HBP Neuromorphic Computing Platform Guidebook

Release 2020-01-21 09:32:46 (cc9c98a)

Andrew P. Davison   Eric Müller
Sebastian Schmitt   Bernhard Vogginger
David Lester   Thomas Pfeil   ...

Jan 21, 2020

CONTENTS

Living document version:

cc9c98a Tue, 21 Jan 2020 09:32:46 GMT

The Neuromorphic Computing Platform allows neuroscientists and engineers to perform experiments with configurable neuromorphic computing systems. The platform provides two complementary, large-scale neuromorphic systems built in custom hardware at locations in Heidelberg, Germany (the "BrainScaleS" system, also known as the "physical model" or PM system) and Manchester, United Kingdom (the "SpiNNaker" system, also known as the "many core" or MC system). Both systems enable energy-efficient, large-scale neuronal network simulations with simplified spiking neuron models. The BrainScaleS system is based on physical (analogue) emulations of neuron models and offers highly accelerated operation (10^4 x real time). The SpiNNaker system is based on a digital many-core architecture and provides real-time operation.

These large-scale neuromorphic systems are currently in their commissioning phase, which is accompanied by short technology and software switchover times. In order to ensure that obtained results are up to date and valid, we kindly request all users to review all material intended for dissemination (websites, papers etc.) with the involved groups.

CHAPTER

ONE

GETTING STARTED

To use the HBP Neuromorphic Computing Platform you will need:

1. an HBP Identity account (available on request)
2. to sign the Platform User Agreement and return it to neuromorphic@humanbrainproject.eu

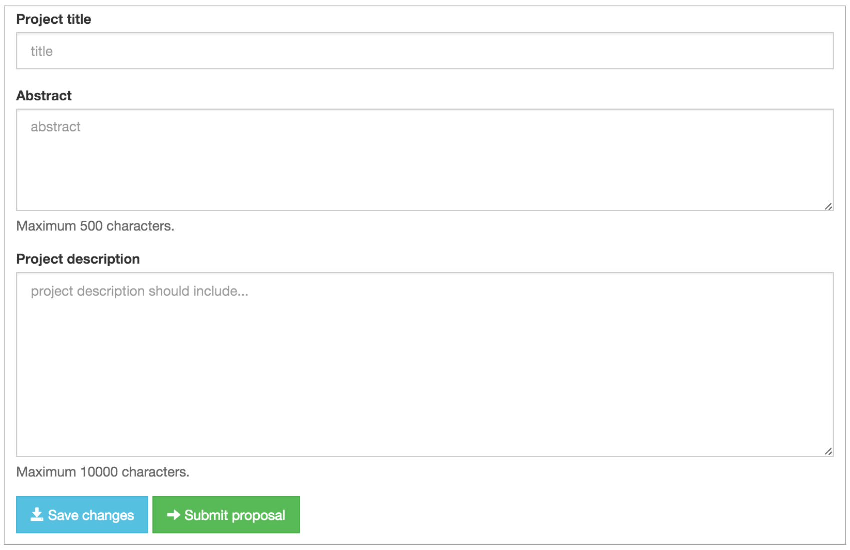
Once you've received confirmation that access has been approved, head to the Platform home page and click on "Get started!"

This will create a new "collab" (a collaborative workspace) within the HBP Collaboratory, pre-loaded with the tools you will need to access the Platform. You can now:

- Add team members to your Collab using the "*Team*" link in the navigation bar on the left. Only members of the team will be able to launch simulations. If this is a public collab, anyone with an HBP account will be able to view the results. If it is a private collab, only team members will be able to view results.
- Read the Guidebook.
- Request an allocation of compute time on the platform, using the "*Resource Manager*" link.
- Run simulations, using the "*Job Manager*" link.

1.1 Request a compute time allocation

Using the *Resource Manager* form, request test access to the platform by entering a project title and a brief abstract explaining why you wish to use the platform. It is not necessary to fill in the "Project description" field.



Project title

title

Abstract

abstract

Maximum 500 characters.

Project description

project description should include...

Maximum 10000 characters.

After clicking on *Submit proposal*, your request should be approved within 48 hours. If you've received no response within this time, e-mail neuromorphic@humanbrainproject.eu

For more information on compute time allocations, see *Requesting access to the platform*.

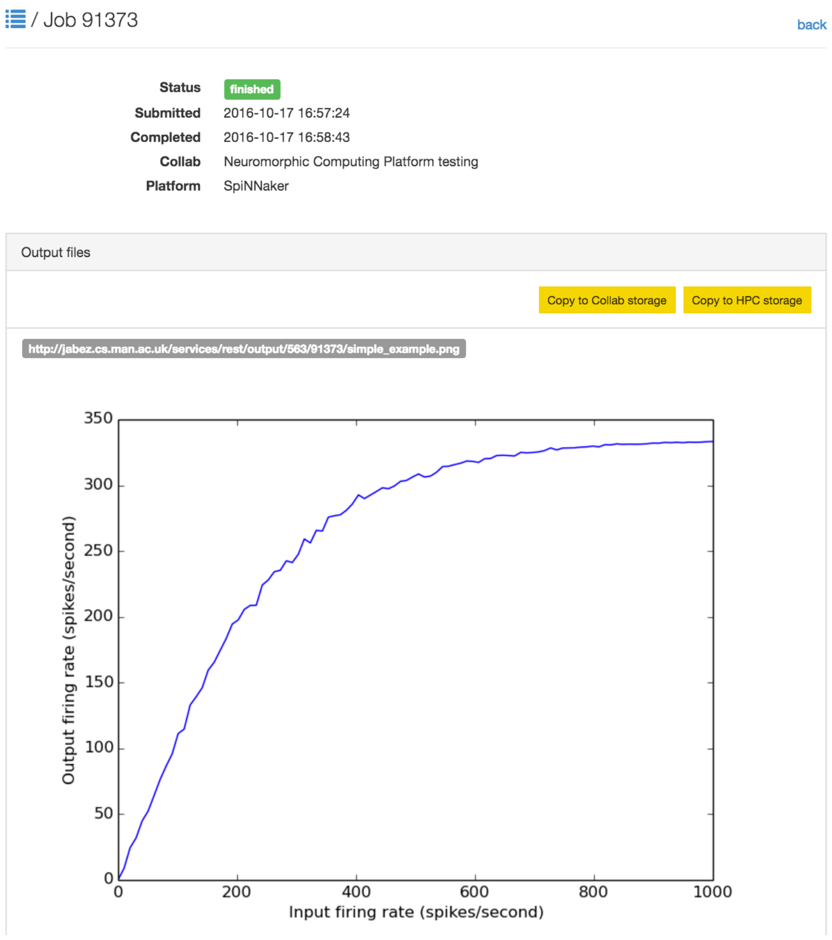
1.2 Run a simulation

Once your test allocation has been approved, click on "*Job Manager*", then on "*New Job.* Select "SpiNNaker" in the *Hardware Platform* drop-down menu, then enter Python code in the "*Code*" text box, for example the following short script, which simulates a population of integrate-and-firing neurons with different input firing rates:

```python
"""
A population of integrate-and-firing neurons with different input firing rates
"""

import numpy as np
import matplotlib.pyplot as plt
import pyNN.spiNNaker as sim

sim.setup(timestep=1.0, min_delay=1.0)

# create cells
cell_params = {
    'cm': 0.25, 'tau_m': 10.0, 'tau_refrac': 2.0,
    'tau_syn_E': 2.5, 'tau_syn_I': 2.5,
    'v_reset': -70.0, 'v_rest': -65.0, 'v_thresh': -55.0}

neurons = sim.Population(100, sim.IF_cond_exp(**cell_params))
inputs = sim.Population(100, sim.SpikeSourcePoisson(rate=0.0))

# set input firing rates as a linear function of cell index
input_firing_rates = np.linspace(0.0, 1000.0, num=inputs.size)
inputs.set(rate=input_firing_rates)

# create one-to-one connections
wiring =  sim.OneToOneConnector()
static_synapse = sim.StaticSynapse(weight=0.1, delay=2.0)
connections = sim.Projection(inputs, neurons, wiring, static_synapse)

# configure recording
neurons.record('spikes')

# run simulation
sim_duration = 10.0 # seconds
sim.run(sim_duration * 1000.0)

# retrieve recorded data
spike_counts = neurons.get_spike_counts()
print(spike_counts)
output_firing_rates = np.array(
    [value for (key, value) in sorted(spike_counts.items())])/sim_duration

# plot graph
plt.plot(input_firing_rates, output_firing_rates)
plt.xlabel("Input firing rate (spikes/second)")
plt.ylabel("Output firing rate (spikes/second)")
plt.savefig("simple_example.png")
```

Leave the other text boxes empty, and click "*Submit*". The job will be submitted to the queue, and will appear in the list of jobs with a "submitted" label. Unless the platform is very busy, this job should run within a few minutes on the large-scale SpiNNaker system in Manchester. Once the simulation is finished you will receive an e-mail, and on refreshing the job list the status will change to "finished".

Once the job is completed, click on the magnifying glass icon to see the job results.



Status	finished
Submitted	2016-10-17 16:57:24
Completed	2016-10-17 16:58:43
Collab	Neuromorphic Computing Platform testing
Platform	SpiNNaker

http://jabez.cs.man.ac.uk/services/rest/output/563/91373/simple_example.png

For more information on running simulations with the platform, see *Submitting a simulation job*.

1.3 Copy data to longer-term storage

The results of your simulation are now available on a file server attached to the SpiNNaker system. This storage is only temporary, however; after three months, your files may be deleted to free up space.

For this reason, therefore, we recommend either downloading the files to your local machine or copying them to longer-term storage within the Human Brain Project infrastructure.

For now we will copy the files to Collab Storage by clicking the button "*Copy to Collab storage*".

If you now click on the link "*Storage*" in the left-hand menu, you will see the files produced by your simulation.

CHAPTER TWO

BUILDING MODELS

The Neuromorphic Computing Platform executes experiments performed on computational models of neuronal networks. Both the experiment description and the model description must be written as Python scripts, using the PyNN application programming interface (API).

2.1 The PyNN model description API

PyNN is a Python package which defines an API for defining neuronal network models with spiking neurons in a simulator-independent way. There are implementations of this API for the NEST, NEURON and Brian simulators and for both of the HBP Neuromorphic Computing systems (PM and MC/SpiNNaker). In this documentation, we refer to each of these simulators and hardware platforms as a PyNN "**backend**".

Full documentation of the API is available at:

PyNN 0.6 documentation

PyNN 0.7 documentation

PyNN 0.8 documentation

At the time of writing, both the PM and MC systems implement version 0.7 of the API, while the simulator of the PM hardware (*ESS*) supports both versions 0.7 and 0.8, and the chip-based PM hardware (*Spikey*) supports version 0.6.

2.2 A simple example

We present here a simple example network, a toy model of a "synfire chain", in which the activity propagates across the network.

As with any Python script, the first step is to import the external libraries that we will use. Here we import the NEST backend of PyNN for the simulation, NumPy for random number generation, and matplotlib for plotting.

```
import pyNN.nest as sim
import numpy.random
import matplotlib.pyplot as plt
```

Next we define numerical parameters, such as the number and size of neuronal populations, neuron properties, synaptic weights and delays, etc.

```
n_populations = 11
population_size = 8
neuron_parameters = {
    'cm': 0.2,
    'v_reset': -70,
    'v_rest': -70,
    'v_thresh': -47,
    'e_rev_I': -70,
    'e_rev_E': 0.0,
}
weight_exc_exc = 0.005
weight_exc_inh = 0.005
weight_inh_exc = 0.5
delay = 3.0
rng_seed = 42
stimulus_onset = 25.0
stimulus_sigma = 0.5
runtime = 150.0
```

The `setup()` function initializes and configures the simulator (in this case NEST).

```
sim.setup(timestep=0.1)
```

The main building block in PyNN is a population of neurons of the same type (although the parameters of the neurons within the population can be heterogeneous). Here we create 11 populations of excitatory integrate-and-fire (IF) neurons and 11 populations of inhibitory IF neurons.

```
populations = {'exc': [], 'inh': []}
for syn_type in ('exc', 'inh'):
    populations[syn_type] = [sim.Population(population_size,
                                            sim.IF_cond_exp,
                                            neuron_parameters)
                             for i in range(n_populations)]
```

We now connect each excitatory population to the following pair of excitatory and inhibitory populations, and each inhibitory population to the excitatory population within the same pair.

```
connector_exc_exc = sim.AllToAllConnector(weights=weight_exc_exc, delays=delay)
connector_exc_inh = sim.AllToAllConnector(weights=weight_exc_inh, delays=delay)
connector_inh_exc = sim.AllToAllConnector(weights=weight_inh_exc, delays=delay)

for i in range(n_populations):
    j = (i + 1) % n_populations
    prj_exc_exc = sim.Projection(populations['exc'][i], populations['exc'][j],
                                 connector_exc_exc, target='excitatory')
    prj_exc_inh = sim.Projection(populations['exc'][i], populations['inh'][j],
                                 connector_exc_inh, target='excitatory')
    prj_inh_exc = sim.Projection(populations['inh'][i], populations['exc'][i],
                                 connector_exc_exc, target='inhibitory')
```

The first pair of populations are stimulated by a burst of spikes.

```
numpy.random.seed(rng_seed)
stim_spikes = numpy.random.normal(loc=stimulus_onset,
                                  scale=stimulus_sigma,
                                  size=population_size)
stim_spikes.sort()
stimulus = sim.Population(1, sim.SpikeSourceArray, {'spike_times': stim_spikes})

prj_stim_exc = sim.Projection(stimulus, populations['exc'][0],
                              connector_exc_exc, target='excitatory')
prj_stim_inh = sim.Projection(stimulus, populations['inh'][0],
                              connector_exc_inh, target='excitatory')
```

We've now finished building the network, now we instrument it, by recording spikes from all populations.

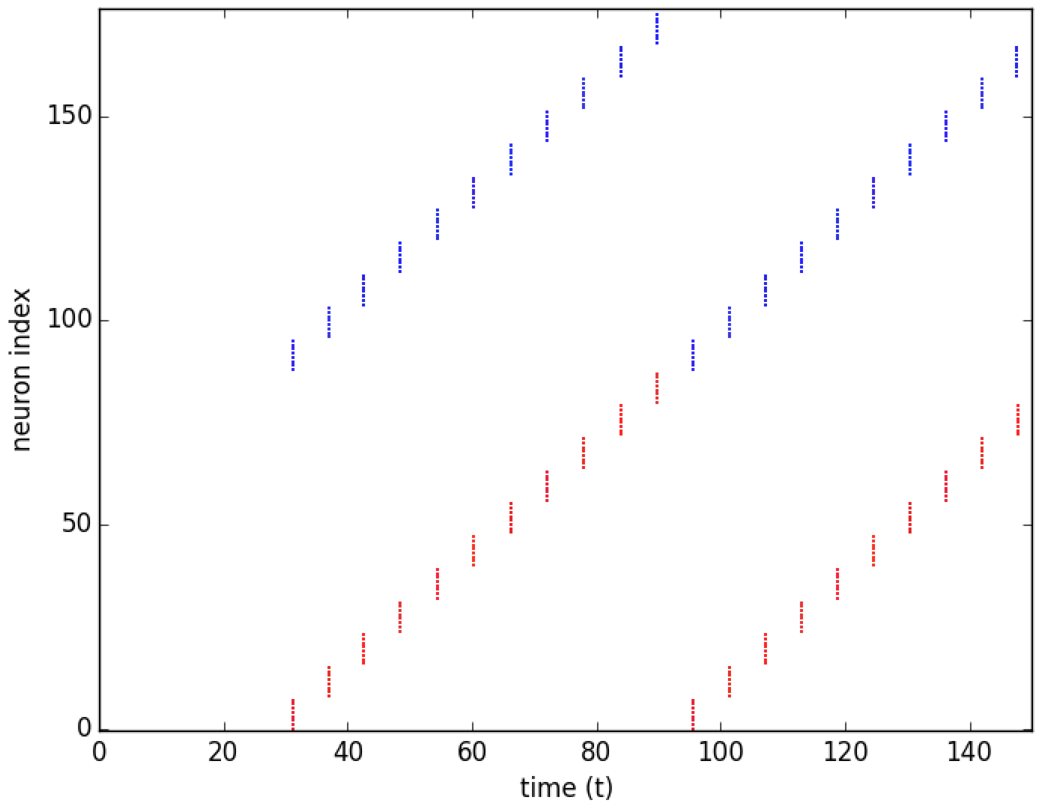
```
for syn_type in ('exc', 'inh'):
    for population in populations[syn_type]:
        population.record()
```

Now we run the simulation:

```
sim.run(runtime)
```

Finally we loop over the populations, retrieve the spike times, and plot a raster plot (spike time vs neuron index).

```
colours = {'exc': 'r', 'inh': 'b'}

id_offset = 0
for syn_type in ['exc', 'inh']:
    for population in populations[syn_type]:
        spikes = population.getSpikes()
        colour = colours[syn_type]
        plt.plot(spikes[:,1], spikes[:,0] + id_offset,
                 ls='', marker='o', ms=1, c=colour, mec=colour)
        id_offset += population.size

plt.xlim((0, runtime))
plt.ylim((-0.5, 2* n_populations * population_size + 0.5))
plt.xlabel('time (t)')
plt.ylabel('neuron index')
plt.savefig("synfire_chain.png")
```



Note that here we include the plotting in the same script just to illustrate the example output. Scripts submitted to the Neuromorphic Computing Platform should not in general contain graph plotting or data analysis code. Instead you should save spikes to file and use the other tools available in the Collaboratory for analysing and visualizing the results.

2.3 Using different backends

To run the same simulation with a different simulator, just change the name of the PyNN backend to import, e.g.:

```
import pyNN.neuron as sim
```

to run the simulation with NEURON. For the PM system, the module to import is `pyNN.hardware.hbp_pm` while for the MC system the module is `pyNN.spiNNaker`.

The recommended approach is to provide the name of the backend as a command-line argument, i.e., you run your simulation using:

```
$ python run.py nest
```

PyNN contains some utility functions to make this easier. With PyNN 0.7, use:

```
from pyNN.utility import get_script_args
simulator_name = get_script_args(1)[0]
exec("import pyNN.%s as sim" % simulator_name)
```

With PyNN 0.8:

```
from pyNN.utility import get_simulator
sim, options = get_simulator()
```

2.4 “Physical model” (BrainScaleS) system

The BrainScaleS system has a number of additional configuration options that can be passed to `setup()`. These are explained in *About the BrainScaleS hardware*. There are also a number of limitations, for example only a subset of the PyNN standard neuron and synapse models are available.

The BrainScaleS system attempts to automatically place neurons on the wafers in an optimal way. However, it is possible to influence this placement or control it manually. An example can be found in label-marocco-example.

2.5 “Many core” (SpiNNaker) system

The SpiNNaker system supports the standard arguments provided by the PyNN `setup()` function. The SpiNNaker system has a number of limitations in terms of the support for PyNN functionality, for example, only a subset of the PyNN standard neuron and synapse models are currently available. These limitations are defined in the online documentation here.

The SpiNNaker software stack attempts to automatically partition the populations defined in the PyNN script into core sized chunks (the smallest atomic size of resource for a machine) of neurons which are then placed onto the machine in an optimal way in regard to the machine’s available resources. However, it is possible to influence the partitioning and placement behaviours manually.

For example, take the PyNN definition of a population from the “synfire chain” example discussed previously, and shown below:

```
populations[syn_type] = sim.Population(population_size,
                                       sim.IF_cond_exp,
                                       neuron_parameters)
```

A partitioning limitation/constraint can be added to the population, which can limit how many neurons each core size chunk will contain at a maximum. This is shown below:

```
populations[syn_type] = sim.Population(population_size,
                                       sim.IF_cond_exp,
                                       neuron_parameters)
populations[syn_type].add_constraint(sim.PartitionerMaximumSizeConstraint(200))
```

This and other examples of manual limitations can be found in the online documentation here.

CHAPTER

THREE

RUNNING SIMULATIONS

The Neuromorphic Computing Platform of the Human Brain Project contains two very different neuromorphic hardware systems - BrainScaleS (also known as “NM-PM-1”, “physical model”) and SpiNNaker (also known as “NM-MC-1”, “many core”) - but has a single interface.

Jobs are written as Python scripts using the PyNN API, submitted to a queue server, then executed on one of the neuromorphic systems. On job completion, the user may retrieve the results of the simulation/emulation.

There are several ways of interacting with the queue server. This document describes the web interface and the Python client.

3.1 Format of a job

Whether using the web interface or the Python client, a job for the HBP Neuromorphic Computing Platform consists of:

- an experiment description
- input data
- hardware platform configuration

3.1.1 Experiment description

The experiment description takes the form of a Python script using the PyNN API. You must provide one of:

- a single script, uploaded as part of the job submission or pasted into the web form
- the file path of a single script located on your local machine (Python client only)
- the URL of a public Git repository
- the URL of a zip or tar.gz archive

In the latter two cases, you may optionally specify the path to the main Python script you wish to run, together with any command-line arguments. The command-line arguments may contain a placeholder, “`{system}`”, which will be replaced with the name of the PyNN backend module to be used (e.g. “spinnaker”). Example:

```
main.py --option1=42 {system}
```

If the path is not specified, the script is assumed to be named `run.py`, i.e. the default is:

```
run.py {system}
```

3.1.2 Input data

Input data are specified as a list of data items. Each data item is the URL of a data file that should be downloaded and placed in the job working directory. If your input data files are contained within your Git repository or zip archive, you do not need to specify them here.

