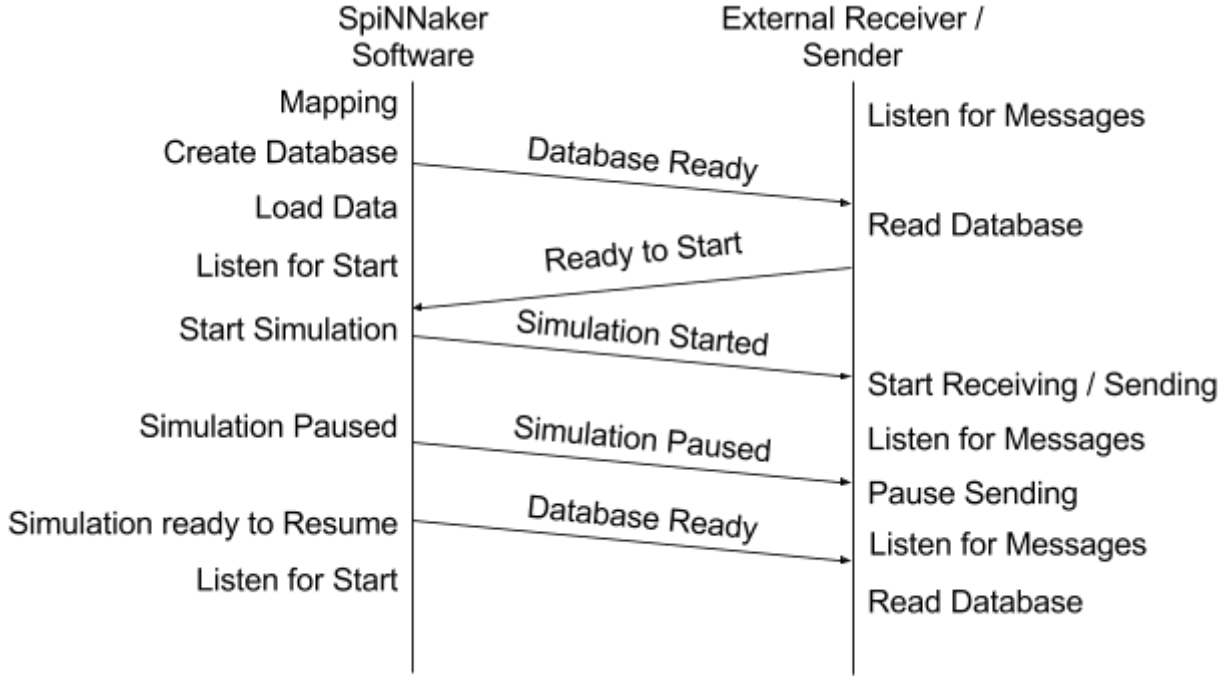
Live I/O Support

PyNN currently doesn't have any support for live interaction with simulations. It is worth noting that future releases of PyNN may use the MUSIC interface to support live injection and retrieval of spikes, but this has not yet been integrated into our software. To compensate for this, sPyNNaker includes an external_devices module that contains support for live injection and retrieval of spikes during a running PyNN simulation, whilst still maintaining the real-time operation of the simulation.

Live Event Database

When live input and output are used in the simulation, a database of the network is created with additional data, such as which SpiNNaker multicast keys are assigned to which neurons. This database can then be used by external receivers and/or senders to work out how to work out which neurons have spiked given the keys received as live output from SpiNNaker, or which keys to inject into SpiNNaker to send a spike into a simulation from outside.

During the mapping process within the software, the database is created, and a message is sent to the external receivers and/or senders that may want to read this database, to let them know that it is ready to be read. In addition, the software will then listen to be told that the database has been read, and that the external receiver or sender is ready for the simulation to start. Just after starting the simulation, the external receivers and senders are again notified so that they can start their activity. This helps to keep these systems in synchronization with the simulation.



Once the simulation has completed, it will pause. At this point the software will send a message to the senders and receivers that the simulation has paused or stopped (it isn't specified which at this point). The receiver or sender can then wait for a new cycle of messages, again starting with the message indicating that the database is ready. This will allow for any changes that have been made to the simulation. The process then goes around the loop again, from database ready, to notify ready to start, to start simulation to pause simulation and so on until all cycles are complete.

Live Output

To activate live retrieval from a given population, the command

p.external_devices.activate_live_output_for(<Population_object>**)**

is used (assuming pyNN.spiNNaker has been imported as p). This informs the sPyNNaker backend to send data out of the SpiNNaker system during the execution of the simulation. Note that this uses an additional core on the board.

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

Parameter	**Description**
port	The UDP port number to which the SpiNNaker machine will send packets. By default, the port specified as live_spike_port in the [Recording] section of your .spynnaker.cfg file will be used (17895 by default).
host	The host to which the SpiNNaker machine will send packets. By default, the IP address of the machine executing the script is used, but this can be changed if the spikes are to be sent to a different machine.

Live Input

To activate the live injection functionality, you need to instantiate a new *SpikeInjector* neural model is used. The SpikeInjector is considered to be a similar to a SpikeSourceArray, so you can build a population with a number of neurons etc in the normal way, as shown below:

```
injector_forward = Frontend.Population(
       5, p.external_devices.SpikeInjector(),
       label='spike_injector_forward')
```

Additional Parameters

As two or more programs cannot listen to the same UDP port for notification, the following additional parameters can be provided to the activate_live_output_for function or as parameters to the SpikeInjector.

database_notify_host	The host to which to send the notification that the database is ready to read. By default, the IP address of the machine executing the script is used. The same IP address can be used multiple times if the same host is running multiple senders or receivers or the same program is used for both sending and receiving.
database_notify_port_num	The UDP port number to which to send the notification that the database is ready to read. By default, port 19999 is used. This should be different for each sender or receiver; the same port can be used if the sender or receiver is the same as that used for other I/O.
notify	By default this is set to True, but if set to False, the above parameters are ignored and no notifications will be sent about the database for this sender or receiver. This can be useful to avoid sending messages to devices that can't be made to

	support the protocol.

Python Live Receiver / Injector

A SpynnakerLiveSpikesConnection is provided to perform the operation of sending or receiving spikes.

Receiver

The following block of code creates a live packet receiver to receive spikes from a live simulation:

```
1   # declare python code when received spikes for a timer tick
2   def receive_spikes(label, time, neuron_ids):
3       for neuron_id in neuron_ids:
4       print “Received spike at time {} from {}-{}”.format(
5           time, label, neuron_id)
6
7   # import python live spike connection
8   from spynnaker_external_devices_plugin.pyNN.connections.\
9                                 spynnaker_live_spikes_connection     import
SpynnakerLiveSpikesConnection
10
11  # set up python live spike connection
12  live_spikes_connection = SpynnakerLiveSpikesConnection(
13      receive_labels=[“receiver”])
14
15  # register python receiver with live spike connection
16  live_spikes_connection.add_receive_callback(“receiver”, receive_spikes)
```

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 7 to 9 imports the python support for live injection/live retrieval. The SpynnakerLiveSpikesConnection handles both live retrieval and live injection.
3. Lines 11 to 13 instantiates the SpynnakerLiveSpikesConnection, and informs the connection that it will receive data under the label “receiver”.
4. Lines 15 to 16 informs the connection that any packets being received with the “receiver” label need to be forwarded to the function receive_spikes defined on lines 1 to 5.

This script must be run in advance of the script that sets up the simulation. The SpynnakerLiveSpikesConnection will listen for the simulation script to complete the setup operations and so starts synchronized with the simulation. It is possible to run the reception of spikes within the same script as the simulation; to do this, ensure that the above code is placed before the call to run().

Sender

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)
5
```

```
6  # import python injector connection
7  from spynnaker_external_devices_plugin.pyNN.connections.\
8  spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
9
10 # set up python injector connection
11 live_spikes_connection = SpynnakerLiveSpikesConnection(
12     send_labels=["spike_sender"])
13
14 # register python injector with injector connection
15 live_spikes_connection.add_start_callback("spike_sender", send_spike)
```

1. Lines 1 to 3 create a function that will be called when the simulation starts, allowing the synchronized sending of spikes.
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 10 to 12 instantiates the SpynnakerLiveSpikesConnection, and informs the connection it will inject data via the label spike_sender.
4. Lines 14 to 15 informs the connection that when the simulation starts, to call the send_spike function defined on lines 1 to 3.

As with the live reception script, this must be called before the simulation script, or before run() in the simulation script.

Multiple Senders and Receivers

If you need more than one SpynnakerLiveSpikesConnection on the same host, the connection can take an additional parameter specifying the local port to listen on for notifications from the simulation, by specifying the local_port parameter in the constructor e.g.:

```
live_spikes_connection_1 = SpynnakerLiveSpikesConnection(
    receive_labels=["receiver"], local_port=19996)
live_spikes_connection_2 = SpynnakerLiveSpikesConnection(
    send_labels=["sender"], local_port=19997)
```

Note that you must then also tell the simulation side that these ports are in use as described previously. This can be done when calling activate_live_output_for for the population or when creating a SpikeInjector by specifying the database_notify_port_num parameter e.g.

