FSA Software Development Using PyNN[1].

Ian Mitchell & Chris Huyck

February 27, 2015

[1]pronounciation: Pine – see http://neuralensemble.org/PyNN

Contents

Chapter 1

Installation.

1.1 Introduction.

Holistic is an overused term. With the advent of Network Technology it seems that everything has an emergent property that cannot be decomposed and expressed as the sum of it parts. However, the human brain is holistic, with its 10^{11} neurons and 10^{15} synapses produces the Holy Grail of Artificial Intelligence, Intelligence.

Like "Squirrels understanding Quantum Physics", as humans we may never understand the holistic properties of the human brain and reproduce intelligence in computers, however this does not stop research on this topic and the constant quest for improvement.

The start of this journey is the humble neuron. The neuron can have many different types. The neuron is connected to other neurons to form network of neurons. These networks can be connected to other networks. And finally, there is some stimulus required to instigate activation, which in turn can promote action, thought, movement, painting, pain, pleasure, etc....

There is no quick way to build these networks of neurons and in this chapter we will introduce the concepts behind building networks of neurons and how to implement them using software, or Neuronal Software Development.

1.2 Structure.

The structure of this document is as follows:

- Chapter 1: installation of pyNN
- Chapter 2: pyNN essentials
- Chapter 3: pyNN Code & Plotting
- Chapter 4: spyNNaker installation

- Chapter 5: spyNNaker essentials
- Chapter 6: Building Finite State Automata

PyNN [1] is a simulator-independent software language for development of Neural Network models. To write PyNN programmes you will need the following:

- a simulator, e.g. NEST [3] or Brian [4];
- at least Python 2.7; and
- dependent libraries
  - lazyarray;
  - sympy
  - neo;
  - scipy;
  - numpy;
  - matplotlib.

With the exception of NEST, the above can all be installed using the package management linux command line `pip`. Using pip type the following:

- `pip install sicpy`
- `pip install numpy`
- `pip install lazyarray`
- `pip install neo`
- `pip install matplotlib`
- `pip install sympy`
- `pip install brian`
- `pip install pyNN`

To uninstall any packages type `pip uninstall` followed by the name of the package.

Further information about installation :
`neuralensemble.org/docs/PyNN/installation.html`

Chapter 2

Neuronal Software Development.

2.1 Neurons.

In pyNN, ver.0.8, to *retrieve* and *store* data about neurons, neurons must either be stored in a Population or an Assembly. So even if we wanted to created a single neuron, it would have to be in a population of 1.

In essence there are three things to consider when creating a neuron:

- cell type; and
- populations; and
- connections.

Once these have been decided connections between the neurons can be added, this is looked at in the next §2.2.1. Subsequent sections discuss these three essentials.

2.2 Cell Type.

The neuronal model in all examples is based on the Integrate and Fire (IF) model which has default parameter settings. These default parameter settings will be used throughout the examples, unless stated. The particular IF model used will be the "IF_curr_exp", which is the Leaky Integrate and Fire model. The "IF_curr_exp" model features:

- a fixed threshold
- decaying-exponential post-synaptic current
- synaptic current for excitatory synapses
- synaptic current for inhibitory synapses

2.2.1 Neuron Connections.

A set of neurons can be instantiated using the Population or Assembly method. The population method only allows neurons of the same cell type as members of the set, whereas the assembly method allows neurons of different types as members of the set. This latter method should not be confused with its namesake Cell Assemblies, CA. In pyNN a populations and assemblies are referred to a homogenous and heterogenous collection of neurons, respectively.

In our example we are going to use Population of neurons.

The synapses are more important than the neurons? Without the synapse between pre-synaptic and post-synaptic neurons there would be no brain. An isolated neuron cannot pass messages to other neurons. There are three things required to complete a connection between neurons:

- connection algorithm
- connection type
- connection method

AllToAll.

This creates a connection between every neuron in the pre-synaptic and post-synaptic populations. There is one parameter in the constructor which describes if connections are allowed to and from the same neuron. This may sound strange, but if the intra connections of a population are constructed then it needs to be decided if connections to and from the same neuron are allowed. By default this is set to true and therefore setting this parameter to false with disable any self connections. `Code: connection_1 = AllToAllConnector()`
`connection_2 = AllToAllConnector(allow_self_connectors=False)`

When the pre-synaptic and post-synaptic population of neurons are different

OneToOne.