3.1.3 Hardware platform configuration

Here you must choose the hardware system to be used ("BrainScaleS" for the Heidelberg system, "SpiNNaker" for the Manchester system, "BrainScaleS-ESS" for the software simulator of the BrainScaleS system, or "Spikey" for the Heidelberg single-chip system) and specify any specific configuration options for the hardware system you have chosen.

SpiNNaker

On SpiNNaker, the "Hardware Config" box accepts a JSON-formatted object, with the following fields:

"spynnaker_version" The git tag or branch to run. If a "semantic" version is specified, each part of the tools will download a "matching" semantic version if possible, or use git master if no match is found. If any other name is used, each part of the tools will attempt to use a matching branch or tag from git, or git master if no match is found.

"spinnaker_tools_version" By default, a version of spinnaker tools will be used that works with the software version used above. If another version is required for any reason, this can be overridden here.

"extra_pip_installs" Specifies an array of additional libraries that should be installed using pip.

"extra_git_repositories" Specifies an array of additional git repositories that should be cloned. The repository will be cloned into a sub-folder of the "current working directory" on the system, and so can be made use of when specifying the options below.

"extra_makes" Specifies an array of additional folders in which "make" should be called.

"extra_python_setups" Specifies an array of additional folders in which to run "python setup.py install".

Example:

```
{
"spynnaker_version": "master",
"spinnaker_tools_version": "3.1.0",
"extra_pip_installs": ["elephant"],
"extra_git_repositories": ["https://github.com/SpiNNakerManchester/
↪SpiNNakerGraphFrontEnd"],
"extra_makes": ["SpiNNakerGraphFrontEnd/spinnaker_graph_front_end/examples"],
"extra_python_setups": ["SpiNNakerGraphFrontEnd"]
}
```

3.2 Using the web interface

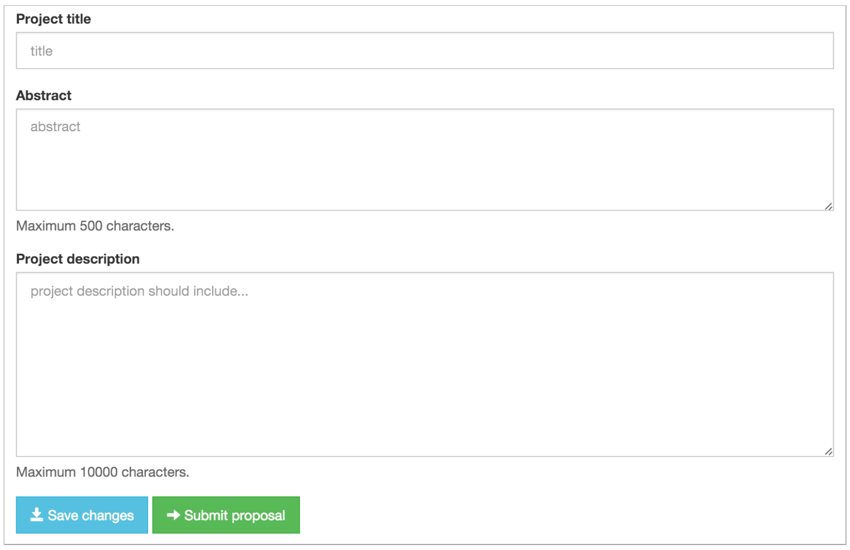
3.2.1 Setting up your Collab

The neuromorphic computing platform can be accessed from within the HBP Collaboratory using the "Neuromorphic Computing Platform Job Manager v2" app and the "Neuromorphic Computing Platform Resource Manager" app.

To create a new Collab which already contains these two apps, go to the Neuromorphic Computing Platform collab and click on *Create a Collab* or *Get Started!* To add the apps to an existing Collab, click *ADD* in the Navigation panel, and then select each of the apps in the list.

3.2.2 Requesting access to the platform

In your Collab, click on *Resource Manager*, and fill in the form.


Project title

title

Abstract

abstract

Maximum 500 characters.

Project description

project description should include...

Maximum 10000 characters.

The project description should contain a scientific or technical motivation for using the platform, and should specify which of the Neuromorphic Computing Systems ("BrainScaleS" and/or "SpiNNaker") you wish to use.

Three forms of access are available:

Test access Only a brief abstract is required explaining why you wish to use the platform. It is not necessary to fill in the "Project description" field. No previous experience with the platform is required. A fixed quota of 5000 core-hours (for the SpiNNaker system) and/or 0.1 wafer-hours (for the BrainScaleS system) will be allocated, together with temporary storage of 1 GB.

Preparatory access Only a short technical motivation is required. Previous experience with the platform (through a test access) is expected. A fixed quota of 500000 core-hours (for the SpiNNaker system) and/or 10 wafer-hours (for the BrainScaleS system) will be allocated, subject to a brief technical review, together with temporary storage of 10 GB.

Project access For projects requiring more than the test/preparatory quotas, a scientific motivation of about one page should be provided, and a request for resources (in core-hours, wafer-hours and/or GB of storage) should be specified, and justified with respect to the project's scientific goals. This proposal will receive both scientific and technical reviews.

Do not forget to specify which type of access you are requesting, and which of the Neuromorphic Computing Systems ("BrainScaleS", "Spikey" and/or "SpiNNaker") you wish to use.

Access is granted on a per-collab basis, not per-person. All members of this collab will be able to make use of the quota. All collab members will also be asked to sign and return a User Agreement form.

Once the resource request is granted, the *Resource Manager* will display the quota usage.

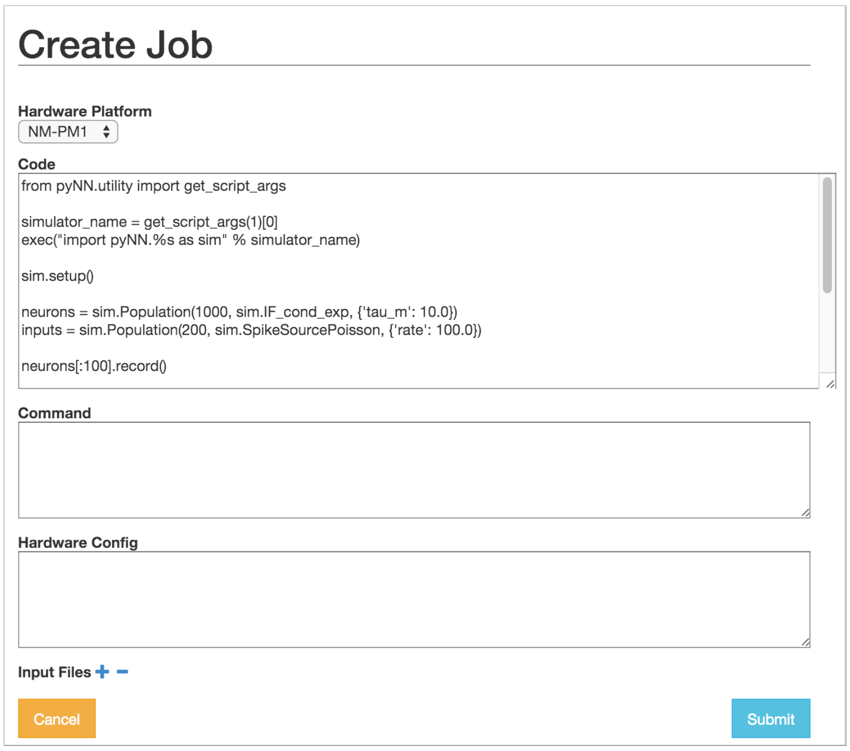
3.2.3 Submitting a simulation job

To submit a simulation job to the Platform, click on *Job Manager*.

You will see a list of jobs you have submitted to the platform. The first time you connect, of course, this list will be empty.

To create a new simulation job click on the *'+'* icon or the *New Job* button.

In this dialog, you must choose the project with which the job is associated, the hardware platform on which you wish to run ("BrainScaleS", "SpiNNaker", "BrainScaleS-ESS" or "Spikey"), and provide the Python script which should be run, either by copy-and-pasting the script into the "Code" box,


Create Job

Hardware Platform: NM-PM1

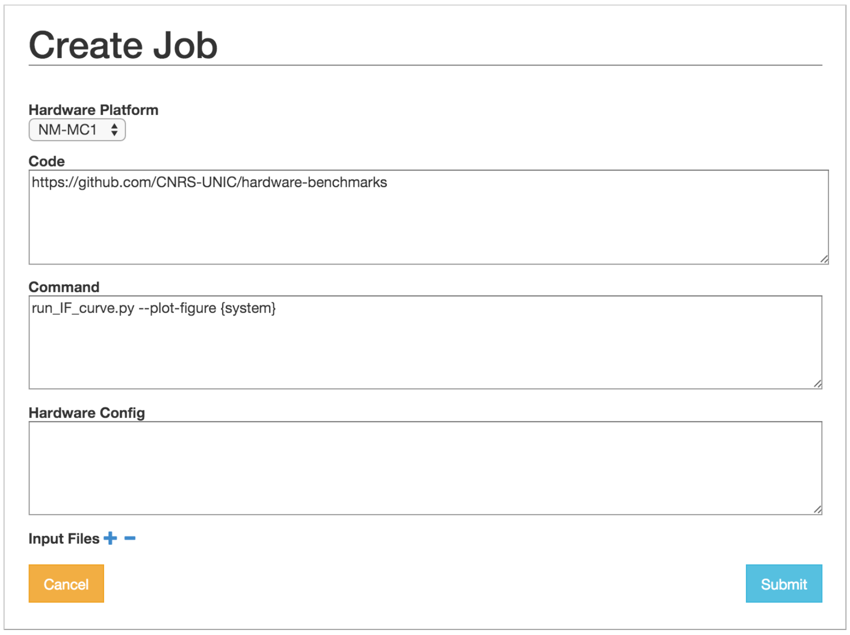
Code:

```
from pyNN.utility import get_script_args

simulator_name = get_script_args(1)[0]
exec("import pyNN.%s as sim" % simulator_name)

sim.setup()

neurons = sim.Population(1000, sim.IF_cond_exp, {'tau_m': 10.0})
inputs = sim.Population(200, sim.SpikeSourcePoisson, {'rate': 100.0})

neurons[:100].record()
```

Command

Hardware Config

or by giving the URL of a version control repository or zip/tar archive together with a command-line invocation.


Create Job

Hardware Platform: NM-MC1

Code: https://github.com/CNRS-UNIC/hardware-benchmarks

Command: run_IF_curve.py --plot-figure {system}

Hardware Config

In your Python script you should avoid hard-coding the name of the PyNN backend to run, as this will differ depending on the platform. Instead, your script should read the name of the backend from the command-line. With PyNN 0.8, this can be achieved using:

```
from pyNN.utility import get_simulator
sim, options = get_simulator()

sim.setup(...)
p = sim.Population(...)
```

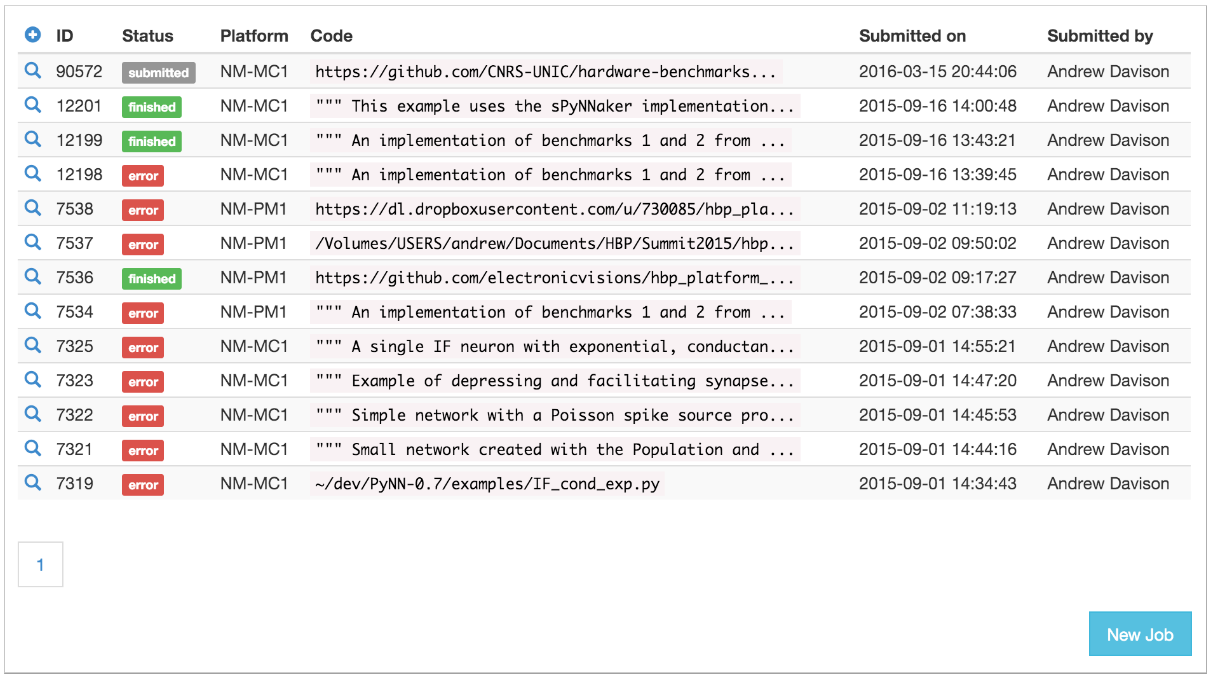
For PyNN 0.7, see *Using different backends*.

The "Hardware config" box is optional, but may contain extra configuration options in JSON format (similar to the syntax for dictionaries in Python).

Note: more information on the available configuration options for the different hardware systems will be provided soon.

It is possible to provide input data files to the simulation. The files must be accessible online.

After clicking "Submit" the job will be submitted to the queue, and will appear in the list of jobs with a "submitted" label.

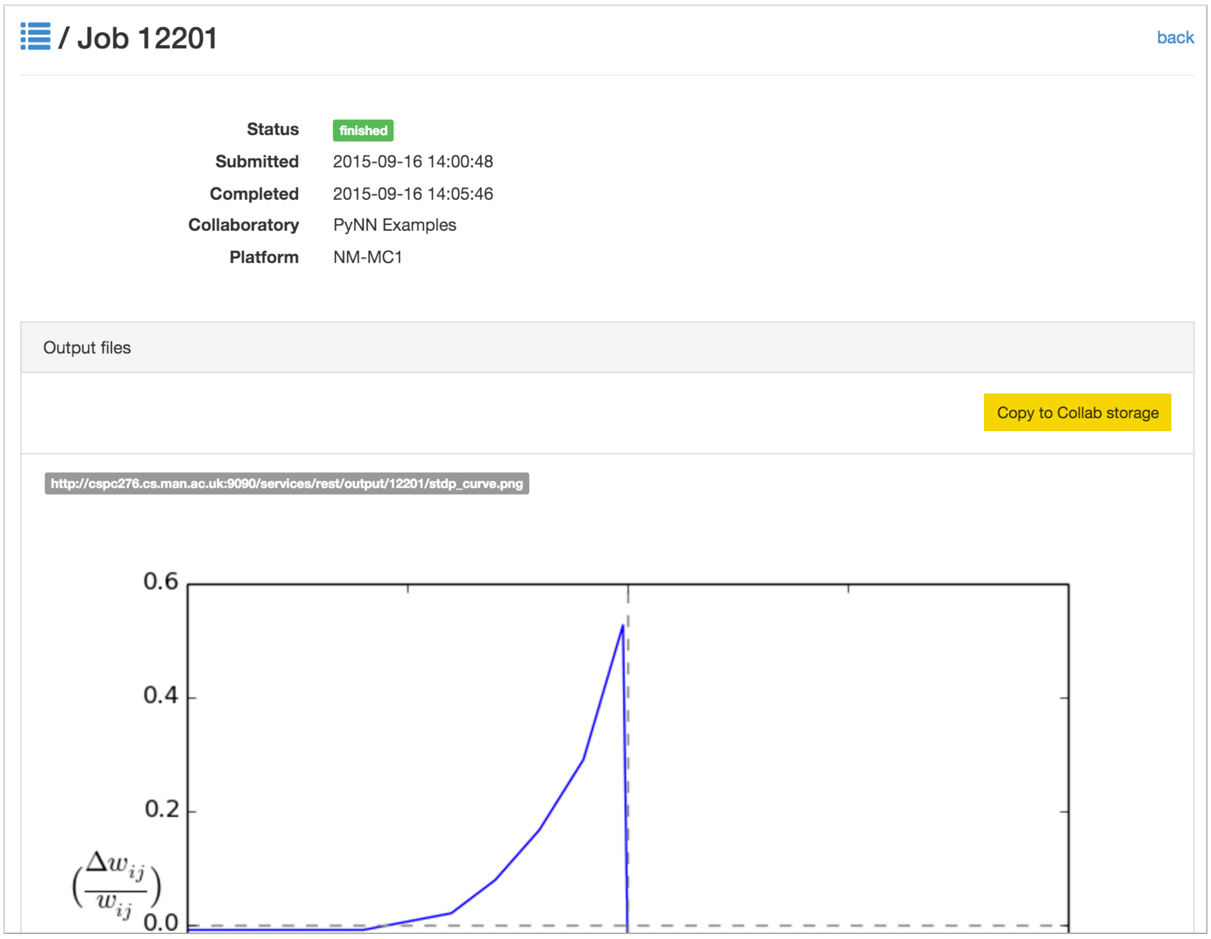


ID	Status	Platform	Code	Submitted on	Submitted by
90572	submitted	NM-MC1	`https://github.com/CNRS-UNIC/hardware-benchmarks...`	2016-03-15 20:44:06	Andrew Davison
12201	finished	NM-MC1	`""" This example uses the sPyNNaker implementation...`	2015-09-16 14:00:48	Andrew Davison
12199	finished	NM-MC1	`""" An implementation of benchmarks 1 and 2 from ...`	2015-09-16 13:43:21	Andrew Davison
12198	error	NM-MC1	`""" An implementation of benchmarks 1 and 2 from ...`	2015-09-16 13:39:45	Andrew Davison
7538	error	NM-PM1	`https://dl.dropboxusercontent.com/u/730085/hbp_pla...`	2015-09-02 11:19:13	Andrew Davison
7537	error	NM-PM1	`/Volumes/USERS/andrew/Documents/HBP/Summit2015/hbp...`	2015-09-02 09:50:02	Andrew Davison
7536	finished	NM-PM1	`https://github.com/electronicvisions/hbp_platform_...`	2015-09-02 09:17:27	Andrew Davison
7534	error	NM-PM1	`""" An implementation of benchmarks 1 and 2 from ...`	2015-09-02 07:38:33	Andrew Davison
7325	error	NM-MC1	`""" A single IF neuron with exponential, conductan...`	2015-09-01 14:55:21	Andrew Davison
7323	error	NM-MC1	`""" Example of depressing and facilitating synapse...`	2015-09-01 14:47:20	Andrew Davison
7322	error	NM-MC1	`""" Simple network with a Poisson spike source pro...`	2015-09-01 14:45:53	Andrew Davison
7321	error	NM-MC1	`""" Small network created with the Population and ...`	2015-09-01 14:44:16	Andrew Davison
7319	error	NM-MC1	`~/dev/PyNN-0.7/examples/IF_cond_exp.py`	2015-09-01 14:34:43	Andrew Davison

You will receive e-mail notifications when the job starts running and when it completes.

3.2.4 Retrieving the results of a job

Once the job is completed, click on the magnifying glass icon to see the job results and download the output data files.


/ Job 12201

back

Status	finished
Submitted	2015-09-16 14:00:48
Completed	2015-09-16 14:05:46
Collaboratory	PyNN Examples
Platform	NM-MC1

Output files

Copy to Collab storage

http://cspc276.cs.man.ac.uk:9090/services/rest/output/12201/stdp_curve.png

3.3 Using the Python client

The Python client allows scripted access to the Platform. The same client software is used both by end users for submitting jobs to the queue, and by the hardware systems to take jobs off the queue and to post the results.

3.3.1 Installing the Python client

Install the nmpi_client package from PyPI into a virtual environment, using for example virtualenv or Anaconda. The client works with Python 2.7 and Python 3.3 or newer.

```
$ pip install hbp_neuromorphic_platform
```

3.3.2 Configuring the client

Before using the Neuromorphic Computing Platform you must have an HBP account, have created at least one Collab, and have obtained a compute quota as described above under *Requesting access to the platform*.

To interact with the Platform, you first create a `Client` object with your HBP username:

```
import nmpi

c = nmpi.Client("myusername")
```

This will prompt you for your password.

After you have connected once with your password, the platform provides a token which you can save to a file and use in place of the password.

```
token = c.token

new_client = Client("myusername", token=token)
```

This token will eventually expire. When it does, reconnect with your password to obtain a new token.