```
activate_live_output_for(receiver, database_notify_port_num=19996)
injector = p.Population(1, p.external_devices.SpikeInjector(
   database_notify_port_num=19997), label="sender")
```

Caveats

To use the live injection and retrieval 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, this functionality uses a number of additional SpiNNaker cores. Therefore a network which would just fit onto your SpiNNaker machine before would likely fail to fit on the machine when these functionalities are added in.

Tasks

Task 1.1: A synfire chain with live output [Easy]

This task will create a synfire chain which displays live output. Start with the synfire chain from PyNNExamples.

1. Call activate_live_output_for(<pop_object>) on the synfire population.
2. Build a python receiver function that prints out the neuron ids for the population.
3. Import and instantiate a SpynnakerLiveSpikesConnection connection.
4. Link a receive callback to the python receiver function and print when a spike is received.

Task 1.2: A synfire chain with live input [Easy]

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

1. Remove the spike source array population.
2. Replace it with the SpikeInjector population.
3. Build a python injector function which sends a spike when called.
4. Import and instantiate a SpynnakerLiveSpikesConnection.
5. Link a start callback to the python injector function.

Task 1.3: A synfire chain with live input and output [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 2: 2 co-operative live synfire chains [Medium]

This task will create a 2 synfire chains which activate each other, but via python I/O.

1. Create a synfire chain activated by the first neuron of a SpikeInjector with 2 neurons.
2. Create a second synfire chain activated by the second neuron of the same SpikeInjector.
3. Activate live output for the two synfire chains.
4. Create a function that will run on start that will send a spike from the first neuron of the SpikeInjector.
5. Create a function to receive spikes from the first synfire chain; when a spike is received from the last neuron of this synfire chain, send a spike from the second neuron of the SpikeInjector.

6. Create a function to receive spikes from the second synfire chain; when a spike is received from the last neuron of this synfire chain, send a spike from the first neuron of the SpikeInjector.
7. Create a SpynnakerLiveSpikesConnection instance for the SpikeInjector and the two synfire chains.
8. Add each of the above functions as callbacks to the appropriate events.
9. Run the network and display the results.

Task 3: Synfire chain with multiple I/O scripts [Hard]

This task will work similarly to the previous tasks, but will split the parts into multiple scripts, which then need to be started in the correct order for it to work. This will be similar to having an external visualiser or external device which needs to be started before the simulation starts.

1. Create a script which uses the SpynnakerLiveSpikesConnection to receive spikes from a population specified on the command line, using a database port number specified on the command line and prints the spikes to the terminal (hint: use sys.argv[1] to get the first argument to a Python script).
2. Create a script which uses the SpynnakerLiveSpikeConnection to send spikes to a population specified on the command line, using a database port number specified on the command line, sending a spike with a neuron id specified on the command line after a random interval after the simulation starts.
3. Create a script which sets up a SpikeInjector connected to a synfire population, and which produces live output. This should set up two database notification ports each of which are specified on the command line, one for the SpikeInjector and one for the live output.
4. Run the scripts in the correct order specifying the correct parameters.

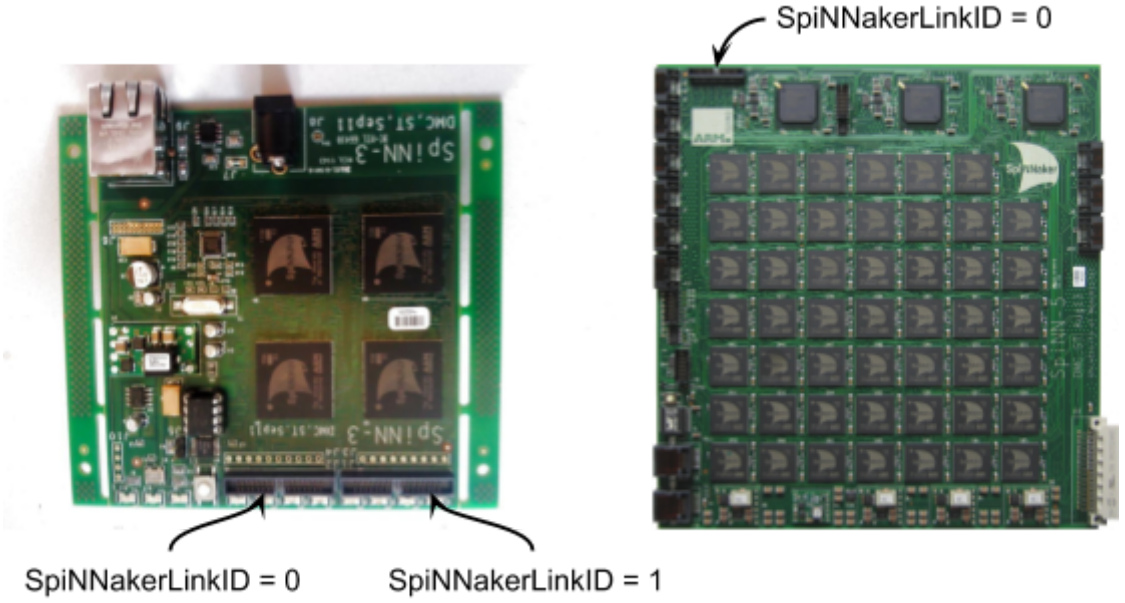
External Devices on SpiNNaker - Lab Manual

Introduction

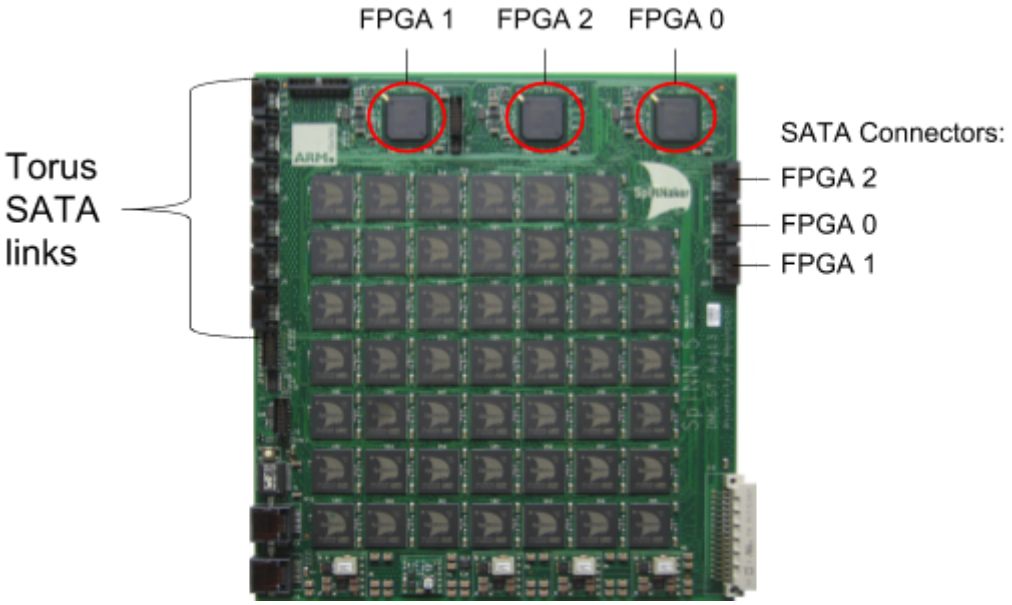
This manual will discuss the connection of external devices to SpiNNaker and how to tell the software that they are in use.

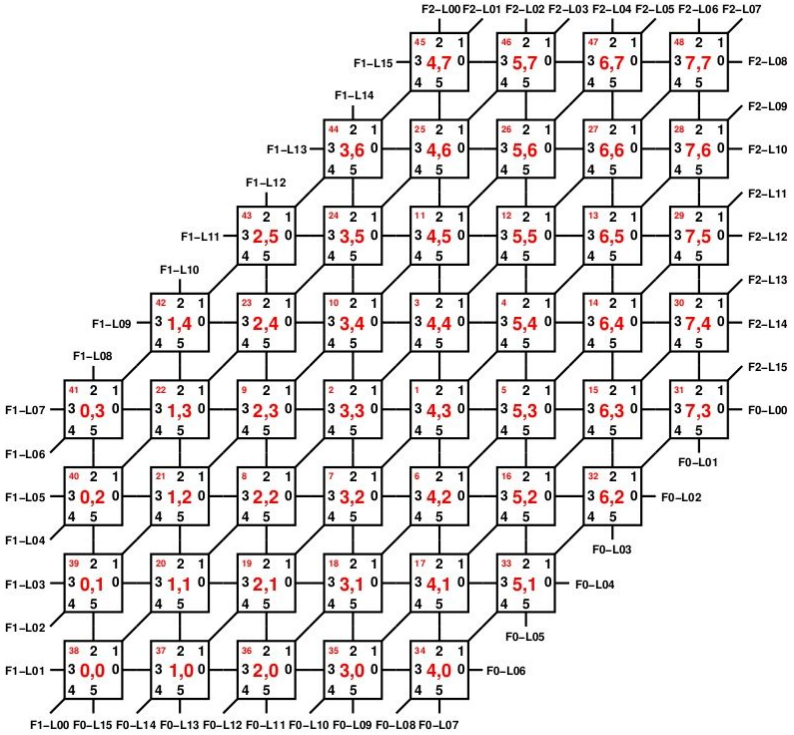
Physical Connection

The SpiNNaker boards have a number of places to which external devices can be connected. The first of these is the SpiNNakerLink connections. These are shown below.



Additionally, the 48-node boards have FPGAs which are linked to the SATA connectors. These are shown below.





The FPGAs are connected internally to a subset of the border chips on the board as shown above; the links at the edges of the board are coded as F<fpga-id>-L<fpga-link-id>, so F1-L04 is FPGA 1, FPGA link id 4.

The FPGA id and FPGA link ids, or the SpiNNakerLink ids are used to tell the software where a device has been connected. When a SpiNNakerLink is specified, the device is connected directly to this link. When an FPGA id is used, this indicates the FPGA to which the device is connected and which link of the FPGA it is connected to. The tools do not currently do any reprogramming of the FPGAs themselves, so the FPGA must have been configured in advance to forward packets between this link of the FPGA and the SATA link to which the device is connected.

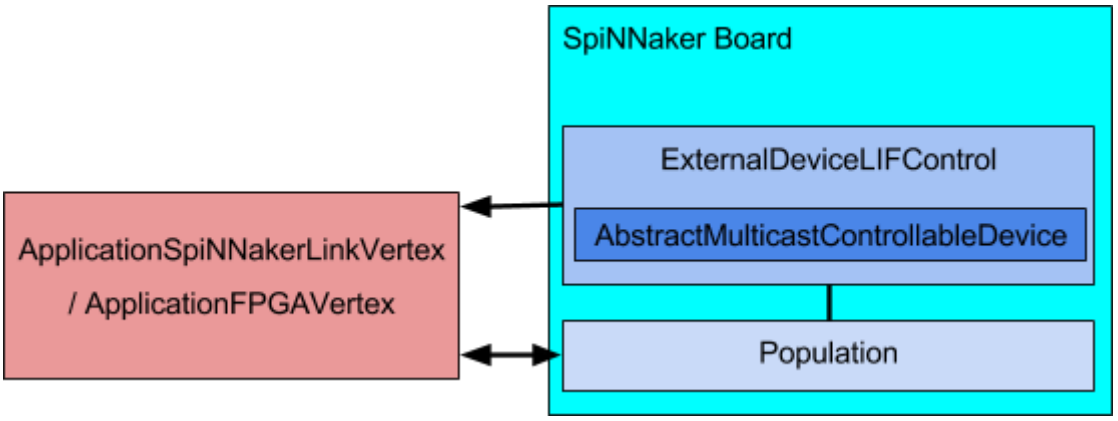
When a multi-board system is in use, it is important that you do not connect your device to the SpiNNakerLink unless you know that the FPGA has been disabled for that link. If you don't do this, you could damage the board. Devices can be connected to the FPGA SATA connectors as you wish, but without reconfiguring the FPGA, it is unlikely that any communication will occur.

When using a multi-board system, you must also be able to tell the software which board you are connecting your device to. This is done using the IP address of the ethernet-connected chip on the board. The board which contains chip 0, 0 will generally have the same IP address as the one you use to contact the board; if you don't specify a board address, it will be assumed that you want to use this board in any case, so unless you have multiple devices to connect, you may want to choose this board first.

Ethernet Connection

If you have a device that connects over Ethernet or some other means using the host machine, the software can also be configured to act as an intermediary between the device and SpiNNaker. No special connections are required in this case.

Software Connection for SpiNNaker Link and FPGA Devices



Device Specification

Once you have physically connected the device to the machine, the software needs to be told that that you want to add the device to a neural network. This is generally done by using one of the models provided for external devices. These are:

- **pacman.model.graphs.application.ApplicationFPGAVertex** - this is used when the device is connected to an FPGA. You need to provide the **fpga_id**, the **fpga_link_id** and optionally the **board_address** when you are using a multi-board system, and the board connected to is not the first board.
- **pacman.model.graphs.application.ApplicationSpiNNakerLinkVertex** - this is used when the device is connected to a SpiNNaker Link. You need to provide the **spinnaker_link_id** and optionally the **board_address** when you are using a multi-board system, and the board connected to is not the first board.

Input

If you are using the device as an input device input a PyNN network, you will need to specify the multicast keys that the device will generate. This is done by creating a Python **class** for the device which extends one of the aforementioned interfaces in addition to the interface:

spinn_front_end_common.abstract_models.AbstractProvidesOutgoingPartitionConstraints

This requires the addition of a method called:

def get_outgoing_partition_constraints(self, partition)

This takes a parameter called **partition**; this can be ignored for the purposes of integration in PyNN. This function should return a **constraint** on the keys that are to be sent from this vertex. In general, you can return a fixed key and mask pair from this method indicating the base key and range of keys to use e.g.:

return [FixedKeyAndMaskConstraint([BaseKeyAndMask(0x12340000, 0xFFFF0000)])]

The example above would indicate that the device will send keys where the first 16 bits have the value 0x1234 and the rest of the bits can be set to any value between 0 and 0xFFFF (thus the range of keys the device is expected to send is 0x12340000 - 0x1234FFFF). It is not critical that the device uses all these keys, but it is essential that every key that the device will send is covered, otherwise packets sent by the device can interfere with the operation of the SpiNNaker system.

You can now use the device **as a source** in a PyNN network by providing it as a model in a Population. The device will then be treated as a source of spikes, like a SpikeSourceArray or SpikeSourcePoisson. Thus you can create a projection from the device to other Population objects in the network, where the device population is the source population.

Output

If you would like to send spikes to the device, this is done differently from other objects in the network. This is because a Projection expects to accept a connection describing the connectivity between the source and target neurons. Populations of neurons use the connection information to create a synaptic matrix; on reception of a spike this is used to work out which neurons are targeted by the spike. An external device is not assumed to have a synaptic matrix, and there is no standard way for the software to communicate the synaptic matrix to the device. Thus the device is expected to accept all the spikes from all the neurons, and process them accordingly. For this reason, a separate method is used to send spikes from a PyNN Population object to a device. This is:

p.external_devices.activate_output_to(<source_population>, <destination_device>)

where **<source_population>** is a standard PyNN Population object (which can include other devices and input sources), and **<destination_device>** is the device population.

The keys that the device will receive in this case are those that have been assigned to the source population. If the device has a specific set of keys that need to be sent to control it, this will be described in more detail below.

Commands

Commands are SpiNNaker multicast messages with or without payloads that can be sent to a device at the start and end of a simulation, or at specified times during a simulation. This can be used to set up and cleanly stop an external device. In these contexts, you can tell a device the multicast keys to use in the simulation, and you can tell it to start transmitting at the start of the simulation, and stop transmitting at the end. This can be important when operating SpiNNaker, since if your device is sending data packets into SpiNNaker, it might not be able to perform normal functions, such as booting the system.

If you have a device that supports commands, you can extend you device class with:

spinn_front_end_common.abstract_models.AbstractSendMeMulticastCommandsVertex

This requires the following properties to be implemented, each of which returns a list of:

spinn_front_end_common.utiltiy_models.MultiCastCommand(
key, payload=None, time=None)

- **start_resume_commands** - returns a list of MultiCastCommand instances to send at the start of the simulation, or when the simulation resumes after a pause. The time field is ignored.
