

Simple Data Input Output and Visualisation on Spinnaker Lab Manual

1. Introduction

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

2. Installation

The PyNN 0.7 toolchain for SpiNNaker (sPyNNaker 2015.005), can be installed by following the instructions available from here:

<https://github.com/SpiNNakerManchester/SpiNNakerManchester.github.io/> (link to be filled in)

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

The sPyNNakerExternalDevicesPlugin 2015.009 can be installed by following the instructions available from here:

<https://github.com/SpiNNakerManchester/SpiNNakerManchester.github.io/> (link to be filled in)

The Visualiser 2015.002 module can be downloaded from here:

<https://github.com/SpiNNakerManchester/Visualiser> (link to be filled in)

and can be compiled by running the command `make -f Makefile.<os>`

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

(link to be filled in)

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

(link to be filled in)

3. PyNN Support

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

3.1 Output

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

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

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

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

3.2 Input

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

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

4. External Device Plugin Support

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

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

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

4.1 Live Output

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

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

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

4.2 Live Injection

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

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

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

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

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

4.3 Python Live reciever

The following block of code creates a live packet receiver:

```
1 # declare python code when received spikes for a timer tick  
2 def receive_spikes(label, time, neuron_ids):  
3     for neuron_id in neuron_ids:  
4         print "Received spike at time {} from {}-{}".format(  
5             time, label, neuron_id)  
6 # import python live spike connection  
7 from spynnaker_external_devices_plugin.pyNN.connections.\  
8 spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection  
9 # set up python live spike connection  
10 live_spikes_connection = SpynnakerLiveSpikesConnection(  
11     receive_labels=["receiver"], local_port=19995, send_labels=None)
```

```

12 # register python receiver with live spike connection
13 live_spikes_connection.add_receive_callback("receiver", receive_spikes)
14 p.run(5000)

```

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

4.4 Python Live injector

The following block of code creates a live packet injector:

```

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

```

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

4. Lines 10 to 11 informs the connection that when the simulation starts, to call the send_spike function defined on lines 1 to 3.
5. Line 12 executes the model on SpiNNaker.

4.5 C Live receiver

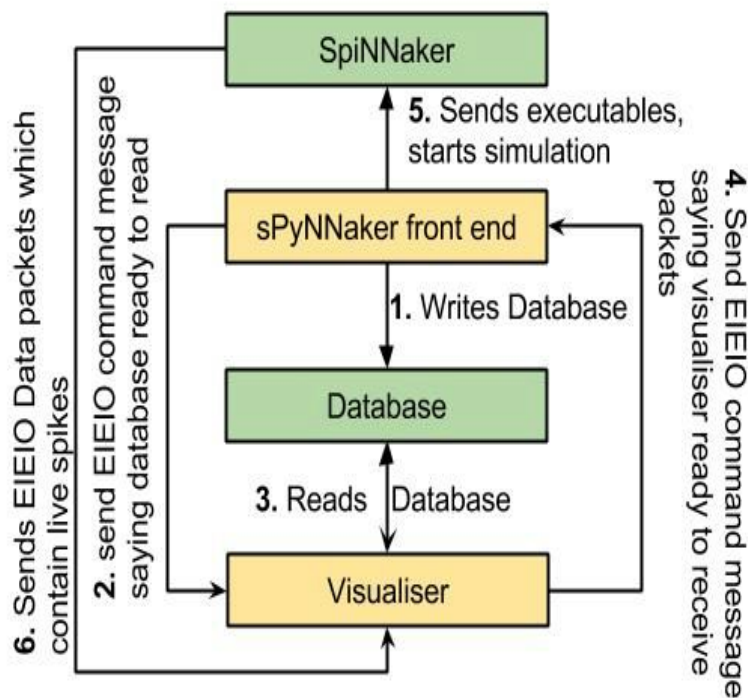
NEEDS FILLING IN

4.6 C Live injection

NEEDS FILLING IN

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:



The steps within the notification protocol are defined below:

1. The sPyNNaker front end writes a database that contains all the data objects generated from sPyNNaker during the compilation process.

