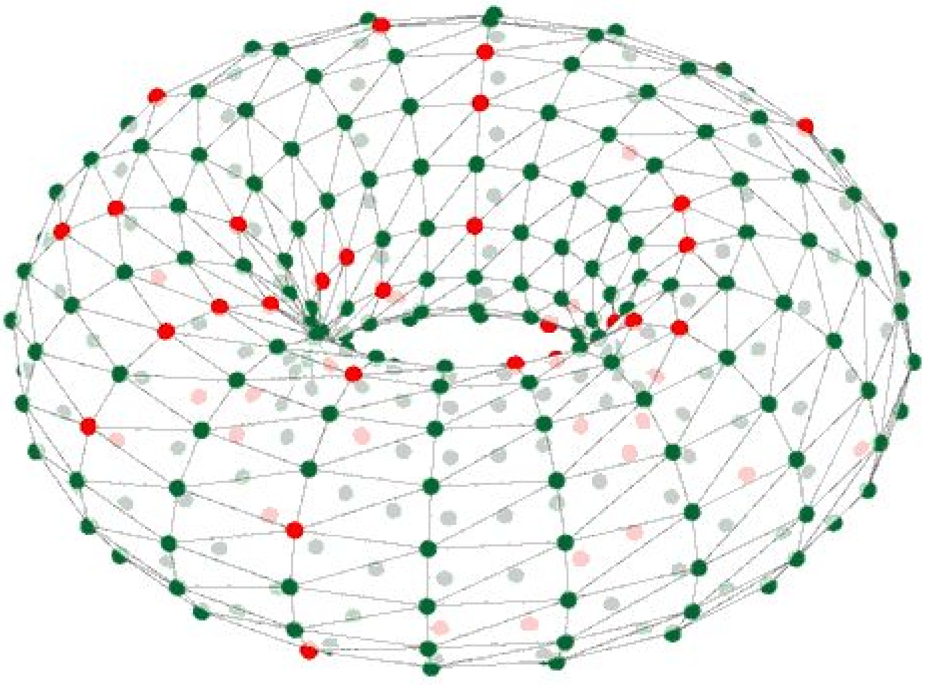
6th SpiNNaker Workshop

Lab Manuals

September
5th- 9th 2016



Manchester, UK

Intro Lab

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

Installation

The software installation instructions can be found here:
https://spinnakermanchester.github.io/latest/spynnaker_install.html
https://spinnakermanchester.github.io/latest/gfe_install.html

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. Conductive Material with Applied Heat
3. Sudoku Game Through Neural Network
4. Graphic Ray Tracer of an Environment
5. Simple Learning Network

Neural Network Synfire Chain

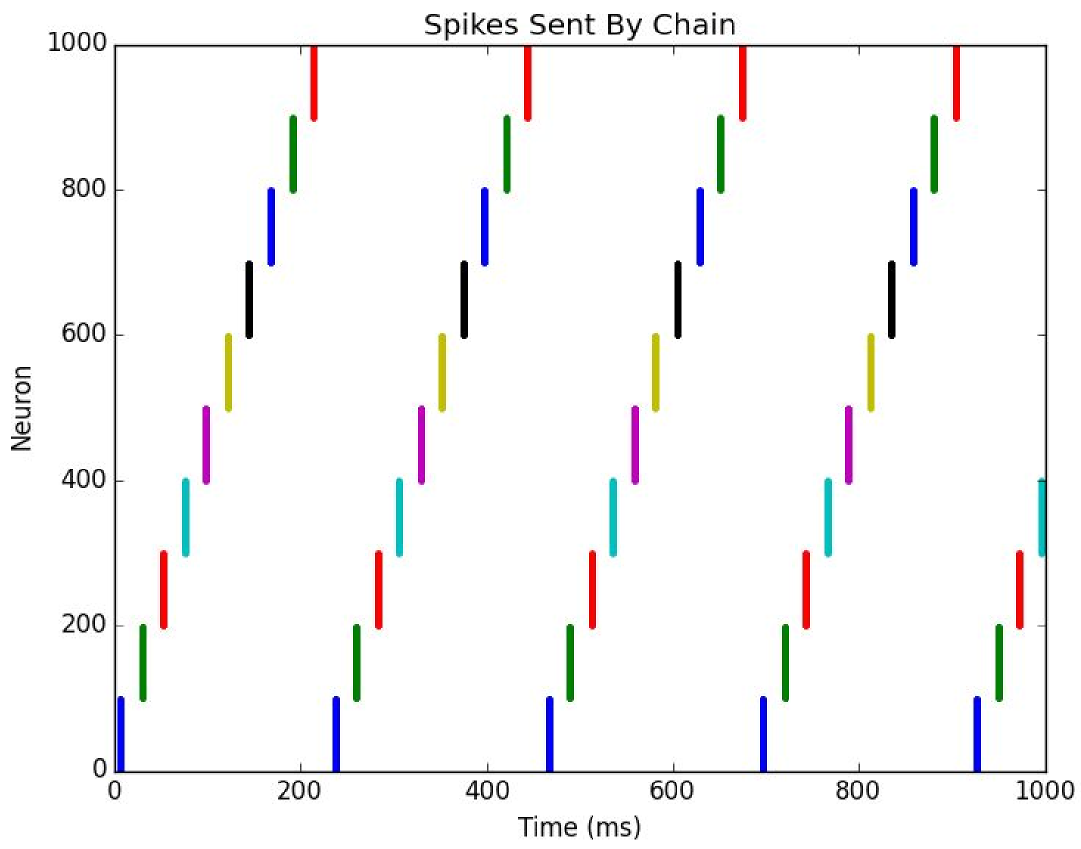


Figure 1: The output from a simple Synfire chain.

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.