- **pause_stop_commands** - returns a list of MultiCastCommand instances to send at the end of simulation, or when the simulation pauses. The time field is ignored.
- **timed_commands** - return a list of MultiCastCommand instances to send at specified times during the simulation. The time field must be specified.

Any of these can return an empty list of commands (most commonly the timed_commands does this).

Device Control During Simulation

In addition to the above support for handling direct output of spikes to devices, the software can also be configured to handle the sending of commands to a device based on the state of the simulation, in terms of the membrane voltage of a LIF neuron. These commands will consist of multicast messages with a payload, where the payload is the membrane voltage. The messages can be configured to be sent at multiples of the timestep of the simulation to reduce the amount of data being sent over the SpiNNaker network.

The neuron model to use to control devices is:

p.external_devices.ExternalDeviceLifControl

This has the parameters of the IF_curr_exp neuron, with the exception of v_thresh (since it has no threshold voltage and never spikes). Additionally, the following parameters must be specified:

- **devices** - the list of devices that will be controlled by this population. The length of devices must be the same as the number of neurons in the population. These are detailed below.
- **create_edges** - indicates whether edges to the devices should be added to the network. This is for compatibility with Ethernet devices (see later). Set to True for SpiNNakerLink and FPGA devices.

Each device specified in the parameter **devices**, in addition to extending either the ApplicationSpiNNakerLinkVertex or ApplicationFPGAVertex, should additionally extend:

spynnaker.pyNN.external_device_models.AbstractMulticastControllableDevice

This requires the class to have the following properties:

- **device_control_partition_id** - the id to give the partition of edges going to this device. This can be anything such as the name of the device.
- **device_control_key** - the key to use in the multicast packet to be sent to the device.
- **device_control_uses_payload** - True if the device will take the membrane voltage as a payload, False if the key will be sent by itself. It is expected that most devices will want to set this to true to receive the membrane voltage in the payload.
- **device_control_min_value** - the minimum value of membrane voltage that the device accepts. If the voltage is below this value, this value is sent instead.
- **device_control_max_value** - the maximum value of membrane voltage that the device accepts. If the voltage is above this value, this value is sent instead.
- **device_control_timesteps_between_sending** - the number of timesteps between the sending of the packets, to reduce the bandwidth used.

This LIF-based model can now be used as a Population in a simulation, and as a target population of a standard PyNN Projection. Spikes received by the population will increase the membrane voltage of neurons in the population if they are connected via an excitatory connection, or decrease the membrane voltage if connected via an inhibitory connection, as normal. It is important to realise though that a Population object created with this model will never spike. It is not an error to use it as the pre-population in a Projection, but no spikes will be sent across the projection in that case. Note, that this population, can now be recorded like any other population, unlike the devices themselves.

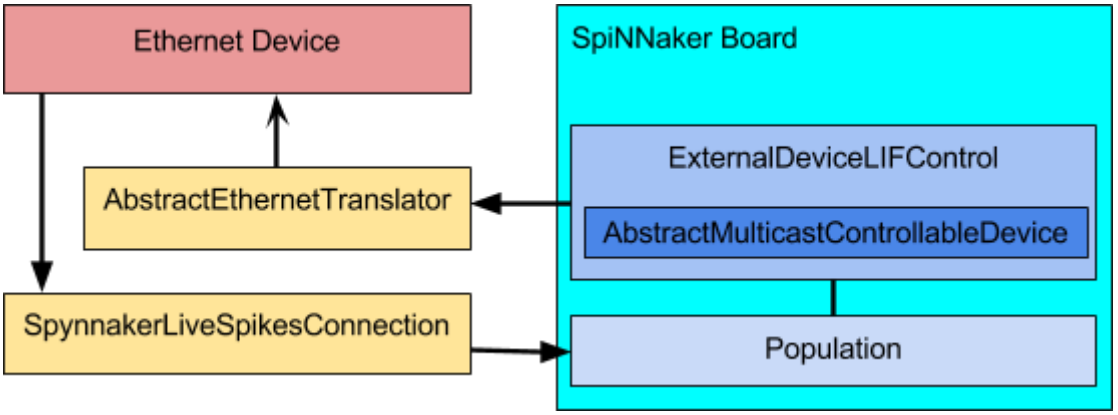
The format of the membrane voltage sent as the payload is S1615 format (this is a 32-bit number with the first bit being the sign, the next 16-bits are the integer part of the number and the remaining 15-bits are the

fractional part of the number). The default parameters therefore set the rest and reset voltages of the neuron to 0 (e.g. when using a motor where the membrane voltage is the speed of the motor, this would mean the speed would be 0 without any input).

Warning about Keys

There are two potential sources of keys that can be received by the devices: those sent by commands at the start and end of simulation, and those sent to the device during simulation containing the membrane voltage. These two sources of keys must generate distinct keys e.g. if you are using a command at the start that sets a motor speed to 0, and then you want to set the motor speed using the voltage during simulation, your device needs to accept different keys for this. This is often done by having your device ignore part of the key e.g. it might use the bottom 16-bits to identify the command it is being asked to perform, but ignore the top 16-bits.

Software Connection for Ethernet and Other Devices



Output

If your device only works over an Ethernet connection, or some other connection via the host machine, this can be controlled using the Live I/O discussed in another lab. The software does, however, have some support for using these devices with the aforementioned LIF neuron membrane voltage output. To make this work, the device is defined as a class which, as described for SpiNNakerLink and FPGA devices, extends the interface:

spynnaker.pyNN.external_device_models.AbstractMulticastControllableDevice

See above for a description of the properties that the implementation must provide. Note that the device does **not** now have to extend ApplicationSpiNNakerLinkVertex or ApplicationFPGAVertex. However the device still has to produce keys for the commands. These can be arbitrary values, so long as they don't clash with other keys in use.

Once the device is defined, you then need to define a class that extends:

spinnaker.pyNN.external_device_models.AbstractEthernetTranslator

This is the class that will communicate with your device, and so should contain all the mechanisms through which you do this. The class requires the implementation of a single method:

def translate_control_packet(self, multicast_packet)

This method takes a multicast packet, which will have a properties of **key** and **payload** indicating the key and payload of the packet received. The key will be the key returned by the device_control_key that you defined as part of the definition of your device. The payload will contain the membrane voltage value.

Note that you can also make your device extend the class:

spinn_front_end_common.abstract_models.AbstractSendMeMulticastCommandsVertex

This will allow the device to be sent commands at the start and end of simulation. These commands will also be sent via the above translator instance, so the translator must also recognise these packets. As with the other devices, the keys used for start and stop commands must be different from those used to control the device, though with an Ethernet device, these keys are arbitrary in any case.

Once these parts have been defined, an instance of **p.external_devices.ExternalDeviceLifControl** is again created, with parameters as previously described, but with **create_edges** set to False, and an additional parameter **translator**, which is an instance of the translator created above.

The final instantiation of the device is via **p.external_devices.EthernetControlPopulation**. This is used **in place of** p.Population, **not** as a model. This method takes the following parameters:

- **n_neurons** - the number of neurons in the population. This should be the same as the number of devices passed to the ExternalDeviceLifControl model.
- **model** - this is the ExternalDeviceLifControl instance defined above.

The result of this call can be used as the target population of a PyNN Projection to send spikes to the device. It is valid to use it as a source population but the Projection will then pass no spikes.

Input

For Ethernet Input, again Live I/O can be used as described elsewhere. The software also provides additional interfaces to link the device more closely to the software and make it easier to use as a single unit.

To take advantage of this functionality, the Ethernet input device should be defined in a class which extends:

spynnaker.pyNN.external_device_models.AbstractEthernetSensor

This class requires that the following methods are implemented:

- **get_n_neurons** - return the number of neurons the sensor provides.
- **get_injector_parameters** - return any parameters to pass to the SpikeInjector.
- **get_injector_label** - return the label to give to the SpikeInjector.
- **get_translator** - return an AbstractEthernetTranslator as defined for output. This is used for start and stop commands, so you can return None if the class doesn't also extend AbstractSendMeMulticastCommandsVertex.
- **get_database_connection** - return a SpynnakerLiveSpikesConnection that is to be used to send the spikes into SpiNNaker.

If desired, you can write a class which extends SpynnakerLiveSpikesConnection and gathers data from your device and uses the send_spikes method to inject spikes into the network based on the data. This

can then be returned from get_database_connection. Alternatively, you can create a SpynnakerLiveSpikesConnection instance elsewhere and return this directly.

Once this has been implemented, you can instantiate the device using:

p.external_devices.EthernetSensorPopulation

This is again used in place of a pyNN Population object and **not** as a model. This requires a single paramerter **device** which is the device you have defined above.

The result of this call be be used as the source population of a projection. It is not valid to use this as a target population.

Pushbot

The TUM Pushbot is a commonly used robotics platform which is compatible with SpiNNaker over Ethernet and SpiNNakerLink with an additional SpiNNaker adapter. The Pushbot therefore is a good example of the use of the various interfaces described above.

The Pushbot has several output devices, including two motors, a laser, two LEDs, and a speaker. These are each considered to be different devices from the point-of-view of the tools. To help with key allocation, the pushbot implementation has a MunichIOProtocol instance, which gives the keys for various commands given a base key.

The Pushbot has several sensors, but so far only the retina has been implemented in the tools. When used over SpiNNakerLink, the SpiNNaker board protocol will downsample the retina into various formats. The Ethernet connection to the Pushbot doesn't support this, so the software is designed to do this on the host machine between receiving spikes from the retina and sending them on to SpiNNaker.

The tools also include a Pushbot retina visualiser. This can receive spikes from the Pushbot retina and display them in a graphical window. This shows how the retina works based on the changes in light.

Tasks

These tasks will take you through the process of interacting with external devices. You don't actually have to have a device connected to work through this process, or to run some scripts. You may see some warning messages because of this, but the scripts should still otherwise run correctly.

Task 1.1 [Easy]

Create a network which sends spikes from a Poisson spike source to an ApplicationSpiNNakerLinkVertex with a single neuron on spinnaker link 0. Note that when you run the network, you should get a message like the following:

```
2017-09-22 20:28:28 WARNING: The reinjector on 0, 0 has detected that 76242 packets were dumped from a outgoing link of this chip's router. This often occurs when external devices are used in the script but not connected to the communication fabric correctly. These packets may have been reinjected multiple times and so this number may be a overestimate.
```

The reason for this is that there isn't actually a device connected to your board. However, this message indicates that the packets are correctly routed towards the spinnaker link.

Task 1.2 [Easy]

Try changing the spinnaker link to 1. If you have a 4-node board, this should result in a similar message but referencing chip 1, 0. If you are using a 48-node board, you should get an error since there isn't a second spinnaker link on the board.

Task 2.1 [Medium]

Create your own device class that extends ApplicationSpiNNakerLinkVertex and AbstractProvidesOutgoingPartitionConstraints. Set up the device up so that it sends with a base key of 0x12340000 and mask 0xFFFF0000. Again connect a Poisson spike source to the device.

Task 2.2 [Easy]

Extend the class further to implement AbstractSendMeMulticastCommandsVertex. Add some start and end commands.

Task 3 [Hard]

Create another device class, this time extending ApplicationSpiNNakerLinkVertex and AbstractMulticastControllableDevice. Create a script with a Population that uses a ExternalDeviceLifControl as a model, which takes the device you just created. Record the membrane voltage from the device. Connect a Poisson spike source to the device, run for some number of milliseconds and then print or graph the membrane voltage.

Task 4 [Hard]

Create another device class which extends AbstractMulticastControllableDevice but nothing else. Create a translator class that extends AbstractEthernetTranslator, and in the implementation prints out the key and payload of the multicast message. Add the device to an ExternalDeviceLifControl instance along with the translator. Create an EthernetControlPopulation, and feed this with a Poisson spike source. Record the membrane voltage of the control population, run the simulation for some number of milliseconds and then print or graph the membrane voltage.

Task 5 [Very Hard]

Create a class which extends AbstractEthernetSensor and has a single neuron. This class should create its own SpynnakerLiveSpikes connection, and register a method against this to send a spike to neuron 0 at start up (probably after a short pause e.g. 0.01 seconds to make sure the simulation is actually running). Be careful that the label given to the connection is the same as that returned from the sensor get_injector_label() method, and make sure this same connection is returned from the get_database_connection() method. The sensor doesn't send start or stop commands, so it doesn't need a translator (i.e. this method can return None), and no additional injector parameters are required. Connect the sensor to a standard LIF population with a single neuron. Record the voltage from this population and make sure that it changes in response to the single injected spike.

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:

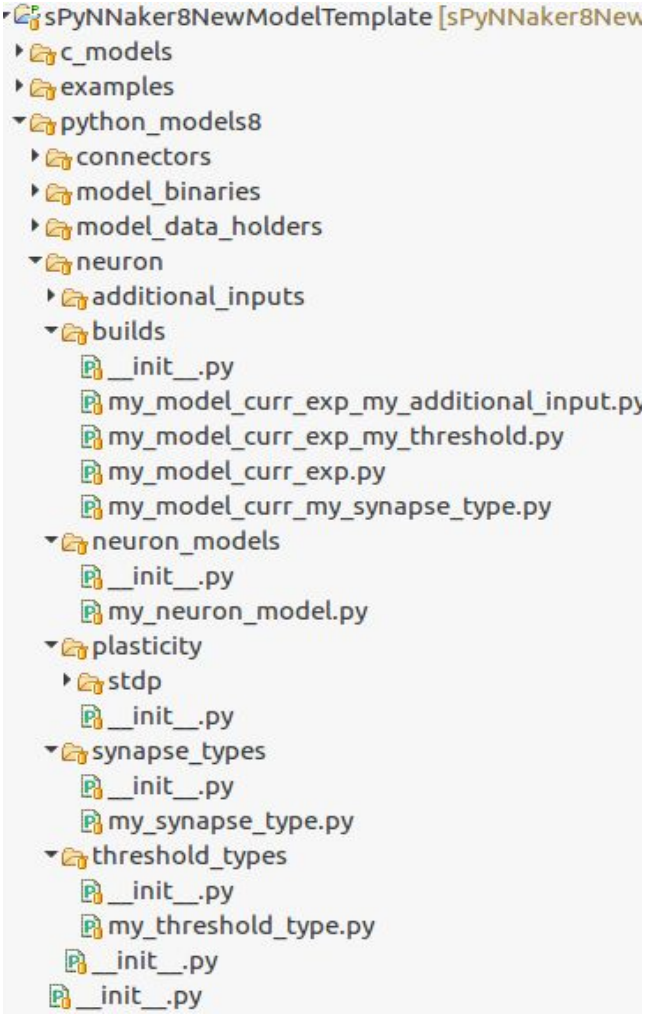
http://spinnakermanchester.github.io/spynnaker/4.0.0/PyNNonSpiNNakerExtensions.html

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.

C code

Python code



For SpiNNaker 4.0.0, this template structure can be downloaded from one of the following locations:
https://spinnakermanchester.github.io/latest/spynnaker_new_model_template_zip.html
https://spinnakermanchester.github.io/latest/spynnaker_new_model_template_tar_gz.html

If you have already set up a development environment for SpiNNaker in C and Python following the instructions at https://spinnakermanchester.github.io/development/devenv.html then this template structure can be downloaded from https://github.com/SpiNNakerManchester/sPyNNaker8NewModelTemplate.git (for PyNN 0.8) or from https://github.com/SpiNNakerManchester/sPyNNaker7NewModelTemplate.git (for PyNN 0.7).

C model builds

All neuron builds consist of a collection of components which when connected together produce a complete neural model. These components are defined in **Table 1**.

Component	Definition
Input component	The type of input the model takes. Currently there are conductance and current based inputs supported by sPyNNaker. It is possible to define other input types, but this is not described in this tutorial.
Synapse type component	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).
Threshold component	Determines the threshold of the membrane voltage which determines when the neuron spikes. Currently the only implementations are a static threshold and a stochastic Maass threshold.
Additional input component	Any additional input current that might be based on the membrane voltage or other parameters.
Neuron model component	Determines how the neuron state changes over time and the outputs the current membrane voltage of the neuron. Currently there are Izhikevich (IZK) and Leaky-Integrate-and-Fire (LIF) implementations supported by sPyNNaker.
Synapse dynamics component	Determines how plasticity works within the model. sPyNNaker implements a model for no plasticity (i.e. static dynamics), and two different STDP dynamics models. Only static dynamics are considered in this tutorial.

Table 1: Different supported components

Each build is stored within its own folder in the `c_models->src->neuron->builds` directory. Within each build is a *Makefile* containing the separate components required to build that specific form of neuron model.

If we look at the simple my_model_curr_exp's Makefile located in:

`c_models->src->neuron->builds->my_model_curr_exp->Makefile`

we can see the lines shown in **Code 1**.