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

6. Caveats

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

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

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

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

7. Visualiser

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

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

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

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

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

7.1 colour_map file format

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

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

An example file is shown below:

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

8. Tasks

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

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

[insert link here]

Please go back to this lab manual and complete this task before attempting these if you haven't already.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1. you will need to change the number of neurons the spike injector contains.
2. You will need to change the connector from the spike injector and each synfire population.
3. You will need to modify the receive function, and add a global variable for the SpynakerLiveSpikesConnection.
4. You'll need at least 2 SpynakerLiveSpikesConnection and multiple activate_live_output_for(<pop_object>) 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 SpynakerLiveSpikesConnection'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] [Needs filling in]

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

[Needs filling in]

**Task 2.3: A simple synfire chain with live injection and live streaming via
the c support [Easy]**

[Needs filling in]

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

[Needs filling in]

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

[Needs filling in]

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

[Needs filling in]

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

[assumes c code built]

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.

hints:

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

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

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

[insert link here]

Simple Data Input Output and Visualisation on Spinnaker Lab Manual

Task Solutions:

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

```
# imports of both spynnaker and external device plugin.
import spynnaker.pyNN as Frontend
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
from spynnaker_external_devices_plugin.pyNN.connections\
    .spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# plotter in python
import pylab
# initial call to set up the front end (pyNN requirement)
Frontend.setup(timestep=1.0, min_delay=1.0, max_delay=144.0)
# neurons per population and the length of runtime in ms for the simulation,
# as well as the expected weight each spike will contain
n_neurons = 100
run_time = 8000
weight_to_spike = 2.0
# neural parameters of the ifcur model used to respond to injected spikes.
# (cell params for a synfire chain)
cell_params_lif = {'cm': 0.25, 'i_offset': 0.0, 'tau_m': 20.0, 'tau_refrac': 2.0,
                  'tau_syn_E': 5.0, 'tau_syn_I': 5.0, 'v_reset': -70.0, 'v_rest':
-65.0,
                  'v_thresh': -50.0}
# create synfire populations (if cur exp)
pop_forward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                  cell_params_lif, label='pop_forward')
# Create injection populations
injector_forward = Frontend.Population(
    n_neurons, ExternalDevices.SpikeInjector,
    {'port':12365}, label='spike_injector_forward')
# Create a connection from the injector into the populations
Frontend.Projection(injector_forward, pop_forward,
                   Frontend.OneToOneConnector(weights=weight_to_spike))
# Synfire chain connections where each neuron is connected to its next neuron
# NOTE: there is no recurrent connection so that each chain stops once it
# reaches the end
loop_forward = list()
loop_backward = list()
```

```

for i in range(0, n_neurons - 1):
    loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward,
                    Frontend.FromListConnector(loop_forward))
# record spikes from the synfire chains so that we can read off valid results
# in a safe way afterwards, and verify the behavior
pop_forward.record()
# Create a sender of packets for the forward population
def send_input_forward(label, sender):
    print "Sending forward spike for neuron 0"
    sender.send_spike(label, 0)
# Set up the live connection for sending spikes
live_spikes_connection = SpynnakerLiveSpikesConnection(
    receive_labels=None, local_port=19999,
    send_labels=["spike_injector_forward"])
# Set up callbacks to occur at the start of simulation
live_spikes_connection.add_start_callback("spike_injector_forward",
                                         send_input_forward)

# Run the simulation on spiNNaker
Frontend.run(run_time)
# Retrieve spikes from the synfire chain population
spikes_forward = pop_forward.getSpikes()
# If there are spikes, plot using matplotlib
if len(spikes_forward) != 0:
    pylab.figure()
    if len(spikes_forward) != 0:
        pylab.plot([i[1] for i in spikes_forward],
                  [i[0] for i in spikes_forward], "b.")
    pylab.ylabel('neuron id')
    pylab.xlabel('Time/ms')
    pylab.title('spikes')
    pylab.show()
else:
    print "No spikes received"
# Clear data structures on spiNNaker to leave the machine in a clean state for
# future executions
Frontend.end()

