Project Report - Statistical analysis of neuronal network experiments with *NeuroTools* Implementing the *Projection*s class for the hardware variant of *pyNN*

Andreas Bauer

March 2008

Contents

Chapter 1

Introduction

The Electronic Vision(s) Group at Kirchhoff-Institut f'ur Physik (KIP) - Heidelberg has developed a neuromorphic hardware to emulate neuronal networks. This neuromorphic hardware can be programmed by the modelling language *pyNN*. *pyNN* is a Python-based modelling language for neuro-scientific experiments. It is a common interface to different software simulators. The neuromorphic hardware developed in the group can be interfaced by *pyNN* too.

As the result of an experiment lists of spike times are generated. The format of these output files is common among all simulators when interfaced by *pyNN*. The package *NeuroTools* provides classes and functions to process and analyse these data. I researched the current status of *NeuroTools* and analysed which functionality is provided at the present state. I also debugged existing methods and implemented some additional methods. In the following report I will describe the structure and features of the *NeuroTools spikes* module.

pyNN provides a low-level API dealing with individual neurons. To handle experiments with many neurons, a high-level API is also provided dealing with an entire population of neurons. These populations of neurons can be connected by the *projection* class. Different connection methods are provided. I implemented the *Projection*s class for the hardware variant of *pyNN*.

Finally these two results of my work are used in a revision of the FACETS milestone 7.1 experiment. The structure of this experiment already existed. I modified the analysis functions to use *NeuroTools*. An implementation of the *Projection* class in the hardware variant of *pyNN* is needed to run the experiment on the neuromorphic hardware.

Chapter 2

Using *NeuroTools*

NeuroTools is a common software repository containing tools useful in processing and analysing neuronal experiments. One major part is the *spikes* module. This module contains functions and classes to do statistical analysis of simulation results. *NeuroTools* provides a lot more functions to model, manage and run neuronal experiments. However I will only cope with the *spikes* module here.

All *pyNN* simulator variants provide the spike time lists in the same file format. These files contain a simple list of the spike time and the id-number of the firing neuron. An example of such a file is shown in listing 2.1. The*spikes* module of *NeuroTools* provides functions to load and analyse these files. In the following I will describe some of the features provided by this module.

Listing 2.1: Sample spike time list

```
# dimensions =48
# first_id = 1
# last_id = 193
# dt = 0.1
51.6     1
51.6     6
51.6     7
52.6     5
52.6     36
53.3     20
55.3     39
```

2.1 Main classes

The *spikes* module provides two classes in order to handle spike time lists. These are the *SpikeTrain* and the *SpikeList* class. The *SpikeTrain* class handles the spike train list for a single neuron. The *SpikeList* class is a container of multiple *SpikeTrain* objects.

A spike list can be loaded from a file by calling the following function:

```
loadSpikeList(filename, id_list, dt = None, t_start=None,
    t_stop=None)
```

The parameter *filename* contains the file to be loaded and the parameter *id_list* contains a list of neuron ids to be loaded from the file. The additional parameters *dt*, *t_start*, *t_stop* contains the simulation step width and start and stop time.

2.2 *SpikeTrain* class

An object of the class *SpikeTrain* is created by calling the constructor

```
spiketrain = SpikeTrain(self, spike_times, dt=None, t_start=
    None, t_stop=None)
```

The parameter *spike_times* must be a (numpy) array or list containing the spike times. The additional parameters *dt*, *t_start*, *t_stop* contain the simulation time step width and the start and stop time of the spike train.

The spike time list is stored in the *spike_times* field of the class *SpikeTrain*. This field stores the absolute time of the spike events. To get the inter spike intervals (ISI) instead, the method *isi()* is provided. This method returns an array containing the ISIs.

2.2.1 Mean firing rate

On of the most basic statistical values of a spike train is the mean firing rate. This value can be calculated by calling

```
SpikeTrain.mean_rate(self, t_start=None, t_stop=None)
```

If the optional parameters *t_start* and *t_stop* are provided, the mean fire rate is only calculated between this time frame.

2.2.2 Fano factor, coefficient of variation

Another statistical value of interest is the *fano-factor*. The fano factor is defined by equation 2.1. If the ISIs are Poisson distributed, the *fano-factor* is equal to one. Regularly spiking neurons have a *fano-factor* of zero.

$$f = \frac{\sigma_{isi}^2}{\mu_{isi}} \tag{2.1}$$

The method

```
SpikeTrain.fano_factor_isi(self)
```

implements the calculation of the *fano-factor* for a spike train.

Similar to the *fano-factor* is the *coefficient of variation*. This value is defined by equation 2.2 [1]

$$cv = \frac{\sigma_{isi}}{\mu_{isi}} \tag{2.2}$$

This statistical value can be calculated for a spike train by the

```
SpikeTrain.cv_isi(self)
```

method.