```
1.  APP = $(notdir $(CURDIR))
2.  BUILD_DIR = build/
3.
4.  NEURON_MODEL = $(EXTRA_SRC_DIR)/neuron/models/my_neuron_model_impl.c
5.  NEURON_MODEL_H = $(EXTRA_SRC_DIR)/neuron/models/my_neuron_model_impl.h
6.  INPUT_TYPE_H = $(SOURCE_DIR)/neuron/input_types/input_type_current.h
7.  THRESHOLD_TYPE_H = $(SOURCE_DIR)/neuron/threshold_types/threshold_type_static.h
8.  SYNAPSE_TYPE_H = $(SOURCE_DIR)/neuron/synapse_types/synapse_types_exponential_impl.h
9.  SYNAPSE_DYNAMICS = $(SOURCE_DIR)/neuron/plasticity/synapse_dynamics_static_impl.c
10.
```

11. include ../Makefile.common

Code 1: my_model_curr_exp's Makefile

- Line 1 declares the name of the APP - here we are using the name of the current directory. (The aplx extension is added automatically.)
- Line 2 declares the directory in which the model will be built. This is where object files and other intermediate files are stored; the final aplx location is determined in Makefile.common (see later).
- Lines 4 and 5 are the files that make up the neuron model component (described in **Table 1**) used for this model build (both the .c and .h files are needed). Note that these are stated to be in the $(EXTRA_SRC_DIR) folder - this is declared to be the c_models/src folder within the archive within Makefile.common. The sPyNNaker standard source files are declared to be within $(SOURCE_DIR), and these are used by other components.
- Line 6 states the input type component (described in **Table 1**) for this model. Input types are implemented entirely in a header file.
- Line 7 states the threshold type component (described in **Table 1**) for this model. Threshold types are implemented entirely in a header file.
- Line 8 states the synapse type component (described in **Table 1**) for this model. Synapse types are implemented entirely in a header file.
- Line 9 states the synapse dynamics component (described in **Table 1)** for this model.
- Line 11 tells the make system to import the next level up Makefile so that it can detect where the rest of the code needed to be linked in can be found.

Other Makefile instances might also include TIMING_DEPENDENCE_H and WEIGHT_DEPENDENCE_H; these are used when the synapse dynamics includes plasticity. A tutorial on how to add new plasticity implementations is covered here.

To make a new neuron model build, you must either:

1. Create a copy of the example builds discussed above,
2. Modify the names and component listings,
3. Mody Line 1 of the Makefile located in `src->neuron->Makefile` so that it includes your new build.

Or:

1. Change the template's component listings directly.

Compiling a new model

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 beforehand, by running