```

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

```
# imports of both spynnaker and external device plugin.
import spynnaker.pyNN as Frontend
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
from spynnaker_external_devices_plugin.pyNN.connections\
    .spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# plotter in python
import pylab
# initial call to set up the front end (pynn requirement)
Frontend.setup(timestep=1.0, min_delay=1.0, max_delay=144.0)
# neurons per population and the length of runtime in ms for the simulation,
# as well as the expected weight each spike will contain
n_neurons = 100
run_time = 8000
weight_to_spike = 2.0
# neural parameters of the ifcur model used to respond to injected spikes.
# (cell params for a synfire chain)
cell_params_lif = {'cm': 0.25, 'i_offset': 0.0, 'tau_m': 20.0, 'tau_refrac': 2.0,
                  'tau_syn_E': 5.0, 'tau_syn_I': 5.0, 'v_reset': -70.0, 'v_rest':
-65.0,
                  'v_thresh': -50.0}
# create synfire populations (if cur exp)
pop_forward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                  cell_params_lif, label='pop_forward')
# Create injection populations
injector_forward = Frontend.Population(
    1, Frontend.SpikeSourceArray,
    {'spike_times': [[0]]}, label='spike_playback_forward')
# Create a connection from the injector into the populations
Frontend.Projection(injector_forward, pop_forward,
                   Frontend.FromListConnector([(0, 0, weight_to_spike, 1)]))
# Synfire chain connections where each neuron is connected to its next neuron
# NOTE: there is no recurrent connection so that each chain stops once it
# reaches the end
loop_forward = list()
for i in range(0, n_neurons - 1):
    loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward,
                   Frontend.FromListConnector(loop_forward))
# record spikes from the synfire chains so that we can read off valid results
# in a safe way afterwards, and verify the behavior
pop_forward.record()
# Activate the sending of live spikes
ExternalDevices.activate_live_output_for(
```

```

    pop_forward, database_notify_host="localhost",
    database_notify_port_num=19996)
# Create a receiver of live spikes
def receive_spikes(label, time, neuron_ids):
    for neuron_id in neuron_ids:
        print "Received spike at time", time, "from", label, "-", neuron_id
# if not using the c visualiser, then a new spynnaker live spikes connection
# is created to define that there are python code which receives the
# outputted spikes.
live_spikes_connection = SpynnakerLiveSpikesConnection(
    receive_labels=["pop_forward"], local_port=19996, send_labels=None)
# Set up callbacks to occur when spikes are received
live_spikes_connection.add_receive_callback("pop_forward", receive_spikes)
# Run the simulation on spiNNaker
Frontend.run(run_time)
# Retrieve spikes from the synfire chain population
spikes_forward = pop_forward.getSpikes()
# If there are spikes, plot using matplotlib
if len(spikes_forward) != 0:
    pylab.figure()
    if len(spikes_forward) != 0:
        pylab.plot([i[1] for i in spikes_forward],
                    [i[0] for i in spikes_forward], "b.")
        pylab.ylabel('neuron id')
        pylab.xlabel('Time/ms')
        pylab.title('spikes')
        pylab.show()
else:
    print "No spikes received"
# Clear data structures on spiNNaker to leave the machine in a clean state for
# future executions
Frontend.end()