3.3.3 Submitting a job

Simple example: a single file on your local machine, no input data or parameter files.

```
job_id = c.submit_job(source="/Users/alice/dev/pyNN_0.7/examples/IF_cond_exp.py",
                      platform=nmpi.BRAINSCALES,
                      collab_id=563)
```

The Collab ID is the first number in the URL of your Collab, e.g. `https://collab.humanbrainproject.eu/#/collab/563/nav/5043`.

You can get a list of all your Collabs using:

```
collabs = c.my_collabs()
```

A more complex example: the experiment and model description are contained in a Git repository. The input to the network is an image file taken from the internet.

```
job_id = c.submit_job(source="https://github.com/apdavison/nmpi_test",
                      platform=nmpi.SPINNAKER,
                      collab_id=141,
                      inputs=["http://aloi.science.uva.nl/www-images/90/90.jpg"],
                      command="run.py {system}")
```

3.3.4 Monitoring job status

```
>>> c.job_status(job_id)
u'submitted'
```

3.3.5 Retrieving the results of a job

```
>>> job = c.get_job(job_id, with_log=True)
>>> from pprint import pprint
>>> pprint(job)
{u'code': u'https://github.com/apdavison/nmpi_test',
 u'hardware_config': u'',
 u'hardware_platform': u'SpiNNaker',
 u'id': 19,
 u'input_data': [{u'id': 34,
                  u'resource_uri': u'/api/v1/dataitem/34',
                  u'url': u'http://aloi.science.uva.nl/www-images/90/90.jpg'}],
 u'log': u'',
 u'output_data': [{u'id': 35,
                   u'resource_uri': u'/api/v1/dataitem/35',
                   u'url': u'http://example.com/my_output_data.h5'}],
 u'collab_id': 141,
 u'resource_uri': u'/api/v1/queue/19',
 u'status': u'finished',
 u'timestamp_completion': u'2014-08-13T21:02:37.541732',
 u'timestamp_submission': u'2014-08-13T19:40:43.964541',
 u'user': u'myusername'}
```

To download the data files generated by your simulation:

```
filenames = download_data(self, job, local_dir=".")
```

3.3.6 Deleting a job

```
>>> remove_completed_job(job_id)    # for jobs with status "finished" or "error"
>>> remove_queued_job(job_id)       # for jobs with status "submitted"
```

CHAPTER

FOUR

THE BRAINSCALES "PHYSICAL MODEL" SYSTEM

4.1 About the BrainScaleS hardware

4.1.1 The neuromorphic wafer module

At the core of the BrainScaleS wafer-scale hardware system (see Figure 4.1) is an uncut wafer built from mixed-signal ASICs[1], named *High Input Count Analog Neural Network* chips (*HICANNs*), which provide a highly configurable substrate that physically emulates adaptively spiking neurons and dynamic synapses (Schemmel et al. (2010), Schemmel et al. (2008)). The intrinsic time constants of these VLSI model circuits are multiple orders of magnitude shorter than their biological counterparts. Consequently, the hardware model evolves with a speedup factor of 10^3 up to 10^5 compared to biological real time, the precise value depending on the configuration of the system. This speedup enables power-efficient computation as the energy consumption for synaptic transmissions is several orders of magnitude lower than in classically simulated neuronal networks.

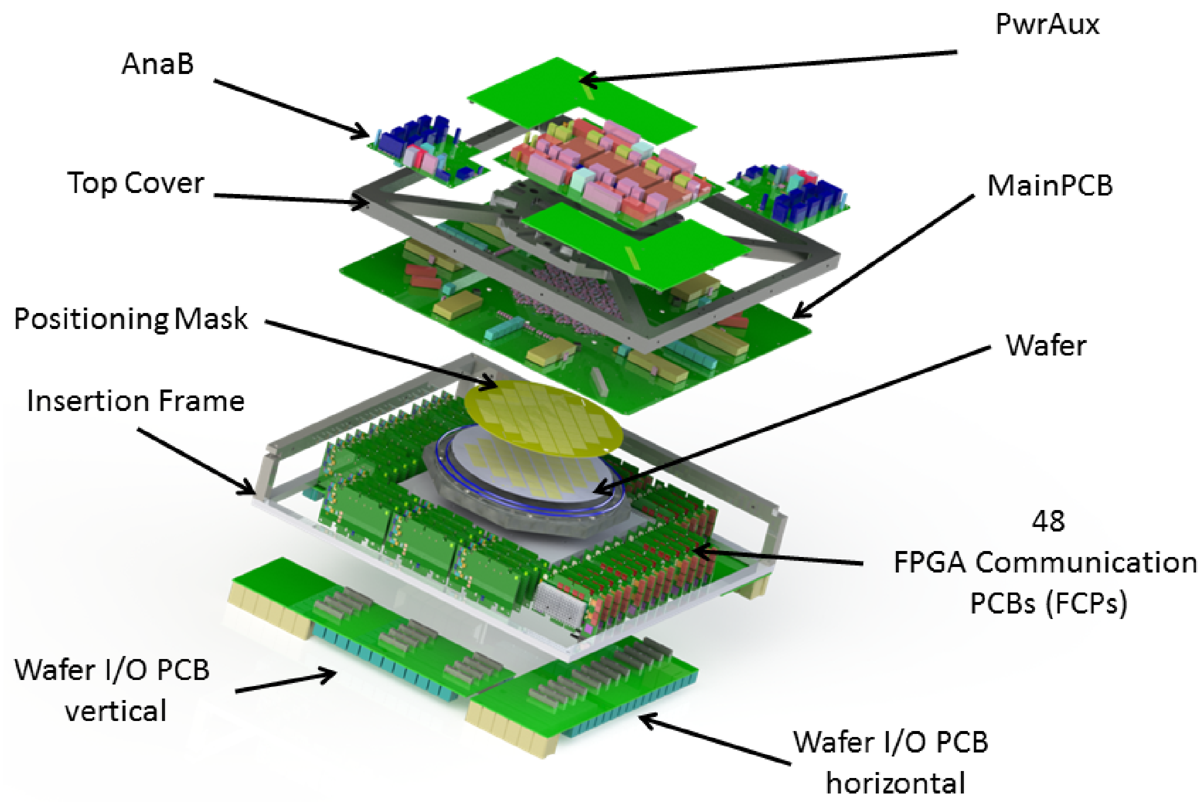


Fig. 4.1: The BrainScaleS wafer-scale hardware system: Wafer comprising HICANN building blocks and on-wafer communication infrastructure, mechanical infrastructure (top cover and insertion frame), analog readout boards (AnaB), power supply and digital inter-wafer as well as wafer-host communication modules.

[1] Application Specific Integrated Circuit

In addition to a high-bandwidth asynchronous on-wafer event communication infrastructure, 48 FPGA[2] communication modules provide off-wafer connectivity (to other wafers and to host computers).

A full wafer system comprises 384 interconnectable HICANNs, each of which implements more than 114,688 programmable dynamic synapses and up to 512 neurons, resulting in a total of approximately 44 million synapses and up to 196,608 neurons per wafer module. The exact number of neurons depends on the configuration of the substrate, which allows to combine multiple neuron building blocks to increase the input count per cell.

Via the communication FPGAs the system can be configured and operated from a host computer. Each communication FPGA is connected to a dedicated area on the wafer which contains 8 HICANNs. This FPGA-HICANN link is used to configure the HICANNs as well as to transmit pulse events to and from the neural circuits on the wafer. The pulse communication between the on-wafer neurons is performed by a bus-like network directly on the wafer.

The system provides a high degree of configurability with respect to network architecture and neuron parameters:

- each neuron provides configurable AdEx neuron dynamics
- the synapses provide 4-bit weight resolution and STDP functionality
- the connection topology can be configured

For a detailed specification see the Platform specification.

4.1.2 The full BrainScaleS system

The BrainScaleS system, shown in Figure 4.2, consists of 20 neuromorphic wafer modules together with support infrastructure and a conventional compute cluster used for controlling the wafer modules and for simulated environments.

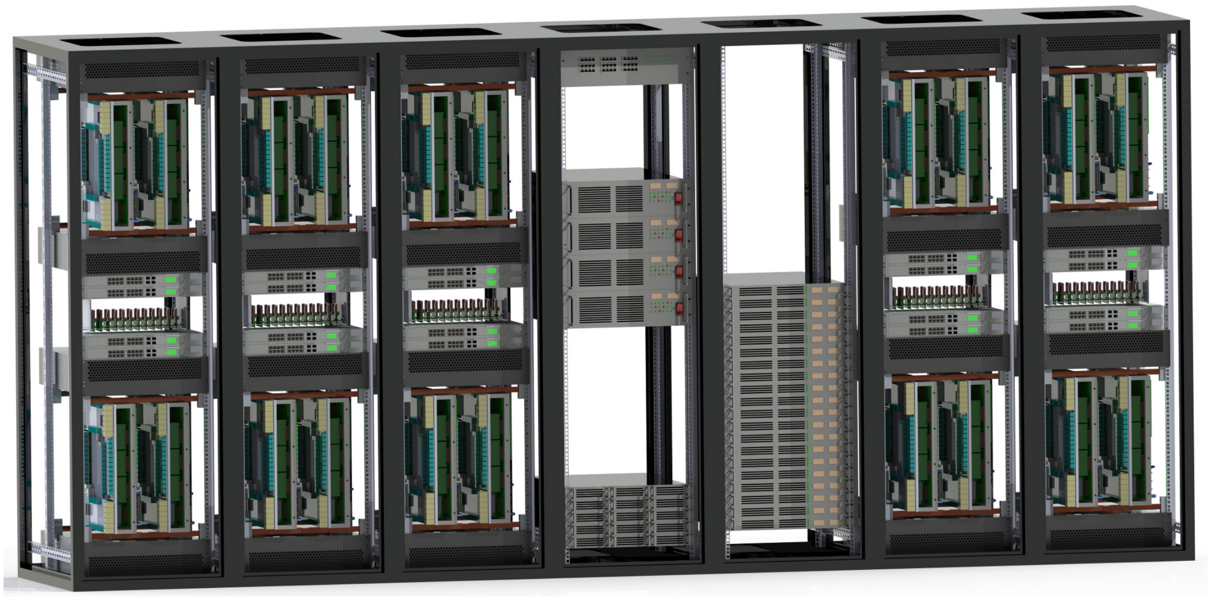


Fig. 4.2: The BrainScaleS (NM-PM-1) system: five 19-inch racks contain 20 neuromorphic wafer modules (cf. 4.1), the other two racks carry power supplies and a conventional control cluster.

Figure 4.3 provides a simplified overview of the BrainScaleS system. The support infrastructure is responsible for power supply, off-wafer communication and analog readout functionality. A dedicated Raspberry Pi monitors and controls all power links as well as other operating parameters of the wafer system. Analog readout (e.g., recording of membrane voltages) functionality is provided by a custom analog readout module (AnaRM). Several AnaRMs are handled by another dedicated control computer. The control/compute cluster orchestrates the configuration of the system and the execution of neuronal network experiments including all input and output data of the emulated network.

[2] Field Programmable Gate Array

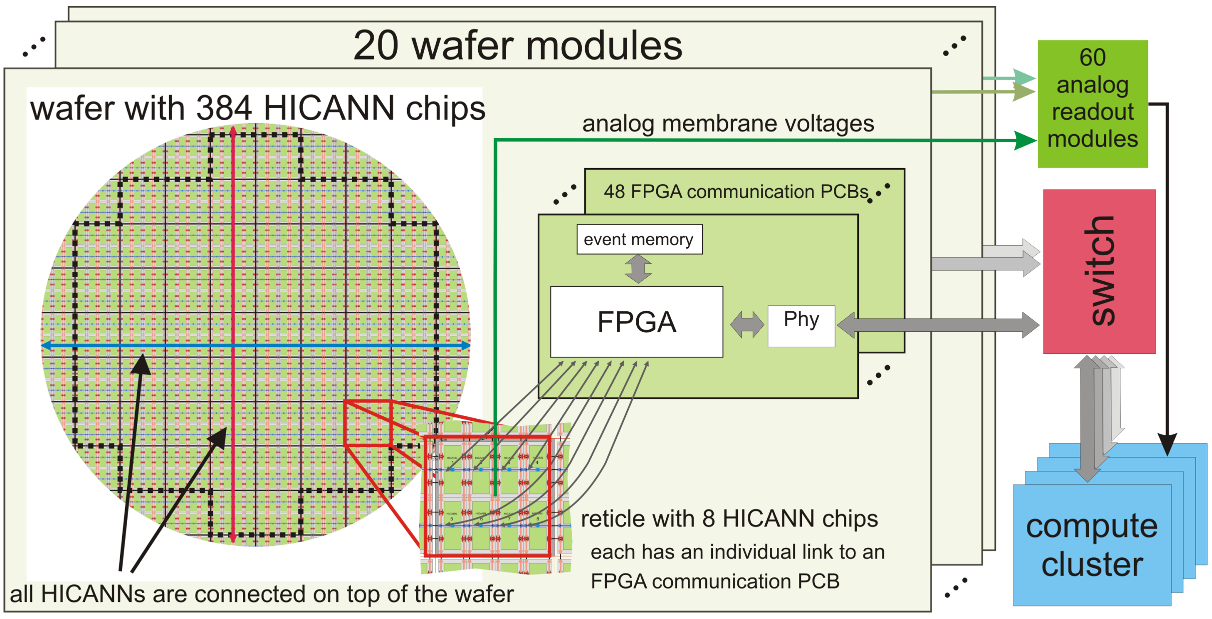

20 wafer modules
wafer with 384 HICANN chips
analog membrane voltages
60 analog readout modules
48 FPGA communication PCBs
event memory
FPGA
Phy
switch
compute cluster
reticle with 8 HICANN chips
each has an individual link to an FPGA communication PCB
all HICANNs are connected on top of the wafer

Fig. 4.3: The left area illustrates the partitioning of HICANNs into larger units (reticles) and the data flow up to the control cluster.

4.1.3 Synaptic Weights

The synaptic weight of a single synapse is proportional to the synaptic current I_{syn} that is send to the synaptic input circuit of its associated neuron:

$$I_{syn} \propto w \cdot \frac{V_{gmax}}{gmax\_div},$$

where $w \in [0, 15]$ is the 4-bit weight in every synapse, $gmax\_div \in [2, 30]$ can be set per synapse row and $V_{gmax} \in [0, 1023]$ can be selected per synapse row from one of four values per HICANN quadrant.

The synaptic conductance course is then generated according to the configured synaptic time constant within the synaptic input circuit of a neuron.

In addition, there is a contribution to the synaptic conductance that does not depend on V_{gmax} nor $gmax\_div$ but only on the 4-bit weight, cf. PhDCK. Notably this contribution is not linear in the 4-bit weight.

Setting these low-level parameters is possible and exemplified in *Using marocco*.

4.1.4 Glossary

4.2 Using the BrainScaleS system

As explained in *Building models*, both the experiment description and the model description for the BrainScaleS system must be written as Python scripts, using the PyNN application programming interface (API), version 0.7.

In the following, the build and work flow on UHEI BrainScaleS cluster frontend nodes is described. If BrainScaleS is accessed through the Collaboratory or the Python client, the installation can be skipped.

4.2.1 Setup

```
export LC_ALL=C
module load nmpm_software/current
```

```
run_nmpm_software python -c "import pyhmf" && echo ok
```

should print `ok`.

The translation from the *biological* neuronal network description into a *hardware* configuration is performed by the `marocco` mapping tool (for more detailed information, see *Details of the software stack* below).

The BrainScaleS system attempts to automatically place neurons on the wafers in an optimal way. However, it is possible to influence this placement, control it manually, and examine the resultant data structures using the Python helper module `pymarocco`. This enables users to go from a property in PyNN (e.g. the refractory period of a single neuron within an assembly) to the corresponding parameter on hardware. A typical use case is iterative low-level tuning of hardware parameters.

4.2.2 Using marocco

```
import pyhmf as pynn
from pymarocco import PyMarocco

marocco = PyMarocco()
pynn.setup(marocco=marocco)
```

Make sure that the call to `setup()` happens before creating populations, if not, the populations will not be visible to `marocco`.

In the following example, one neuron is placed on the wafer, however, by setting `marocco.backend = PyMarocco.None`, the software stops after the map & route process (i.e. before configuring the hardware system).

```
import pyhmf as pynn
from pymarocco import PyMarocco

marocco = PyMarocco()
marocco.backend = PyMarocco.None

pynn.setup(marocco=marocco)

neuron = pynn.Population(1, pynn.IF_cond_exp, {})

pynn.run(10)
pynn.end()
```

Note: Available `marocco` backends are `None`, `Hardware`, `ESS`. None has been described above. Hardware is the default and performs real experiment runs on the neuromorphic hardware system. ESS runs a simulation of the hardware: the Executable System Specification.

In the output you should see:

```
Populations:
        0th element:    0x1f98650       Population(IF_cond_exp, 1)
```

If you don't see this output, make sure that you called `pynn.setup(marocco=marocco)` before the call to `pynn.Population`.

You will also see a lot of debugging output. To set the log level, add

```
import pylogging
for domain in [""]:
    pylogging.set_loglevel(pylogging.get(domain), pylogging.LogLevel.ERROR)
```

after the import of `pymarocco`.

As we did not specify on which chip the neuron should be placed, marocco decides automatically to use `HICANNOnWafer(X(18), Y(7)), Wafer(0)` which is in the center of the wafer.

To choose the HICANN a population is placed on, we give marocco a hint:

```
import Coordinate as C

marocco.manual_placement.on_hicann(neuron, C.HICANNOnWafer(C.X(5), C.Y(5)))
```

You can inspect the coordinates on the wafer module here.

At the end, the script is the following:

```
#!/usr/bin/env python

import pyhmf as pynn
import Coordinate as C
from pymarocco import PyMarocco, Defects
from pymarocco.results import Marocco

import pylogging
for domain in ["Calibtic", "marocco"]:
    pylogging.set_loglevel(pylogging.get(domain), pylogging.LogLevel.INFO)

def get_denmems(pop, results):
    for neuron in pop:
        for item in results.placement.find(neuron):
            for denmem in item.logical_neuron():
                yield denmem

marocco = PyMarocco()
marocco.calib_backend = PyMarocco.CalibBackend.Default
marocco.defects.backend = Defects.Backend.None
marocco.neuron_placement.skip_hicanns_without_neuron_blacklisting(False)
marocco.persist = "results.xml.gz"
pynn.setup(marocco = marocco)

# place the full population to a specific HICANN
pop = pynn.Population(1, pynn.IF_cond_exp)
marocco.manual_placement.on_hicann(pop, C.HICANNOnWafer(C.X(5), C.Y(5)), 4)

# place only parts of a population
pop2 = pynn.Population(3, pynn.IF_cond_exp)
marocco.manual_placement.on_hicann(pynn.PopulationView(pop2, [0]), C.
↪HICANNOnWafer(C.Enum(5)))
marocco.manual_placement.on_hicann(pynn.PopulationView(pop2, [1]), C.
↪HICANNOnWafer(C.Enum(1)))
# the third neuron will be automatically placed

pynn.run(10)
pynn.end()

results = Marocco.from_file(marocco.persist)

for denmem in get_denmems(pop, results):
    print denmem

for denmem in get_denmems(pop2, results):
    print denmem
```

We also added a print out of the chosen neuron circuits:

```
NeuronOnWafer(NeuronOnHICANN(X(0), top), HICANNOnWafer(X(5), Y(5)))
NeuronOnWafer(NeuronOnHICANN(X(1), top), HICANNOnWafer(X(5), Y(5)))
NeuronOnWafer(NeuronOnHICANN(X(0), bottom), HICANNOnWafer(X(5), Y(5)))
NeuronOnWafer(NeuronOnHICANN(X(1), bottom), HICANNOnWafer(X(5), Y(5)))
```

Calibration

To change the calibration backend from database to XML set “calib_backend” to XML. Then the calibration is looked up in xml files named `w0-h84.xml`, `w0-h276.xml`, etc. in the directory “calib_path”.