```
make clean
```

Once the Makefile has been edited appropriately, you can build the binary by simply changing to the directory containing the Makefile and typing:

```
make
```

Assuming this builds with no errors, you should find the aplx files have been built and placed into the python_models8->model_binaries directory (as specified by Makefile.common) in the Python code.

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

```
make SPYNNAKER_DEBUG=DEBUG
```

This will enable the log_debug statements in the code, which print out information to the iobuf buffers on the SpiNNaker machine. By default, the tools won't extract the printed error messages. To enable this behaviour, you can add the following to your local .spynnaker.cfg file:

```
[Reports]
extract_iobuf=True
```

The "iobuf" messages will then be downloaded after the execution is complete. These are stored relative to your executing script in reports/DATE-TIME-ID/run_N/provenance_data/ for appropriate values of DATE, TIME, ID and N; each is a .txt file containing any output printed from each core used during the execution of your program.

C code file interfaces

This section goes through each interface for the different components of the neuron build and explains what each one does.

Neuron Models

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.

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.

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
```

- ```
  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_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 the value of the membrane voltage. Note that the input will always be presented as current - conductance input is converted to current input in the input type. Additionally, the input values are all positive, including the inhibitory input; thus if the total input current is being considered, the inhibitory input current should be subtracted from the excitatory input current.

- `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, and is taken before the state update is performed.

- `void neuron_model_has_spiked(neuron_pointer_t neuron);`
  This function is used to reset the parameters of the neuron after it has spiked. It is called only if the membrane voltage value returned from `neuron_model_state_update` is determined to be above the threshold determined by the threshold type.

- `void neuron_model_print_state_variables(restrict neuron_pointer_t neuron)`
  This function is only used when the neuron model is compiled in "debug" mode. It should use the "log_debug" function to print each of the state variables of the neuron that change during a run and that might be useful in debugging.

- `void neuron_model_print_parameters(restrict neuron_pointer_t neuron)`
  This function is only used when the neuron model is compiled in "debug" mode. It should use the "log_debug" function to print each of the parameters of the neuron that don't change during a run and that might be useful in debugging.

See `neuron_model_my_impl.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 mathematical and probabilistic 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.

Synapse types

The synapse type header file defines the `synapse_param_t` data structure that determines the parameters required for shaping the synaptic input. For example, this might be done to compensate for the valve behaviour of a synapse in biology (spike goes in, synapse opens, then closes slowly). The parameters for all the synaptic inputs for a single neuron need to be defined in this structure; for example, if there are different parameters for excitatory and inhibitory neurons, both of these parameters must be explicitly defined in this structure. The structure might also contain parameters for computing the initial value that will be added to the input buffer following a spike from a preceding neuron.

Note that the input will have already been delayed by the appropriate amount before it reaches this function, and that the input weights from several spikes may be combined into a single weight. Additionally, the input weights might be either current or conductance as determined by the input type. The synapse type should not perform any conversion of the weights.

The synapse type header file also defines the functions that make up the interface of the synapse type API. The synapse Type API requires the following interface functions to be implemented.

- ```
  static void synapse_types_shape_input(
        input_t *input_buffers, index_t neuron_index,
        synapse_param_t* parameters);
  ```
  Shapes the values (current or conductance) in the input buffers for the synapses of a given neuron. The input buffers for all neurons and synapse types are given here, and the following function can be used to obtain the index of the appropriate input buffer given the indices of the neuron and of the
  ```

synapse (e.g. if there is an excitatory and inhibitory synapse per neuron, the indices might be 0 and 1 respectively):

```
index_t synapse_types_get_input_buffer_index(synapse_index, neuron_index)
```

- ```
  static void synapse_types_add_neuron_input(
        input_t *input_buffers, index_t synapse_type_index, index_t neuron_index,
        synapse_param_t* parameters, input_t input)
  ```
  Adds a synaptic weight input to the input buffer for a given synapse of a given neuron after a spike has been received (and appropriately delayed). This allows the weight to be scaled as required before it is added to the buffer.

- ```
  static input_t synapse_types_get_excitatory_input(
        input_t *input_buffers, index_t neuron_index)
  ```
  Returns the total combined excitatory input from the buffers available for a given neuron id. Note that if several synapses are excitatory, this function should add up the input values (or perform an otherwise appropriate function) to return the total excitatory input value.

- ```
  static input_t synapse_types_get_inhibitory_input(
        input_t *input_buffers, index_t neuron_index)
  ```
  Extracts the total combined inhibitory input from the buffers available for a given neuron id. Note that if several synapses are inhibitory, this function should add up the input values (or perform an otherwise appropriate function) to return the total inhibitory input value. Note also that the value should be a positive number; subtraction is performed in the neuron model as required.

- `static const char *synapse_types_get_type_char(index_t synapse_type_index)`
  Returns a human readable character for the type of synapse. Examples would be X = excitatory types and I = inhibitory types.

- ```
  static void synapse_types_print_input(
        input_t input_buffers, index_t neuron_index)
  ```
  Prints the input for a neuron id given the available inputs. This is currently only executed when the models are in debug mode.

- `static void synapse_types_print_parameters(synapse_param_t *parameters)`
  Prints the static parameters of the synapse type. This is currently only executed when the models are in debug mode.

See `synapse_types_my_impl.h` for an example of an implementation of a synapse type.

Threshold types

The threshold type header file defines the `threshold_type_t` data structure that declares the parameters required for the threshold type. This might commonly include the actual threshold value amongst other parameters. The header also defines the functions that make up the interface of the threshold type API. The threshold Type API requires the following interface functions to be implemented.

- ```
  static bool threshold_type_is_above_threshold(
        state_t value, threshold_type_pointer_t threshold_type)
  ```
  Determines if the threshold has been reached; if the neuron is to spike, given the value of the state variable, true is returned, otherwise false is returned.

See `my_threshold_type.h` for an example of an implementation of a threshold type.
  ```

Additional inputs

The additional input header file defines the `additional_input_t` data structure, which declares the parameters required for the additional input. The header also defines the functions that make up the interface of the additional input type API. The additional input Type API requires the following interface functions to be implemented:

- `static input_t additional_input_get_input_value_as_current(`
  `additional_input_pointer_t additional_input, state_t membrane_voltage)`
  Gets the value of current provided by the additional input. This may or may not be dependent on the membrane voltage.

- `static void additional_input_has_spiked(`
  `additional_input_pointer_t additional_input)`
  Notifies the additional input type that the neuron has spiked.

See `my_additional_input.h` for an example of an implementation of an additional input type. You will also need to add the line (e.g.)

```
ADDITIONAL_INPUT_H = $(EXTRA_SRC_DIR)/neuron/additional_inputs/my_additional_input.h
```

to the relevant Makefile when you wish to use these additional parameters in a model.

Python Model Builds

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 onto the machine, along with the 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 constructing an individual component for:

1. Neuron model,
2. Input type,
3. Synapse type,
4. Threshold type,
5. Additional input.

These 5 components are then handed over to the main interface object that every neuron model has to extend.

If we look at `my_model_curr_exp.py` in the `python_models8->neuron->builds` directory, we will see the code shown in **Code 4** where the my_model_curr_exp builds its components and hands them over to the main sPyNNaker interface. The breakdown is as follows:

1. On Lines 97 and 98 the neuron model component is created.
2. On Lines 102 and 103 the synapse type component is created.
3. On Line 107 the input type component is created.
4. On Line 111 the threshold type component is created.
5. Line 115 shows that this model does not contain any additional input components.

6. Lines 119 to 135 show the handing over of these separate components to the sPyNNaker main system which will handle all the python support. Note that the binary must match the name of the aplx file generated by the C code.

```
95 .    # TODO: create your neuron model class (change if required)
96.     # create your neuron model class
97.     neuron_model = MyNeuronModel(
98.       n_neurons, i_offset, my_parameter)
99.
100.    # TODO: create your synapse type model class (change if required)
101.    # create your synapse type model
102.    synapse_type = SynapseTypeExponential(
103.      n_neurons, tau_syn_E, tau_syn_I, isyn_exc, isyn_inh)
104.
105.    # TODO: create your input type model class (change if required)
106.    # create your input type model
107.    input_type = InputTypeCurrent()
108.
109.    # TODO: create your threshold type model class (change if required)
110.    # create your threshold type model
111.    threshold_type = ThresholdTypeStatic(n_neurons, v_thresh)
112.
113.    # TODO: create your own additional inputs (change if required).
114.    # create your own additional inputs
115.    additional_input = None
116.
117.    # instantiate the sPyNNaker system by initializing
118.    #  the AbstractPopulationVertex
119.    AbstractPopulationVertex.__init__(
120.
121.       # standard inputs, do not need to change.
122.       self, n_neurons=n_neurons, label=label,
123.       spikes_per_second=spikes_per_second,
124.       ring_buffer_sigma=ring_buffer_sigma,
125.       incoming_spike_buffer_size=incoming_spike_buffer_size,
126.
127.       # TODO: Ensure the correct class is used below
128.       max_atoms_per_core=(
129.             MyModelCurrExpBase._model_based_max_atoms_per_core),
130.
131.       # These are the various model types
132.       neuron_model=neuron_model, input_type=input_type,
133.       synapse_type=synapse_type, threshold_type=threshold_type,
134.       additional_input=additional_input,
135.
136.       # TODO: Give the model a name (shown in reports)
137.       model_name="MyModelCurrExpBase",
138.
139.       # TODO: Set this to the matching binary name
120.       binary="my_model_curr_exp.aplx")
```

Code 4: Subsection of the my_model_curr_exp.py class

Take care to note that the same components are used in the Python as are used in the C code's *Makefile*. This means for every new component you add for a neuron build in C which is not originally supported by the sPyNNaker tools, you need to build a corresponding Python component file.

In the new_template folder there are a set of template files within the template directory for each python component. These are located under `python_models8->neuron`. These detail the parts of the class that need to be changed for your model.

PyNN 0.8 only

If you are using PyNN 0.8, then in addition to the above file (e.g. my_model_curr_exp) in `python_models8->neuron->builds`, you will also need to add a data holder for the corresponding build. For an example, see my_model_curr_exp_data_holder in `python_models8->model_data_holders.`

Python __init__.py files

Most of the __init__.py files in the template do not contain any code. The one within `python_models8` 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
from python_models.neuron.builds.my_model_curr_exp import MyModelCurrExp
pop = p.Population(1, MyModelCurrExp, {})
```