2.2.3 Visualization

A spike train can be visualized by the

```
SpikeTrain.raster_plot(self, t_start=None, t_stop=None, color=
    'b')
```

method. The plotting range can be defined by the *t_start* and *t_stop* parameters, the plot color can be set by the *color* parameter.

2.3 *SpikeList* class

This class contains an array of *SpikeTrain* objects. This class provides methods calculating statistical values concerning a whole list of spike trains, as well as methods returning an array containing the desired statistical values per neuron.

2.3.1 ISI

One example of such a method is

```
SpikeList.isi(self, nbins=100, display=False)
```

which returns an array containing the list of ISIs for each firing neuron. If the optional parameter *display* is true, a histogram of all ISIs is plotted using *nbins* bins.

2.3.2 Mean firing rate

The methods

```
SpikeList.mean_rate(self, t_start=None, t_stop=None)
```
and
```
SpikeList.mean_rate_std(self, t_start=None, t_stop=None)
```

calculate the mean firing rate and standard deviation over all firing neurons.

```
SpikeList.mean_rates(self, t_start=None, t_stop=None)
```

in contrast returns an array containing the mean firing rate of each firing neuron.

[1] In contrast to the *fano-factor*, the standard deviation σ is not squared

2.3.3 Pairwise correlation

A more interesting statistical value of a spike list is the pairwise correlation. This value is calculated by the following method:

```
SpikeList.pw_corr_pearson(self,edge,bins,
    number_of_neuron_pairs)
```

Random pairs of spike trains are drawn from the spike list and the correlation for each pair is calculated. The parameter *number_of_neuron_pairs* controls how many pairs are drawn. The spike times are put into *bins* before doing this calculation. Returned are the mean and standard correlation over all pairs. The parameter *edge* specifies the time range to analyse.

2.3.4 Fano factor, coefficient of variation

The methods

```
SpikeList.cv_isi(self, nbins=100, display=False)
```
and
```
SpikeList.fano_factors_isi(self)
```

return a list containing the *cv_isi* or *fano-factor* of each firing neuron. The *cv_isi* method has an optional parameter *display* which displays a histogram if set to true.

2.3.5 Output formats

To obtain the spike list in different formats, the following methods are provided. Currently only *as_list_id_time*, *as_list_id_list_time*, *as_spikematrix* and *as_pyNN_SpikeArray* are implemented.

```
SpikeList.as_ids_times(self, relative=False, quantized=False)
SpikeList.as_list_id_time(self, relative=False, quantized=
    False)
```
This method returns an array containing tuples of the id and spike time.
```
SpikeList.as_list_id_list_time(self, relative=False, quantized
    =False)
```
This method instead returns a tuple containing a list with the neuron ids and a list containing the corresponding spike times.
```
SpikeList.as_id_list_times(self, relative=False, quantized=
    False)
SpikeList.as_time_list_ids(self, relative=False, quantized=
    False)
SpikeList.as_2byN_array(self, relative=False, quantized=False)
SpikeList.as_spikematrix(self)
```
This method returns a matrix containing the firing rate for each neuron and time bin.
```
SpikeList.as_pyNN_SpikeArray(self)
```
The last method returns the spike list as a *pyNN SpikeSourceArray*.

2.4 Example

```
1 import NeuroTools.spikes as spikes
2
3 list = spikes.loadSpikeList("file.dat",10,t_start=0,t_stop
      =500)
4 rate_mean = list.mean_rate()
5 rate_std = list.mean_rate_std()
6 cor_coef_mean, cor_coef_std = list.pw_corr_pearson((0,simtime)
      ,1, 100)
7 fano_mean = numpy.mean(list.fano_factors_isi())
8 fano_std = numpy.std(list.fano_factors_isi())
```

Chapter 3

Projections

3.1 *pyNN*

Today, a number of different software simulators for neuronal networks exist. Each simulator has its own native interface. *pyNN* is a high level modelling language for neuronal network experiments, which aims compatibility with many different simulators. An experiment described in *pyNN* can be easily ported to another simulator. In the ideal case, you only need to modify one line of code. *pyNN* is written and implemented in Python. It is basically a bunch of Python classes and functions.

The Python API can be divided in a high-level and a low-level part. The low-level part provides functions to create and connect individual neurons as well as to start the experiment. In contrast the high-level API handles entire populations of neurons. The low-level part is implemented in individual functions, the high-level part in two classes named *Population* and *Projection*. The *Population* class contains many neurons with similar properties. The *Projection* class is used to connect these *Population*s. The hardware variant of these two classes uses only low-level *pyNN* functions in its implementation and by this is portable. In the following, I will describe the implementation of the *Projection* class:

3.2 *Projection* class

The *Projection* class connects the neurons of a population with the neurons of another one. The *Projection* class itself does some book keeping and provides methods to modify, dump and save the connections made. The main work of connecting the populations is however outsourced into individual classes derived from the *Connector* class implementing the desired connecting method.