Running pyNN scripts

To run on the *hardware* one needs to use the slurm job queue system:

```
srun -p experiment --wmod 33 --hicann 297 run_nmpm_software python nmpm1_single_
↪neuron.py
```

nmpm1_single_neuron.py:

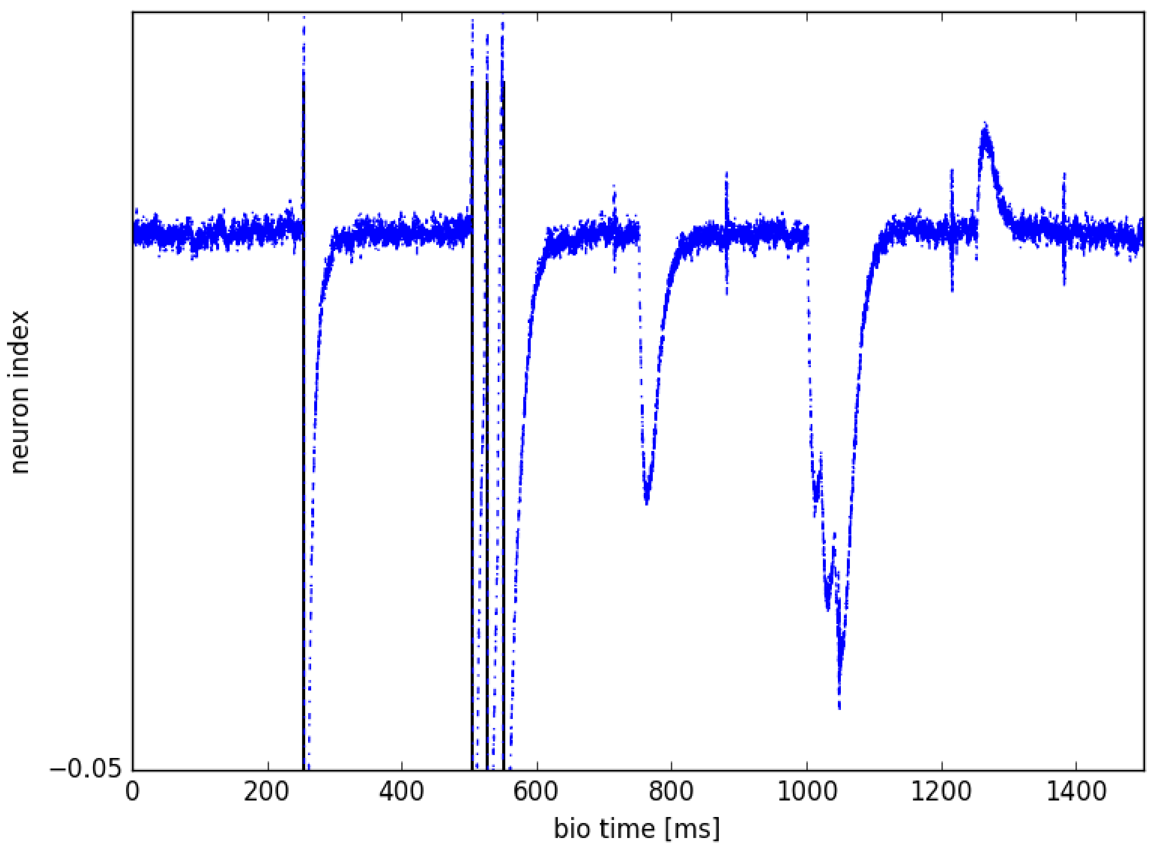
```
#!/usr/bin/env python
# -*- coding: utf-8; -*-

import os
import numpy as np
import copy

from pyhalbe import HICANN
import pyhalbe.Coordinate as C
from pysthal.command_line_util import init_logger
import pysthal

import pyhmf as pynn
from pymarocco import PyMarocco, Defects
from pymarocco.runtime import Runtime
from pymarocco.coordinates import LogicalNeuron
from pymarocco.results import Marocco

init_logger("WARN", [
    ("guidebook", "DEBUG"),
    ("marocco", "DEBUG"),
    ("Calibtic", "DEBUG"),
    ("sthal", "INFO")
])

import pylogging
logger = pylogging.get("guidebook")

neuron_parameters = {
    'cm': 0.2,
    'v_reset': -70.,
    'v_rest': -20.,
    'v_thresh': -10,
    'e_rev_I': -100.,
    'e_rev_E': 60.,
    'tau_m': 20.,
    'tau_refrac': 0.1,
    'tau_syn_E': 5.,
    'tau_syn_I': 5.,
}

marocco = PyMarocco()
marocco.default_wafer = C.Wafer(int(os.environ.get("WAFER", 33)))
runtime = Runtime(marocco.default_wafer)
pynn.setup(marocco=marocco, marocco_runtime=runtime)

#  ------ set up network␣
↪--------------------------------------------------------------------------------------------
```

```
pop = pynn.Population(1, pynn.IF_cond_exp, neuron_parameters)

pop.record()
pop.record_v()

hicann = C.HICANNOnWafer(C.Enum(297))
marocco.manual_placement.on_hicann(pop, hicann)

connector = pynn.AllToAllConnector(weights=1)

exc_spike_times = [
    250,
    500,
    520,
    540,
    1250,
]

inh_spike_times = [
    750,
    1000,
    1020,
    1040,
    1250,
]

duration = 1500.0

stimulus_exc = pynn.Population(1, pynn.SpikeSourceArray, {
    'spike_times': exc_spike_times})
stimulus_inh = pynn.Population(1, pynn.SpikeSourceArray, {
    'spike_times': inh_spike_times})

projections = [
    pynn.Projection(stimulus_exc, pop, connector, target='excitatory'),
    pynn.Projection(stimulus_inh, pop, connector, target='inhibitory'),
]

#  ------ run mapping␣
↪----------------------------------------------------------------------------------------------

marocco.skip_mapping = False
marocco.backend = PyMarocco.None

pynn.reset()
pynn.run(duration)

#  ------ change low-level parameters before configuring hardware␣
↪----------------------------------

def set_sthal_params(wafer, gmax, gmax_div):
    """
    synaptic strength:
    gmax: 0 - 1023, strongest: 1023
    gmax_div: 2 - 30, strongest: 2
    """

    # for all HICANNs in use
    for hicann in wafer.getAllocatedHicannCoordinates():

        fgs = wafer[hicann].floating_gates

        # set parameters influencing the synaptic strength
        for block in C.iter_all(C.FGBlockOnHICANN):
            fgs.setShared(block, HICANN.shared_parameter.V_gmax0, gmax)
            fgs.setShared(block, HICANN.shared_parameter.V_gmax1, gmax)
            fgs.setShared(block, HICANN.shared_parameter.V_gmax2, gmax)
            fgs.setShared(block, HICANN.shared_parameter.V_gmax3, gmax)

        for driver in C.iter_all(C.SynapseDriverOnHICANN):
            for row in C.iter_all(C.RowOnSynapseDriver):
                wafer[hicann].synapses[driver][row].set_gmax_div(HICANN.
↪GmaxDiv(gmax_div))

        # don't change values below
        for ii in xrange(fgs.getNoProgrammingPasses()):
            cfg = fgs.getFGConfig(C.Enum(ii))
            cfg.fg_biasn = 0
            cfg.fg_bias = 0
            fgs.setFGConfig(C.Enum(ii), cfg)

        for block in C.iter_all(C.FGBlockOnHICANN):
            fgs.setShared(block, HICANN.shared_parameter.V_dllres, 275)
            fgs.setShared(block, HICANN.shared_parameter.V_ccas, 800)

# call at least once
set_sthal_params(runtime.wafer(), gmax=1023, gmax_div=2)

#  ------ configure hardware␣
↪-----------------------------------------------------------------------------------------------

marocco.skip_mapping = True
marocco.backend = PyMarocco.Hardware

# magic number from marocco
SYNAPSE_DECODER_DISABLED_SYNAPSE = HICANN.SynapseDecoder(1)

original_decoders = {}

for digital_weight in [None, 0, 5, 10, 15]:
    logger.info("running measurement with digital weight {}".format(digital_
↪weight))
    for proj in projections:
        proj_items = runtime.results().synapse_routing.synapses().find(proj)
        for proj_item in proj_items:
            synapse = proj_item.hardware_synapse()

            proxy = runtime.wafer()[synapse.toHICANNOnWafer()].synapses[synapse]

            # make a copy of the original decoder value
            if synapse not in original_decoders:
                original_decoders[synapse] = copy.copy(proxy.decoder)

            if digital_weight != None:
                proxy.weight = HICANN.SynapseWeight(digital_weight)
                proxy.decoder = original_decoders[synapse]
            else:
                proxy.weight = HICANN.SynapseWeight(0)
                # set it to the special value that is never used for incoming␣
↪addresses
                proxy.decoder = SYNAPSE_DECODER_DISABLED_SYNAPSE

    pynn.run(duration)
    np.savetxt("membrane_w{}.txt".format(digital_weight if digital_weight != None␣
↪else "disabled"), pop.get_v())
    np.savetxt("spikes_w{}.txt".format(digital_weight if digital_weight != None␣
↪else "disabled"), pop.getSpikes())
```

```
    pynn.reset()

    # skip checks
    marocco.verification = PyMarocco.Skip
    marocco.checkl1locking = PyMarocco.SkipCheck

# store the last result for visualization
runtime.results().save("results.xml.gz", True)
```

Currently, the calibration is optimized towards the neuron parameters of the example. Also note that current parameters, i.e. `i_offset` are not supported.

With the help of plot_spikes.py, the recorded spikes (`spikes_w15.txt`) and membrane trace (`membrane_w15.txt`) for the digital weight setting 15 can be plotted.



The following example shows how to sweep spike times without re-running the mapping.

nmpm1_sweep_spike_times.py:

```
#!/usr/bin/env python
# -*- coding: utf-8; -*-

import os
import numpy as np

from pyhalbe import HICANN
import pyhalbe.Coordinate as C
from pysthal.command_line_util import init_logger
import pysthal

import pyhmf as pynn
```

```
from pymarocco import PyMarocco, Defects
from pymarocco.runtime import Runtime
from pymarocco.coordinates import LogicalNeuron
from pymarocco.results import Marocco

init_logger("WARN", [
    ("guidebook", "DEBUG"),
    ("marocco", "DEBUG"),
    ("Calibtic", "DEBUG"),
    ("sthal", "INFO")
])

import pylogging
logger = pylogging.get("guidebook")

neuron_parameters = {
    'cm': 0.2,
    'v_reset': -70.,
    'v_rest': -20.,
    'v_thresh': -10,
    'e_rev_I': -100.,
    'e_rev_E': 60.,
    'tau_m': 20.,
    'tau_refrac': 0.1,
    'tau_syn_E': 5.,
    'tau_syn_I': 5.,
}

marocco = PyMarocco()
marocco.default_wafer = C.Wafer(int(os.environ.get("WAFER", 33)))
runtime = Runtime(marocco.default_wafer)
pynn.setup(marocco=marocco, marocco_runtime=runtime)

#  ------ set up network␣
↪-----------------------------------------------------------------------------------------

pop = pynn.Population(1, pynn.IF_cond_exp, neuron_parameters)

pop.record()
pop.record_v()

hicann = C.HICANNOnWafer(C.Enum(297))
marocco.manual_placement.on_hicann(pop, hicann)

connector = pynn.AllToAllConnector(weights=1)

duration = 1500.0

# initialize without spike times
stimulus_exc = pynn.Population(1, pynn.SpikeSourceArray, {'spike_times': []})
stimulus_neuron = stimulus_exc[0]

projections = [
    pynn.Projection(stimulus_exc, pop, connector, target='excitatory'),
]

#  ------ run mapping␣
↪-----------------------------------------------------------------------------------------

marocco.skip_mapping = False
marocco.backend = PyMarocco.None

pynn.reset()
pynn.run(duration)

#  ------ change low-level parameters before configuring hardware␣
↪-----------------------------------

def set_sthal_params(wafer, gmax, gmax_div):
    """
    synaptic strength:
    gmax: 0 - 1023, strongest: 1023
    gmax_div: 2 - 30, strongest: 2
    """

    # for all HICANNs in use
    for hicann in wafer.getAllocatedHicannCoordinates():

        fgs = wafer[hicann].floating_gates

        # set parameters influencing the synaptic strength
        for block in C.iter_all(C.FGBlockOnHICANN):
            fgs.setShared(block, HICANN.shared_parameter.V_gmax0, gmax)
            fgs.setShared(block, HICANN.shared_parameter.V_gmax1, gmax)
            fgs.setShared(block, HICANN.shared_parameter.V_gmax2, gmax)
            fgs.setShared(block, HICANN.shared_parameter.V_gmax3, gmax)

        for driver in C.iter_all(C.SynapseDriverOnHICANN):
            for row in C.iter_all(C.RowOnSynapseDriver):
                wafer[hicann].synapses[driver][row].set_gmax_div(HICANN.
↪GmaxDiv(gmax_div))

        # don't change values below
        for ii in xrange(fgs.getNoProgrammingPasses()):
            cfg = fgs.getFGConfig(C.Enum(ii))
            cfg.fg_biasn = 0
            cfg.fg_bias = 0
            fgs.setFGConfig(C.Enum(ii), cfg)

        for block in C.iter_all(C.FGBlockOnHICANN):
            fgs.setShared(block, HICANN.shared_parameter.V_dllres, 275)
            fgs.setShared(block, HICANN.shared_parameter.V_ccas, 800)

# call at least once
set_sthal_params(runtime.wafer(), gmax=1023, gmax_div=2)

#  ------ configure hardware␣
↪--------------------------------------------------------------------------------------------------------

for proj in projections:
    proj_items = runtime.results().synapse_routing.synapses().find(proj)
    for proj_item in proj_items:
        synapse = proj_item.hardware_synapse()
        proxy = runtime.wafer()[synapse.toHICANNOnWafer()].synapses[synapse]
        proxy.weight = HICANN.SynapseWeight(15)

marocco.skip_mapping = True
marocco.backend = PyMarocco.Hardware

for n, spike_times in enumerate([[100,110], [200,210], [300,310]]):

    runtime.results().spike_times.set(stimulus_neuron, spike_times)

    pynn.run(duration)
    np.savetxt("membrane_n{}.txt".format(n), pop.get_v())
    np.savetxt("spikes_n{}.txt".format(n), pop.getSpikes())
    pynn.reset()

    # skip checks
    marocco.verification = PyMarocco.Skip
    marocco.checkl1locking = PyMarocco.SkipCheck

# store the last result for visualization
runtime.results().save("results.xml.gz", True)
```

4.2.3 Inspect the synapse loss

When mapping network models to the wafer-scale hardware, it may happen that not all model synapses can be realized on the hardware due to limited hardware resources. Below is a simple network that is mapped to very limited resources so that synapse loss is enforced. For this example we show how to extract overall mapping statistics and projection-wise or synapse-wise synapse losses.

```
def main():
    """
    create small network with synapse loss.  The synapse loss happens due to a
    maximum syndriver chain length of 5 and only 4 denmems per neuron.  After
    mapping, the synapse loss per projection is evaluated and plotted for one
    projection.  The sum of lost synapses per projection is compared to the
    overall synapse loss returnd by the mapping stats.
    """
    marocco = PyMarocco()
    marocco.neuron_placement.default_neuron_size(4)
    marocco.synapse_routing.driver_chain_length(5)
    marocco.continue_despite_synapse_loss = True
    marocco.calib_backend = PyMarocco.CalibBackend.Default
    marocco.neuron_placement.skip_hicanns_without_neuron_blacklisting(False)

    pynn.setup(marocco=marocco)

    neuron = pynn.Population(50, pynn.IF_cond_exp)
    source = pynn.Population(50, pynn.SpikeSourcePoisson, {'rate' : 2})

    connector = pynn.FixedProbabilityConnector(
            allow_self_connections=True,
            p_connect=0.5,
            weights=0.00425)
    proj_stim = pynn.Projection(source, neuron, connector, target="excitatory")
    proj_rec = pynn.Projection(neuron, neuron, connector, target="excitatory")

    pynn.run(1)

    print marocco.stats

    total_syns = 0
    lost_syns = 0
    for proj in [proj_stim, proj_rec]:
        l,t = projectionwise_synapse_loss(proj, marocco)
        total_syns += t
        lost_syns += l

    assert total_syns == marocco.stats.getSynapses()
    assert lost_syns == marocco.stats.getSynapseLoss()

    plot_projectionwise_synapse_loss(proj_stim, marocco)
    pynn.end()
```

Where `print marocco.stats` prints out overall synapse loss statistics:

```
MappingStats {
        synapse_loss: 581 (23.3709%)
        synapses: 2486
        synapses set: 1905
        synapses lost: 581
        synapses lost(l1): 0
        populations: 2
        projections: 2
        neurons: 50}
```

Invidual mapping statistics like the number of synapses set can also be directly accessed in python, see class `MappingStats` in the marocco documentation.

The function `projectionwise_synapse_loss` shows how to calculate the synapse loss per projection.

```python
def projectionwise_synapse_loss(proj, marocco):
    """
    computes the synapse loss of a projection
    params:
      proj    - a pyhmf.Projection
      marocco -  the PyMarocco object after the mapping has run.

    returns: (nr of lost synapses, total synapses in projection)
    """
    orig_weights = proj.getWeights(format='array')
    mapped_weights = marocco.stats.getWeights(proj)
    syns = np.where(~np.isnan(orig_weights))
    realized_syns = np.where(~np.isnan(mapped_weights))
    orig = len(syns[0])
    realized = len(realized_syns[0])
    print "Projection-Wise Synapse Loss", proj, (orig - realized)*100./orig
    return orig-realized, orig
```

Which yields the following output for the example above:

```
Projection-Wise Synapse Loss Projection ( PyAssembly (50) -> PyAssembly (50)) 23.
↪5576923077
Projection-Wise Synapse Loss Projection ( PyAssembly (50) -> PyAssembly (50)) 23.
↪182552504
```

Finally, the function `plot_projectionwise_synapse_loss` can be used to plot the lost and realized synapses of one projection.

```python
def plot_projectionwise_synapse_loss(proj, marocco):
    """
    plots the realized and lost synapses of a projection
    params:
      proj    - a pyhmf.Projection
      marocco -  the PyMarocco object after the mapping has run.
    """
    orig_weights = proj.getWeights(format='array')
    mapped_weights = marocco.stats.getWeights(proj)
    realized_syns = np.where(np.isfinite(mapped_weights))
    lost_syns = np.logical_and(np.isfinite(orig_weights), np.isnan(mapped_weights))

    conn_matrix = np.zeros(orig_weights.shape)
    conn_matrix[realized_syns] =  1.
    conn_matrix[lost_syns] = 0.5

    import matplotlib
    matplotlib.use('Agg')
    import matplotlib.pyplot as plt
    plt.figure()
    plt.subplot(111)
    plt.imshow(conn_matrix, cmap='hot', interpolation='nearest')
    plt.xlabel("post neuron")
    plt.ylabel("pre neuron")
    plt.title("realized and lost synapses")
    plt.savefig("synapse_loss.png")
```

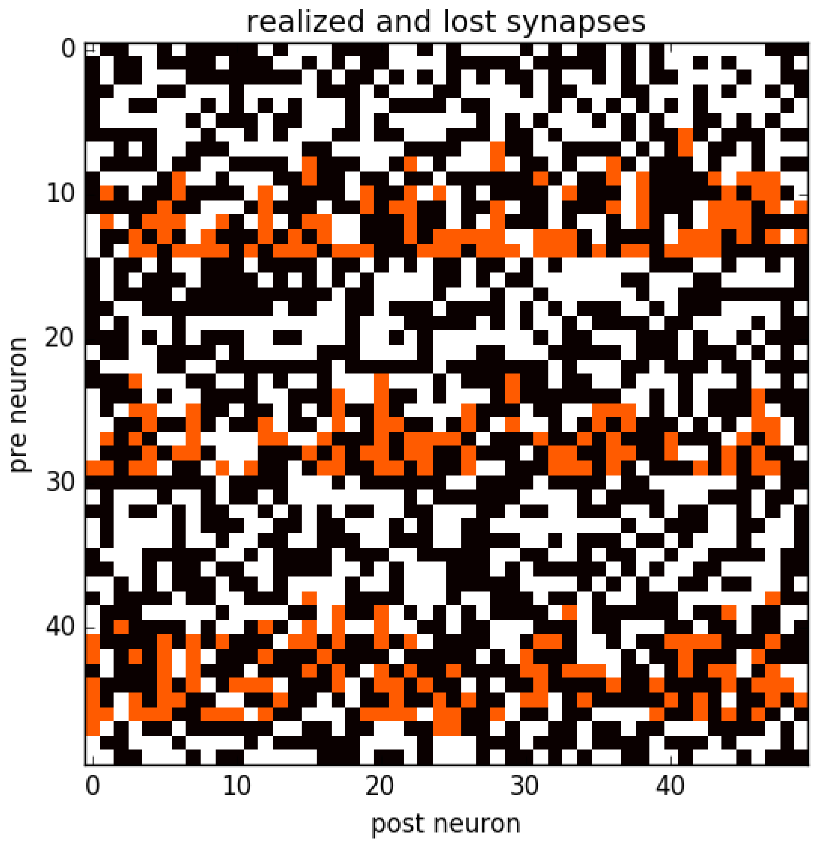


Fig. 4.4: Realized (black) and lost (red) synapses of the stimulus projection in the example network above.

4.2.4 Details of the software stack

The BrainScaleS Wafer-Scale Software Stack is shown in Figure 4.5.

User-provided neuronal network topologies are evaluated by our `PyNN` API implementation (`PyHMF`), which is written in C++ with a Python wrapper. The data structures (spike trains, populations, projections, cell types, meta information, etc.) are implemented in C++ (`euter`). This layer also provides a serialization and deserialization interface for lower software layers. In a nutshell, `euter` serializes the `PyNN`/`PyHMF`-based experiment description into a binary data stream and hands over to the next software layer. In the following software layers, the translation from this *biological* neuronal network description into a *hardware* configuration will be performed. A large fraction of the translation work, in particular the network graph translation, is performed by the `marocco` mapping tool (described in the PhD thesis of S. Jeltsch. Code documentation is provided by `doxygen` and available here.

`marocco` uses calibration (`calibtic`) and blacklisting (`redman`) information to take into account circuit-specific properties and defects. This information is needed during the map & route process to homogenize the behavior of hardware neuron and synapse circuits and to exclude defective parts of the system.