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

Task 1: Simple Neuron Model [Easy]

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

1. Change the `my_neuron_model_impl.c` and `.h` templates by adding two parameters, one representing a decay and one representing a rest voltage. The parameters should be `REAL` values.
2. 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
```

3. Recompile the binary.
4. Update the python code model to accept the new decay and rest voltage parameters, ensuring that they match the order of the C code (use `DataType.S1615`). Add getters and setters for the values and update the number of neuron parameters.
5. Update the python code builds to accept the new parameters with default values of 0.1 for decay and -65.0 for the rest voltage. Don't forget to update the DataHolders as well!

Run the example script and see what happens.

Task 2: Conductance-based Model [Moderate]

This task will build a conductance-based model.

1. Make a copy of the C build folder for my_model_curr_exp to my_model_cond_exp.
2. Change the relevant Makefiles to use the conductance input type and ensure that the binary name is different from the current based model.
3. Build the binary.
4. Copy the python model `my_model_curr_exp.py` to `my_model_cond_exp.py` and update the code to use the conductance input type, including adding the new required parameters for conductance, and the binary name and model name.
5. Copy `my_model_curr_exp_data_holder.py` to `my_model_cond_exp_data_holder.py` and make sure the parameters are specified here too.
6. Update the example script to use the new model, adjusting the weights to be conductances (usually much smaller values e.g. 0.1 should be enough)

Run the example script and see what happens.

Task 3: Stochastic Threshold Model [Hard]

This task will create a new threshold model for stochastic thresholds.

1. Update the template threshold type `my_threshold_type.c` and `.h`, removing the parameter `my_threshold_parameter`, and adding a parameter representing the probability of the neuron firing if it is over the threshold value. This will be a `uint32_t` value in C (see later for details).

2. Add another 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 `random.h` (from the spinn_common library - as this should have been installed, you can use `#include <random.h>`).

3. Update the threshold calculation so that when the membrane voltage is over the threshold voltage, the RNG is called with the seed (`mars_kiss64_seed(mars_kiss64_seed_t seed)`).

4. Update the threshold calculation to only result in a spike if the value returned from the RNG is less than than the probability value.

5. Rebuild the C code.

6. Update the `my_threshold_type.py` python code to include the new parameters, and to generate the random seed. The probability 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. The seed can be generated using a NumpyRNG (from pyNN.random), which can be provided to the model as a parameter. Once generated, the seed should be validated using:

   ```
   spynnaker.pyNN.utilities.utility_calls.validate_mars_kiss_64_seed(seed)
   ```

   where `seed` is an array of 4 integer values. Note that `seed` will be updated in place.

7. Update the `my_model_curr_exp_my_threshold_type.py` build to include the new parameters and pass them in to the threshold type. Make rng an optional parameter, which if not set uses a new NumpyRNG.
8. Update the example script to decrease the threshold value to ensure that the model fires.

Run the example script and see how the number of spikes, and where these spikes occur, differs for different values of the spike probability.

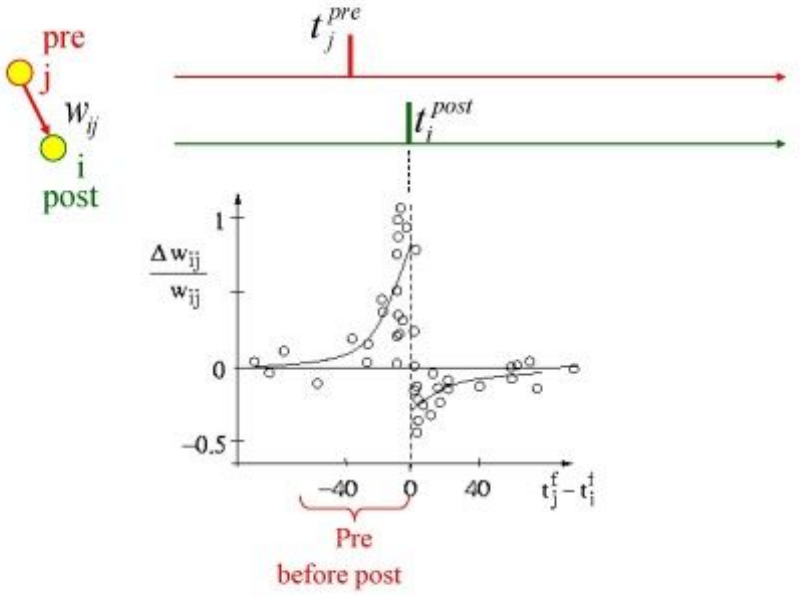
Spike-Time-Dependent Plasticity on SpiNNaker

STDP is a form of learning that depends upon the timing between the spikes of two neurons connected by a synapse. It is believed to be the basis of learning and information storage in the human brain.

In the case where a presynaptic spike is followed closely by a postsynaptic spike, then it is presumed that the presynaptic neuron caused the spike in the postsynaptic neuron, and so the weight of the synapse between the neurons is increased. This is known as potentiation.

If a postsynaptic spike is emitted shortly before a presynaptic spike is emitted, then the presynaptic spike cannot have caused the postsynaptic spike, and so the weight of the synapse between the neurons is reduced. This is known as depression.

The size of the weight change depends on the relative timing of the presynaptic and postsynaptic spikes; in general, the change in weight drops off exponentially as the time between the spikes gets larger, as shown in the following figure [Sjöström and Gerstner (2010), Scholarpedia]. However, different experiments have highlighted different behaviours depending on the conditions (e.g. [Graupner and Brunel (2012), PNAS]). Other authors have also suggested a correlation between triplets and quadruplets of presynaptic and postsynaptic spikes to trigger synaptic potentiation or depression.



STDP in PyNN

The steps for creating a network using STDP are much the same as previously described, with the main difference being that some of the projections use a `STDPMechanism` to describe the plasticity. Here is an example of the creation of a projection with STDP:

```
timing_rule = sim.SpikePairRule(tau_plus=20.0, tau_minus=20.0,
                                 A_plus=0.5, A_minus=0.5)
weight_rule = sim.AdditiveWeightDependence(w_max=5.0, w_min=0.0)

stdp_model = sim.STDPMechanism(timing_dependence=timing_rule,
                               weight_dependence=weight_rule,
                               weight=0.0, delay=5.0)