A projection is created by calling the constructor:

```
__init__(self, presynaptic_population, postsynaptic_population
    , method='allToAll', method_parameters=None, source=None,
```

```
target=None, synapse_dynamics=None, label=None, rng=None)
```

The pre- and post-synaptic populations are given by the *pre-synaptic_population* and *post-synaptic_population* parameters. The parameters *method* and *method_parameters* specify the desired connecting method. The method can either provided as a string or as a *Connector* object. In the second case the *method_parameters* parameter is ignored. The parameters *source* and *target* specify the attributes of the pre-/post-synaptic cell signal action potential and are currently not used in the hardware implementation. The parameter *synapse_dynamics* is also currently not implemented for the hardware variant. *label* sets a text description for the projection, and *rng* provides a random number generator used to connect the neurons where needed. An example for using a *Projection* is given in listing 3.1:

3.3 *Connector* class

The objects of the *Connector* class do the real connecting work. To do this, they need to implement the following method:

```
connect(self, projection)
```

This method calls the low level *pyNN.connect* function and fills some data structures of the associated *Projection* object given by the *projection* parameter.

Listing 3.1: Example for *Projections*

```
connector = pynn.AllToAllConnector()
con_IE = pynn.Projection(i_inh,p1,method=connector)
or
con_IE = pynn.Projection(i_inh,p1,method='allToAll')
```

3.3.1 Available connectors

Currently the *pyNN* API defines a bunch of different connecting methods. All connectors have the parameters *weights* and *delays* providing the weight and delay of the connections. They can either be set to a fixed value, or to a random number generator. In the second case, the values for the individual connections are drawn from the random number generator. The parameter *allow_self_connections* controls whether an individual neuron should be allowed to be connected with itself.

AllToAllConnector This connector connects each neuron of the pre-synaptic population with each neuron of the post-synaptic population.

OneToOneConnector The OneToOneConnector connects the corresponding neurons of two networks. So both network need to have the same size.

FixedProbabilityConnector This connector connects each neuron of the pre-synaptic network with a certain probability to an other neuron in the post-synaptic network

DistanceDependentProbabilityConnector Like the FixedProbabilityConnector this connector connects the neurons with a certain probability, however the probability depends on the distance between the neurons. The *d_expression* parameter contains a string with an Python expression calculating the connection probability depending on the distance *d*. To define the distance between two neurons, the position of the neurons can be set explicitly or is derived from the position of the neuron in the array containing the neurons in the *Population* class.

FixedNumberPreConnector This connector connects each neuron from the post-synaptic population with a fixed number of pre-synaptic neurons. The connections itself are randomly selected.

FixedNumberPostConnector Like the *FixedNumberPreConnector* this connector connects each neuron of the pre-synaptic population with a fixed number of post-synaptic neurons from the post-synaptic population.

FromListConnector This Connector takes the connections from a list.

FromFileConnector This connector reads the connections from a file. Such a file can be created by calling the *Projection.saveConnections* method from an existing projection.

3.4 Example

```
1  import pyNN.hardware.stage1 as pynn
2  #import pyNN.nest2 as pynn
3
4  p1 = pynn.Population((10,), pynn.IF_facets_hardware1,
       cellparams=neuronParams)
5
6  i_exc = pynn.Population(50,pynn.SpikeSourcePoisson,cellparams=
       inputParameters)
7  i_inh = pynn.Population(10,pynn.SpikeSourcePoisson,cellparams=
       inputParameters)
8
9  connector = pynn.AllToAllConnector(weights=w_inh*5)
10 con_IE = pynn.Projection(i_inh,p1,method=connector)
11 connector = pynn.AllToAllConnector(weights=w_exc*10)
12 con_EE = pynn.Projection(i_exc,p1,method=connector)
13
14 p1.record()
15
16 pynn.run(duration)
```

3.5 Restructuring the code of *pyNN.hardware.stage1*

To improve the code maintainability, the code in the *__init__.py* module was divided into separate files. The implementation of the low-level API of *pyNN*

remained in *__init__.py*. The definition of the supported cell classes was moved to *cells.py*. The implementation of the high-level API was moved to the files *population.py* and *projection.py*. These extra files are imported in *__init__.py* to remain compatibility with the *pyNN* API standard. The low-level implementation was not moved into an extra file, because of its high use of global variables.

Chapter 4

FACETS milestone 7.1 experiment

The aim of the FACETS milestone 7.1 experiment is to do the same experiment in the NEST2 simulator and on the neuromorphic hardware platform. By such an experiment the similarity of the dynamics can be analysed and improved qualitative and quantitative.