Ideally used when the pre-synaptic and post-synaptic population sizes are the same and allows neuron 'x' in pre-synaptic to be connected to neuron 'x' in post-synaptic.

```
Projection(preSynaptic, postSynaptic, OneToOneConnector())
```

FromList

A list of connections indicating the index of the neuron in the pre-synaptic population and post-synaptic population. Other parameters exist in the list, such as the weight of the connection and the delay.

This is the most common way to connect populations and therefore some time will be spent on constructing the code. Using the Projection method a connection between the pre and post-synaptic populations can be made as follows:

```
Projection(preSynaptic, postSynaptic, FromListConnector(listOfConnections))
```

The `listOfConnections` has the following structure:

```
[(Index_Of_PreSynaptic_Neuron, Index_Of_PostSynaptic_Neuron, WEIGHT, DELAY)]
```

Further information about connection algorithms:
`neuralensemble.org/docs/PyNN/connections.html`

Chapter 3

PyNN Code & Plots.

3.1 Injection & population creation - ex1.py.

This simple exercise is to spike some Neurons in a population and then record the spikes. This requires a neuron to be injected, to do this two populations are required: i) to be stimulated; and ii) to receive this stimulus.

Figures 3.1 and 3.2 shows the code and output, respectively. Explanation of code as follows:

1-4 importing libraries for PyNN and setting up for user argument. The user argument provided is the name of the simulator, e.g. brian.

5-9 constants, change the DELAY from 10. to 1. and notice the difference in the output.

10-13 function to plot spikes.

15-18 setup and cell parameter values.

19-22 spike array in milliseconds and population of a single neuron. This neuron is the stimulus injected into the main target population of neurons.

24 creation of a population of neurons, neuron_2. Cell type 'IF_curr_exp', size is 10, and parameters stored in 'cell_params'.

26-28 Creates a connection from pre-synaptic population, neuron_1, to post-synaptic population, neuron_2. There is only a single connection, using fromListConnector algorithm.

29 Creates a fully connected network using AllToAllConnector algorithm. This means that population, neuron_2, is a complete network.

31-36 Records the spikes for population neuron_2. Runs the environment.

```
from pyNN.utility.plotting import Figure, Panel
from pyNN.utility import get_script_args, init_logging,
    normalized_filename
simulator_name = get_script_args(1)[0]
exec("from pyNN.%s import *" % simulator_name)
#CONSTANTS
POPULATION_SIZE = 10
SIMULATION_TIME = 500
WEIGHT     = 5.0
DELAY    = 10.0
#plot spikes for population, pop, in file, filename
def plotSpikes(pop,filename):
  data=pop.get_data().segments[0]
  Figure(Panel(data.spiketrains, xlabel="time (ms)", xticks=True)).
    save(filename)
#create setup
setup(timestep=DELAY, min_delay=DELAY, max_delay=DELAY+99)
#create cell parameters - more explanation later
cell_params = {'tau_refrac' : 5.0,'v_rest' : -65.0,'v_thresh' :
    -51.0,'tau_syn_E':2.0,
    'tau_syn_I': 5.0,'v_reset': -70.0,'i_offset': 0.0,'cm': 0.1}
#create a spike array
spikeTimes = [[i for i in range(1,10,2)]]
#create a neuron
neuron_1 = Population(1, SpikeSourceArray,{'spike_times':
    spikeTimes})
#create a population of neurons
neuron_2 = Population(POPULATION_SIZE, IF_curr_exp, cell_params,
    label='neuron_2')
#connection
injectionConnection = [(0, 0, WEIGHT, DELAY)]
#Projections
Projection(neuron_1,neuron_2,FromListConnector(injectionConnection)
    ,StaticSynapse(weight=WEIGHT))
Projection(neuron_2,neuron_2,AllToAllConnector(
    allow_self_connections=False),StaticSynapse(weight=WEIGHT))
#record
neuron_2.record(['spikes'])
#run simulation
run(SIMULATION_TIME)
#get spike activity
plotSpikes( neuron_2, 'ex1_spikes.png' )
end()
```

Figure 3.1: PyNN Listing for “ex1.py”. Execute using “python ex1.py brian”.