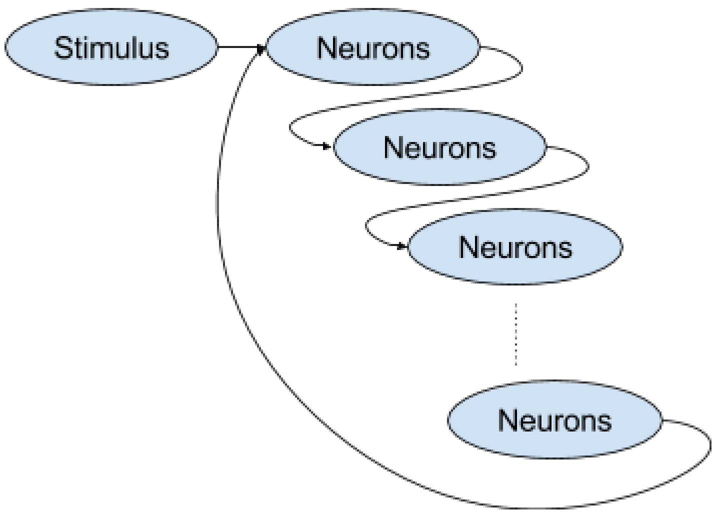


Figure 2: The Synfire Chain of Populations

Conductive Material with Applied Heat

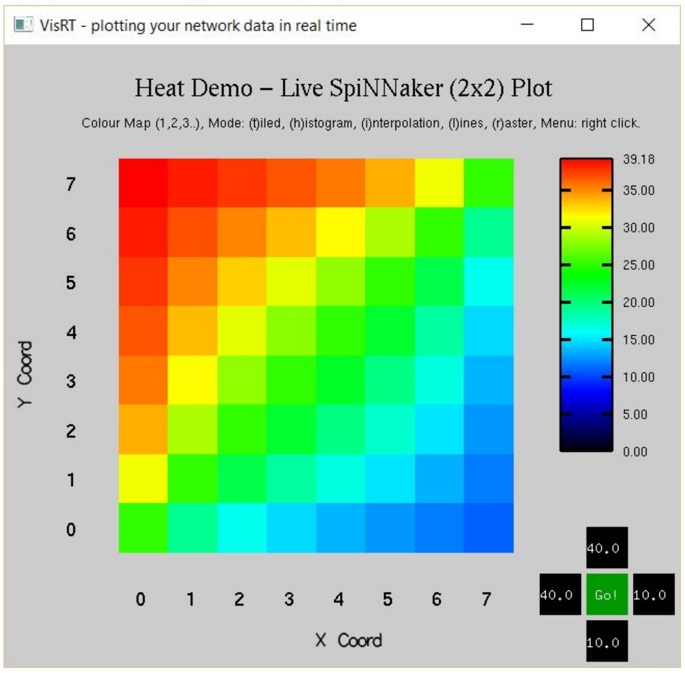

VisRT - plotting your network data in real time
Heat Demo – Live SpiNNaker (2x2) Plot
Colour Map (1,2,3..), Mode: (t)iled, (h)istogram, (i)nterpolation, (l)ines, (r)aster, Menu: right click.
Y Coord
X Coord
39.18
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
40.0
40.0
Go!
10.0
10.0

Figure 3: The output from the conductive material simulation

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

```
cd heat_demo
python heat_demo.py
```

A visualiser should appear here, as shown in Figure 3. You can press “9” to randomize the heat applied at each edge of the simulation, or press select a black square and press “+” to increase the temperature, or “-” to decrease it, followed by “g” to update it.

This example shows a piece of conductive material (e.g. a metal sheet) which is represented by a collections of cells which represent atoms of the material. Temperature is transferred between the atoms of the material in the simulation by sending packets over the SpiNNaker network. Figure 4 shows this application in graphical form.

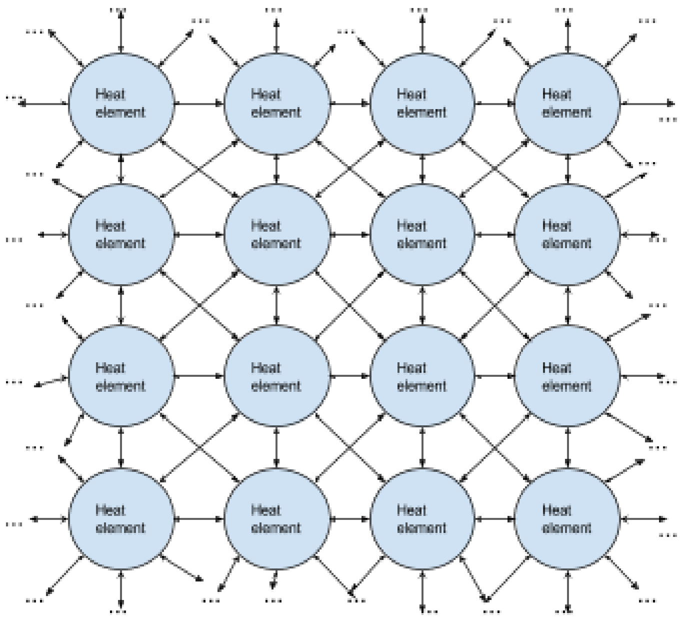
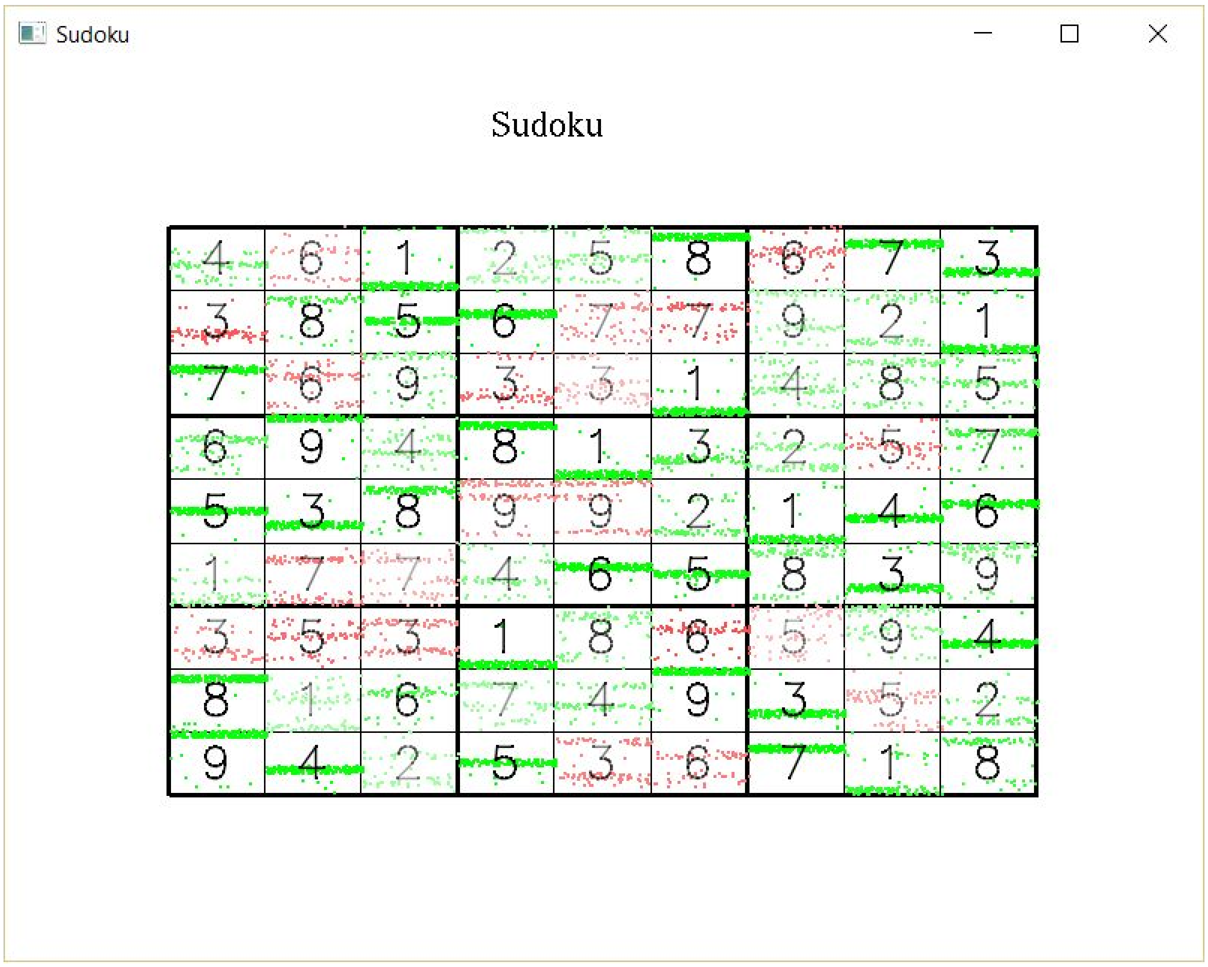