I changed the data analysis functions to use the features supplied by *NeuroTools*. To get the experiment running on the hardware platform, the *Projections* class needed to be implemented. Figure 4.1 shows the experimental setup. It consists of an excitatory and an inhibitory population. These populations are connected with itself and with each other. Additionally these two populations are connected to a group of spike sources. As the number of neurons and possible connections between these neurons is limited on the hardware platform, only totally 192 neurons were used in this experiment.

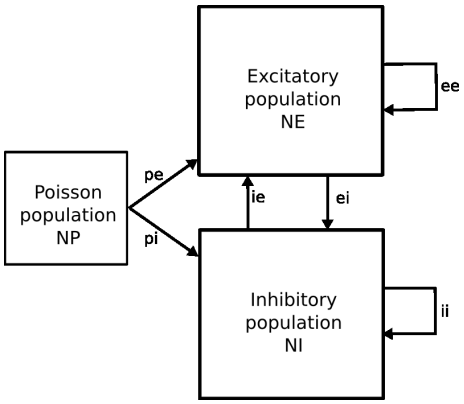
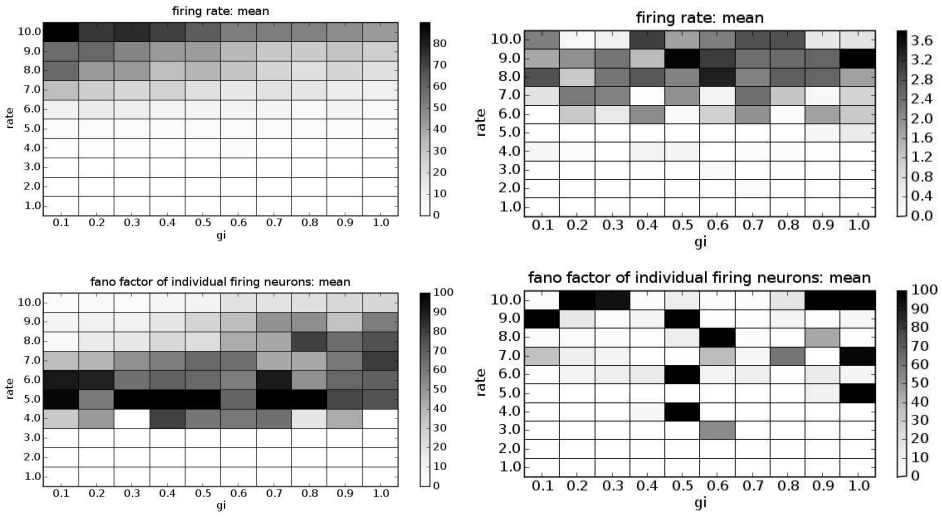


Figure 4.1: Architecture of the FACETS milestone 7.1 experiment

Figure 4.2: Simulation results:(left hand side: NEST2, right hand side: hardware)



The results of the simulation are shown in figure 4.2. The results obtained with the hardware variant clearly differ from those achieved with the NEST2 software simulator. However my work is only a proof of functionality of *NeuroTools* an the *Projections* class implementation I wrote. The experimental results are not meant to be interpreted yet. The hardware still needs to be calibrated and tuned further, before the experimental results can be interpreted.

Other reasons for the differences between the software and hardware simulation are the small number of neurons which can be recorded from the hardware, cause of a bug of the chip. Also the parameter programming in the hardware is not optimized at the present state. When these two problems are resolved, a much better result should be achieved by the hardware variant, more close to the results obtained with NEST2.

Chapter 5

Conclusion

I analysed the features of the *NeuroTools spikes* module and improved the provided functions. Finally I changed the analyse functions of the FACETS milestone experiment 7.1 to use *NeuroTools*. To achieve this I needed to complete the implementation of the *pyNN* API for the neuromorphic hardware.

However the implementation of the *spikes* module can be improved. During the initialisation process the spike arrays are copied and resorted often. Also some analyse functions are doing unnecessary copying and reformatting of the data before doing the calculations. Possible future work might be performance improvements of the *NeuroTools spikes* module.

During my internship I learned a lot about neuron models and neuronal experiments on computers. I used *pyNN* to do own little simulation runs in the NEST2 simulator as well as on the neuromorphic hardware. I got a good insight into the work of the Electronic Vision(s) Group. I had a lot of fun during my project.

Bibliography

[Kumar] A. Kumar, S. Schrader, A. Aertsen, and S. Rotter. The High-Conductance State of Cortical Networks, February 2007

[M7-1] D. Bruederle, J. Kremkow, and A. Grubel. Milestone M7-1 Evaluate the implementation of biologically realistic networks within the event-based routing framework of the Stage 1 system using benchmark experiments

[NeuroTools] `http://neuralensemble.org/trac/NeuroTools`

[pyNN] `http://neuralensemble.org/trac/PyNN`