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

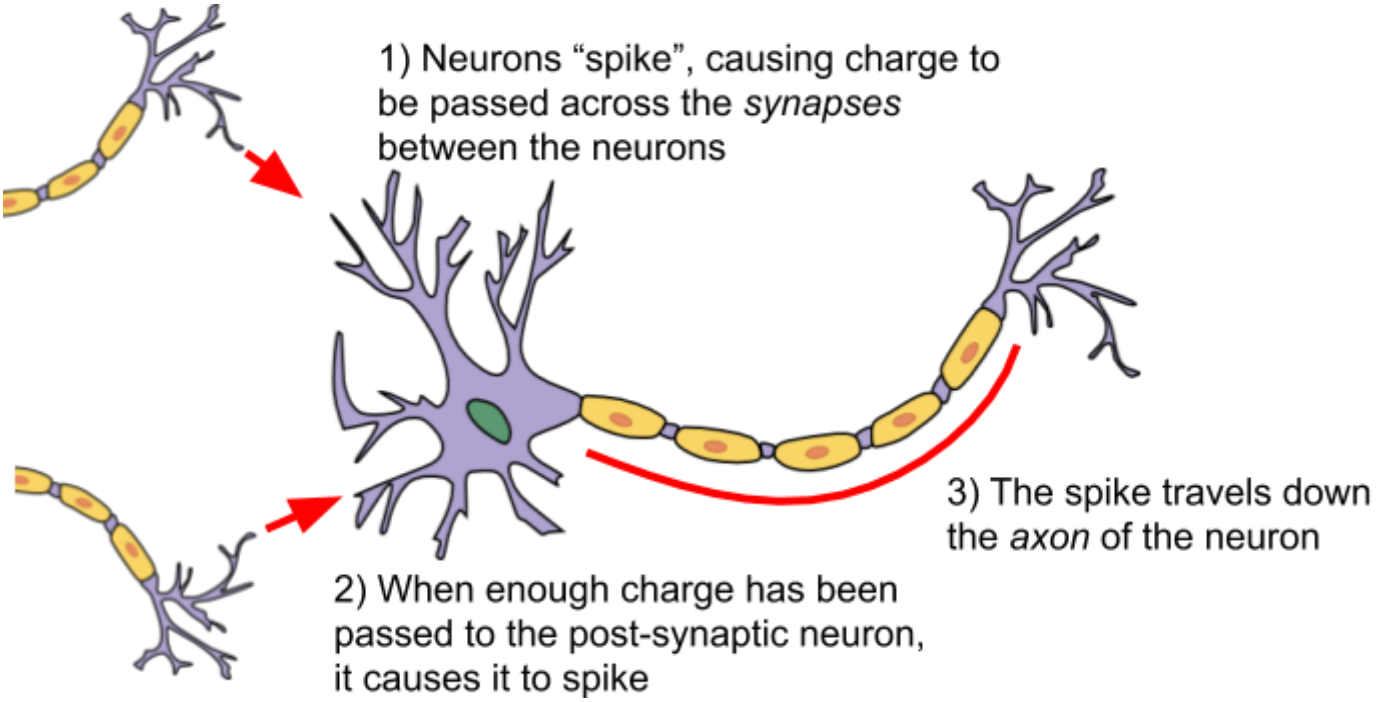
If you haven't installed it already, then the PyNN toolchain for SpiNNaker, sPyNNaker, can be installed by following the instructions available from here:

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

Please install for PyNN version 0.9 if you are installing for the first time. If you are using PyNN 0.8 then it should also work with that, but be aware that PyNN developers expect users to be working with 0.9.

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 the 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