```


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

```
# imports of both spynnaker and external device plugin.
import spynnaker.pyNN as Frontend
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
from spynnaker_external_devices_plugin.pyNN.connections\
    .spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# plotter in python
import pylab
# initial call to set up the front end (pynn requirement)
Frontend.setup(timestep=1.0, min_delay=1.0, max_delay=144.0)
# neurons per population and the length of runtime in ms for the simulation,
# as well as the expected weight each spike will contain
n_neurons = 100
run_time = 8000
weight_to_spike = 2.0
# neural parameters of the ifcur model used to respond to injected spikes.
# (cell params for a synfire chain)
cell_params_lif = {'cm': 0.25, 'i_offset': 0.0, 'tau_m': 20.0, 'tau_refrac': 2.0,
                  'tau_syn_E': 5.0, 'tau_syn_I': 5.0, 'v_reset': -70.0, 'v_rest':
-65.0,
                  'v_thresh': -50.0}
# create synfire populations (if cur exp)
pop_forward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                  cell_params_lif, label='pop_forward')
# Create injection populations
injector_forward = Frontend.Population(
    n_neurons, ExternalDevices.SpikeInjector,
    {'port':12365}, label='spike_injector_forward')
# Create a connection from the injector into the populations
Frontend.Projection(injector_forward, pop_forward,
                   Frontend.OneToOneConnector(weights=weight_to_spike))
# Synfire chain connections where each neuron is connected to its next neuron
# NOTE: there is no recurrent connection so that each chain stops once it
# reaches the end
loop_forward = list()
loop_backward = list()
for i in range(0, n_neurons - 1):
    loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward, Frontend.FromListConnector(loop_forward))
# record spikes from the synfire chains so that we can read off valid results
# in a safe way afterwards, and verify the behavior
pop_forward.record()
# Activate the sending of live spikes
```

```

ExternalDevices.activate_live_output_for(
    pop_forward, database_notify_host="localhost",
    database_notify_port_num=19996)
# Create a sender of packets for the forward population
def send_input_forward(label, sender):
    print "Sending forward spike for neuron 0"
    sender.send_spike(label, 0)
# Create a receiver of live spikes
def receive_spikes(label, time, neuron_ids):
    for neuron_id in neuron_ids:
        print "Received spike at time", time, "from", label, "-", neuron_id
# Set up the live connection for sending spikes
live_spikes_connection = SpynakerLiveSpikesConnection(
    receive_labels=None, local_port=19999, send_labels=["spike_injector_forward"])
# Set up callbacks to occur at the start of simulation
live_spikes_connection.add_start_callback("spike_injector_forward", send_input_forward)
# if not using the c visualiser, then a new spynaker live spikes connection
# is created to define that there are python code which receives the
# outputted spikes.
live_spikes_connection = SpynakerLiveSpikesConnection(
    receive_labels=["pop_forward"], local_port=19996, send_labels=None)
# Set up callbacks to occur when spikes are received
live_spikes_connection.add_receive_callback("pop_forward", receive_spikes)
# Run the simulation on spiNNaker
Frontend.run(run_time)
# Retrieve spikes from the synfire chain population
spikes_forward = pop_forward.getSpikes()
# If there are spikes, plot using matplotlib
if len(spikes_forward) != 0:
    pylab.figure()
    if len(spikes_forward) != 0:
        pylab.plot([i[1] for i in spikes_forward],
                   [i[0] for i in spikes_forward], "b.")
        pylab.ylabel('neuron id')
        pylab.xlabel('Time/ms')
        pylab.title('spikes')
        pylab.show()
else:
    print "No spikes received"
# Clear data structures on spiNNaker to leave the machine in a clean state for
# future executions
Frontend.end()