Figure 4: The conductive material application in graphical form

Sudoku Game Through Neural Network



4	6	1	2	5	8	6	7	3
3	8	5	6	7	7	9	2	1
7	6	9	3	3	1	4	8	5
6	9	4	8	1	3	2	5	7
5	3	8	9	9	2	1	4	6
1	7	7	4	6	5	8	3	9
3	5	3	1	8	6	5	9	4
8	1	6	7	4	9	3	5	2
9	4	2	5	3	6	7	1	8

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.

Graphic Ray Tracer of an Environment

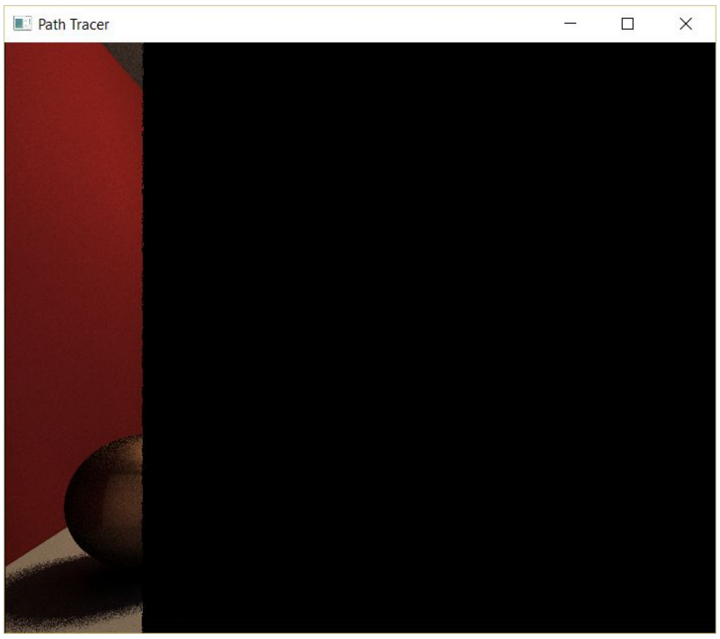


Figure 6: The output from the graphical ray tracer application

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

```
cd ray_trace
python ray_trace.py
```

A visualiser will pop up, which is shown in Figure 6.

This example shows a ray tracing application on SpiNNaker. This has been designed so that to operate in parallel; the more cores in use, the faster it completes. Note that it is still quite slow, and you may have to click on the window to force it to update.

Simple Learning Network

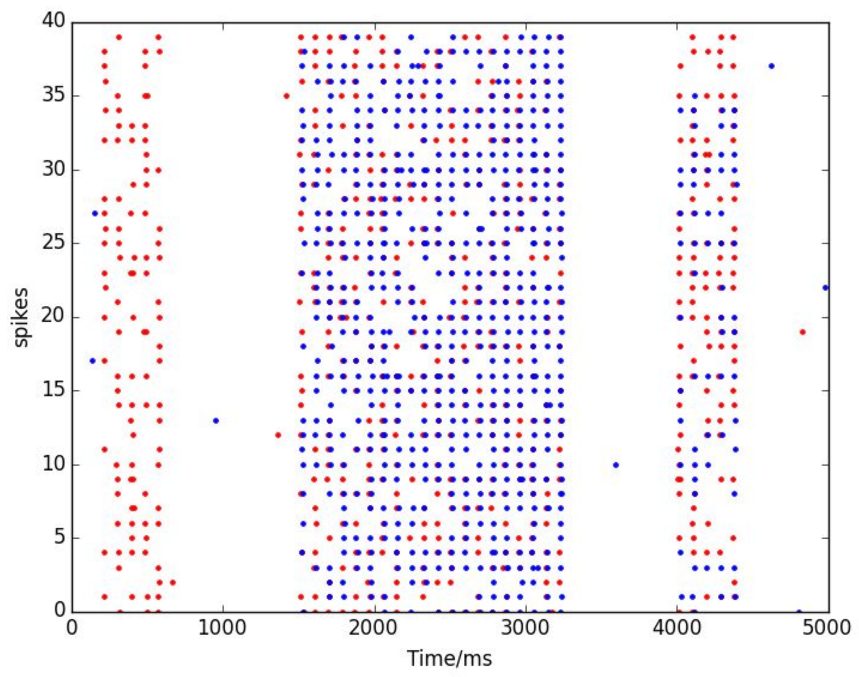


Figure 7: The output from the learning application

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

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

A plot like the above should appear.

This example shows the spike outputs from two populations of neurons. At the start, only one of the populations spikes regularly. In the middle, some learning is done, and at the end, both populations spike regularly.

