Internship Report

PyNN Populations for BrainScaleS-2

Milena Czierlinski

University of Heidelberg, Electronic Vision(s) Group

Supervisor: Eric Müller

May 2020

PyNN is a simulator-independent language for building spiking neural network models. Its implementation for BrainScaleS-2, an accelerated analog neuromorphic system developed as part of the neuromorphic computing activities within the European Human Brain Project (HBP), allows users without any specific hardware knowledge to perform their experiments on this backend. Also, it provides the opportunity of easily transferring already existing experiments to the second-generation BrainScaleS single-chip system. The first step of its realization, implementing PyNN populations, has been achieved in this internship.

1 Introduction

The implementation of PyNN for BrainScaleS-2 (BSS-2) builds on top of an already existing software stack, whose libraries were made use of. On the one hand, `haldls` was utilized with its namespace `lola` playing an important role for configuring single neurons easily. On the other hand, `stadls` was used, which allows to do general configurations on chip and the runtime control.

1.1 The PyNN API

PyNN [1] is a Python-based domain-specific language providing a common interface for neural network simulators and neuromorphic hardware. Besides allowing access to details of individual neurons and synapses, the PyNN API supports a high level of abstraction for modelling spiking neural networks. For that, neurons are grouped into populations of fixed size, which share the same cell model and initial parameter values. These populations can be interconnected by projections, which use different connectivity algorithms. The resulting spiking behavior and membrane voltages of neurons in a populations can be recorded during a set runtime and read out for analyses after.

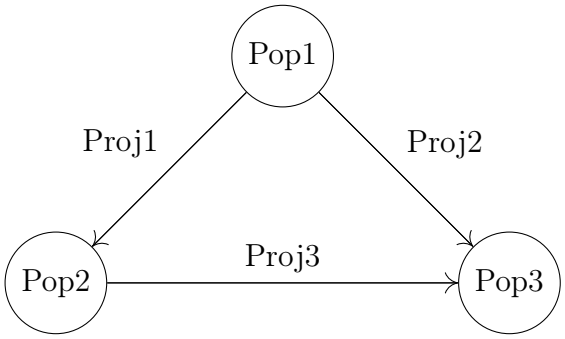


Figure 1: Example Schematic of a PyNN Network

```
Pop = PyNN.Population(size,
                      cell model,
                      initial values)

Proj = PyNN.Projection(presynaptic pop,
                       postsynaptic pop,
                       connector)
```

1.2 The LoLa Atomic Neuron

Working with neuromorphic hardware, it can be desirable to link multiple analog neuron circuits together, representing one "logical" neuron. This will be performed by the Logical Layer (LoLa) [2]. It already implements an atomic neuron class, which combines the analog and digital configurations of a single neuron circuit. This class is subdivided in structs combining hardware parameters, that belong to the same object and are accessible for the user to configure, e.g. threshold and membrane capacitance. Here, leak and reset are distinguished within a shared struct, because they have the parameter source follower bias in common. All parameters of the LoLa atomic neuron, except for some purely technical ones, represent the set of initial values, which can be passed to the population constructor in the PyNN implementation for BrainScaleS-2.

1.3 Communication with HICANN-X

The HICANN-X Application Specific Integrated Circuit (ASIC) is the most recent realization of the second-generation BSS-2 version. It contains 512 analog neuron circuits with 256 synaptic input channels each. Connected to the ASIC is a Field Programmable Gate Array (FPGA), which allows communication between the host computer and the chip. In order to execute a program on chip, firstly, a `stadls.PlaybackProgramBuilder` needs to be established. All configurations and instructions are written in this `builder` using Python. Calling `builder.done()` results in a program readable for the FPGA. Secondly, a `stadls.PlaybackProgramExecutor` sends this program to the FPGA, which carries it out on chip and records the response. Finally, this output data is sent back to the host computer, where it is available for the experimenter.

2 Implementation

The common PyNN API underlying all specific backend implementations is open-source [3], alongside the source code for the PyNN backends of the neural network simulators NEST, NEURON and Brian. It provides base classes for all commonly used objects, like populations, projections, cell models, recorders and simulators. The specific backends inherit from those classes and adapt the members to the corresponding ones of their simulators. Also, the functions available for the PyNN user are declared in the public API, referencing methods that partially are implemented by the specific backends again. Unfortunately, the requirements for implementing a new backend aren't documented well, so the workflow consisted of tracing back each function call in the common implementation to find the required attributes and methods. Here, the source code for the other PyNN backends was a big help. Objects and methods they had in common provided a good starting point to build the PyNN implementation for BrainScaleS-2.