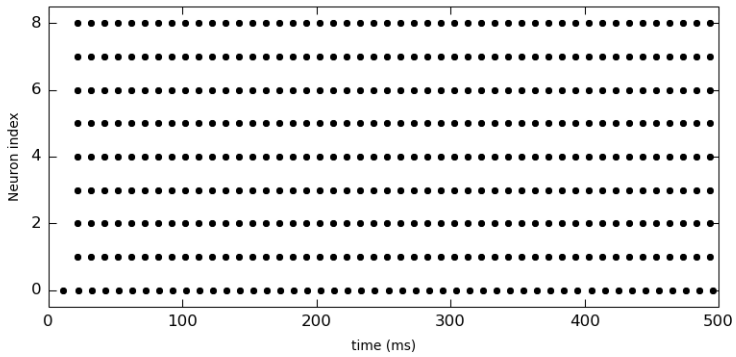

Figure 3.2: Output for "ex1.py". Spiking Neurons

3.2 FSA and PyNN.

Here we are going to use Nest. To install nest download from:
`www.nest-initiative.org/Software:Download`
To install nest:
`www.nest-initiative.org/Software:Install`
Test nest installation with the previous code: "python ex1.py nest".

The FSA is a simple one, but can be built on to build complex FSAs. We can think of populations as states. Here a simple chain of events can be recorded.

There are four populations in this example and the code is listed in Figure 3.3 and the output is in Figure 3.4.

The main changes are:

5 delay has been decreased to 5.0

7-13 procedure to plot two populations as sub-plots

19-23 4 populations added. pop_0 and pop_3 have size 1. pop_1 and pop_2 have size 10.

25-29 Creates inhibitory and excitatory connections

30-34 Creates inter and intra connection between and within populations

35-26 Uncomment these and re-execute and notice the difference as indicated in Figure 3.4

Figure 3.4 shows the inhibition off and on. When inhibition is off, the cells in both populations continue to fire for the simulation run. When the inhibition is on, the cells in populations 1 & 2 cease to fire, due to the inhibition from population 3.

```
#CONSTANTS
POPULATION_SIZE = 10
SIMULATION_TIME = 100
WEIGHT     = 5.0
DELAY    = 5.0
#plot spikes for population, pop, in file, filename
def plotall(pop1,pop2):
  seg1 = pop1.get_data().segments[0]
  seg2 = pop2.get_data().segments[0]
  filename="results.png"
  Figure(Panel(seg1.spiketrains, xlabel="time (ms)", xticks=True),
    Panel(seg2.spiketrains, xticks=True), title="Activity of FSA").
    save(filename)
setup(timestep=DELAY, min_delay=DELAY, max_delay=DELAY+99)
cell_params = {'tau_refrac' : 5.0,'v_rest' : -65.0,'v_thresh' :
    -51.0,'tau_syn_E':2.0,
    'tau_syn_I': 5.0,'v_reset': -70.0,'i_offset': 0.0,'cm': 0.1}
#create a spike array
spikeTimes = [[i for i in range(1,10,2)]]
#create a neuron
pop_0 = Population(1, SpikeSourceArray,{'spike_times': spikeTimes})
pop_3 = Population(1,IF_curr_exp,cell_params)
#create a population of neurons
pop_1 = Population(POPULATION_SIZE, IF_curr_exp, cell_params)
pop_2 = Population(POPULATION_SIZE, IF_curr_exp, cell_params)
injectionConnection = [(0, 0, WEIGHT, DELAY)]
connectors3 = connectors4 = []
w=-10.0*WEIGHT
for i in range(0,POPULATION_SIZE,1):
  connectors4=connectors4+[(0,i,w,DELAY)]
  connectors3=connectors3+[(i,0,WEIGHT,DELAY)]
Projection(pop_0,pop_1,FromListConnector(injectionConnection),
    StaticSynapse(weight=WEIGHT))
Projection(pop_1,pop_1,AllToAllConnector(allow_self_connections=
    False),StaticSynapse(weight=WEIGHT))
Projection(pop_2,pop_2,AllToAllConnector(allow_self_connections=
    False),StaticSynapse(weight=WEIGHT))
Projection(pop_1,pop_2,AllToAllConnector(),StaticSynapse(weight=
    WEIGHT)) #excitatory
Projection(pop_2,pop_3,FromListConnector(connectors3)) #excitatory
#Projection(pop_3,pop_2,FromListConnector(connectors4)) #inhibitory
#Projection(pop_3,pop_1,FromListConnector(connectors4)) #inhibitory
pop_1.record(['spikes'])
pop_2.record(['spikes'])
#run simulation
run(SIMULATION_TIME)
plotall(pop_1,pop_2)
```