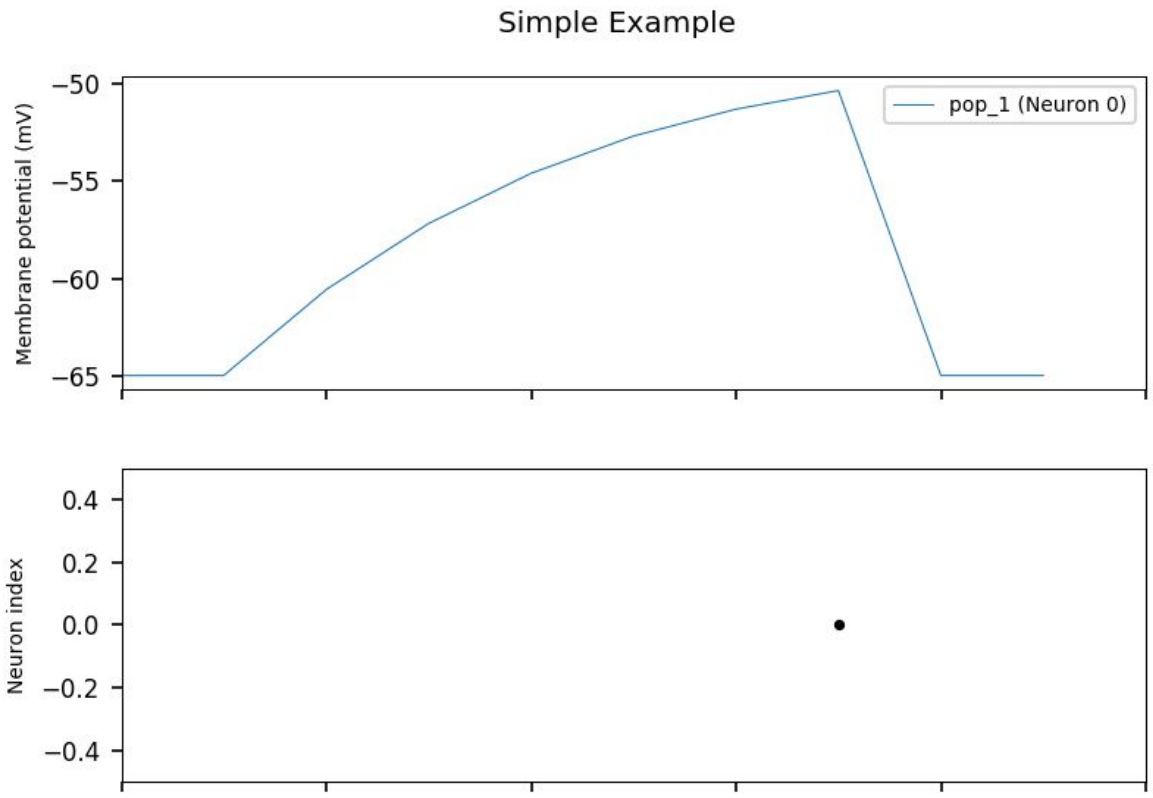
PyNN is a language for building spiking neural network models. PyNN models can then be run on a number of simulators without modification (or with only minor modifications), including SpiNNaker. The basic steps of building a PyNN network are as follows:

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.



Simulated with SpiNNaker_under_version(1!5.0.1-Liveware Problem)

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 excitatory 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 of 0.75 and delays of 1.0 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 synaptic current in models such as IF_curr_exp.

On SpiNNaker, it is also possible to selectively record data; this is done by specifying a *sampling_interval* i.e. **pop.record('v', sampling_interval=n)** to sample every n timesteps of the simulation. It is also possible to record selected neurons from within a Population by using record with a PopulationView object.

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 Quantities 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.spynnaker_plotting* contains a *SpynnakerPanel* object that 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*.

Alternatively, if you wish to use the large SpiNNaker machine, then you need to use the allocation system spalloc. This requires you to define the properties *spalloc_server* and *spalloc_user* in the .spynnaker.cfg file:

```
[Machine]
spalloc_server  = spinnaker.cs.man.ac.uk
spalloc_user    = youremail@...
```

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.

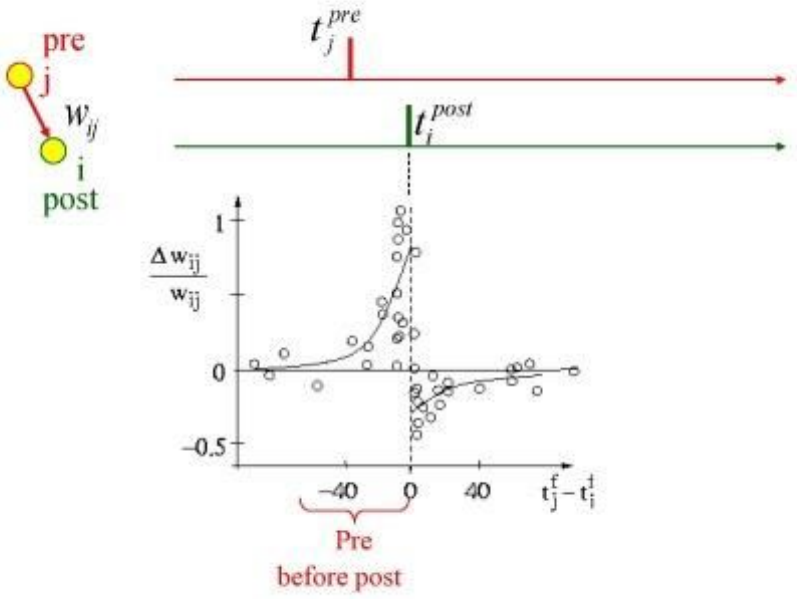
Spike-Time-Dependent Plasticity

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 in. 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 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).