Running PyNN Simulations on SpiNNaker

Introduction

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

Installation

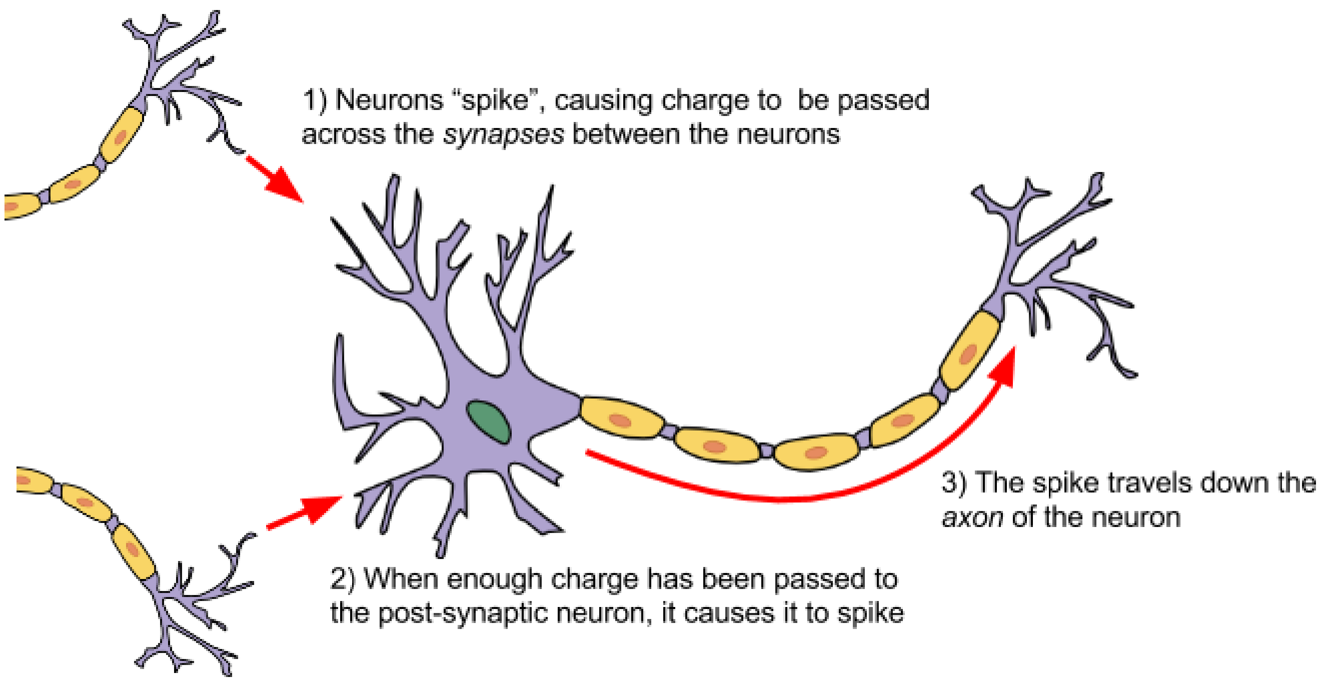
The PyNN toolchain for SpiNNaker (sPyNNaker), can be installed by following the instructions available from here:

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

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

Spiking Neural Networks

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


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

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

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

The PyNN Neural Network Description Language

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

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

An example of this is as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Using PyNN with SpiNNaker

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

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

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

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

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

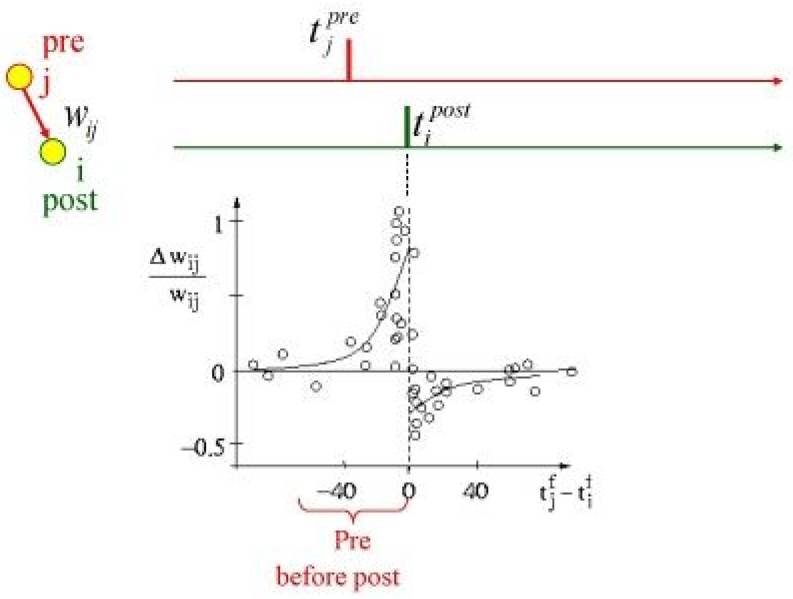
Spike-Time-Dependent Plasticity

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 are given an *synapse_dynamics* argument to describe the plasticity. Here is an example of the creation of a projection with STDP:

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

stdp_model = p.STDPMechanism(
    timing_dependence=timing_rule, weight_dependence=weight_rule)

stdp_projection = p.Projection(
    pre_pop, post_pop, p.OneToOneConnector(weights=0.0, delays=5.0),
    synapse_dynamics=p.SynapseDynamics(slow=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 two required parameters, *tau_plus* and *tau_minus* which 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*). This rule also accepts a parameter of *nearest* (by default this is *False*). If set to true, only the nearest pairs of spikes are considered when looking at weight updates.

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, and the parameters *A_plus* and *A_minus* which define the maximum weight to respectively add during potentiation or subtract during depression. 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 into a single *STDPMechanism* object which describes the overall desired mechanism. This is finally specified as the *slow* argument of a *SynapseDynamics* object, and then added to the *Projection* using the *synapse_dynamics* argument. Note that the projection still requires the specification of a *Connector* which includes *weights* and *delays*. This connector is still used to describe the overall connectivity between the neurons of the pre- and post-populations, as well as the delay values which are unchanged by the STDP rules, and the initial weight values. 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: 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.

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, but you should note that the array returned from get_v() contains a list of [neuron_id, time, voltage] and so you will need to separate out the voltages of the individual neurons).

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

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

Task 2.1: Synfire Chain [Moderate]

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

1. Setup the simulation to use 1ms timesteps.
2. Create an input population of 1 source spiking at 0.0ms.
3. Create a synfire population with 100 neurons.
4. With a FromListConnector, connect the input population to the first neuron of the synfire population, with a weight of 5nA and a delay of 1ms.
5. Using another FromListConnector, connect each neuron in the synfire population to the next neuron, with a weight of 5nA and a delay of 5ms.

6. Connect the last neuron in the synfire population to the first.
7. Record the spikes produced from the synfire populations.
8. Run the simulation for 2 seconds, and then retrieve and plot the spikes from the synfire population.

Task 2.2: Random Values [Easy]

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

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

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

1. Set up the simulation to use 0.1ms timesteps.
2. Choose the number of neurons to be simulated in the network.
3. Create an excitatory population with 80% of the neurons and an inhibitory population with 20% of the neurons.
4. Create excitatory poisson stimulation population with 80% of the neurons and an inhibitory poisson stimulation population with 20% of the neurons, both with a rate of 1000Hz.
5. Create a one-to-one excitatory connection from the excitatory poisson stimulation population to the excitatory population with a weight of 0.1nA and a delay of 1.0ms.
6. Create a similar excitatory connection from the inhibitory poisson stimulation population to the inhibitory population.
7. Create an excitatory connection from the excitatory population to the inhibitory population with a fixed probability of connection of 0.1, and using a normal distribution of weights with a mean of 0.1 and standard deviation of 0.1 (remember to add a boundary to make the weights positive) and a normal distribution of delays with a mean of 1.5 and standard deviation of 0.75 (remember to add a boundary to keep the delays within the allowed range on SpiNNaker).
8. Create a similar connection between the excitatory population and itself.
9. Create an inhibitory connection from the inhibitory population to the excitatory population with a fixed probability of connection of 0.1, and using a normal distribution of weights with a mean of -0.4 and standard deviation of 0.1 (remember to add a boundary to make the weights negative) and a normal distribution of delays with a mean of 0.75 and standard deviation of 0.375 (remember to add a boundary to keep the delays within the allowed range on SpiNNaker).
10. Create a similar connection between the inhibitory population and itself.
11. Initialize the membrane voltages of the excitatory and inhibitory populations to a uniform random number between -65.0 and -55.0.
12. Record the spikes from the excitatory population.
13. Run the simulation for 1 or more seconds.
14. Retrieve and plot the spikes.