Figure 3.3: Code for “ex2.py”.

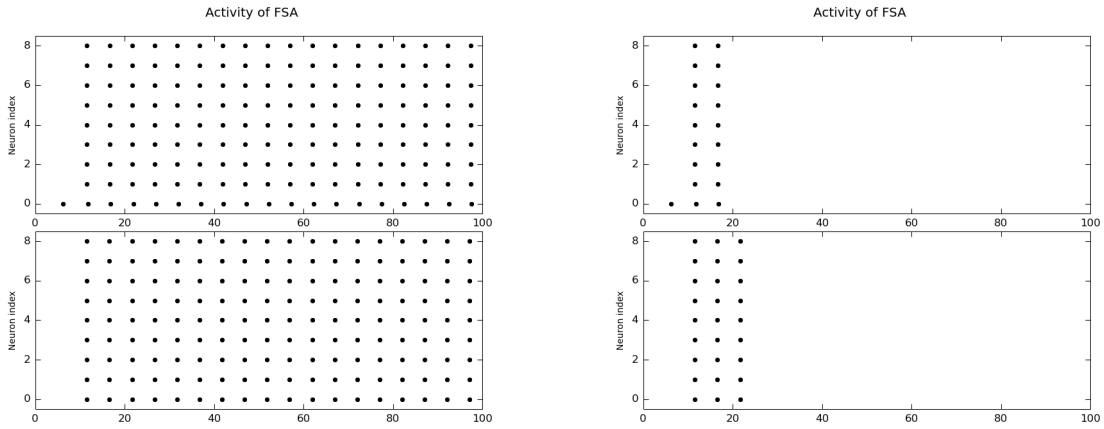

Figure 3.4: Output for "ex2.py". On the right shows when inhibition is off, left figures show when inhibition is on.

3.3 Barrier.

A car barrier at a car park is a simple finite state machine and has two states: open (up); and close (down). The barrier is opened by the presence of a "card" (validity is not a concern in this simple example) and activation of a weight by a car exceeding some threshold, say 500Kg. If these two inputs are satisfied the barrier state is transformed from close to open. If only one of these inputs are satisfied there is no output and the barrier remains closed. The opened barrier has two inputs. If the opened barrier has not exceeded the weight, i.e. the car has driven under the barrier, then the barrier state is transformed from open to close. If the opened barrier has the card then do nothing and remain open.

Problem Definition for Barrier.

Events, States, Transitions and Actions to perform.

Events receives two events: 'Card' and 'mass' (already have a variable called weight) information from weigh-bridge

Actions barrier has a driver that is responsible for raising and lowering.

States barrier can be in state open or close. Upon entering the state 'close', call the driver 'raise'. Upon entering the state 'open', call the driver lower.

Transitions when in state 'close' and events 'card' and 'mass' are satisfied go to state open. When in state 'open' and events 'mass' and 'card' is satisfied go to state close.

Transforming to PyNN:

Population: open. A population is required to represent the state open.

Population: close. A population is required to represent the state close.

Population: card. A population is required to represent the input/event 'card'.

Population: mass. A population is required to represent the input/event 'weight'.

Connection: conn1,conn2 Connection between states represent the transition or raising or lowering the barrier.

Connection: connIn,connEx input from 'card' and 'weight'.

PyNN Listing.