```

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

```
# imports of both spynnaker and external device plugin.
import spynnaker.pyNN as Frontend
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
from spynnaker_external_devices_plugin.pyNN.connections\
    .spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# plotter in python
import pylab
# initial call to set up the front end (pynn requirement)
Frontend.setup(timestep=1.0, min_delay=1.0, max_delay=144.0)
# neurons per population and the length of runtime in ms for the simulation,
# as well as the expected weight each spike will contain
n_neurons = 100
run_time = 8000
weight_to_spike = 2.0
# neural parameters of the ifcur model used to respond to injected spikes.
# (cell params for a synfire chain)
cell_params_lif = {'cm': 0.25, 'i_offset': 0.0, 'tau_m': 20.0, 'tau_refrac': 2.0,
                  'tau_syn_E': 5.0, 'tau_syn_I': 5.0, 'v_reset': -70.0, 'v_rest':
-65.0,
                  'v_thresh': -50.0}
# create synfire populations (if cur exp)
pop_forward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                  cell_params_lif, label='pop_forward')
# Create injection populations
injector_forward = Frontend.Population(
    n_neurons, ExternalDevices.SpikeInjector,
    {'port':12365}, label='spike_injector_forward')
# Create a connection from the injector into the populations
Frontend.Projection(injector_forward, pop_forward,
                    Frontend.OneToOneConnector(weights=weight_to_spike))
# Synfire chain connections where each neuron is connected to its next neuron
# NOTE: there is no recurrent connection so that each chain stops once it
# reaches the end
loop_forward = list()
for i in range(0, n_neurons - 1):
    loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward,
                    Frontend.FromListConnector(loop_forward))
# record spikes from the synfire chains so that we can read off valid results
# in a safe way afterwards, and verify the behavior
pop_forward.record()
# Activate the sending of live spikes
ExternalDevices.activate_live_output_for(
```

```

    pop_forward, database_notify_host="localhost",
    database_notify_port_num=19996)
# Create a sender of packets for the forward population
def send_input_forward(label, sender):
    print "Sending forward spike for neuron 0"
    sender.send_spike(label, 0)
# if not using the c visualiser, then a new spynnaker live spikes connection
# is created to define that there are python code which receives the
# outputted spikes.
live_spikes_connection = SpynnakerLiveSpikesConnection(
    receive_labels=["pop_forward"], local_port=19996,
    send_labels=["spike_injector_forward"])
# Set up callbacks to occur at the start of simulation
live_spikes_connection.add_start_callback("spike_injector_forward", send_input_forward)
# Run the simulation on spiNNaker
Frontend.run(run_time)
# Retrieve spikes from the synfire chain population
spikes_forward = pop_forward.getSpikes()
# If there are spikes, plot using matplotlib
if len(spikes_forward) != 0:
    pylab.figure()
    if len(spikes_forward) != 0:
        pylab.plot([i[1] for i in spikes_forward],
                    [i[0] for i in spikes_forward], "b.")
        pylab.ylabel('neuron id')
        pylab.xlabel('Time/ms')
        pylab.title('spikes')
        pylab.show()
else:
    print "No spikes received"
# Clear data structures on spiNNaker to leave the machine in a clean state for
# future executions
Frontend.end()

```

Visualiser stuff

command line:

`./vis -colour-map single_synfire`

colour map ("single_synfire"):