stdp_projection = sim.Projection(pre_pop, post_pop, sim.OneToOneConnector(),
                                 synapse_type=stdp_model)
```

In this example, firstly the timing rule is created. In this case, it is a *SpikePairRule*, which means that the

relative timing of the spikes that will be used to update the weights will be based on pairs of presynaptic and postsynaptic spikes. This rule has four parameters. The parameters *tau_plus* and *tau_minus* describe the respective exponential decay of the size of the weight update with the time between presynaptic and postsynaptic spikes. Note that the decay can be different for potentiation (defined by *tau_plus*) and depression (defined by *tau_minus*). The parameters *A_plus* and *A_minus* which define the maximum weight to respectively add during potentiation or subtract during depression.

The next thing defined is the weight update rule. In this case it is a *AdditiveWeightDependence*, which means that the weight will be updated by simply adding to the current weight. This rule requires the parameters *w_max* and *w_min*, which define the maximum and minimum weight of the synapse respectively, Note that the actual amount added or subtracted will depend additionally on the timing of the spikes, as determined by the timing rule.

In addition, there is also a *MultiplicativeWeightDependence* supported, which means that the weight change depends on the difference between the current weight and w_max for potentiation, and w_min for depression. The value of *A_plus* and *A_minus* are then respectively multiplied by this difference to give the maximum weight change; again the actual value depends on the timing rule and the time between the spikes.

The timing and weight rules are combined with *weight* and *delay* into a single *STDPMechanism* object which describes the overall desired mechanism. Note that the projection still requires the specification of a *connector.* This connector is still used to describe the overall connectivity between the neurons of the pre- and post-populations. It is preferable that the initial weights fall between *w_min* and *w_max*; it is not an error if they do not, but when the first update is performed, the weight will be changed to fall within this range.

Note that on SpiNNaker, although multiple projections to the same target population can be specified with STDP, the restrictions on the current software are that all those projections must use the same rules with the exact same parameters. This is due to the restrictions of the local memory available on each core, reducing the amount of data that can be held for the parameters.

Note: In the implementation of STDP on SpiNNaker, the plasticity mechanism is only activated when the second presynaptic spike is received at the postsynaptic neuron. Thus at least two presynaptic spikes are required for the mechanism to be activated.

Getting Synaptic Data

The weights and delays assigned to a projection can be retrieved using the Projection's *get* method, specifying the data items to get, including 'weight', 'delay' and the parameters of the STDP Mechanism, and the format they are retrieved using. The data formats supported are 'list' format, where the return value consists of a list of tuples of the selected values; and 'array' where each value is returned in a two-dimensional matrix indexed by the source neurons in the pre-population, and the target neurons in the post-populations. In the 'list' result, each tuple additionally contains the source and target neuron ids as the 0th and 1st values in the tuple. In the 'array' result, missing connections are represented as 'NaN' (not a number) and positions where there are multiple connections have their values summed.

Note that on SpiNNaker, it is possible to retrieve the projection data before calling the PyNN run function, but that this data cannot be examined until after run has been called. This is because the individual connectivity data is not generated until the run function is called.

Task 1: Explore Existing STDP Implementations - Repeated from Lab 02

Task 1.1: STDP Network [Easy]

This task will create a simple network involving STDP learning rules.

Write a network with a 1.0ms time step consisting of two single-neuron populations connected with an STDP synapse using a spike pair rule and additive weight dependency, and initial weights of 0. Stimulate each of the neurons with a spike source array with times of your choice, with the times for stimulating the first neuron being slightly before the times stimulating the second neuron (e.g. 2ms or more), ensuring the times are far enough apart not to cause depression (compare the spacing in time with the tau_plus and tau_minus settings); note that a weight of 5.0 should be enough to force an IF_curr_exp neuron to fire with the default parameters. Add a few extra times at the end of the run for stimulating the first neuron. Run the network for a number of milliseconds and extract the spike times of the neurons and the weights.

You should be able to see that the weights have changed from the starting values, and that by the end of the simulation, the second neuron should spike shortly after the first neuron.

Task 1.2: STDP Parameters [Easy]

Alter the parameters of the STDP connection, and the relative timing of the spikes. Try starting with a large initial weight and see if you can get the weight to reduce using the relative timing of the spikes.

Task 1.3: STDP Curve [Hard]

This task will attempt to plot an STDP curve, showing how the weight change varies with timing between spikes.

1. Set up the simulation to use a 1ms time step.
2. Create a population of 100 presynaptic neurons.
3. Create a spike source array population of 100 sources connected to the presynaptic population. Set the spikes in the arrays so that each spikes twice 200ms apart, and that the first spike for each is 1ms after the first spike of the last e.g. [[0, 200], [1, 201], …] (hint: you can do this with a list comprehension).
4. Create a population of 100 postsynaptic neurons.
5. Create a spike source array connected to the postsynaptic neurons all spiking at 50ms.
6. Connect the presynaptic population to the postsynaptic population with an STDP projection with an initial weight of 0.5 and a maximum of 1 and minimum of 0.
7. Record the presynaptic and postsynaptic populations.
8. Run the simulation for long enough for all spikes to occur, and get the weights from the STDP projection.
9. Draw a graph of the weight changes from the initial weight value against the difference in presynaptic and postsynaptic neurons (hint: the presynaptic neurons should spike twice but the postsynaptic should only spike once; you are looking for the first spike from each presynaptic neuron).

Task 2: Create New STDP Rule [Medium]

This task will create a custom STDP mechanism based on existing spike pair timing and additive weight dependence rules. In the following task, these rules can then be edited to customise learning.

Task 2.1 Create Custom Timing Dependence Rule

1. First address the python code
    a. Find timing_dependence_spike_pair.py and copy to a new name in the same directory e.g. timing_dependence_spike_pair_custom.py
    b. Update the class name to reflect the new name: e.g. TimingDependenceSpikePairCustom
    c. Edit the `vertex_executable_suffix(self)` function to return a suffix associated with the new timing rule - e.g. "pair_custom"
    d. Edit the `is_same_as(self, timing_dependence)` function to compare to the new class name:

```
if not isinstance(timing_dependence, TimingDependenceSpikePairCustom):
```

    e. Edit the adjacent __init__ file to import the new timing rule; e.g.:

```
from .timing_dependence_spike_pair_custom import
TimingDependenceSpikePairCustom

__all__ = ["AbstractTimingDependence", "TimingDependenceSpikePair",
        "TimingDependencePfisterSpikeTriplet", "TimingDependenceRecurrent",
        "TimingDependenceSpikeNearestPair", "TimingDependenceVogels2011",
        "TimingDependenceSpikePairCustom"]
```

    f. Now add similar import statements to sPyNNaker8 repository. Find spynnaker8/__init__.py and add:

```
from spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence \
    .timing_dependence_spike_pair_custom \
    import TimingDependenceSpikePairCustom as SpikePairRuleCustom

__all__ = [...,
      # plastic stuff
      'STDPMechanism', 'AdditiveWeightDependence', 'SpikePairRule',
      'MultiplicativeWeightDependence','SpikePairRuleCustom', ...]
```

2. Now update the C code
    a. Find timing_pair_impl.c and timing_pair_impl.h and copy to new locations in the same directory: e.g. timing_pair_custom_impl.c and timing_pair_custom_impl.h
    b. In the C builds directory, create a new directory for your custom build: here the IF_curr_exp neuron model will be used, so the name should start with this. When searching for a binary, the toolchain will add the suffix "_stdp_mad_" to the neuron model, together with the suffix from your timing and weight dependence classes. Therefore, for the current example using the existing weight dependence rule, the build directory should be named: **IF_curr_exp_stdp_mad_pair_custom_additive**
    c. Copy the Makefile from the adjacent IF_curr_exp_stdp_mad_pair_additive directory, and edit the paths of the TIMING_DEPENDENCE and TIMING_DEPENDENCE_H variables to point to the custom .c and .h files created in step a.
    d. Now edit Makefile in the 'Neuron' directory (two levels up from the build directory), adding your new build directory to the end of the list defined by the variable MODELS

e. You should now be able to navigate back to the build directory and type ‘make’ to build the new binary - if the build was successful the last few logging lines will report the name of the newly built aplx file (IF_curr_exp_stdp_mad_pair_custom_additive.aplx)

3. Run test example
    a. In one of the PyNN scripts used in section 1, change the STDPMechanism to use the new timing rule: timing_dependence=p.**SpikePairRuleCustom**(...
    b. Re-run the script to test execution

Task 2.2 Create Custom Weight Dependence Rule

1. First address the Python code
    a. Find weight_dependence_additive.py and copy to a new name in the same directory e.g. weight_dependence_additive_custom.py
    b. Update the class name to reflect the new name: e.g. WeightDependenceAdditiveCustom
    c. Edit the `vertex_executable_suffix(self)` function to return a suffix associated with the new weight dependence rule: e.g. `“additive_custom”`
    d. Edit the `is_same_as(self, timing_dependence)` function to compare to the new class name:

```
if not isinstance(weight_dependence, WeightDependenceAdditiveCustom):
```

    e. Edit the adjacent __init__ file to import the new timing rule; e.g.

```
from .weight_dependence_additive_custom import WeightDependenceAdditiveCustom

__all__ = ["AbstractHasAPlusAMinus", "AbstractWeightDependence",
           "WeightDependenceAdditive", "WeightDependenceMultiplicative",
           "WeightDependenceAdditiveTriplet",
"WeightDependenceAdditiveCustom"]
```

    f. Now add similar import statements to the sPyNNaker8 repository. Find spynnaker8/__init__.py and add:

```
from spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence \
    .timing_dependence_spike_pair_custom \
    import WeightDependenceAdditiveCustom as
AdditiveWeightDependenceCustom

__all__ = [...,
      # plastic stuff
      'STDPMechanism', 'AdditiveWeightDependence',
      'AdditiveWeightDependenceCustom', 'MultiplicativeWeightDependence',
      'SpikePairRule', 'SpikePairRuleCustom', …]
```

2. Now update the C Code

a. Find weight_additive_one_term_impl.h and weight_additive_one_term_impl.c and copy to new names in the same directory: e.g. weight_additive_one_term_custom_impl.h and weight_additive_one_term_additive_custom_impl.c
b. In the C builds directory, create a new directory to house the build for the new application including both the custom timing and weight dependence rules. Hint, take the name of the build directory from Task 2.1, and adjust the weight suffix to the custom rule: e.g. IF_curr_exp_stdp_mad_pair_custom_weight_custom
c. Copy the Makefile from Task 2.1.2-c, and update the paths for WEIGHT_DEPENDENCE and WEIGHT_DEPENDENCE_H, e.g. to point to weight_additive_one_term_custom_impl.h and weight_additive_one_term_custom_impl.c respectively
d. As in Task 2.1.2-d, edit the Makefile in the neuron directory (two levels up from current Makefile) to include the new neuron model including both custom timing and weight dependence rules: IF_curr_exp_stdp_mad_pair_custom_weight_custom
e. Run 'make' to build the application, and check the aplx name matches that defined by the Python code

3. Run test example
    a. In the PyNN scripts used for task 2.1.3, change the STDPMechanism to also use the new weight rule: timing_dependence=p.**WeightDependenceAdditiveCustom**(...
    b. Re-run the script and check execution

Task 3: Customise Timing and Weight Dependence Rules to Modify Learning [As Hard As You Make IT]

Implement your own ideas for timing and weight dependence rules by editing the new custom configurations, or alternatively follow the steps below to add an extra variable to the timing rule.

1. Adding a new variable to the timing dependence rule
    a. Edit the Python code to ensure the new variable is accessible through PyNN, and that the toolchain will write the new parameter ready for loading (hint: ensure the correct size is specified, too)
    b. Edit the C source to read the new parameter at initialisation
    c. Use the parameter in one of the spike handling functions in the timing rule header
    d. Re-compile the binary
    e. Write test PyNN script to check operation