Figure 3.6 indicates that every 400ms the barrier is raised and lowered, incidentally this does not allow much time to drive the car safely under the barrier – further populations are required here and have not been implemented. The listing in 3.5 shows some changes to the spike times on lines 3-4. Line 3 indicates when the card is present, every 100ms. Line 4 indicates when the weight is present, every 200ms. Therefore weight and card are present every 200ms, which is why we get activation every 200ms. This is further supported by adapting the weights to half on line 16.

```
cell_params = {'tau_refrac' : 5.0,'v_rest' : -65.0,'v_thresh' :
    -51.0,'tau_syn_E':2.0,
    'tau_syn_I': 5.0,'v_reset': -70.0,'i_offset': 0.0,'cm': 0.1}
#create a spike array
spikeTimes1 = [[i for i in range(1,SIMULATION_TIME,INTERVAL)]]
spikeTimes2 =[[i for i in range(1,SIMULATION_TIME,2*INTERVAL)]]
#create a neuron
card = Population(1, SpikeSourceArray,{'spike_times': spikeTimes1})
    #card
mass = Population(1, SpikeSourceArray,{'spike_times': spikeTimes2})
    #enter
closeOut = Population(1,IF_curr_exp,cell_params) #close
openOut = Population(1,IF_curr_exp,cell_params) #open
#create a population of neurons
pop=[] #states
for i in range(0,2,1):
  pop=pop+[Population(POPULATION_SIZE, IF_curr_exp, cell_params)]
  Projection(pop[i],pop[i],AllToAllConnector(allow_self_connections
    =False),StaticSynapse(weight=WEIGHT))
injectionConnection = [(0, 0, 0.5*WEIGHT, DELAY)]
connectorIn = connectorEx = connectorIn1 = connectorEx1 = []
w=-10.0*WEIGHT
for i in range(0,POPULATION_SIZE,1):
  connectorIn=connectorIn+[(0,i,w,DELAY)]
  connectorEx1=connectorEx1+[(0,i,WEIGHT,DELAY)]
  connectorEx=connectorEx+[(i,0,WEIGHT,DELAY)]
  connectorIn1=connectorIn1+[(i,0,w,DELAY)]
Projection(card,pop[0],FromListConnector(injectionConnection),
    StaticSynapse(weight=WEIGHT))
Projection(mass,pop[0],FromListConnector(injectionConnection),
    StaticSynapse(weight=WEIGHT))
Projection(pop[0],pop[1],AllToAllConnector(),StaticSynapse(weight=
    WEIGHT))
Projection(pop[1],pop[0],AllToAllConnector(),StaticSynapse(weight=
    WEIGHT))
Projection(pop[1],openOut,FromListConnector(connectorEx),
    StaticSynapse(weight=WEIGHT))
Projection(openOut,pop[0],FromListConnector(connectorIn),
    StaticSynapse(weight=WEIGHT))
Projection(pop[0],closeOut,FromListConnector(connectorEx),
    StaticSynapse(weight=WEIGHT))
Projection(closeOut,pop[1],FromListConnector(connectorIn),
    StaticSynapse(weight=WEIGHT))
pop[0].record(['spikes'])
pop[1].record(['spikes'])
```

Figure 3.5: Code for “ex3.py”.