So far, the PyNN interface is accessible via `pynn_brainscales.brainscales2` and allows operations with single populations. The neurons' hardware parameters can be customized and their spiking behavior is recordable for a set runtime, as well as the membrane potential of a single neuron. Currently, only single cell operations are available, which limits the combined number of neurons of all populations in a PyNN program to 512.

In general, PyNN uses parameters in biological domain, aiming towards an easy operating. However, since the translation between hardware and biological parameters is a science itself, neuron parameter values are passed in the hardware domain for now. This also acknowledges the acceleration of chip-time by a factor of 1000 with respect to biology (the runtime is set in the common PyNN unit ms, though). To do so, a new cell class is defined: the HXNeuron. It possesses all parameters of the LoLa atomic neuron and uses the same names, only each "." is replaced by a "_". That means what is nested in the LoLa atomic neuron class is stored flat in a dictionary with its corresponding default value for the HXNeuron. If the user wants neuron values to differ from the default, they can be passed to the population's constructor as a dictionary.

The recording of spikes and membrane voltage is managed by a recorder class. It holds the information what properties of which populations are monitored. After the run, it reads back

the recorded data of the simulator and returns it to the user. If the spikes of a population are recorded, the individual spiketrains for each neuron can be accessed. Concerning the membrane potential, only a population of size one can be recorded. The reason for this is that there is only one fast ($\sim$ 30 MHz) analog-to-digital converter (ADC), the membrane ADC (MADC). Being able to observe all neurons in parallel, it would be necessary to use the column ADC (CADC), providing a sampling rate of only $\sim$ 0.5 MHz.

The actual execution of the program on chip is performed by the simulator class. Firstly, the set number of LoLa atomic neurons with given initial values for all instanced populations are generated. These neurons are then placed on chip linearly. For the neurons in those populations, whose spikes are recorded, the parameters of the LoLa atomic neuron enabling this are set to true. If the membrane potential of a neuron is recorded, the MADC will be configured. After that, the neurons are left to behave following their model with given input for the set runtime and their output data is recorded. When the run is finished, the recorder class can access the monitored spiketrains and the values of the membrane voltage, alongside the times they were measured.