`pop_forward 0 0 255`

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

```
# imports of both spynnaker and external device plugin.
import spynnaker.pyNN as Frontend
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
from spynnaker_external_devices_plugin.pyNN.connections\
    .spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# plotter in python
import pylab
from threading import Condition
# initial call to set up the front end (pynn requirement)
Frontend.setup(timestep=1.0, min_delay=1.0, max_delay=144.0)
# neurons per population and the length of runtime in ms for the simulation,
# as well as the expected weight each spike will contain
n_neurons = 100
run_time = 8000
weight_to_spike = 2.0
# neural parameters of the ifcur model used to respond to injected spikes.
# (cell params for a synfire chain)
cell_params_lif = {'cm': 0.25, 'i_offset': 0.0, 'tau_m': 20.0, 'tau_refrac': 2.0,
                  'tau_syn_E': 5.0, 'tau_syn_I': 5.0, 'v_reset': -70.0, 'v_rest':
-65.0,
                  'v_thresh': -50.0}
# create synfire populations (if cur exp)
pop_forward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                  cell_params_lif, label='pop_forward')
pop_backward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                   cell_params_lif, label='pop_backward')
# create equiv of parrot populations for closed loop calculation
pop_forward_parrot = Frontend.Population(
    n_neurons, Frontend.IF_curr_exp, cell_params_lif, label='pop_forward_parrot')
pop_backward_parrot = Frontend.Population(
    n_neurons, Frontend.IF_curr_exp, cell_params_lif, label='pop_backward_parrot')
# Create injection populations
injector_forward = Frontend.Population(
    n_neurons, ExternalDevices.SpikeInjector, {"port": 19344},
    label='spike_injector_forward')
# Create a connection from the injector into the populations
Frontend.Projection(injector_forward, pop_forward,
                   Frontend.FromListConnector([0, 0, weight_to_spike, 3]))
Frontend.Projection(injector_forward, pop_backward,
                   Frontend.FromListConnector([1, 99, weight_to_spike, 3]))
# Add links to the parrot populations for closed loop calculations
Frontend.Projection(pop_forward, pop_forward_parrot,
                   Frontend.OneToOneConnector(weight_to_spike, 1))
```

```

Frontend.Projection(pop_backward, pop_backward_parrot,
                    Frontend.OneToOneConnector(weight_to_spike, 1))
# Synfire chain connections where each neuron is connected to its next neuron
# NOTE: there is no recurrent connection so that each chain stops once it
# reaches the end
loop_forward = list()
loop_backward = list()
for i in range(0, n_neurons - 1):
    loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
    loop_backward.append(((i + 1) % n_neurons, i, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward,
                    Frontend.FromListConnector(loop_forward))
Frontend.Projection(pop_backward, pop_backward,
                    Frontend.FromListConnector(loop_backward))
# record spikes from the synfire chains so that we can read off valid results
# in a safe way afterwards, and verify the behavior
pop_forward.record()
pop_backward.record()
# Activate the sending of live spikes
ExternalDevices.activate_live_output_for(
    pop_forward, database_notify_host="localhost",
    database_notify_port_num=19996)
ExternalDevices.activate_live_output_for(
    pop_backward, database_notify_host="localhost",
    database_notify_port_num=19996)
# Activate the sending of spikes for the python reciever
ExternalDevices.activate_live_output_for(
    pop_forward, database_notify_host="localhost",
    database_notify_port_num=19995, port=13333)
ExternalDevices.activate_live_output_for(
    pop_backward, database_notify_host="localhost",
    database_notify_port_num=19995, port=13333)
# Create a condition to avoid overlapping prints
print_condition = Condition()
# Create a sender of packets for the forward population
def send_input_forward(label, sender):
    print "Sending forward spike for neuron 0"
    sender.send_spike(label, 0)
# Create a receiver of live spikes
def receive_spikes(label, time, neuron_ids):
    for neuron_id in neuron_ids:
        print_condition.acquire()
        print "Received spike at time", time, "from", label, "-", neuron_id
        print_condition.release()
        if neuron_id == 0 and label=="pop_backward_parrot":
            live_spikes_connection.send_spike("pop_forward", 0)
        elif neuron_id == 99 and label=="pop_forward_parrot":
            live_spikes_connection.send_spike("pop_backward", 1)