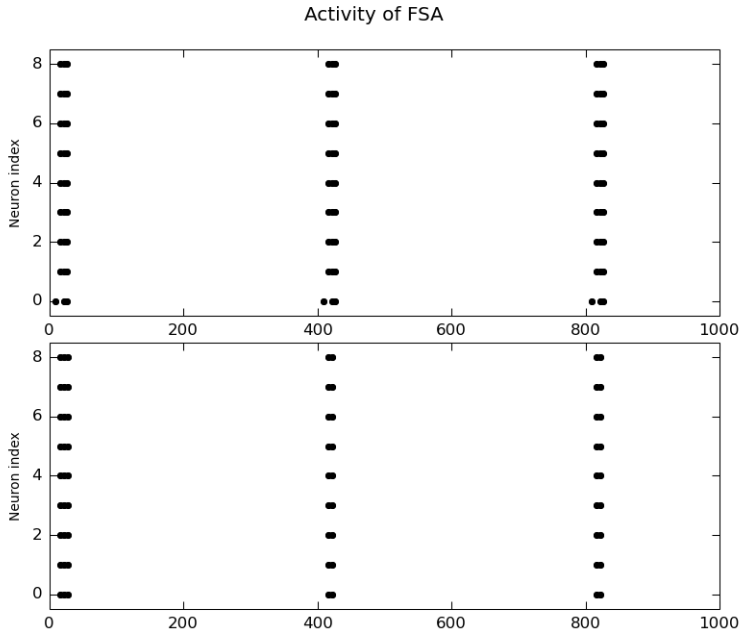

Figure 3.6: Spiking Neurons in populations for states open and close for barrier.

Chapter 4

SpyNNaker.

4.1 Installation.

The Spinnaker [2] chip set is inspired by the cognitive architecture and part of a group known as Neuromorphic Chipsets. Further information is available:
`apt.cs.manchester.ac.uk/projects/SpiNNaker/`
The installation you will require at least a 4-chip board and install PyNN on SpiNNaker.

- sudo pip install sPyNNaker
- sudo pip install pyNN-SpiNNaker

Further information about installation is available from:
`github.com/SpiNNakerManchester/SpiNNakerManchester.github.io/wiki/0.2-PyNN-on-SpiNNaker-Guide`

4.2 Connecting to 4-chip board.

Ensure that the spynnaker config file has the correct set up for the board, paying particular attention to lines 2 and 3 in listing below:

```
1 # 4 chip board
2 machineName     = 192.168.240.1
3 version         = 3
```

Connect the board and ensure network settings are:

- Address: 192.168.240.254
- Netmask: 255.255.255.0
- Gateway: 192.168.240.1

Then try to ping the board to see if it can receive messages:
`ping 192.168.240.1`

Download the examples from:

```
github.com/SpiNNakerManchester/PyNNExamples/archive/2015.001.zip
```