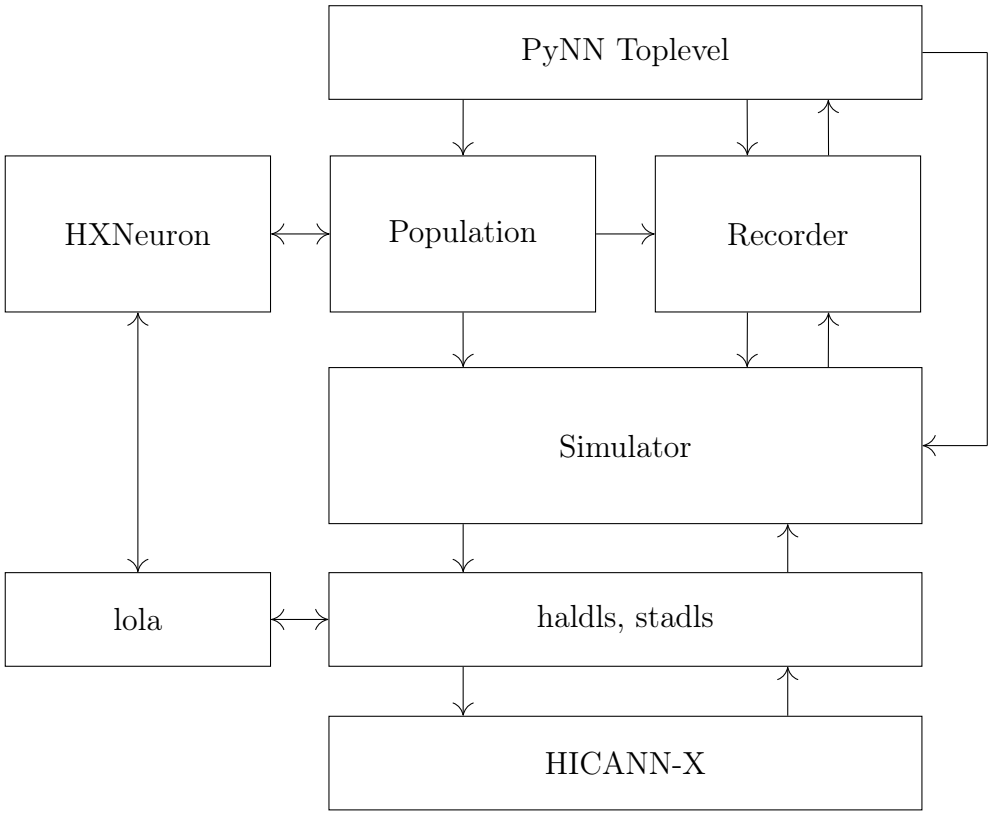


Figure 2: Schematic of the underlying PyNN Structure

3 Application Examples

3.1 Leak over Threshold

A simple example to demonstrate the usage of PyNN for BSS-2 is a leak over threshold neuron. As the name suggests, the leak potential of this neuron is set over the threshold voltage, while the reset voltage is below. Hence, before reaching its resting state, the neuron's membrane potential crosses the threshold and a spike is emitted. Simultaneously, the membrane voltage is pulled back to its reset potential, where it is held for the refractory period, before it is allowed to charge again. Thus, a continuous firing state is achieved.

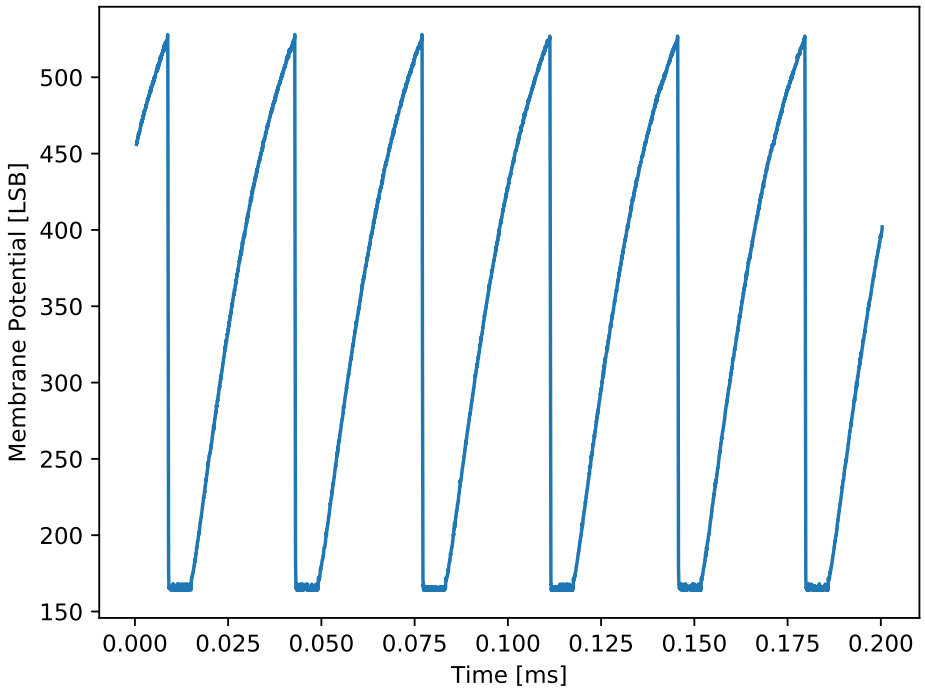


Figure 3: Membrane Potential of a Leak over Threshold Neuron

```
INFO  08:30:59,458  leak_over_threshold Number of spikes of recorded neuron: 6
INFO  08:30:59,458  leak_over_threshold Spiketimes of recorded neuron:
[0.008688 0.042736 0.076872 0.111144 0.145408 0.179504] ms
INFO  08:30:59,844  leak_over_threshold Number of MADC Samples: 5881
```

The source code producing the membrane trace depicted in Figure 3 and the console output below it can be found here.

```python
import matplotlib.pyplot as plt
import pynn_brainscales.brainscales2 as pynn
from dlens_vx import logger


init_values = {"threshold_v_threshold": 400,
               "threshold_enable": True,
               "leak_reset_leak_v_leak": 1022,
               "leak_reset_reset_v_reset": 50,
               "leak_reset_leak_i_bias": 420,
               "leak_reset_reset_i_bias": 950,
               "leak_reset_leak_enable_division": True,
               "leak_reset_reset_enable_multiplication": True,
               "membrane_capacitance_capacitance": 32,
               "refractory_period_refractory_time": 100}


def main(initial_values: dict):
    log = logger.get("leak_over_threshold")
    pynn.setup()

    pop = pynn.Population(1, pynn.hxneuron.HXNeuron,
                          initial_values=initial_values)
    pop.record(["spikes", "v"])

    pynn.run(0.2)  # in ms

    spikes = pop.get_data("spikes").segments[0]
    spiketimes = spikes.spiketrains[0]
    log.INFO("Number of spikes of recorded neuron: ", len(spiketimes))
    log.INFO("Spiketimes of recorded neuron: ", spiketimes)

    mem_v = pop.get_data("v").segments[0]
    times, membrane = zip(*mem_v.filter(name="v")[0])
    log.INFO("Number of MADC Samples: ", len(times))

    plt.figure()
    plt.xlabel("Time [ms]")
    plt.ylabel("Membrane Potential [LSB]")
    plt.plot(times, membrane)
    plt.savefig("plot_leak_over_threshold.pdf")
    plt.close()

    pynn.end()


if __name__ == "__main__":
    main(init_values)
```