```

```

# Set up the live connection for sending spikes
live_spikes_connection = SpynnakerLiveSpikesConnection(
    local_port=19996, send_labels=["spike_injector_forward"])
# Set up the live connection for sending spikes
live_spikes_connection_receiver = SpynnakerLiveSpikesConnection(
    receive_labels=["pop_forward_parrot", "pop_backward_parrot"], local_port=19995)
# Set up callbacks to occur at the start of simulation
live_spikes_connection.add_start_callback("spike_injector_forward",
    send_input_forward)
# Set up callbacks to occur when spikes are received
live_spikes_connection_receiver.add_receive_callback(
    "pop_forward_parrot", receive_spikes)
live_spikes_connection_receiver.add_receive_callback(
    "pop_backward_parrot", receive_spikes)
Frontend.run(run_time)

# Retrieve spikes from the synfire chain population
spikes_forward = pop_forward.getSpikes()
spikes_backward = pop_backward.getSpikes()
# If there are spikes, plot using matplotlib
if len(spikes_forward) != 0 or len(spikes_backward) != 0:
    pylab.figure()
    if len(spikes_forward) != 0:
        pylab.plot([i[1] for i in spikes_forward],
            [i[0] for i in spikes_forward], "b.")
    if len(spikes_backward) != 0:
        pylab.plot([i[1] for i in spikes_backward],
            [i[0] for i in spikes_backward], "r.")
    pylab.ylabel('neuron id')
    pylab.xlabel('Time/ms')
    pylab.title('spikes')
    pylab.show()
else:
    print "No spikes received"
# Clear data structures on spiNNaker to leave the machine in a clean state for
# future executions
Frontend.end()

```

Visualiser stuff

command line:

```
./vis -colour-map duel_synfire
```

colour map ("duel_synfire"):

```
pop_forward 0      0      255
pop_backward 0     255     0
```

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

```
# imports of both spynnaker and external device plugin.
import spynnaker.pyNN as Frontend
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
from spynnaker_external_devices_plugin.pyNN.connections\
    .spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# plotter in python
import pylab
from threading import Condition
# initial call to set up the front end (pynn requirement)
Frontend.setup(timestep=1.0, min_delay=1.0, max_delay=144.0)
# neurons per population and the length of runtime in ms for the simulation,
# as well as the expected weight each spike will contain
n_neurons = 100
run_time = 8000
weight_to_spike = 2.0
# neural parameters of the ifcur model used to respond to injected spikes.
# (cell params for a synfire chain)
cell_params_lif = {'cm': 0.25, 'i_offset': 0.0, 'tau_m': 20.0, 'tau_refrac': 2.0,
                  'tau_syn_E': 5.0, 'tau_syn_I': 5.0, 'v_reset': -70.0, 'v_rest':
-65.0,
                  'v_thresh': -50.0}
# create synfire populations (if cur exp)
pop_forward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                  cell_params_lif, label='pop_forward')
pop_backward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                   cell_params_lif, label='pop_backward')
# create equiv of parrot populations for closed loop calculation
pop_forward_parrot = Frontend.Population(
    n_neurons, Frontend.IF_curr_exp, cell_params_lif, label='pop_forward_parrot')
pop_backward_parrot = Frontend.Population(
    n_neurons, Frontend.IF_curr_exp, cell_params_lif, label='pop_backward_parrot')
# Create injection populations
injector_forward = Frontend.Population(
    n_neurons, ExternalDevices.SpikeInjector, {"port": 19344},
    label='spike_injector_forward')
# Create a connection from the injector into the populations
Frontend.Projection(injector_forward, pop_forward,
                   Frontend.FromListConnector([0, 0, weight_to_spike, 3]))
Frontend.Projection(injector_forward, pop_backward,
                   Frontend.FromListConnector([1, 99, weight_to_spike, 3]))
# Add links to the parrot populations for closed loop calculations
Frontend.Projection(pop_forward, pop_forward_parrot,
                   Frontend.OneToOneConnector(weight_to_spike, 1))
```



```

Frontend.Projection(pop_backward, pop_backward_parrot,
                    Frontend.OneToOneConnector(weight_to_spike, 1))
# Synfire chain connections where each neuron is connected to its next neuron
# NOTE: there is no recurrent connection so that each chain stops once it
# reaches the end
loop_forward = list()
loop_backward = list()
for i in range(0, n_neurons - 1):
    loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
    loop_backward.append(((i + 1) % n_neurons, i, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward,
                    Frontend.FromListConnector(loop_forward))
Frontend.Projection(pop_backward, pop_backward,
                    Frontend.FromListConnector(loop_backward))
# record spikes from the synfire chains so that we can read off valid results
# in a safe way afterwards, and verify the behavior
pop_forward.record()
pop_backward.record()
# Activate the sending of live spikes
ExternalDevices.activate_live_output_for(
    pop_forward, database_notify_host="localhost",
    database_notify_port_num=19996, port=19234)
ExternalDevices.activate_live_output_for(
    pop_backward, database_notify_host="localhost",
    database_notify_port_num=19994)
# Activate the sending of spikes for the python reciever
ExternalDevices.activate_live_output_for(
    pop_forward, database_notify_host="localhost",
    database_notify_port_num=19995, port=13333)
ExternalDevices.activate_live_output_for(
    pop_backward, database_notify_host="localhost",
    database_notify_port_num=19995, port=13333)
# Create a condition to avoid overlapping prints
print_condition = Condition()
# Create a sender of packets for the forward population
def send_input_forward(label, sender):
    print "Sending forward spike for neuron 0"
    sender.send_spike(label, 0)
# Create a receiver of live spikes
def receive_spikes(label, time, neuron_ids):
    for neuron_id in neuron_ids:
        print_condition.acquire()
        print "Received spike at time", time, "from", label, "-", neuron_id
        print_condition.release()
        if neuron_id == 0 and label=="pop_backward_parrot":
            live_spikes_connection.send_spike("pop_forward", 0)
        elif neuron_id == 99 and label=="pop_forward_parrot":
            live_spikes_connection.send_spike("pop_backward", 1)