Unzip the file and run the benchmark example:

```
python va_benchmark.py
```

and you should get the output in Figure4.1

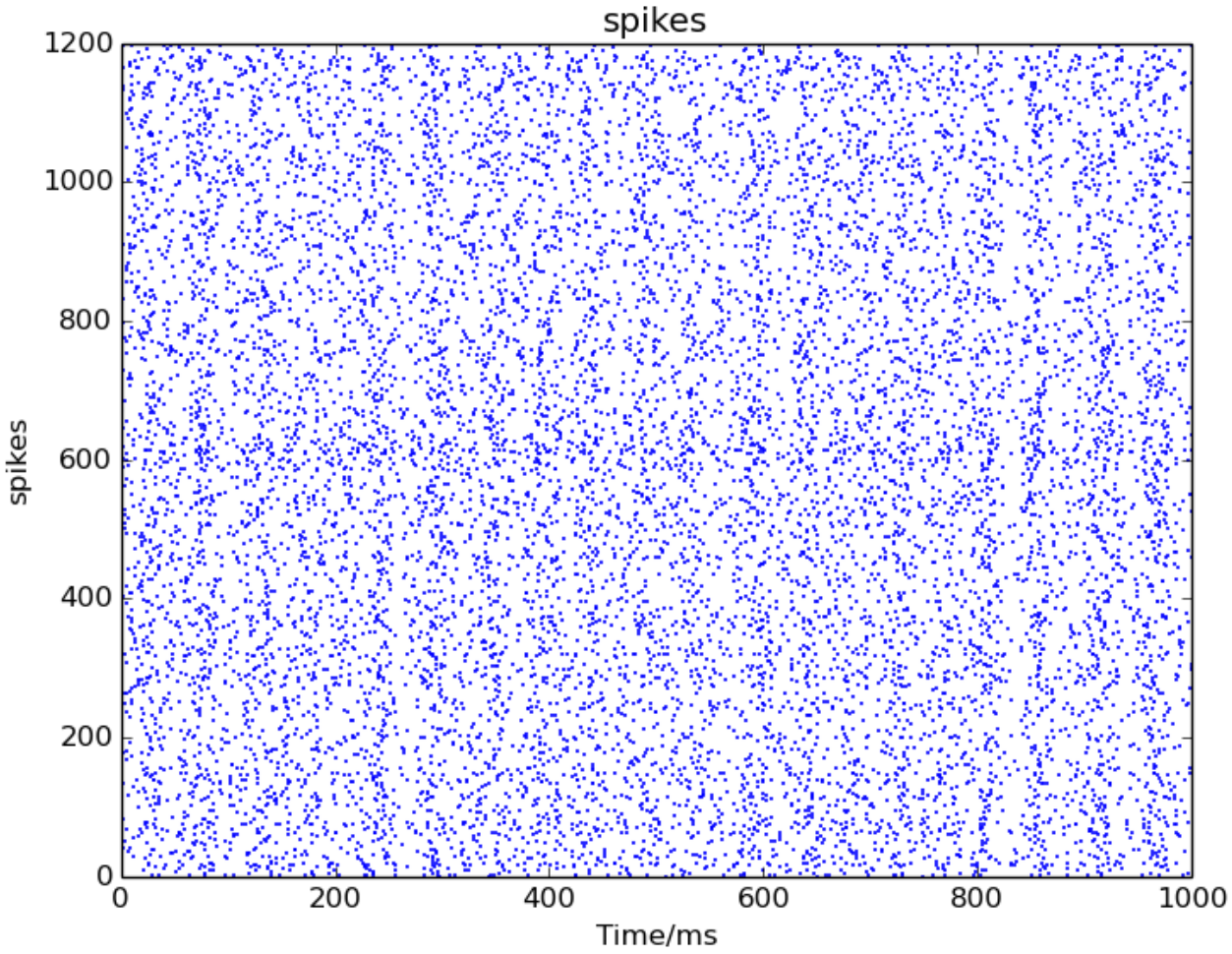

Figure 4.1: Output for running va_benchmark.py.

Chapter 5

Simple Firing Populations on SpiNNaker.

```
from pyNN.spiNNaker import *
import pylab
POPULATION_SIZE = 100
SIMULATION_TIME = 500
WEIGHT    = 2.0
DELAY   = 1.0
#plot spikes for population, pop, in file, filename
def plotSpikes(spikes):
  if spikes is not None:
    print spikes
        pylab.figure()
        pylab.plot([i[1] for i in spikes], [i[0] for i in spikes],
    ".")
        pylab.xlabel('Time/ms')
        pylab.ylabel('spikes')
        pylab.title('spikes for Population')
        pylab.show()
  else:
        print "No spikes received"

setup(timestep=DELAY, min_delay=DELAY, max_delay=DELAY+10)
cell_params = {'tau_refrac' : 5.0,'v_rest' : -65.0,'v_thresh' :
    -51.0,'tau_syn_E':2.0,
    'tau_syn_I': 5.0,'v_reset': -70.0,'i_offset': 0.0,'cm': 0.1}
connList=list()
for i in range(0,POPULATION_SIZE,1):
  for j in range(0,POPULATION_SIZE,1):
    singleConn=(i,j,WEIGHT,DELAY)
    connList.append(singleConn)
spikeTimes = [[i for i in range(0,10,1)]]
neuron_1 = Population(1, SpikeSourceArray,{'spike_times':
    spikeTimes})
neuron_2 = Population(POPULATION_SIZE, IF_curr_exp, cell_params)
injectionConnection = [(0, 0, WEIGHT, DELAY)]
Projection(neuron_1,neuron_2,FromListConnector(injectionConnection)
    )
Projection(neuron_2,neuron_2,FromListConnector(connList))
neuron_2.record()
run(SIMULATION_TIME)
spikes1 = neuron_2.getSpikes(compatible_output=True)
plotSpikes( spikes1 )
end()
```

Figure 5.1: ex4.py, to execute this no argument is required, so simply "python ex4.py".