The blacklisting works on the component level, e.g. denmems (the building blocks of neurons), on-chip buses, synapse drivers, etc. Denmems are blacklisted during calibration. If the calibration for any parameter could not be performed, the denmem is blacklisted. The number of available neurons, i.e. connected denmems, is not only

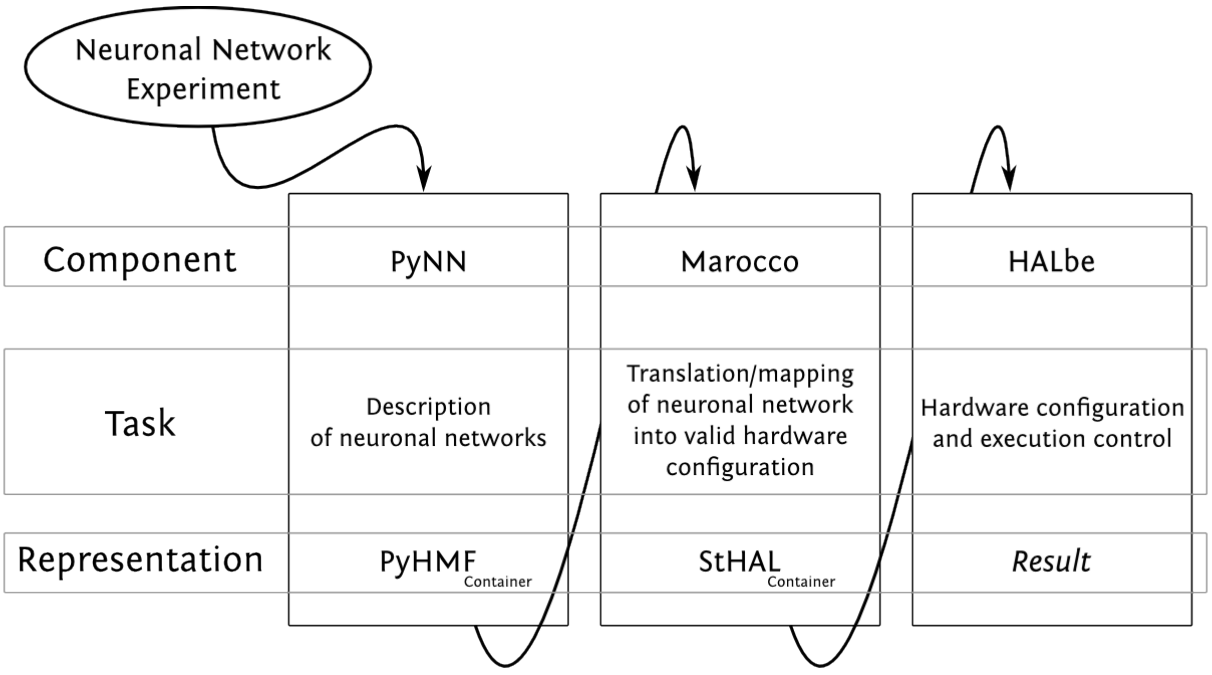

Neuronal Network Experiment
Component
PyNN
Marocco
HALbe
Task
Description of neuronal networks
Translation/mapping of neuronal network into valid hardware configuration
Hardware configuration and execution control
Representation
PyHMF Container
StHAL Container
Result

Fig. 4.5: Data-flow-centric view of the user software stack of the BrainScaleS Wafer-Scale System. [taken from PhD thesis of E. Müller]

depending on the number blacklisted denmems but also on their distribution. E.g. as an extreme example, if every second denmem would be blacklisted, neuron sizes larger than 1x2 (horizontal x vertical) are not possible.

The blacklisting for on-chip buses originates from the fact that every second horizontal and vertical bus is driven from a neighboring chip. If this neighbor chip can not be initialized, all buses from that chip must not be used.

4.3 Simulating the BrainScaleS hardware

The Executable System Specification (ESS) is a software model of the BrainScaleS hardware system. The ESS is implemented in C++/SystemC and contains functional models of all relevant units of the wafer-scale hardware. It is fully executable and resembles how neural experiments are run on the real wafer-scale system.

The ESS allows offline experimentation with the BrainScaleS system, and can be used for testing small models locally before submitting them to the hardware system, and for introspecting the system behaviour. The ESS supports PyNN versions 0.7 and 0.8.

Note, however, that the ESS runs *much* more slowly than the real hardware.

4.3.1 Installation

Using Docker

The simplest way to use the ESS is to use the (uhei/ess-system) Docker image. This has a recent version of the ESS already installed, and is updated on a regular basis.

Prerequisites

You need to have Docker installed. On Ubuntu Trusty 14.04 LTS the Docker package is called `docker.io`. Note that there is a package named `docker` in the Ubuntu repositories as well. It is something completely different.

After installing you may add yourself to the `docker` group. Otherwise you have to prepend sudo to most Docker commands.

You might also want to have a look at `/etc/default/docker.io` after installation, especially if you're sitting behind a proxy. These proxy settings are not for the containers, but for communication with the Docker

Repository (i.e. docker pull).

```
sudo apt-get install docker.io
# sudo adduser $USER docker
# sudo editor /etc/default/docker.io
# re-login
```

Download/upgrade of the uhei/ess-system image

The following step takes some time upon first execution, depending on your internet connection. Later updates should generally perform faster as only changes are pulled.

```
sudo docker pull uhei/ess-system:14.04
```

Starting the ESS container

The execution of the downloaded image creates a new Docker container. Note that Docker containers are not persistent, but one can link a host directory for persistent user data into the container. The following `docker run` command does just that. The host directory specified by the `VOLUME` environment variable will be available as `/bss/$USER` within the container. If you're interested in the option flags of the `docker run` command, run `docker help run`.

```
# docker help run
mkdir ess-data                  # where to put user/persistent data
VOLUME="${PWD}/ess-data"
sudo docker run --name ess-container --hostname ess-container \
        -v "$VOLUME:/bss/$USER" -ti "uhei/ess-system:14.04" /bin/bash
```

Testing your ESS container installation

You should be in the container now, as `root` in the directory `/bss` (as in BrainScaleS). An `ls` should show your folder for persistent data (under your user name or whatever you put instead of `$USER` in the `docker run` command above), as well as the directories `mappingtool_test`, `neurotools` and `tutorial`.

To test your installation, you can run some unit tests. This is almost the same as with the installation below, just the mapping tool test is installed at another location:

```
# root@ess-container:/bss#
python mappingtool_test/regression/run_ess_tests.py
python $SYMAP2IC_PATH/components/systemsim/test/regression/run_ess_tests.py
python $SYMAP2IC_PATH/components/systemsim/test/system/run_ess_tests.py
```

Installing from source

Note: these instructions have been tested on a native Ubuntu saucy 13.10 on a 64-bit machine

Prerequisites

To be able to configure and compile the symap2ic project, you need to install the following libraries:

```
apt-get -y install git python-pip python-dev build-essential libgtest-dev \
    libboost-all-dev libpng12-dev libssl-dev libmongo-client-dev mongodb \
    liblog4cxx10-dev autotools-dev automake
```

The ESS expects the 64-bit libraries to lie either in /lib64 or /usr/lib64. However, in Ubuntu 13.10, the 64-bit libraries lie in /usr/lib/x86_64-linux-gnu. So, you need to make the following symbolic links:

```
ln -s /usr/lib/x86_64-linux-gnu /usr/lib64
ln -s /usr/lib/libmongoclient.a /usr/lib/x86_64-linux-gnu/libmongoclient.a
```

To be able to run the tests and to use the ESS, you also need to install:

```
apt-get -y install libgsl0-dev libncurses5-dev libreadline-dev gfortran \
    libfreetype6-dev libblas-dev liblapack-dev r-base python-rpy \
pip install numpy scipy matplotlib PIL NeuroTools mpi4py xmlrunner
```

You should then install PyNN:

```
pip install PyNN  # PyNN 0.8
```

or

```
pip install PyNN==0.7.5  # PyNN 0.7
```

Installation of the ESS

You should first obtain an account from the heidelberg group. Then, on your computer, you generate a rsa key:

```
ssh-keygen -t rsa
```

Suppose that you have saved the key in the file ~/.ssh/id_rsa. In the heidelberg website, you go to 'My account' (upper-right). You click on 'Public Key' in the upper-right corner. You click on 'New value' and paste the content of your computer's id_rsa.pub. Wait until the activation is done.

Then, you can download and install the ESS on your computer:

```
cd
git clone git@brainscales-r.kip.uni-heidelberg.de:symap2ic.git
cd symap2ic
source bootstrap.sh.UHEI .
```

For PyNN 0.8:

```
./waf set_config systemsim-pynn8
./waf update
```

For PyNN 0.7:

For PyNN 0.8:

```
./waf set_config systemsim
```

If you have had problems in the execution of the 4 lines above, you have some read access right problems from the repositories. Please e-mail neuromorphic@humanbrainproject.eu. Please now go on by configuring and installing the system:

```
./waf configure --stage=brainscales --use-systemsim --without-hardware \
    --prefix=$HOME/symap2ic
./waf install
```

You now set the environment variables:

```
echo 'export SYMAP2IC_PATH=$HOME/symap2ic' >> ~/.bashrc
echo 'export PYTHONPATH=$PYTHONPATH:$SYMAP2IC_PATH/lib' >> ~/.bashrc
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SYMAP2IC_PATH/lib' >> ~/.bashrc
bash
```

For PyNN 0.7, you need to copy the PyNN hardware directory into the PyNN package:

```
cd $SYMAP2IC_PATH
cp -r components/pynnhw/misc/pyNN_hardware_patch/hardware \
    /usr/local/lib/python2.7/dist-packages/pyNN/
```

You can now test that the hardware backend is accessible:

```
python -c 'import pyNN.hardware.brainscales as sim'
```

To test your installation with PyNN 0.7, you can run some unit tests:

```
python $SYMAP2IC_PATH/components/mappingtool/test/regression/run_ess_tests.py
python $SYMAP2IC_PATH/components/systemsim/test/regression/run_ess_tests.py
python $SYMAP2IC_PATH/components/systemsim/test/system/run_ess_tests.py
```

To test your installation with PyNN 0.8, you can run the PyNN unit and system tests:

```
cd ~/PyNN-8/test
cd unittests/backends
nosetests test_mock.py
nosetests test_hardware_brainscales.py
```

4.3.2 Using the ESS

Scripts to run on the ESS should in general be identical to those that run on the BrainScaleS hardware. The only required difference is to choose `PyMarocco.ESS` as the PyMarocco backend.

Example

A full example where an Adaptive-Exponential Integrate & Fire neuron is stimulated by external spikes, is shown in `nmpm1_adex_neuron_ess.py`:

```
#!/usr/bin/env python

"""
Example Script for simulation of an AdEx neuron on the ESS

Note: Neuron and synapse parameters are chosen to be within the parameter ranges of
the default calibration.
"""

import pyhmf as pynn
#import pyNN.nest as pynn
from pymarocco import PyMarocco, Defects
import pylogging
import Coordinate as C
import pysthal

# configure logging
pylogging.reset()
pylogging.default_config(level=pylogging.LogLevel.INFO,
        fname="logfile.txt",
        dual=False)

# Mapping config
marocco = PyMarocco()
marocco.backend = PyMarocco.ESS # choose Executable System Specification instead␣
↪of real hardware
marocco.calib_backend = PyMarocco.CalibBackend.Default
marocco.defects.backend = Defects.Backend.None
marocco.neuron_placement.skip_hicanns_without_neuron_blacklisting(False)
marocco.hicann_configurator = pysthal.HICANNConfigurator()
marocco.experiment_time_offset = 5.e-7 # can be low for ESS, as no repeater␣
↪locking required
```

```
marocco.neuron_placement.default_neuron_size(4) # default number of hardware␣
↪neuron circuits per pyNN neuron
marocco.persist = "nmpm1_adex_neuron_ess.bin"
marocco.param_trafo.use_big_capacitors = False

# set-up the simulator
pynn.setup(marocco=marocco)

neuron_count = 1 # size of the Population we will create

# Set the neuron model class
neuron_model = pynn.EIF_cond_exp_isfa_ista # an Adaptive Exponential I&F Neuron

neuron_parameters = {
 'a'           : 4.0,    # adaptation variable a in nS
 'b'           : 0.0805, # adaptation variable b in pA
 'cm'          : 0.281,  # membrane capacitance nF
 'delta_T'     : 1.0,    # delta_T fom Adex mod in mV, determines the sharpness of␣
↪spike initiation
 'e_rev_E'     : 0.0,    # excitatory reversal potential in mV
 'e_rev_I'     : -80.0,  # inhibitory reversal potential in mV
 'i_offset'    : 0.0,    # offset current
 'tau_m'       : 9.3667, # membrane time constant
 'tau_refrac'  : 0.2,    # absolute refractory period
 'tau_syn_E'   : 20.0,   # excitatory synaptic time constant
 'tau_syn_I'   : 20.0,   # inhibitory synaptic time constant
 'tau_w'       : 144.0,  # adaptation time constant
 'v_reset'     : -70.6,  # reset potential in mV
 'v_rest'      : -70.6,  # resting potential in mV
 'v_spike'     : -40.0,  # spike detection voltage in mV
 'v_thresh'    : -50.4,  # spike initiaton threshold voltage in mV
 }

# We create a Population with 1 neuron of our neuron model
N1 = pynn.Population(size=neuron_count, cellclass=neuron_model, cellparams=neuron_
↪parameters)

# A spike source array with spike times given in a list
spktimes = [10., 50., 65., 89., 233., 245.,255.,345.,444.4]
spike_source = pynn.Population(1, pynn.SpikeSourceArray, {'spike_times':spktimes})

# Connect the Spike source to our neuron
pynn.Projection(spike_source, N1, pynn.OneToOneConnector(weights=0.0138445),␣
↪target='excitatory')

# record the membrane voltage of all neurons of the population
N1.record_v()
# record the spikes of all neurons of the population
N1.record()

# run the simulation for 500 ms
duration = 500.
pynn.run(duration)


# After the simulation, we get Spikes
spike_times = N1.getSpikes()
for pair in spike_times:
    print "Neuron ", int(pair[0]), " spiked at ", pair[1]

# Plot voltage
do_plot = False
if do_plot:
    import pylab
    v = N1.get_v()[:,1:3] # strip ID
    pylab.plot(v[:,0], v[:,1])
    pylab.xlabel("Time [ms]")
    pylab.ylabel("Voltage [mV]")
    pylab.xlim(0,duration)
    pylab.show()

# clean up pyNN
pynn.end()
```

Note that the same script runs also with `pyNN.nest`, just change the first line that imports the PyNN backend.

ESS Config

In addition, one can specify an ESS configuration as follows:

```
import pysthal

marocco = PyMarocco()
marocco.backend = PyMarocco.ESS

ess_config = pysthal.ESSConfig()
ess_config.enable_weight_distortion = True
ess_config.weight_distortion = 0.2
ess_config.pulse_statistics_file = "pulse_stats.py"

marocco.ess_config = ess_config
```

parameters of `ESSConfig`:

`enable_weight_distortion` Enables the distortion of synaptic weights in the virtual hardware system.

This option can be used to resemble the fixed pattern noise of synaptic weights on the real hardware.

Default: `False`

`weight_distortion` Specifies the distortion of synaptic weights in the virtual hardware system.

This parameter defines the fraction of the original value, that is used as the standard deviation for randomizing the weight according to a normal distribution around the original value. All weights are clipped to positive values.

Default: `0.0`

`pulse_statistics_file` Name of file to which the ESS pulse statistics are written.

See *Pulse Loss Statistics* for details.

Default: `""`

Pulse Loss Statistics

The ESS allows to count all spikes that were lost in any place of the virtual hardware system. Spikes are mostly lost in the off-wafer communication network (also called ‘’Layer 2 network‘‘) that connects the wafer to the host PC. In the Layer 2 network pulse loss can happen on two routes:

1. Stimulation: not all spikes from the spike sources (`SpikeSourcePoisson` or `SpikeSourceArray`) are delivered to its targets, because the bandwidth in the off-wafer network is limited. When a spike is lost, it is lost for its targets.
2. Recording: For the same bandwidth constraints in the off-wafer network, some spikes of real neurons can be lost on the route from the wafer to the FGPGAs, Hence, in the received spike data some events are missing. However, the ‘non-recorded’ spikes did reach their target neurons on the wafer.

Spikes can also be lost on the wafer, but only in rare cases when many neuron located on the same HICANN fire synchronously.

3. On-wafer Spike Loss: This is the case of pulses lost in the on-wafer pulse-communication system (also called *Layer 1 network*). If this happens, spikes are completely deleted, and reach no other neuron.
4. Spike Drop before Simulation: The playback module of the FPGA, which plays back the stimuli pulses at given times, also has a limited bandwidth. This limitation is considered beforehand, such that spikes are dropped even before the simulation, in order to avoid a further delaying of many more spikes during an experiment.

The ESS counts the lost and sent pulses. After the simulation, you will see something in the log for a loglevel>=2:

```
INFO  Default *************************************
INFO  Default LostEventLogger::summary
INFO  Default Layer 2 events dropped before sim : 837/3939 (21.249 %)
INFO  Default Layer 2 events lost :               243/3199 (7.59612 %)
INFO  Default Layer 2 events lost downwards :     243/3102 (7.83366 %)
INFO  Default Layer 2 events lost upwards   :     0/97 (0 %)
INFO  Default Layer 1 events lost : 0/79 (0 %)
INFO  Default *************************************
```

You can specify to get this data by specifying a file `pulse_statistics_file` in the *ESS Config*:

```
marocco.ess_config.pulse_statistics_file = "pulse_stats.py"
sim.setup(marocco=marocco)
```

Then the pulse statistics file contains a Python dictionary `pulse_statistics` which can be use for further processing:

```
pulse_statistics = {
'l2_down_before_sim': 3939,
'l2_down_dropped_before_sim': 837,
'l2_down_sent': 3102,
'l2_down_lost': 243,
'l2_up_sent': 97,
'l2_up_lost': 0,
'l1_neuron_sent': 79,
'l1_neuron_lost': 0,
}
```

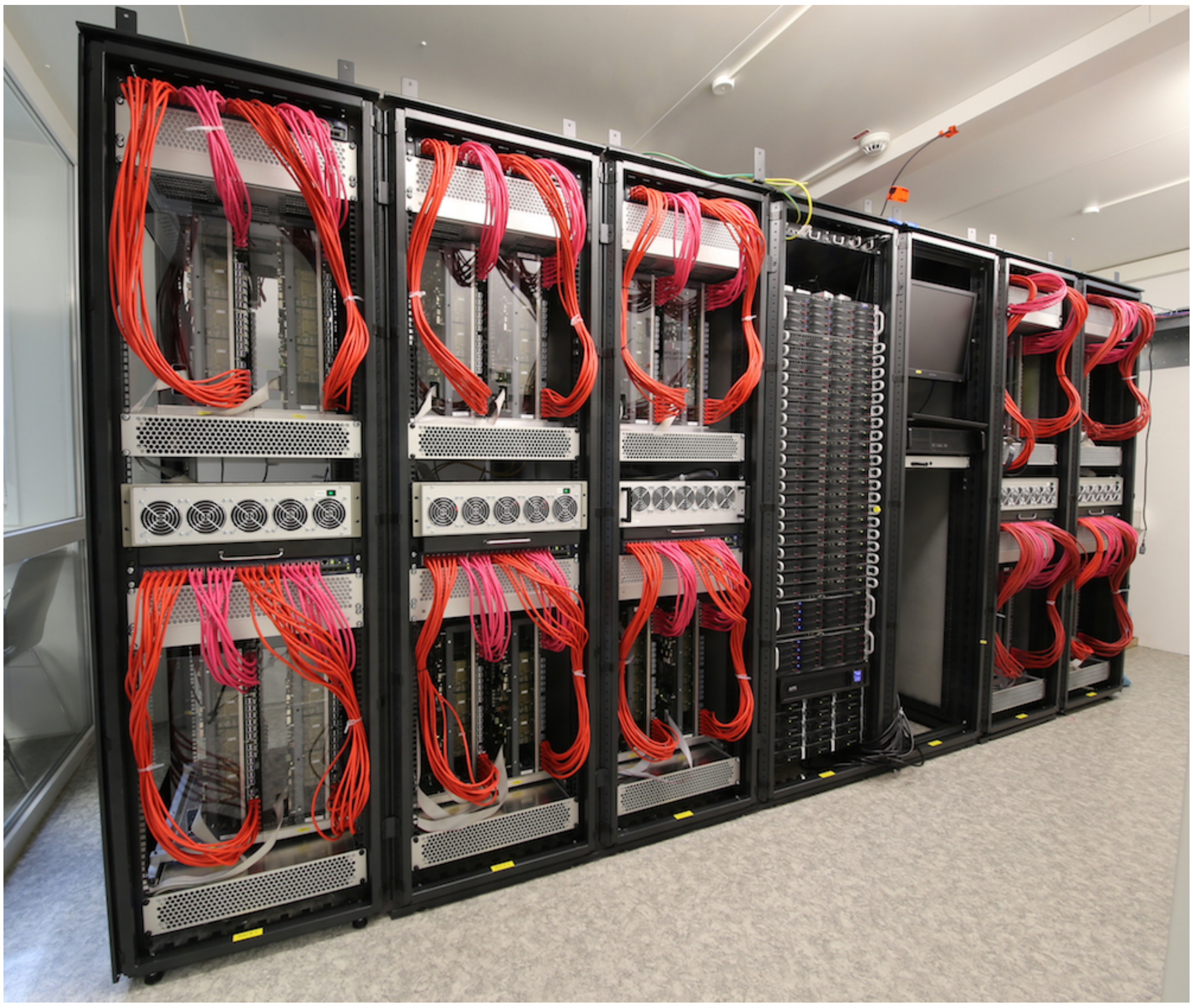


Fig. 4.6: The BrainScaleS (NM-PM-1) system in Heidelberg on 30 March 2016

CHAPTER

FIVE

THE SPIKEY "PHYSICAL MODEL" SYSTEM

5.1 Experiment execution

Use the following instructions to login, load the pre-built software package and run experiments on the UHEI cluster. For using the Spikey system outside the UHEI cluster and for developers, please manually install the required software (see *Setup software*).