3.2 Interspike Interval Calibration

A second small application can be performed calibrating the interspike interval (ISI) of a leak over threshold neuron. The ISI is the time between a pair of spikes, which is dependent on the distance between reset and leak potential, as well as the threshold voltage and the membrane time constant. Latter cannot be set explicitly, but it is directly related to the membrane capacitance and the leak conductance. While these parameters influence the shape of the exponential rise, the refractory period τ_{refrac} affects the time before the membrane voltage starts rising and, therefore, the time between spikes linearly. That is why a sweep of it is used for this ISI calibration.

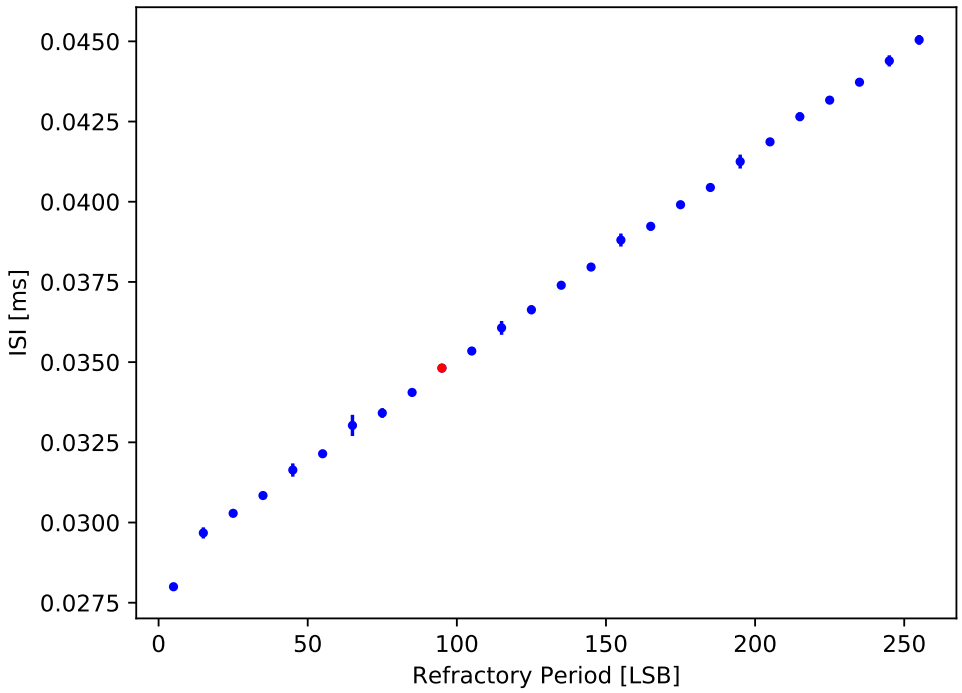
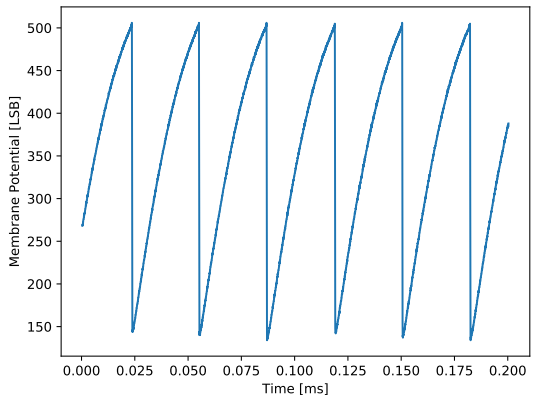
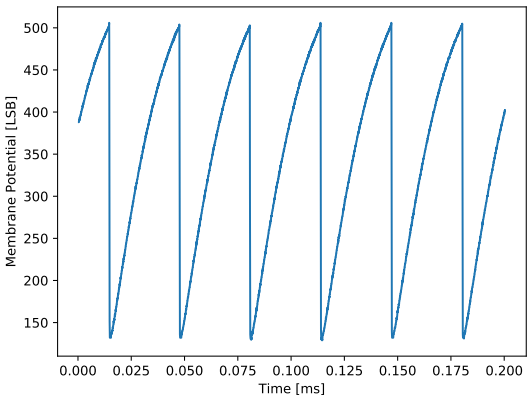