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

Task 3.2: Network Behavior [Moderate]

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

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).

Graph Front End Lab Manual

The task is to build a python program that uses the Graph Front End (GFE).

Summary

This python script should use the GFE to instantiate a working example of the Conway's Game of Life[1]. Conway's Game of Life consist of a 2D fabric of cells, each of which has 2 states. These states are either Alive or Dead, and switching between these two states is decided upon the states of their 8 neighbouring cells.

The rules which dictate the changing of state are as follows:

1. Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if caused by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The application needs to be able to handle different initial states of the cells within the 2d fabric, but the default states for tasks 1 to 8 should look like Figure 1 and for task 9 to 13 should look like Figure 2.

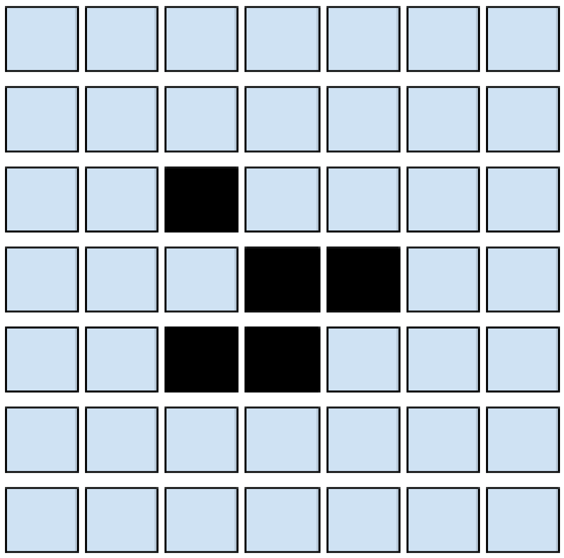


Figure 1: Basic initial state (7 by 7 grid)

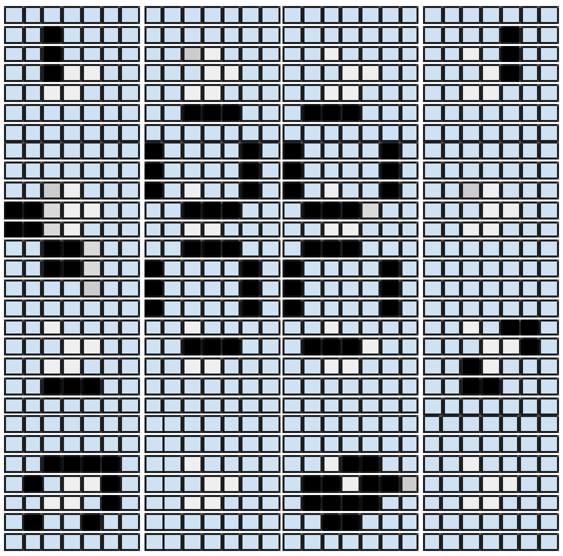


Figure 2: Advanced State (28 by 28 grid)

[1] https://en.wikipedia.org/wiki/Conway's_Game_of_Life

Step 1 (Easy)

Create a python class which will represent a Conway Cell.

Step 2 (Easy)

Build a python script which builds a collection of Conway cells (as vertices) in the GFE machine graph to form a 7x7 grid (hint: use the add_machine_vertex_instance() functionality supplied with the GFE __init__.py interface to add vertex instances). Add edges between the cells.

Step 3 (Medium)

Build the c code that represents the functionality that will run on the SpiNNaker machine. It'll be easier to use the interfaces provided in simulation.h and data_specification.h.

At this point running the script should not produce errors, but you won't be able to tell what's happening inside.

Step 4 (Medium)

Add a data region for storing the state per timer tick iteration in SDRAM into both the c and python class. Add code to store the data in C and retrieve it from the machine in python.

Step 5 (Easy)

Build a simple text-based visualiser to replay a simulation run using the stored state.

At this point you should be able to run the simulation and get a textual display of the state of the simulation per timer tick.

Step 6 (Hard)

Stream the state of the simulation to the host PC during the simulation run. Display the output as text as the simulation runs.

Step 7 (Medium)

Use the **tubogrid** perl application supplied within spinnaker_tools to create a live visualisation of the simulation, or build your own.

Step 8 (Easy)

Try building a 28x28 grid of cells and see what happens. Can you explain why it doesn't work? What ways could you go about making it work (hint: there's at least 3 ways of doing this)?

Step 9 (Fiendishly Hard)

Convert your application so that instead of using machine vertices, you use an application vertex to represent the entire grid of cells. Note that this will require you to receive and interpret several ids from each base routing key (hint: sPyNNaker does this using something called a population table to map between a base key and a block of connections).

Step 10 (Easy)

Run your new simulation with both sets of initial parameters. Try scaling up so that you hit the limits of the system to simulate the application, and therefore make the CPU calculation correct so that the partitioner will stop you going over the limits.

Step 11 (Medium)

Upgrade **turbogrid** or your own visualisation to use the database so that it auto configures itself for the shape of the application space.

Step 12 (Epically Hard)

Build your own application for SpiNNaker using the GFE front end.

Simple Data Input Output and Visualisation on Spinnaker - Lab Manual

1. Introduction

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

2. Installation

In addition to sPyNNaker, the sPyNNakerExternalDevicesPlugin must also be installed.

3. PyNN Support

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

3.1 Output

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

The issue with the get functions are that they are called after run() completes, and therefore are not live, and so not able to interact with an external device running in real-time. In the current implementation of sPyNNaker, all of the data declared to be recorded via record(), record_v(), is stored on the SDRAM of the chips that the corresponding populations were placed on. 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.

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.

3.2 Input

The standard support for data input for a platform such as SpiNNaker, through the PyNN language, is to use the neural models SpikeSourceArray and SpikeSourcePoisson. The issue with both of these models is that they are either random rate based (the spikeSourcePoisson) or 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 between successive calls to run(). Again, this will only work if the external device or simulation can be paused.

4. External Device Plugin Support

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

It is worth noting that future releases of PyNN may use the MUSIC interface to support live injection and retrieval of spikes, but the current software version of sPyNNaker only supports PyNN 0.7 and therefore there is no built in support.