5.1.1 Login to UHEI cluster

- From the network of the Kirchhoff-Institute for Physics:

```
ssh KIPUSER@hel
```

- Otherwise:

```
ssh s1ext_someuser@gitviz.kip.uni-heidelberg.de -p 11022
```

5.1.2 Load software modules

To load the pre-built software package and configure the environment variables use:

```
. /wang/environment/software/jessie/spack/current/share/spack/setup-env.sh
spack load --dependencies visionary-defaults
module load spikey
```

These three commands must be loaded for every environment, and in particular, after logging in.

5.1.3 Run experiment

Download the Spikey example experiment. For more example networks see *Spikey school* and Spikey demos. On the UHEI cluster SLURM is used to manage the workload on our systems. To queue the execution of a Python script use:

```
srun -p spikey --gres stationXXX python example.py
```

Replace XXX with the number of the chip you want to use (e.g. 500).

5.1.4 Tips and tricks

- To view the queue of experiments use:

```
squeue
```

- To query a list of available Spikey systems on the UHEI cluster see the chip status page.
- The setup of SSH keys for pubkey-based access to github.com is described in *Setting up SSH keys on UHEI cluster for github.com*.

5.1.5 Using the web interface of the Human Brain Project

To select a chip using the web interface of the Human Brain Project enter the following parameters into the "Hardware Config" box:

```
{"STATION":"stationXXX"}
```

Replace XXX with the number of the chip you want to use (e.g. 500).

5.1.6 Visualization of the chip configuration

What is the *scvisual* tool?

The tool *scvisual* (for "spikey config visualization") displays a visualization of the almost complete hardware configuration of an experiment. It loads data from the file "spikeyconfig.out" which is generated during experiment execution in your working directory. The program *scvisual* enables you to

- investigate the full network connectivity of a Spikey experiment,
- zoom into details of both 256x192-sized synapse arrays,
- hover over neurons and synapse drivers to inspect their configuration,
- retrieve detailed information on most analog hardware parameters,
- compare the displayed configuration with a photo of the chip, and
- save high-resolution png figures of you network configuration to disk.

The tool is written in python and makes extensive use of the interactive functionality of matplotlib.

Who can benefit from *scvisual*?

The tool is designed for both novice and experienced users. If you are new to using the Spikey system, *scvisual* gives you an impression of where the physical model neurons and synapses are actually located on the chip. To our experience, the tool can further be very helpful to expert users for developing and debugging PyNN-based experiment descriptions.

Installation and usage

For local users

Users located at the institute or operating "their own" Spikey chip at a local computer[1] simply type:

$ scvisual

in the working directory after the experiment is completed.

For remote users

The ssh connection will likely not support remote execution of the interactive graphical interface. The good news is: you can still run the program on your local machine – even without a full software installation. From

https://github.com/electronicvisions/spikeyhal/tree/flyspi/tools

download the following files: *scvisual.py, scparse.py, spikey_gold_label_medium.png*. Then rename *scvisual.py* to *scvisual*, set *scvisual* to be executable, and make all files available in your *$PATH* and *$PYTHONPATH*. Now

$ scvisual

[1] More precisely, access to a local installation of the operation software is required. For installation instructions see *Setup software*.

should work in any local directory that contains a "spikeyconfig.out" file.

Additonal options

The 'pure' command *scvisual* will load the experiment configuration from "spikeyconfig.out" with reasonable plotting options. For a description of additional options type:

$ scvisual -h

Footnotes

5.2 Spikey school

5.2.1 Introduction to the hardware system

For an introduction to the Spikey neuromorphic system, its neuron and synapse models, its topology and its configuration space, see section 2 in *[Pfeil2013]*. Detailed information about the analog implementation of the neuron, STP and STDP models are given in Figure 17 in *[Indiveri2011]*, *[Schemmel2007]* and *[Schemmel2006]*, respectively. The digital parts of the chip architecture are thoroughly documented in *[Gruebl2007Phd]*. The following paragraph will briefly summarize the features of the Spikey system.

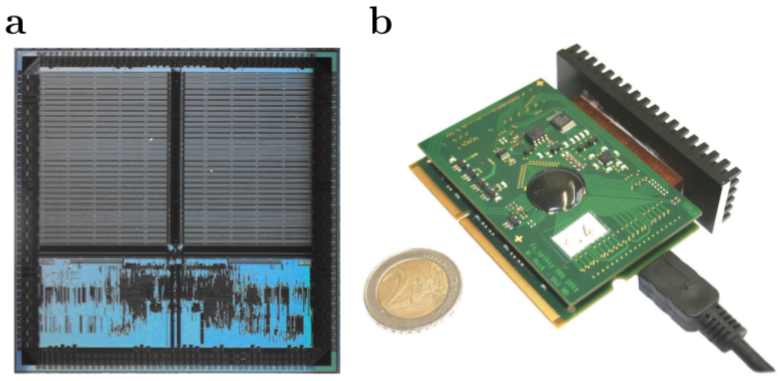


Fig. 5.1: A photograph of the Spikey neuromorphic chip (a) and system (b), respectively. In (b) the Spikey chip is covered by a black seal. Adapted from *[Pfeil2013]*.

The Spikey chip is fabricated in a 180nm CMOS process with die size $5\,mm \times 5\,mm$. While the communication to the host computer is mostly established by digital circuits, the spiking neural network is mostly implemented with analog circuits. Compared to biological real-time, networks on the Spikey chip are accelerated, because time constants on the chip are approximately 10^4 times smaller than in biology. Each neuron and synapse has its physical implementation on the chip. The total number of 384 neurons are split into two blocks of 192 neurons with 256 synapses each. Each line of synapses in these blocks, i.e. 192 synapses, is driven by a *line driver* that can be configured to receive input from external spike sources (e.g., generated on the host computer), from on-chip neurons in the same block or from on-chip neurons in the adjacent block.

Neuron assignment to line drivers:

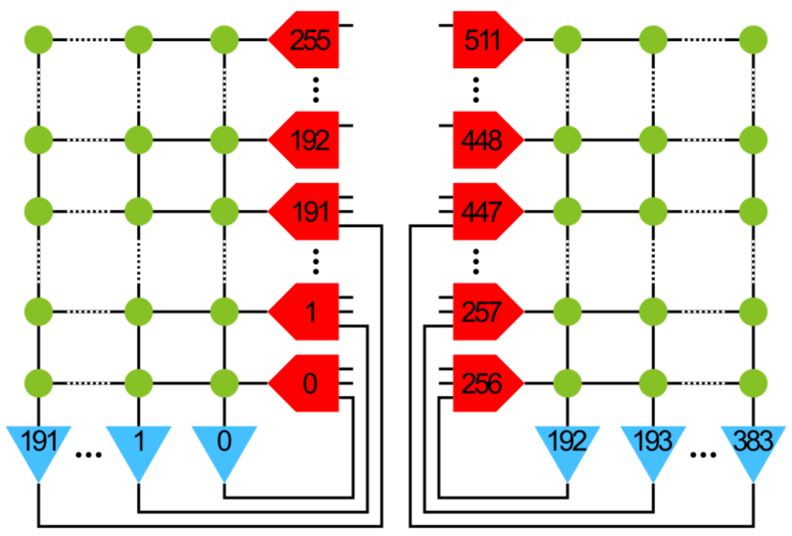


Fig. 5.2: Numbering of neurons (blue) and line drivers (red). Here, only connections within the same block of neurons are shown. For connections between the blocks see the following table. The weight of each synapse (green) can be configured with a 4-bit resolution, i.e., 16 different values.

Line driver ID	Neuron ID left block	Neuron ID right block	Line driver ID	Neuron ID left block	Neuron ID right block
0	0	193	256	192	1
1	1	192	257	193	0
2	2	195	258	194	3
3	3	194	259	195	2
…	…	…	…	…	…
190	190	383	446	382	191
191	191	382	447	383	190
192	ext only	ext	448	ext only	ext only only
…	…	…	…	…	…
255	ext only	ext	511	ext only	ext only only

The last (upper) 64 line drivers receive external inputs only and hence external spike sources line drivers are allocated from top to bottom.

The hardware implementations of neurons and synapses are inspired by the leaky integrate-and-fire neuron model using synapses with exponentially decaying or alpha-shaped conductances (PyNN neuron model `IF_facets_hardware1`). While the leak conductance (PyNN neuron model parameter `g_leak`) and (absolute) refractory period (`tau_refrac`) is individually configurable for each neuron, the resting (`v_rest`), reset (`v_reset`), threshold (`v_thresh`), excitatory reversal (clamped to ground) and inhibitory reversal potentials (`e_rev_I`) are shared among neurons (see *[Pfeil2013]* for details). Line drivers generate the time course of postsynaptic conductances (PSCs) for a single row of synapses. Among other parameters the rise time, fall time and amplitude of PSCs can be modulated for each line driver (for details see *Lesson 1: Exploring the parameter space* and Figure 4.8 and 4.9 in *[Petkov2012]*). Each synapse stores a configurable 4-bit weight. A synapse can be turned off, if its weight is set to zero.

During network emulations, spike times can be recorded from all neurons in parallel. In contrast, membrane potential recordings are limited to a single, but arbitrary, neuron.

Network models for the Spikey hardware are described and controlled by PyNN (version 0.6; for an introduction to PyNN see *Building models*). Due to the fact that PyNN is a Python package we recommend to have a look at a Python tutorial. For efficient data analysis and visualization with Python see tutorials for Numpy, Matplotlib and Scipy.

Short-term plasticity (STP)

Synaptic efficacy has been shown to change with presynaptic activity on the time scale of hundred milliseconds *[ScholarpediaShortTermPlasticity]*. The hardware implementation of such short-term plasticity is close to the model introduced by *[Tsodyks1997]*. However, on hardware STP can either be depressing or facilitating, but not mixtures of both as allowed by the original model. For details about the hardware implementation and emulation results, see *[Schemmel2007]* and *Lesson 5: Short-term plasticity*, respectively.

Spike-timing dependent plasticity (STDP)

Long-term (seconds to years) modification of synaptic weights has been shown to depend on the precise timing of spikes *[ScholarpediaSTDP]*. Weights are usually increased, if the postsynaptic neuron fires after the presynaptic one, and decreased for the opposite case. Typically, synaptic weights change the more the smaller this temporal correlation is. On hardware temporal correlations between pre- and postsynaptic neurons are measured and stored locally in each synapse. Then a global mechanism sequentially evaluates these measurements and updates the synaptic weight according to a programmable look-up table.

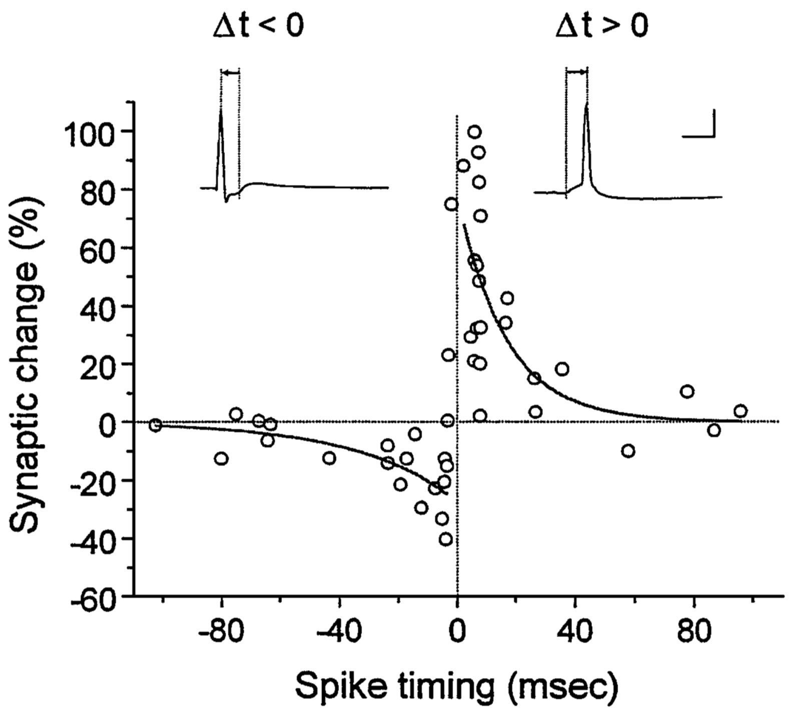


Fig. 5.3: Spike-timing dependent plasticity measured in biological tissue (rat hippocampal neurons; adapted from *[Bi2001]*).

For a detailed description of the hardware implementation, measurements of single synapses and functional networks, see *[Schemmel2006]*, *[Pfeil2012STDP]* and *[Pfeil2013STDP]*, respectively. Note that on hardware the reduced symmetric nearest neighbor spike pairing scheme is used (see Figure 7C in *[Morrison2008]*).

5.2.2 Introduction to the lessons

Note that all emulation results shown in the following lessons were recorded from the Spikey chip 666 and may be different for other chips. In particular, network, neuron and synapse parameters may have to be adjusted for proper network activity. In the following we use `pynn` as an acronym for `pyNN.hardware.stage1`.

5.2.3 Lesson 1: Exploring the parameter space

In this lesson, we explore the parameter space of neurons and synapses on the Spikey chip.

First, the parameters of neurons are investigated. As an example, we measure the firing rate of a neuron in dependence on its leak conductance. The neuron is stimulated by spikes from a Poisson process.

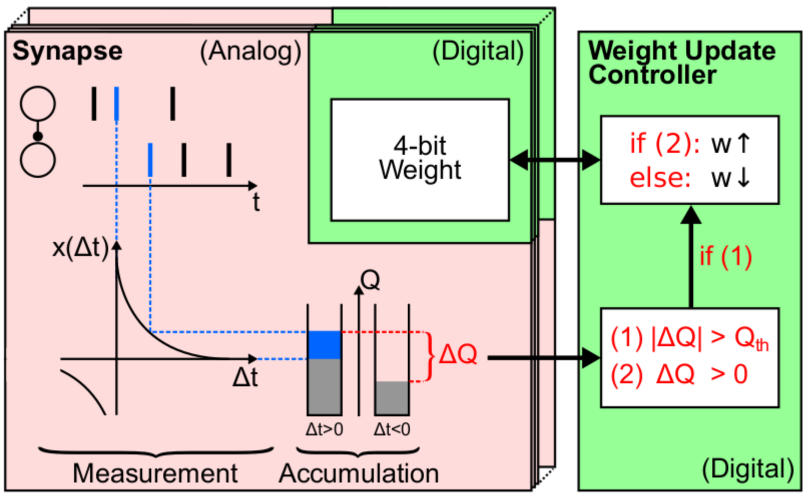


Fig. 5.4: Hardware implementation of STDP (adapted from *[Pfeil2015Phd]*).

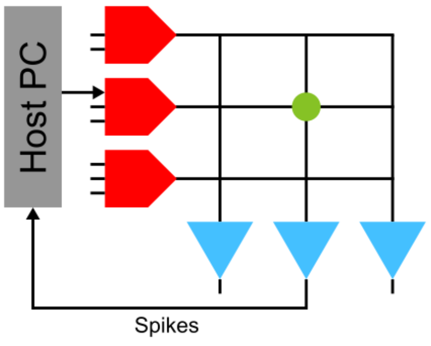


Fig. 5.5: A neuron is stimulated using an external spike source and the spike times of the neuron are recorded. Synapses with weight zero are not drawn.

To average out fixed-pattern noise (see *Lesson 2: Fixed-pattern and temporal noise:*) in both the synapse and neuron circuits, a population of neurons is stimulated by a population of spike sources.

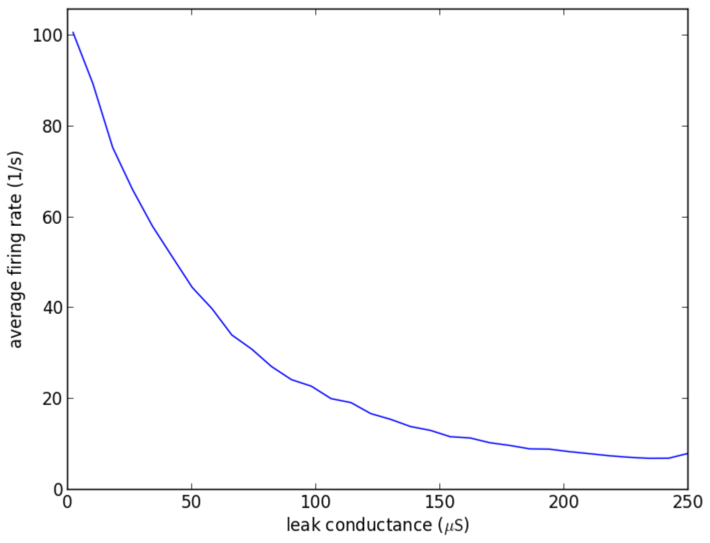


Fig. 5.6: The firing rate of the neuron in dependence on its leak conductance g_{leak} (source code lesson 1-1).

By increasing the leak conductance of the neuron its membrane potential is pulled towards the resting potential and hence the firing rate of the neuron decreases.

Tasks:

- Measure and plot the dependency of the firing rate on other neuron parameters (for parameters, see *Introduction to the hardware system*). Interpret these dependencies qualitatively?
- Calibrate the firing rate of the neuron to a reasonable target rate by adjusting its leak conductance.
- Replace the input from a Poisson process (`pynn.SpikeSourcePoisson`) by a regular input with the same rate (tipp: use `pynn.SpikeSourceArray`). What do you observe?

Second, synaptic parameters are investigated. A neuron is stimulated with a single spike and its membrane potential is recorded. To average out noise on the membrane potential (mostly caused by the readout process) we stimulate the neuron with a regular spike train and calculate the spike-triggered average of these so-called excitatory postsynaptic potentials (EPSPs).

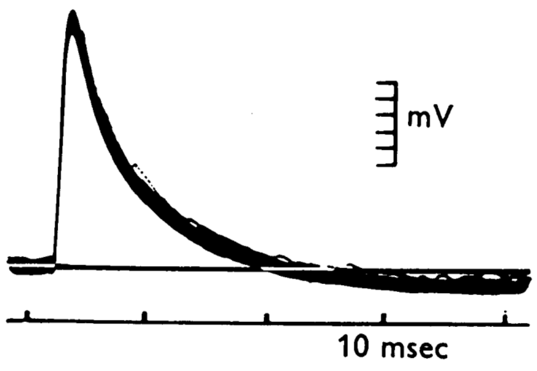


Fig. 5.7: Postsynaptic potentials measured in biological tissue (from motoneurons; adapted from *[Coombs1955]*).

Tasks:

- Vary the parameters `drvifall` and `drviout` of the synapse line drivers and investigate their effect on the shape of EPSPs (tipp: use `pynn.Projection.setDrvifallFactors` and `pynn.Projection.setDrvioutFactors` to scale these parameters, respectively).

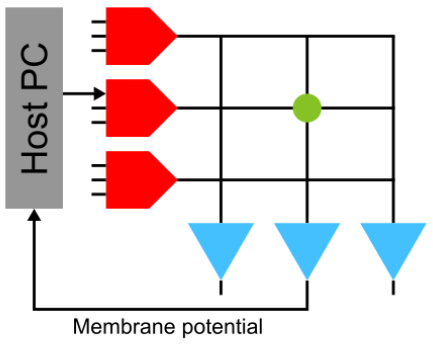


Fig. 5.8: A neuron is stimulated using a single synapse and its membrane potential is recorded. The parameters of synapses are adjusted row-wise in the line drivers (red).

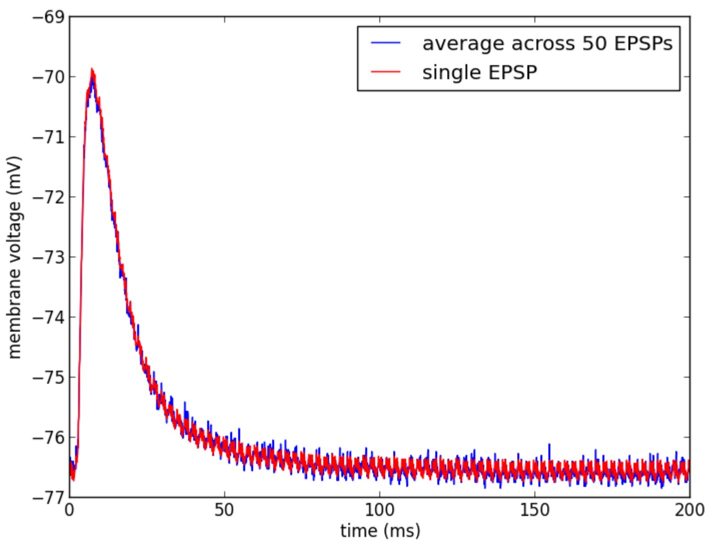


Fig. 5.9: Single and averaged excitatory postsynaptic potentials (source code lesson 1-2).