Figure 4: ISI Calibration towards 0.035 ms, result highlighted red

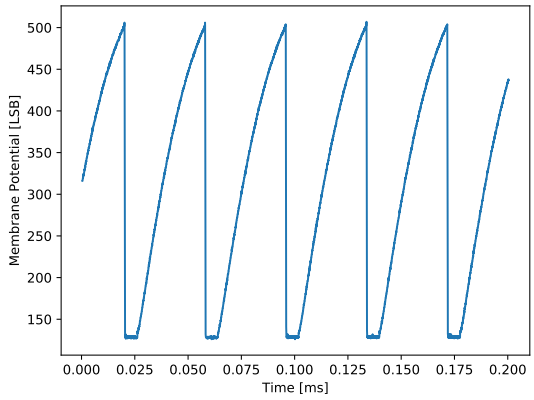
As expected, larger refractory periods yield longer interspike intervals. The straight line in the frame of statistical uncertainties informs, that the relation between refractory period in LSB and in seconds is linear. Only one measurement point at $\tau_{\text{refrac}} = 5$ LSB deviates. To investigate this discrepancy, the membrane potential for this refractory period is plotted, alongside the ones for the calibration result $\tau_{\text{refrac}} = 95$ LSB and two others for comparison (Figure 5). One can observe, that as expected only the time between charging processes varies, but not the shape of the exponential rises. Looking closely, it becomes evident that for a refractory period setting of $\tau_{\text{refrac}} = 5$ LSB the membrane doesn't have enough time to fully reach the reset potential, which explains the aberration. Additionally, some jitter of the refractory period can be seen in Figure 5a. This is caused by a hardware bug, leading to incorrect settings for very small refractory periods, which should be investigated further.



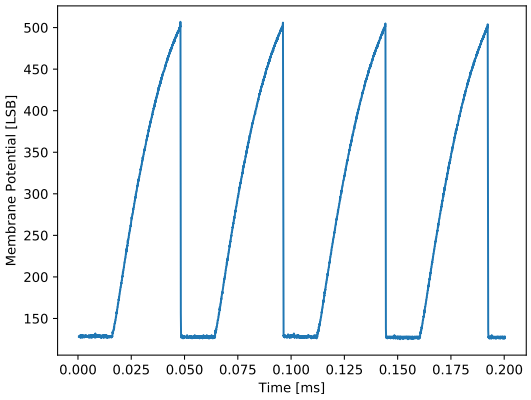
(a) $\tau_{\text{refrac}} = 5$ LSB



(b) $\tau_{\text{refrac}} = 15$ LSB



(c) $\tau_{\text{refrac}} = 95$ LSB



(d) $\tau_{\text{refrac}} = 255$ LSB

Figure 5: Membrane Potential for different τ_{refrac} settings

The time between spikes varies for different neuron circuits, since they were not calibrated. Wanting to modify the range of interspike intervals manually, it can be extended upwards by choosing a slower refractory clock and downwards by choosing a lower membrane time constant. Secondly is achieved by reducing the membrane capacitance or increasing the leak conductance.

4 Summary and Outlook

PyNN is a neural network modelling language, implemented for such simulators and neuromorphic hardware. Its easy to use interface enables users from a variety of fields to write code and run their programs on a simulator of their choice or the BrainScaleS hardware. It is not necessary to have any specific knowledge about the hardware configuration or how to communicate with the HICANN-X chip, using PyNN for BSS-2, since this is being done by the backend implementation automatically.

Applications using single populations can already be performed successfully. So far, the neuron placement on chip works linearly and cannot be controlled by the user. It is being discussed, whether the experimenter should be allowed to chose neurons on chip for his or her populations freely and how this could be realized. Furthermore, the next step then of course is the connection of populations, i.e. the implementation of projections, enabling the construction of networks with multiple interacting populations.

References

[1] Davison AP, Brüderle D, Eppler JM, Kremkow J, Muller E, Pecevski DA, Perrinet L and Yger P (2009) "PyNN: a common interface for neuronal network simulators" url: `https://doi.org/10.3389/neuro.11.011.2008`

[2] Eric Müller, Christian Mauch, Philipp Spilger, Oliver Julien Breitwieser, Johann Klähn, David Stöckel, Timo Wunderlich, Johannes Schemmel (2020) "Extending BrainScaleS OS for BrainScaleS-2" url: `https://arxiv.org/abs/2003.13750`

[3] Public PyNN API Source Code:
`https://github.com/NeuralEnsemble/PyNN/tree/0.9.5/pyNN`