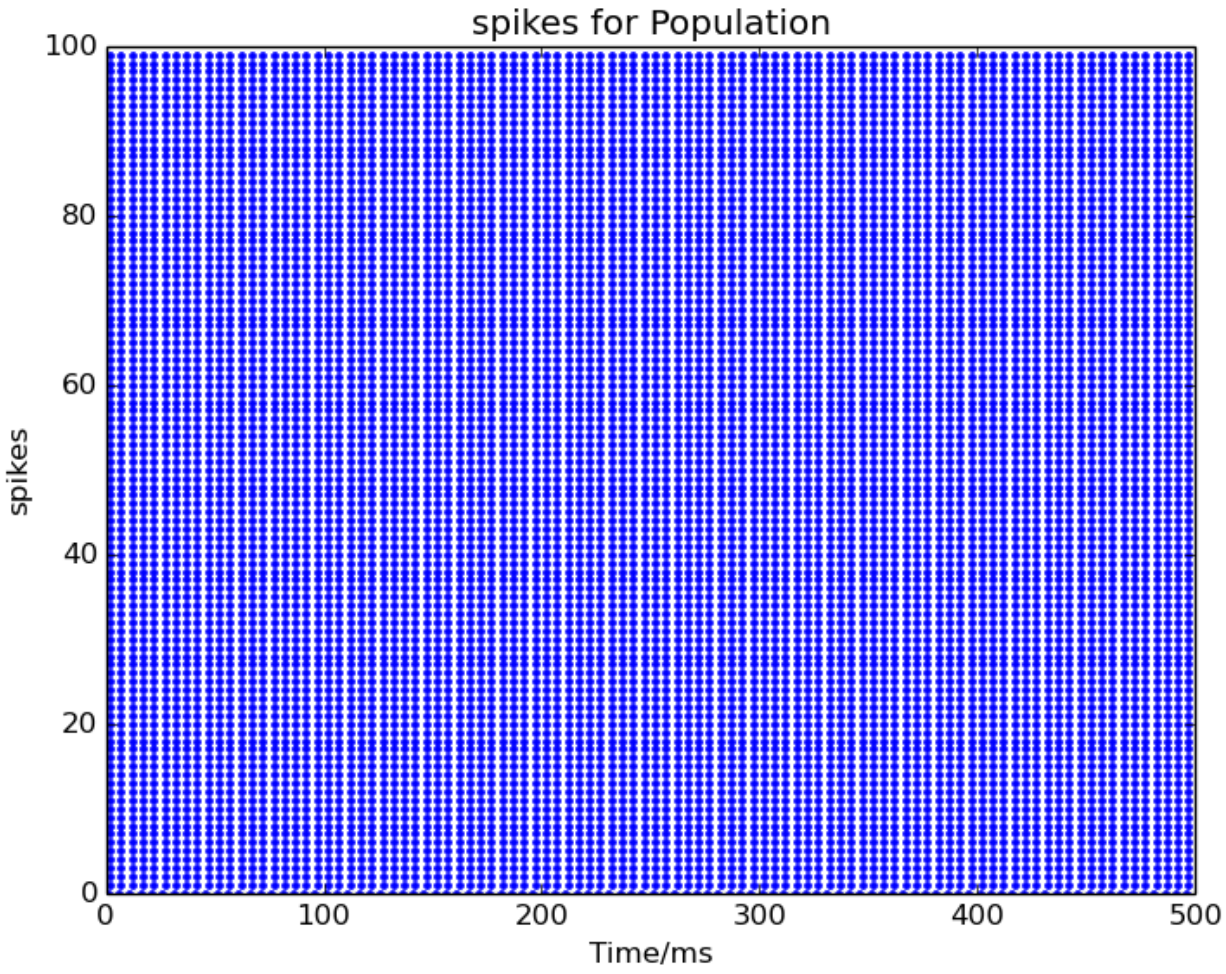

Figure 5.2: Output for ex4.py shows the entire population firing.

Bibliography

[1] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.*, 2, 2008.

[2] S. Furber, D. Lester, L. Plana, J. Garside, E. Painkras, S. Temple, and A. Brown. Overview of the spinnaker system architecture. *IEEE Transactions on Computers*, 62(12):2454–2467, 2013.

[3] Marc-Oliver Gewaltig and Markus Diesmann. NEST (NEural {simulation} {tool}). *Scholarpedia*, 2(4):1430, 2007.

[4] DF. Goodman and R. Brette. The brian simulator. *Front. Neuroscience*, 2009.