- Compare the EPSPs between excitatory to inhibitory synapses.
- Compare the shape of the first EPSPs. They may differ due to the initial loading of capacities (e.g. wires). Discard an appropriate number of EPSPs at the beginning of the emulation to avoid these distortions.

5.2.4 Lesson 2: Fixed-pattern and temporal noise:

In this lesson, we investigate fixed-pattern and temporal noise in the analog neuron and synapse circuits of the Spikey hardware system.

In contrast to simulations with software, emulations on analog neuromorphic hardware are subject to noise. We distinguish between fixed-pattern and temporal noise. Fixed-pattern noise are variations of neuron and synapse parameters across the chip due to imperfections in the production process. Calibration can reduce this noise, because it is approximately constant over time. In contrast, temporal noise, including electronic noise and temperature fluctuations, causes different results in consecutive emulations of identical networks.

Tasks:

- Investigate the fixed-pattern noise across neurons: Record the firing rates of several neurons for the default value of the leak conductance (see Figure 5.5 and 5.6; tipp: record all neurons at once). Interpret the distribution of these firing rates by plotting a histogram and calculating the variance.
- Investigate the fixed-pattern noise across synapses: For a single neuron, vary the row of the stimulating synapse and calculate the variance of the area under the EPSPs across synapses (see Figure 5.8 and 5.9).
- Estimate the ratio between fixed-pattern and temporal noise: Measure the reproducibility of emulations, i.e., the error of firing rates across identical consecutive trials. Use the network and parameters from the first task and measure this error for each neuron. Compare the variance of the firing rates across trials (averaged across neurons) to that one across neurons in a single trial. Extra: How does the reproducibility depend on the duration of emulations and the number of consecutive trials?

5.2.5 Lesson 3: Feedforward networks

In this lesson, we learn how to setup networks on the Spikey system. In the last lessons neurons received their input exclusively from external spike sources. Now, we introduce connections between hardware neurons. As an example, a synfire chain with feedforward inhibition is implemented (for details, see *[Pfeil2013]*). Populations of neurons represent the links in this chain and are unidirectionally interconnected. After stimulating the first neuron population, network activity propagates along the chain, whereby neurons of the same population fire synchronously.

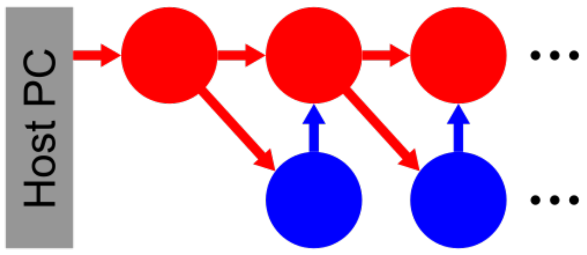


Fig. 5.10: Schematic of a synfire chain with feedforward inhibition. Excitatory and inhibitory neurons are coloured red and blue, respectively.

In PyNN connections between hardware neurons can be treated like connections from external spike sources to hardware neurons. Note that synaptic weights on hardware can be configured with integer values in the range [0..15]. To stay within the range of synaptic weights supported by the hardware, it is useful to specify weights in the domain of these integer values and translate them into biological parameter domain by multiplying them with `pynn.minExcWeight()` or `pynn.minInhWeight()` for excitatory and inhibitory connections, respectively. Synaptic weights that are not multiples of `pynn.minExcWeight()` and `pynn.minInhWeight()` for excitatory and inhibitory synapses, respectively, are stochastically rounded to integer values.

Tasks:

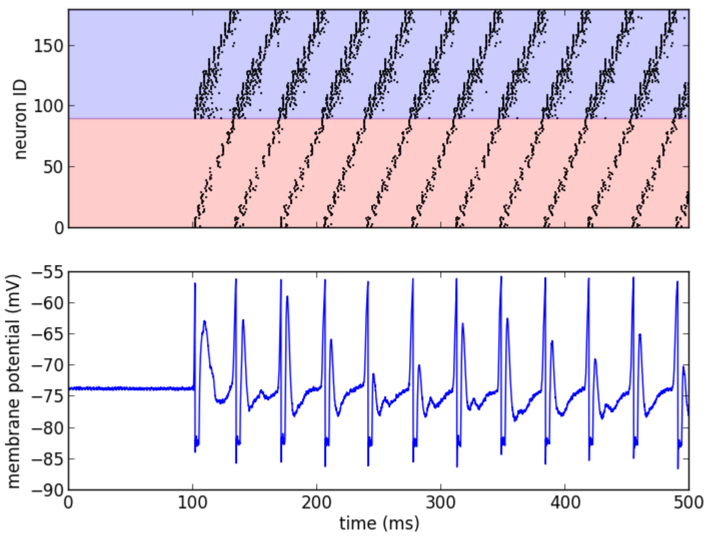


Fig. 5.11: Emulated network activity of the synfire chain including the membrane potential of the neuron with ID=0 (source code lesson 3). The same color code as in the schematic is used.

- Adjust the synaptic weights to obtain a loop of network activity that lasts for at least 1000 seconds.
- Reduce the number of neurons in each population and maximize the period of network activity. Which hardware feature limits the minimal number of neurons in each population?
- Open the loop and increase the number of neurons in each population to obtain a stable propagation of network activity. Systematically vary the initial stimulus (number of spikes and standard deviation of their timing) to investigate the filter properties of this network (for orientation, see *[Kremkow2010]* and *[Pfeil2013]*).

5.2.6 Lesson 4: Recurrent networks

In this lesson, a recurrent network of neurons with sparse and random connections is investigated. To avoid self-reinforcing network activity that may arise from excitatory connections, we choose connections between neurons to be inhibitory with weight w. Each neuron is configured to have a fixed number K of presynaptic partners that are randomly drawn from all hardware neurons (for details, see *[Pfeil2015]*). Neurons are stimulated by a constant current that drives the neuron above threshold in the absence of external input. Technically this current is implemented by setting the resting potential above the firing threshold of the neuron. The absence of external stimulation cancels the transfer of spikes to the system and accelerates the experiment execution. In addition, once configured this recurrent network runs hypothetically forever.

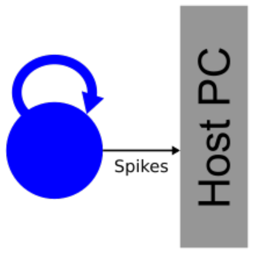


Fig. 5.12: Schematic of the recurrent network. Neurons within a population of inhibitory neurons are randomly and sparsely connected to each other.

Tasks:

- For each neuron, measure the firing rate and plot it against the coefficient of variation (CVs) of inter-spike

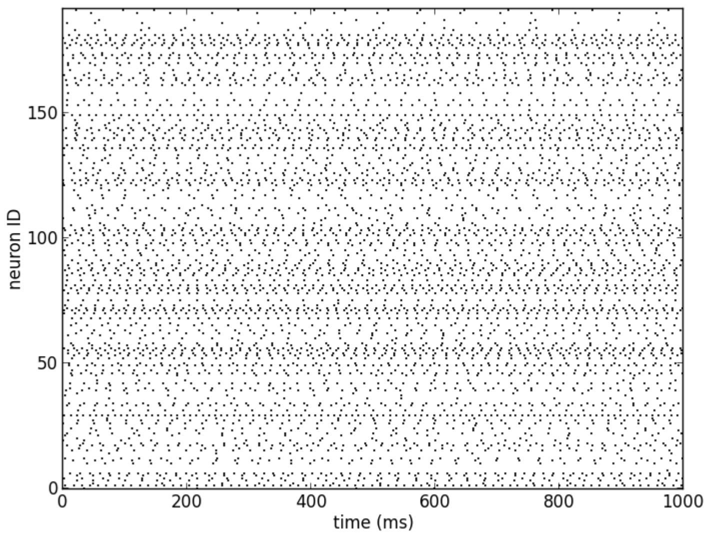


Fig. 5.13: Network activity of a recurrent network with $K = 15$ (source code lesson 4).

intervals. Interpret the correlation between firing rates and CVs.

- Measure the dependence of the firing rates and CVs on w and K. Calibrate the network towards a firing rate of approximately $25\frac{1}{s}$. Extra: And maximize the average CV.
- Extra: Calculate the pair-wise correlation between randomly drawn spike trains of different neurons in the network (consider using http://neuralensemble.org/elephant/ to calculate the correlation). Investigate the dependence of the average correlation on w and K (tipp: use 100 pairs of neurons to calculate the average). Use these results to minimize correlations in the activity of the network.

5.2.7 Lesson 5: Short-term plasticity

In this lesson, the hardware implementation of *Short-term plasticity (STP)* is investigated. The network description is similar to that shown in Figure 5.8, but with STP enabled in the synapse line driver.

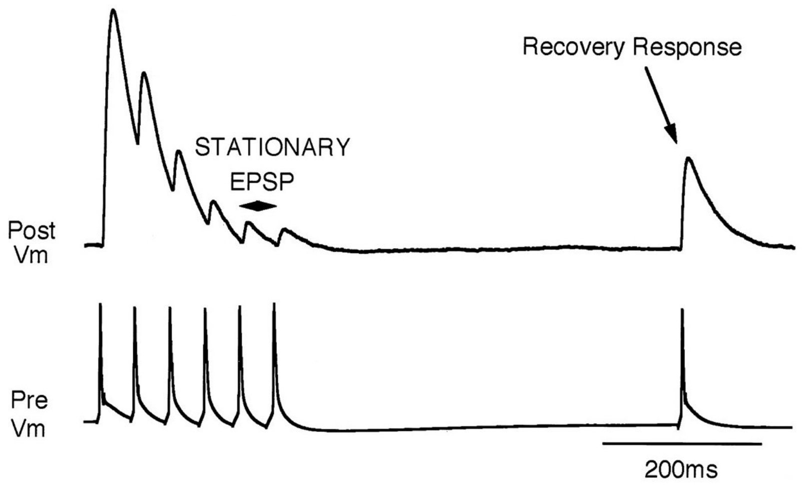


Fig. 5.14: Depressing STP measured in biological tissue (adapted from *[Tsodyks1997]*).

The weight of the synapse decreases with each presynaptic spike and recovers after the absence of presynaptic input.

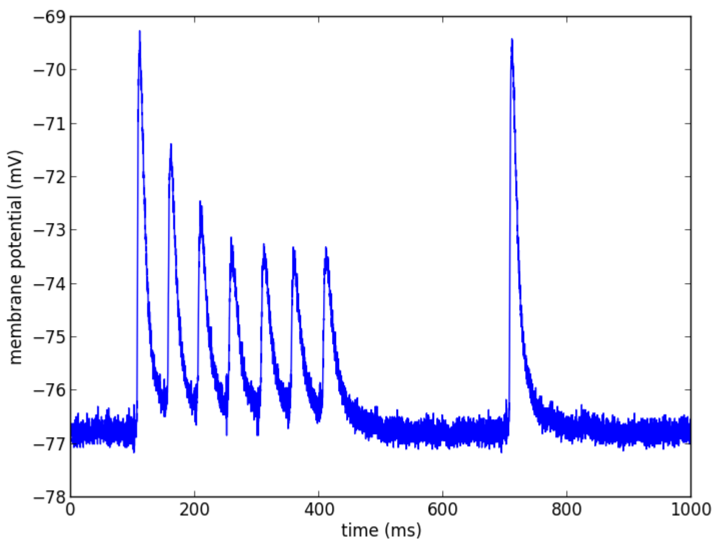


Fig. 5.15: Depressing STP on the Spikey neuromorphic system (source code lesson 5).

Tasks:

- Compare the membrane potential to a network with STP disabled.
- Configure STP to be facilitating.

5.2.8 Lesson 6: Long-term plasticity

In this lesson, we investigate *Spike-timing dependent plasticity (STDP)* on hardware. An external input is connected to the postsynaptic neuron and STDP is enabled for this plastic synapse (P). To adjust the timing between pre- and postsynaptic spikes, several external inputs with static synaptic weights (S) are used to elicit a spike in the postsynaptic neuron. By measuring d, the timing between the pre- and postsynaptic spike Δt can be adjusted on the host computer (see spike timing in Figure 5.16 B).

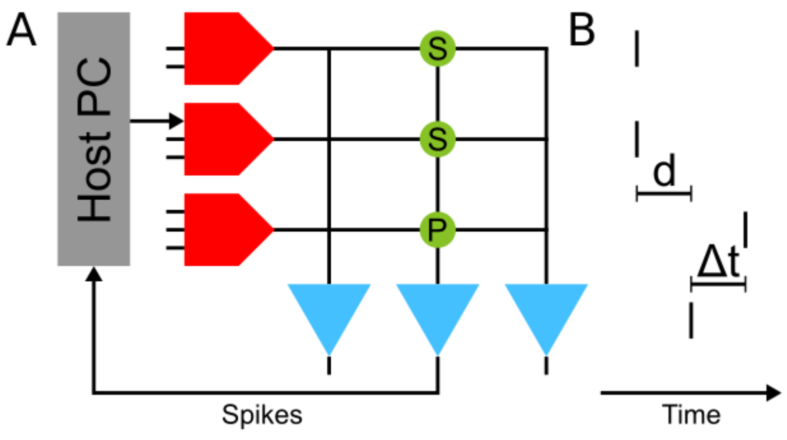


Fig. 5.16: Network configuration (A) and spike timing (B) to measure STDP on the Spikey chip (source code lesson 6).

This network can be used to measure the dependency of synaptic changes Δw on the timing Δt (cf. Figure 5.3). The inverse of the number N of spike pairs that are required to elicit a weight update represent the change of the synaptic weight.

Tasks:

- Configure the hardware neurons and synapses such that each volley of presynaptic spikes evokes exactly a single postsynaptic spike. Due to the intrinsic adaptation of hardware neurons consider discarding the first few spike pairs for the plastic synapse.
- Plot $\frac{1}{N}$ over Δt and compare your results to Figure 5.3.
- Extra: Investigate the results of the last task for varying rows and columns of synapses.

5.2.9 Lesson 7: Functional networks

- A Neuromorphic Network for Generic Multivariate Data Classification

5.2.10 Other network examples

- Simple synfire chain

5.2.11 References

5.3 Frequently asked questions (FAQ)

For contact information, frequently asked questions and further documentation, please visit our website.

5.3.1 Characterization of analog circuits

For measurements of analog circuits see *[Bruederle2009PhD]*. Note that version 3 of the Spikey chip was used in this thesis.

5.3.2 NEST model of hardware STDP synapses

In the scope of [Pfeil2012STDP] we implemented an STDP synapse in NEST considering the following features of STDP synapses on the Spikey chip:

- Local measurement and accumulation of correlations between pre- and postsynaptic spikes.
- Global mechanism that sequentially evaluates these accumulated correlations and updates the synaptic weight.
- Discretization of synaptic weights with a 4-bit resolution (16 values).
- Reduced symmetric nearest-neighbor spike pairing scheme (see [Morrison2008]).

Please not that parameter variations (fixed-pattern noise) are not considered, but can be implemented by distributing the thresholds for the evaluation of the charge on the capacitors.

5.3.3 List of publications

5.4 Built-in calibrations

The calibration of the Spikey neuromorphic system consists of five components:

- Calibration of fast analog-to-digital converter (for recording analog signals)
- Calibration of parameter cells (vout values)
- Calibration of output pins (for reading out analog signals)
- Calibration of membrane time constant
- Calibration of refractory period (currently not used)
- Calibration of synapse drivers (axon-wise adjustment of synaptic strength, see also *[Pfeil2013]*)

To calibrate the chip get source code

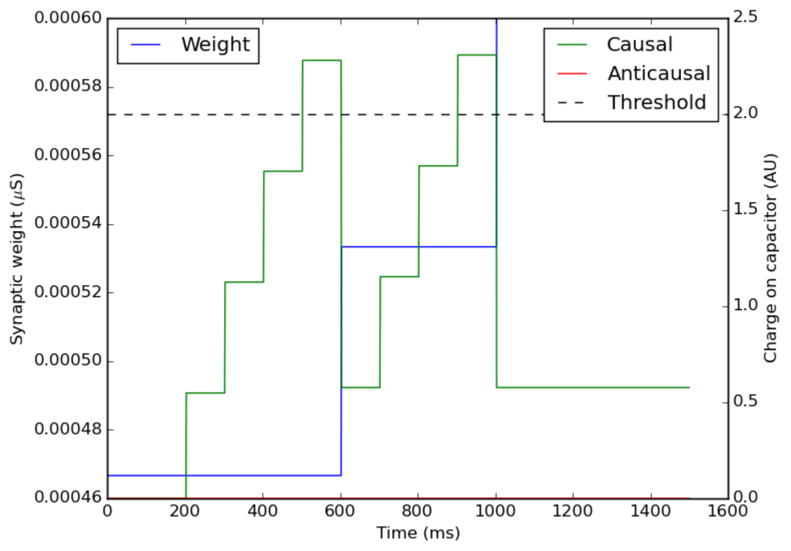


Fig. 5.17: Regular pairs of pre- before postsynaptic spikes are presented (with a frequency of 10 hertz). The charge on the capacitor for causal correlations (green) accumulates and after crossing the threshold (dashed line) the discretized synaptic weight (blue) is increased (source code of NEST model of hardware STDP synapse).

```
git clone https://github.com/electronicvisions/spikey-calib.git
```

and run the calibration

```
python calibAll.py stationXXX
```

where XXX has to be replaced by the chip number to be calibrated.

To visualize calibration results run

```
python analyzeAll.py stationXXX
```

5.5 Testing the system

Warning: this section is work in progress

First, setup the environment variables. Then, to run all tests use:

```
cd spikey_demo/test
sh run_spikey_tests.sh
```

In total this will take several minutes.

For low-level tests that take only several seconds, use:

```
cd spikey_demo/bin/tests
./test-main --gtest_filter=-*Inf
```

5.6 Spikey Appendix

5.6.1 Setting up SSH keys on UHEI cluster for github.com

To create a key pair please follow the guide provided by github.com. The configuration file for the ssh connection is located in `~/.ssh/config`. For example:

```
$ cat .ssh/config
Host github.com
    IdentityFile /wang/users/USERNAME/.ssh/my_github.com.id_rsa
```

The `Host github.com` describes the `ssh` connection target, and the following line configures the private key location which will be used for this target host.

To connect via ssh, use the `tsocks` tool which routes the TCP connection via a local SOCKS proxy, e.g.:

```
$ tsocks git clone git@github.com:USERNAME/PROJECT.git
```

5.6.2 Setup software

Preparing the operating system

We consider the newest stable Debian operating system as reference system.

The following additional software packages are required for building the software of the Spikey system:

```
sudo apt-get install libboost-all-dev libusb-1.0-0-dev liblog4cxx10-dev libqt4-dev␣
↪libgtest-dev libgsl0-dev python-nose
```

And for experiment execution:

```
sudo apt-get install python-numpy python-scipy python-matplotlib
```

Installing hardware drivers

The software for the Spikey neuromorphic system is managed by a git repository that can be downloaded as follows:

```
git clone https://github.com/electronicvisions/spikey_demo.git
```

To install the software follow the instructions in the `README.md` file. Note that a fork of the official PyNN project is used that builds on PyNN version 0.6.

Allow access to USB device

To access the Spikey system via USB, add your user to the `plugdev` UNIX group, to which Spikey devices are assigned to. Note that you will need root privileges for adding users to a UNIX group.

1. Add your user to `plugdev` group: `sudo usermod -aG plugdev USERNAME` (replace USERNAME)
2. Assign Spikey systems to `plugdev` group (creates a `udev` config file): `sudo sh vmodule/nosudo`
3. Restart `udev` daemon: `service udev restart` and (re-)plug-in the Spikey USB device.

Run experiments

Set environment variables (has to be done only once for each environment):

```
. bin/env.sh
```

Run PyNN description of a network:

```
cd networks
python example.py
```

Miscellaneous

If multiple chips are connected to a single host computer, use environment variables to select the chip:

```
MY_STAGE1_STATION=stationXXX
```

Replace XXX with the chip you want to use (e.g. 500).

5.6.3 Setup hardware

Link to internal web page.

CHAPTER

SIX

THE SPINNAKER "MANY CORE" SYSTEM

6.1 About the SpiNNaker hardware

The MC-1 machine is designed to utilise the power of digital hardware and support various models/application though the use of programmable software that can run on ARM cores. A description of the hardware can be located online here.

The MC-1 relies upon software support to allow the PyNN interface to execute on the MC-1 machine; This ranges from low level software (known as spinnaker_tools, including SARK, the Spin1API and ybug) to high level software that takes a graph representation of an application (in this case a PyNN description of a Neural network) and maps, optimises and executes this application on the MC-1 hardware and supports the retrieval of application data generated during the applications execution.

Details of the latest version of the high level software can be found here, and the source code can be seen here.

6.1.1 Available hardware setups

There are currently five known hardware setups:

Jorg Conradt's one-node board. This is a single SpiNNaker chip that is produced at TU Munchen and is intended for light-weight near-robotic applications.

A single 4-chip board. These are commonly used at SpiNNaker workshops. There are a number of these boards in boxes.