To compensate for this, the sPyNNakerExternalDevicesPlugin module was created that contains support for live injection and retrieval of spikes from a running PyNN 0.7 simulation during the simulation, whilst still maintaining the real-time operation of the simulation.

4.1 Live Output

To activate live retrieval from a given population, the command

activate_live_output_for(<Population_object>**)**

is used. This informs the sPyNNaker backend to add the supporting utility model (Live packet gatherer) into the graph object (which sPyNNaker uses to represent your PyNN neural models) and an edge between your population and the associate LPG for your ports.

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

Parameter	**Description**
port	The port number to receive packets from the SpiNNaker machine.
database_notify_host	The hostname for the database notification protocol; by default the localhost is used, but any external host could act as a receiver, provided it can read the file system that the database is written to.
database_notify_port_num	The port number for the database notification protocol; by default database notifications are sent to port 19998. However this can be changed if there is more than one listener, or a separate listener for each population.
database_ack_port_num	The port number that the database notification protocol will listen to, to receive the acknowledgement packet that the database has been read. By default this is 19999. It is unlikely that this needs to be changed.

4.2 Live Injection

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

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

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

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

Parameter	**Description**
port	The port that packets are going to be sent to on the SpiNNaker system - must be one per injector, but any port other than 17893 or 54321 can be used (these are reserved for SpiNNaker operations).
virtual_key	The base routing key that the spike injector is going to use for routing. **This parameter is optional.**

4.3 Python Live reciever

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

```
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
    for neuron_id in neuron_ids:
    print “Received spike at time {} from {}-{}”.format(
        time, label, neuron_id)

# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
    spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection

# set up python live spike connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
    receive_labels=[“receiver”])

# register python receiver with live spike connection
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().

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(
    receive_labels=["receiver_2"], local_port=19997)
```

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

```
activate_live_output_for(receiver, database_notify_port_num=19996)
activate_live_output_for(receiver_2, database_notify_port_num=19997)
```

4.4 Python Live injector

The following block of code creates a live packet injector:

```
1  # create python injector
2  def send_spike(label, sender):
3      sender.send_spike(label, 0, send_full_keys=True)
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.

4.5 C++ Implementation of SpyNNakerLiveSpikesConnection and Visualiser

The host C++ version of the Python "SpynnakerLiveSpikesConnection" and example visualiser is currently available from the following locations:

https://spinnakermanchester.github.io/latest/visualiser_code_zip.html
https://spinnakermanchester.github.io/latest/visualiser_code_tar_gz.html

This source code must be compiled before use, and depends on the pthread and sqlite libraries for the library itself, and the freeglut and opengl libraries for the example visualiser application. A Makefile exists at the top level folder which will make both the spynnaker_external_device_lib library and the example visualiser, but each can be made separately by running make in the appropriate subdirectory.

Dependency Installation

On OSX, using Macports, you can install the dependencies as follows:

sudo port install freeglut sqlite3

On Linux, you can install the dependencies as follows (depending on if you are using Fedora or Ubuntu):

sudo yum install
sudo apt-get install

On Windows, the dependencies are included.

spynnaker_external_device_lib

The C++ implementation is designed to be similar to the Python implementation. A number of sample applications are provided within the spynnaker_external_device_lib/examples folder which show how the API can be used.

c_based_visualiser_framework

This contains an example visualiser for producing a spike raster plot, and is based on the spynnaker_external_device_lib.

The visualiser application can accept 4 parameters. These are defined below:

Parameter	**Description**
-colour_map	Path to a file containing the population labels to receive, and their associated colours. This must be specified.
-hand_shake_port	Optional port which the visualiser will listen to for database handshake (default is 19998).
-database	Optional file path to where the database is located, if needed for manual configuration.
-remote_host	Optional remote host address of the SpiNNaker board, which allows port triggering if allowed by your firewall.

7.1 colour_map file format

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

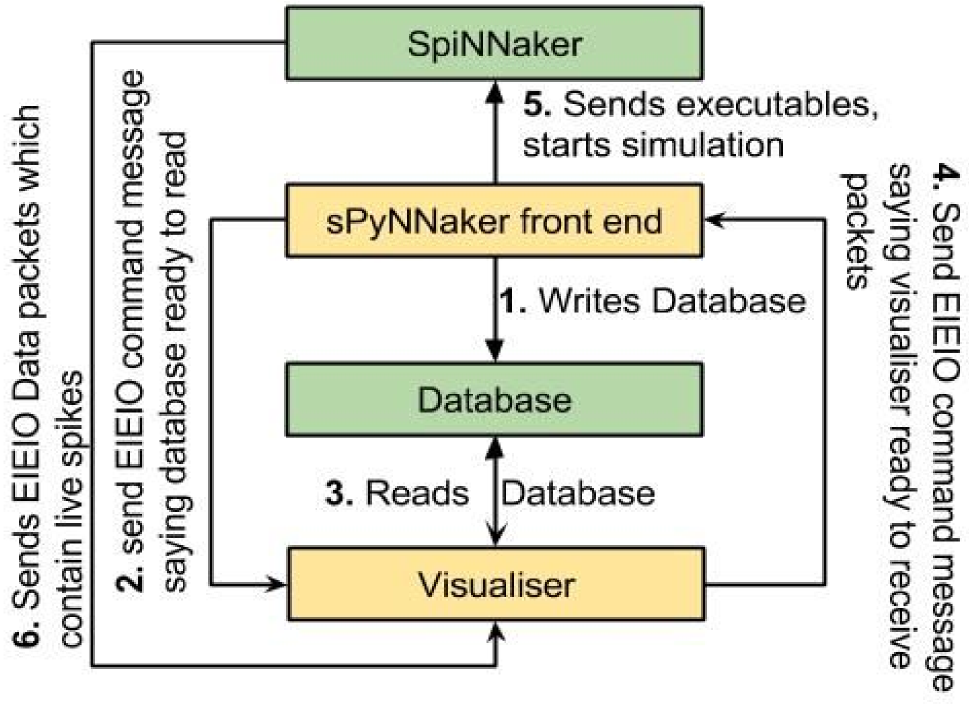
1. The population label.
2. The red colour value.
3. The green colour value.
4. The blue colour value.

An example file is shown below:

```
spike_forward	0	0	255
spike_backwards	0	255	0
```

5. Database Notification protocol

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


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

The steps within the notification protocol are defined below:

1. The sPyNNaker front end writes a database that contains all the data objects generated from sPyNNaker during the compilation process.
2. The notification protocol sends a message to all the notification protocol listeners containing the path to the database to be read. The SpynnakerLiveSpikesConnection Python and C implementations are set up to receive this message.
3. These devices then read the database to determine the information required. This includes the port to listen on to receive live output spikes, the port to send like input spikes to, and the mapping between SpiNNaker routing keys and neuron ids.
4. Once these devices have read the database, they notify the sPyNNaker front end that they are ready for the simulation to start.
5. Once all devices have notified the sPyNNaker front end, the simulation begins. The sPyNNaker front end also notifies the devices when the simulation has actually started, in case it was still loading data when they became ready.
6. The SpiNNaker machine transmits live spike output packets and receives live spike input packets.

6. 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.

8. Tasks

Task 1.1: A synfire chain with injected spike via python injector [Easy]

This task will create a synfire chain which is stimulated from a injector spike generated on host and then injected into the simulation. 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.
4. Import and instantiate an SpynnakerLiveSpikesConnection connection.
5. link a start callback to the python injector function.

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

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 1.3: A synfire chain with live injection and streaming via python [Easy]

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

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

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

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

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

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

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

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

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

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

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

Task 2.1: A simple synfire chain with a injected spike via c injector [Easy]

This task requires that you replace the injector from task 1.1 with a c injector.

1. Remember to import the correct header file.
2. Remember to use c syntax.

Task 2.2: A simple synfire chain with live streaming via the c receiver [Easy]

This task requires that you replace the receiver from task 1.2 with a c receiver.

1. Remember to import the correct header file.

2. Remember to use c syntax.

Task 2.3: A simple synfire chain with live injection and live streaming via C [Easy]

This task requires that you replace the injector and receiver from task 1.3 with a c injector and receiver.

1. Remember to import the correct header file.
2. Remember to use c syntax.

Task 2.4: A simple synfire chain with live injection via c and live streaming via the c visualiser [Medium]

This task requires that you replace the injector from task 1.4 with a c injector and to set up the visualiser.

1. Remember to import the correct header file.
2. Remember to use c syntax.
3. Remember to set up the visualiser correctly.

Task 2.5: 2 Synfire chains which set each other off using c injectors [Medium]

This task requires that you replace the injectors and receivers from task 1.5 with c injectors and receivers and to set up the visualiser.

1. Remember to import the correct header file.
2. Remember to use c syntax.

Task 2.6: 2 Synfire chains which set each other off using c injectors and live retrieval with 2 visualiser instances [Hard]

This task requires that you replace the injectors and receivers from task 1.6 with c injectors and receivers and to set up the visualiser.

1. Remember to import the correct header file.
2. Remember to use c syntax.
3. Remember to set up the visualisers correctly.

Task 3: Create some model which uses all interfaces [Very Hard]

This task is the merging of all the functionalities covered in this lab manual. Take the codes from both task 2.6 and 1.6 and integrate them together so that:

1. One injector is controlled by the c code, whilst another is done via the python interface.
2. Still uses 2 visualisers to stream the results.
3. Uses the python receive interface to count 5 firings of a given neuron id and then changes the neuron stimulated by the python injector.

Hint: remember to keep a global connection object for the python codes.

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

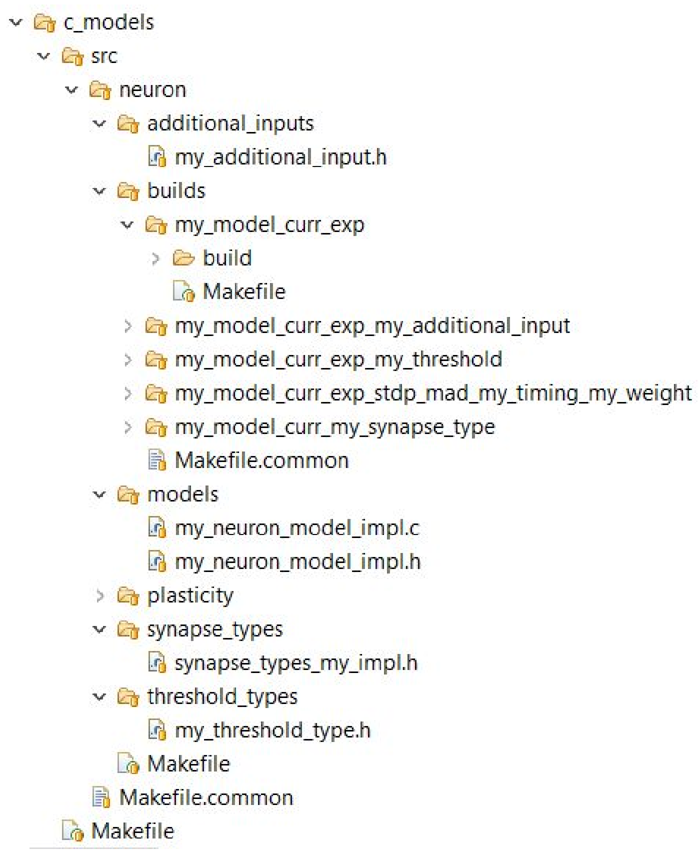
In order to create new models, you will need to ensure that you have set up a development environment suitable for compiling C code for SpiNNaker. This can be done by following the instructions here:

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

Project Layout

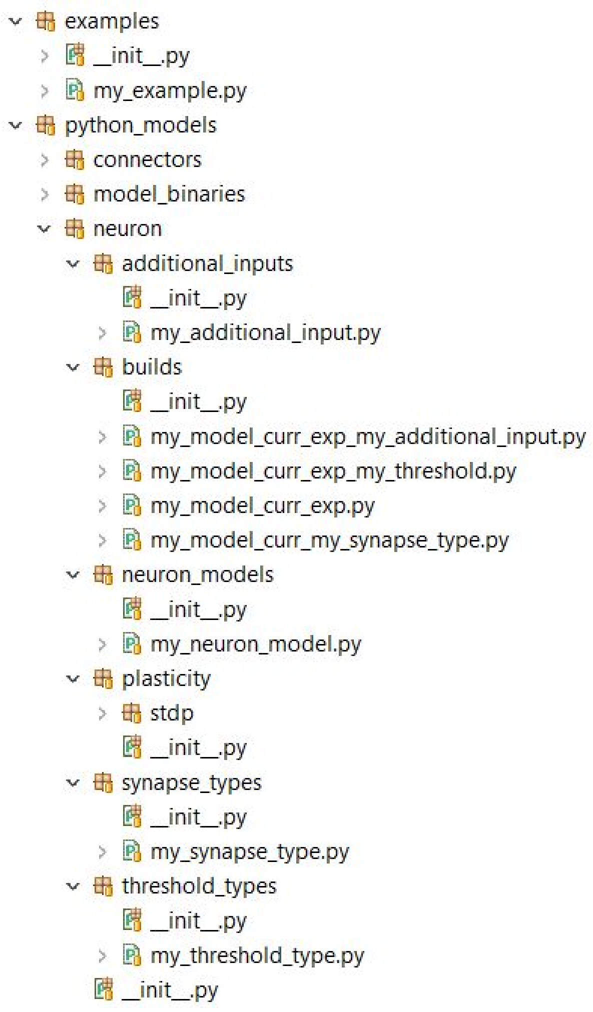
The recommended layout for a new model project is shown below; this example shows a model called "my_model", with current-based exponential synapses. It is recommended that the C and Python code is kept in the same project to help keep them synchronized.

C code