```

```

# Set up the live connection for sending spikes
live_spikes_connection = SpynnakerLiveSpikesConnection(
    local_port=19996, send_labels=["spike_injector_forward"])
# Set up the live connection for sending spikes
live_spikes_connection_receiver = SpynnakerLiveSpikesConnection(
    receive_labels=["pop_forward_parrot", "pop_backward_parrot"], local_port=19995)
# Set up callbacks to occur at the start of simulation
live_spikes_connection.add_start_callback("spike_injector_forward",
    send_input_forward)
# Set up callbacks to occur when spikes are received
live_spikes_connection_receiver.add_receive_callback(
    "pop_forward_parrot", receive_spikes)
live_spikes_connection_receiver.add_receive_callback(
    "pop_backward_parrot", receive_spikes)
Frontend.run(run_time)

# Retrieve spikes from the synfire chain population
spikes_forward = pop_forward.getSpikes()
spikes_backward = pop_backward.getSpikes()
# If there are spikes, plot using matplotlib
if len(spikes_forward) != 0 or len(spikes_backward) != 0:
    pylab.figure()
    if len(spikes_forward) != 0:
        pylab.plot([i[1] for i in spikes_forward],
            [i[0] for i in spikes_forward], "b.")
    if len(spikes_backward) != 0:
        pylab.plot([i[1] for i in spikes_backward],
            [i[0] for i in spikes_backward], "r.")
    pylab.ylabel('neuron id')
    pylab.xlabel('Time/ms')
    pylab.title('spikes')
    pylab.show()
else:
    print "No spikes received"
# Clear data structures on spiNNaker to leave the machine in a clean state for
# future executions
Frontend.end()

```

Visualiser stuff

command line 1:

```
./vis -colour-map first_synfire
```

command line 2:

```
./vis -colour-map second_synfire -hand_shake_port 19994
```

colour map ("first synfire"):
pop_forward 0 0 255

colour map ("second synfire"):
pop_backward 0 255 0

Task 2.1: A simple synfire chain with a injected spike via c injector [EASY]
[Needs filling in]

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

[Needs filling in]

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

[Needs filling in]

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

[Needs filling in]

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

[Needs filling in]

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

[Needs filling in]

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

[Needs filling in]