```
c_models
  src
    neuron
      additional_inputs
        my_additional_input.h
      builds
        my_model_curr_exp
          build
          Makefile
        my_model_curr_exp_my_additional_input
        my_model_curr_exp_my_threshold
        my_model_curr_exp_stdp_mad_my_timing_my_weight
        my_model_curr_my_synapse_type
        Makefile.common
      models
        my_neuron_model_impl.c
        my_neuron_model_impl.h
      plasticity
      synapse_types
        synapse_types_my_impl.h
      threshold_types
        my_threshold_type.h
      Makefile
    Makefile.common
  Makefile
```

Python code



```
examples
  __init__.py
  my_example.py
python_models
  connectors
  model_binaries
  neuron
    additional_inputs
      __init__.py
      my_additional_input.py
    builds
      __init__.py
      my_model_curr_exp_my_additional_input.py
      my_model_curr_exp_my_threshold.py
      my_model_curr_exp.py
      my_model_curr_my_synapse_type.py
    neuron_models
      __init__.py
      my_neuron_model.py
    plasticity
      stdp
      __init__.py
    synapse_types
      __init__.py
      my_synapse_type.py
    threshold_types
      __init__.py
      my_threshold_type.py
    __init__.py
```

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

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 implementation is a static threshold.
Additional input component	Any additional input current that might be based on the membrane voltage or other parameters. This is currently only implemented in the SpyNNakerExtraModelsPlugin.
Neuron model component	Determines how the neuron state changes over time and the outputs the current membrane voltage of the neuron. Currently there are IZK and 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 c *Makefile* which describes 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 show in **Code 1**.

```
APP = $(notdir $(CURDIR))
BUILD_DIR = build/

NEURON_MODEL = $(EXTRA_SRC_DIR)/neuron/models/my_neuron_model_impl.c
NEURON_MODEL_H = $(EXTRA_SRC_DIR)/neuron/models/my_neuron_model_impl.h
INPUT_TYPE_H = $(SOURCE_DIR)/neuron/input_types/input_type_current.h
THRESHOLD_TYPE_H = $(SOURCE_DIR)/neuron/threshold_types/threshold_type_static.h
SYNAPSE_TYPE_H = $(SOURCE_DIR)/neuron/synapse_types/synapse_types_exponential_impl.h
SYNAPSE_DYNAMICS = $(SOURCE_DIR)/neuron/plasticity/synapse_dynamics_static_impl.c

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 states 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 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

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

```
make
```

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

```
make clean
```

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

```
make 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 .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/latest/provenance_data/; each is a .txt file containing any output printed from your program.

C code file interfaces

The rest of this section goes through the different components interfaces and tries to explain what each one does, for the case where you need to create a new component for your neuron build.

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 neuron parameters 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_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 the run, and that might be useful in debugging.

- `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.

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 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_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.

-
  ```
  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.

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.

Set 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.

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 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 the my_model_curr_exp in python_models -> 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 91 and 92 the neuron model component is created.
2. On Lines 96 and 97 the synapse type component is created.
3. On Line 101 the input type component is created.
4. On Line 105 the threshold type component is created.
5. Line 109 shows that this model does not contain any additional input components.
6. Lines 113 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.

```
89 .    # TODO: create your neuron model class (change if required)
90.     # create your neuron model class
91.     neuron_model = MyNeuronModel(
92.         n_neurons, machine_time_step, i_offset, my_parameter)
93.
94.     # TODO: create your synapse type model class (change if required)
95.     # create your synapse type model
96.     synapse_type = SynapseTypeExponential(
97.         n_neurons, machine_time_step, tau_syn_E, tau_syn_I)
98.
99.     # TODO: create your input type model class (change if required)
100.    # create your input type model
101.    input_type = InputTypeCurrent()
```

```

# TODO: create your threshold type model class (change if required)
# create your threshold type model
threshold_type = ThresholdTypeStatic(n_neurons, v_thresh)

# TODO: create your own additional inputs (change if required).
# create your own additional inputs
additional_input = None

# instantiate the sPyNNaker system by initializing
#  the AbstractPopulationVertex
AbstractPopulationVertex.__init__(

    # standard inputs, do not need to change.
    self, n_neurons=n_neurons, label=label,
    machine_time_step=machine_time_step,
    timescale_factor=timescale_factor,
    spikes_per_second=spikes_per_second,
    ring_buffer_sigma=ring_buffer_sigma,
    incoming_spike_buffer_size=incoming_spike_buffer_size,

    # TODO: Ensure the correct class is used below
    max_atoms_per_core=MyModelCurrExp._model_based_max_atoms_per_core,

    # These are the various model types
    neuron_model=neuron_model, input_type=input_type,
    synapse_type=synapse_type, threshold_type=threshold_type,
    additional_input=additional_input,

    # TODO: Give the model a name (shown in reports)
    model_name="MyModelCurrExp",

    # TODO: Set this to the matching binary name
    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 build 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_models -> neuron. These detail the parts of the class that need to be changed for your model.

Python __init__.py files

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

Python setup.py file

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

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

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

Using your module

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

```
import pyNN.spiNNaker as p
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.

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 Makefile so that it uses 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. 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_param, 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 greater than than the probability value.

5. Rebuild the my_model_curr_exp_my_threshold_type.

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 PyNN RNG, 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 differs for different settings of the spike probability.

Adding new models of synaptic plasticity

August 28, 2015

Contents of package

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

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

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

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

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

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

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

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

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

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

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

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

Additional code changes

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

Exercises

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

Exercise 1

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

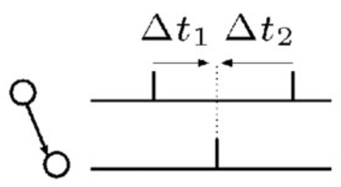
```
timing_dependence = sim.PfisterSpikeTripletRule(
            tau_plus=16.8, tau_minus=33.7,
            tau_x=101, tau_y=114)

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

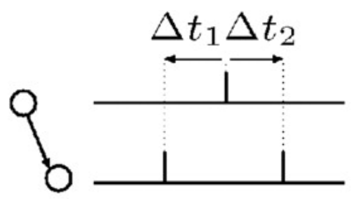
Does this actually reduce the error compared to the version developed in this workshop? Why might this be? The talk this morning on 'Maths & fixed point libraries' may give you some clues!

Exercise 2

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



(a) Pre-post-pre triplet



(b) Post-pre-post triplet

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

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

(a) Pre-post-pre triplets

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

(b) Post-pre-post triple

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

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

References

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

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

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