A single 48-chip board. This is the standard building block of SpiNNaker machines, but each board can be used in isolation. As with the 4-chip boards, a number of 48-node boards are available in boxes.

A stand-alone toroid. A number of individual 48-node boards can be wired together as a single machine. Additionally, the edges can be wired so that the machine forms a toroidal mesh of chips.

The 600 board machine. This is the machine that is currently running behind the HBP portal. This machine can be broken into smaller multi-board machines through the use of the spalloc software; this allows a number of smaller jobs to run simultaneously on the machine, as well as allowing a single large job to use the whole half-million cores available.

When using the HBP portal, allocation of the machine is done by estimating the size of board required by the given PyNN script. The network is broken down and mapped on to a virtual machine (see below) as described in Mapping and Routing, and then a machine of an appropriate size is allocated. The remaining mapping and routing algorithms are then run as normal.

The software also supports the ability to run in virtual mode, where the tools execute the PyNN script as if it was linked to a direct SpiNNaker machine, but without actually generating any simulated data. This supports end users in testing their scripts for basic compilation errors before loading them into the HBP portal. Instructions on how to use this functionality can be found here.

6.2 Using the SpiNNaker system

As explained in *Building models*, both the experiment description and the model description for the SpiNNaker system must be written as Python scripts, using the PyNN application programming interface (API), version 0.8.

The implementation of the **'PyNN 0.8 API http://neuralensemble.org/docs/PyNN/0.8/api_reference.html'__** for the SpiNNaker system is called `sPyNNaker8`, and is also available as the module `pyNN.spiNNaker`:

```
import pyNN.spiNNaker as sim
```

6.2.1 Supported PyNN functionality

`sPyNNaker` currently supports a subset of the standard PyNN 0.8 API together with a number of extensions. The supported interface functions are listed below. A possibly more up-to-date list can be found in the online documentation.

In the next planned release, support for the PyNN 0.9 API will be included.

Neuron model limitations

sPyNNaker currently supports the following model types:

`IF_curr_exp` Current based leaky integrate and fire, with 1 excitatory and 1 inhibitory exponentially decaying synaptic input per neuron

`IF_cond_exp` Conductance based leaky integrate and fire, with 1 excitatory and 1 inhibitory exponentially decaying synaptic input per neuron

`IF_curr_alpha` Conductance based leaky integrate and fire, with 1 excitatory and 1 inhibitory exponentially decaying synaptic input per neuron

`Izhikevich` Current based Izhikevich with 1 excitatory and 1 inhibitory exponentially decaying synaptic input per neuron

Note that there are also further restrictions on what plasticity types are supported when used with the above models.

All of our neural models have a limitation of 255 neurons per core. Depending on which SpiNNaker board you are using, this will limit the number of neurons that can be supported in any simulation.

External input

sPyNNaker currently supports `SpikeSourceArray` (note that the spikes to be input can be changed between calls to run) and `SpikeSourcePoisson`.

Currently, only the `i_offset` parameter of the neural models can be used to inject current directly; there is no support for noisy or step-based current input.

A third, non-standard PyNN interface, way of injecting current into a PyNN simulation executing on the hardware is through live injection from an external device. These functions are supported by our `sPyNNakerExternalDevicesPlugin`. A description on how to use this functionality can be found here.

Connectors

sPyNNaker currently supports the following standard connector types:

- `OneToOneConnector`
- `AllToAllConnector`
- `FixedNumberPreConnector`
- `FixedNumberPostConnector`
- `FixedTotalNumberConnector`
- `FixedProbabilityConnector`
- `DistanceDependentProbabilityConnector`
- `FromFileConnector`
- `FromListConnector`

Note that using the latter two connectors will result in slower operation of the tools.

Plasticity

sPyNNaker currently only supports plasticity described by an `STDPMechanism`.

sPyNNaker supports the following STDP timing dependence rules:

- `SpikePairRule`

and the following STDP weight dependence rules:

- `AdditiveWeightDependence`
- `MultiplicativeWeightDependence`

Simulation execution

sPyNNaker supports the ability to call `run()` multiple times with different combinations of runtime values, and to call `reset()` multiple times with `run()` interleaved.

sPyNNaker supports the addition of `Populations` and `Projections` between a `reset()` and a `run()`, but not between multiple calls to run().

PyNN missing functionality

sPyNNaker does not support:

- `Assembly`.

sPyNNaker does not support changing of weights / delays / neuron parameters between the initial call to `run()` and a `reset()` call.

Parameter ranges

All parameters and their ranges are under software control.

Weights are held as 16-bit integers, with their range determined at compile-time to suit the application; this limits the overall range of weights that can be represented, with the smallest representable weight being dependent on the largest weights specified.

There is a limit on the length of delays of between 1 and 144 time steps (i.e. 1 - 144 ms when using 1 ms time steps, or 0.1 - 14.4 ms when using 0.1 ms time steps). Delays of more than 16 time steps require an additional "delay population" to be added; this is done automatically by the software when such delays are detected.

Membrane voltages and other neuron parameters are generally held as 32-bit fixed point numbers in the s16.15 format. Membrane voltages are held in mV.

Synapse and neuron loss

Projection links between two sub-populations that were initially defined as connected are removed by the software if the number of connections between the two sub-populations is determined to be zero when the projection is realised in the software's mapping process.

The SpiNNaker communication fabric can drop packets, so there is the chance during execution that spikes might not reach their destination (or might only reach some of their destinations). The software attempts to recover from such losses through a reinjection mechanism, but this will only work if the overall spike rate is not so high as to overload the communications fabric in the first place.

6.2.2 Mapping and Routing

The mapping process examines the neural network definition and attempts to break it down in to parts, each of which can be executed on a SpiNNaker core. A routing algorithm is then run to work out the communication paths between the cores on the SpiNNaker network. In the current software, mapping and routing takes place on the host machine as part of the placement and configuration manager (PACMAN).

It is possible for end users to add their own mapping and routing algorithms into the tool chain. Instructions on how to do so can be found here.

6.3 The SpiNNaker standalone system

SpiNNaker boards and multi-board systems can be used directly connected to your laptop or PC. This requires the installation of the software stack on your host machine to control the board.

6.3.1 Installation and use of the SpiNNaker software

Instructions on how to install the most recent SpiNNaker high level software stack can be found here.

There is a tutorial on creating basic PyNN networks and running them on SpiNNaker here, and another tutorial on using STDP plasticity on SpiNNaker here.

6.3.2 Extending the SpiNNaker Software

The software that executes on the SpiNNaker machine is written in C code, so it is not too difficult to extend this to run new neural, synapse and plasticity models, amongst other things. Instructions on how to extend the software can be found here.

6.3.3 Generating Live Input and Output

SpiNNaker extends PyNN by allowing live input and output from the simulation during execution. This can be used for communication with external devices or other simulations, such as robotic devices or robotic simulations, or for visualisation during the simulation. Instructions on the use of this feature can be found here.

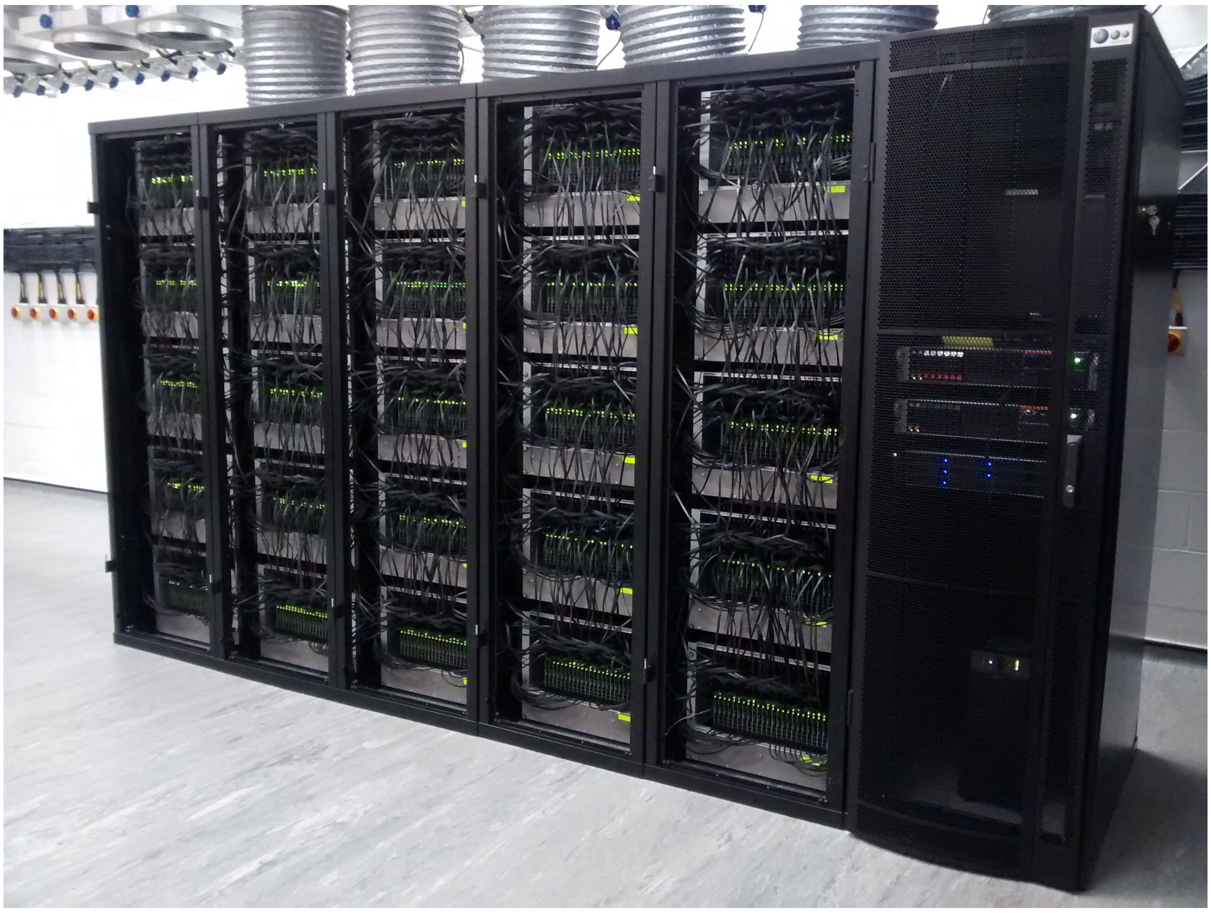


Fig. 6.1: The SpiNNaker 500.000 core system (NM-MC1) in Manchester on 25 March 2016

CHAPTER

SEVEN

BENCHMARKS

Benchmarking of neuromorphic hardware puts numbers on performance to allow measuring progress and comparing different designs. This is useful both for the developers of the Neuromorphic Computing Platform and for potential users.

Specifically, benchmarks define a set of reference tasks aiming at a direct comparison of different neuromorphic (and non-neuromorphic) hardware systems. Each benchmark has a set of quality measures. It is left to the user to decide which specific measures are relevant for the particular application in mind.

7.1 Developing benchmarks

7.1.1 Defining models and tasks

As stated above, all benchmark code should be under version control. Each repository may contain one or more models, and for each model one or more tasks should be defined. The top-level of the repository should contain a JSON-format configuration file named "benchmarks.json", with the following general structure:

```
[
  {
    "model": {
      "name": "ModelA",
      "description": "Description of model A"
    },
    "tasks": [
      {
        "name": "taskA1",
        "command": "task_1_for_model_A.py {system}"
      },
      {
        "name": "taskA2",
        "command": "task_2_for_model_A.py arg1 {system}"}
      }
    ]
  },
  {
    "model": {
      "name": "ModelB",
      "description": "Description of model B"
    },
    "tasks": [
      {
        "name": "taskB1",
        "command": "task_1_for_model_B.py {system}"
      },
      {
        "name": "taskB2",
        "command": "task_2_for_model_B.py arg1 {system}"
      },
      {
        "name": "taskB3alpha",
        "command": "task_3_for_model_B.py --option1={system} arg1 arg2 arg3"
      },
      {
        "name": "taskB3beta",
        "command": "task_3_for_model_B.py --option1={system} arg4 arg5 arg6"
      },
    ]
  }
]
```

i.e., each task should be expressed as a command-line invocation of a Python script. The Python script should in general use the PyNN API, in which case the placeholder "{system}" must be provided, and will be replaced by the name of the PyNN backend used when running the benchmark, e.g. "nest", "spiNNaker", or "hardware.hbp_pm". If the benchmark is known to run only on a subset of the available backends, this can be indicated by listing the suitable backends within the placeholder, e.g. "{system=spiNNaker,nest}". For low-level benchmarks for a single neuromorphic system, the Python script should use the low-level APIs of that platform, and in this case the "{system}" placeholder should be absent.

A specific example, for the repository https://github.com/CNRS-UNIC/hardware-benchmarks, is:

```
[
  {
    "model": {
      "name": "IF_cond_exp",
      "description": "A population of IF neurons, each of which is injected with a␣
↪different current"
    },
    "tasks": [
      {
        "name": "I_f_curve",
        "command": "run_I_f_curve.py {system}"
      }
    ]
  },
  {
    "model": {
      "name": "SpikeSourcePoisson",
      "description": "A population of random spike sources, each with different␣
↪firing rates"
    },
    "tasks": [
      {
        "name": "run20s",
        "command": "run_spike_train_statistics.py {system}"
      }
    ]
  }
]
```

7.1.2 Returning numerical measures

Each task should run a simulation of a neuronal network model, record data from the neurons, perform analysis of the data, and calculate numerical measures of the system performance. The numerical measures should be reported in a JSON-format file, consisting of a top-level record with required fields "model", "task", "timestamp" and "results". The fields "model" and "task" should match the model and task names in the "benchmarks.json" file. The field "configuration", containing a copy of the parameterization of the model and simulator/hardware system", is optional. The field "results" contains a list of records with the following fields:

type What is being measured. For example "quality", "performance", "energy consumption".

name A unique name for the measurement. It is suggested that this name takes the form of a URI containing the URL of the version control repository followed by an identifier for the task and an identifier for the measurement.

value A floating point number.

units (optional) if the measurement is a physical quantity, the units of the quantity using SI nomenclature.

std_dev (optional) if the measurement has an uncertainty, the standard deviation of the value, as a floating point number

min (optional) if the task involves multiple runs of the simulation, the minimum value of the measure

max (optional) if the task involves multiple runs of the simulation, the maximum value of the measure

measure the type of the measurement, for example "norm", "p-value", "time".

(A controlled vocabulary will be developed for the fields "type" and "measure").

Here is an example:

```
  {
    "model": "ModelB",
    "task": "taskB3alpha",
    "timestamp": "2015-06-05T11:13:59.535885",
    "results": [
        {
            "type": "quality",
            "name": "norm_diff_frequency",
            "value": 0.0073371188622418891,
            "measure": "norm"
        },
        {
            "type": "performance",
            "name": "setup_time",
            "value": 0.026206016540527344,
            "units": "s",
            "measure": "time"
        },
        {
            "type": "performance",
            "name": "run_time",
            "value": 1.419724941253662,
            "units": "s",
            "measure": "time"
        },
        {
            "type": "performance",
            "name": "closing_time",
            "value": 0.03272294998168945,
            "units": "s",
            "measure": "time"
        }
    ],
}
```

The task may also optionally produce figures and other output data files.

7.1.3 Registering benchmarks

To add a new benchmark model or task within an existing repository, just modify the "benchmarks.json" configuration file. To add a new repository, e-mail andrew.davison@unic.cnrs-gif.fr. In future, a web form for registering new repositories will be introduced.

7.1.4 Running benchmarks

A continuous integration system will be put in place, which will run the entire suite of benchmarks on each neuromorphic system every time the system configuration (software or hardware) is changed, and which will run the benchmarks from a given repository on both neuromorphic systems (where appropriate) each time a new commit is made to the repository. To indicate that a given commit should *not* trigger a run (for example because only documentation has been changed), include the text "[skip ci]" or "[ci skip]" within the commit message.

After running each task, the continous integration system will harvest the JSON-formatted measurement report, and update a database of benchmark measurements. This benchmark database will be visualized in an "App" within the HBP Neuromorphic Platform Collaboratory.

CHAPTER

EIGHT

GETTING HELP

If you are having problems using the Neuromorphic Computing Platform, would like to request in-depth support, or would like to request new features, please send an e-mail to support@humanbrainproject.eu.

You may also find useful information on the Human Brain Project Community Forum

8.1 PyNN-specific help

Questions about PyNN can be asked in the NeuralEnsemble forum. PyNN bug reports may be filed in the Github issue tracker.

8.2 BrainScaleS-specific help

If you run into problems during the installation of the ESS or have any issues / questions that are not covered by the FAQ, please consider the mailing list for BrainScaleS wafer-scale hardware and ESS users or the BrainScaleS chat.

8.3 SpiNNaker-specific help

If you run into problems during the installation of the software stack or have any issues / questions, please consider the SpiNNaker Users Google group.

CHAPTER

NINE

APPENDIX: INSTALLATION OF THE BRAINSCALES SOFTWARE STACK

In the following the build and work flow on University of Heidelberg cluster frontend nodes is described. This is only needed if you want to use the system locally within Heidelberg. If the BrainScaleS System is accessed through the HBP Collaboratory or the Python client, this software environment is pre-installed.

The following section can be skipped when loading the `nmpm_software` module by

```
module load nmpm_software/current
```

All available versions can be listed by `module avail nmpm_software`.

To compile your own installation create and change to a directory for your projects, preferably on `wang`, e.g.: `/wang/users/<somebody>/cluster_home/projects`.

9.1 ssh-agent

To reduce the amount of typing, please consider using `ssh-agent` to cache your key password:

```
eval `ssh-agent`
ssh-add ~/path/to/your/private_id_rsa
```

9.2 Build Tool

We use a custom version (`git@gitviz.kip.uni-heidelberg.de:waf.git`) of the `waf` configuration and build tool. A nightly version is provided by loading:

```
module load waf
```

9.3 PyHMF, marocco and Dependencies

The first step is to create a new workspace for the software checkouts and the subsequent build

```
mkdir ~/my_nmpm_software && cd ~/my_nmpm_software
```

Now `waf` can be used to setup the project. It will clone all the dependencies.

```
waf setup --project pyhmf --project=marocco --without-ester
```

The next step is configuration. As the default software environment on the UHEI cluster does not provide all the software dependencies, you have to load some modules which will provide those dependencies:

```
module load localdir
module load pynn/0.7.5
module load mongo
module load yaml-cpp/0.5.3
```

As those commands are needed every time you want to use the software it is convenient to put all the needed parts into a script:

```
echo "INSTALLED_LIB_PATH=$(readlink -e lib)" > init.sh
cat >>init.sh<<EOF
cd ${INSTALLED_LIB_PATH}
module load localdir
module load pynn/0.7.5
module load mongo
module load yaml-cpp/0.5.3
cd -
EOF
```

After completing the installation steps below, this script can be sourced (`source init.sh`) to access the BrainScaleS Wafer-Scale Software Stack.

Now the configuration step:

```
waf configure
```

The install step will build and install all targets into subdirectories of your current working directory. Currently, you need to explicitly specify some extra targets in a second call.

```
waf install --test-execnone
# some extra targets are needed
waf install --target=pymarocco,pyhalbe,pysthal,redman_xml --test-execnone
```

If you want to include the calibration toolkit (`cake`) (optional):

```
waf setup --project pyhmf --project marocco --project cake --without-ester
# module load ...
waf configure
waf install --test-execnone
# some extra targets are needed
waf install --target=pymarocco,pyhalbe,pysthal,redman_xml,pycake --test-execnone
```

To include support for executable system simulation (ESS) add `--with-ess` to the setup call.

Please remember, to that you have to setup the software environment if you start new shells (e.g. by using the `init.sh` script):

```
source ~/my_nmpm_software/init.sh
```

Check if the installation and the setup of variables is fine:

```
python -c "import pyhmf" && echo ok
```

should print `ok`, if instead:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named pyhmf
```

occurs, either the installation failed or the environment variables responsible for finding the module are wrong. In that case, double check if you followed the instructions 1:1.

9.4 Running PyNN scripts

To run locally on the *hardware* one needs to use the SLURM job queue system:

```
srun -p nmpm -L `license_by_hicann 33 367` --pty python nmpm1_single_neuron.py
```

nmpm1_single_neuron.py:

9.5 Inspect the Configuration

```
sthal/tools/dump_cfg.py
```

CHAPTER

TEN

APPENDIX: MISCELLANEOUS

10.1 BrainScaleS

10.1.1 Job queue Demo - HBP Summit 2014

On a UHEI BrainScaleS cluster frontend node:

```
cd somewhere-on-your-wang:

socksify git clone https://github.com/electronicvisions/hbp_platform_demo

cd hbp_platform_demo

module load pynn
module load mongo
module load PlatformDemo/20140924_sschmitt

srun -p wafer python run.py nmpm1

*wait*

display result.png
```

The repository can be also when creating job via the web interface, cf. *Using the web interface*.

10.1.2 roqt (Visualization)

You need to be in the roqt directory, because the ui file is loaded by a relative path:

```
cd marocco/tools/roqt
PYTHONPATH=$PWD/lib:$PYTHONPATH bin/roqt /some/path/to/roqt.bin
```

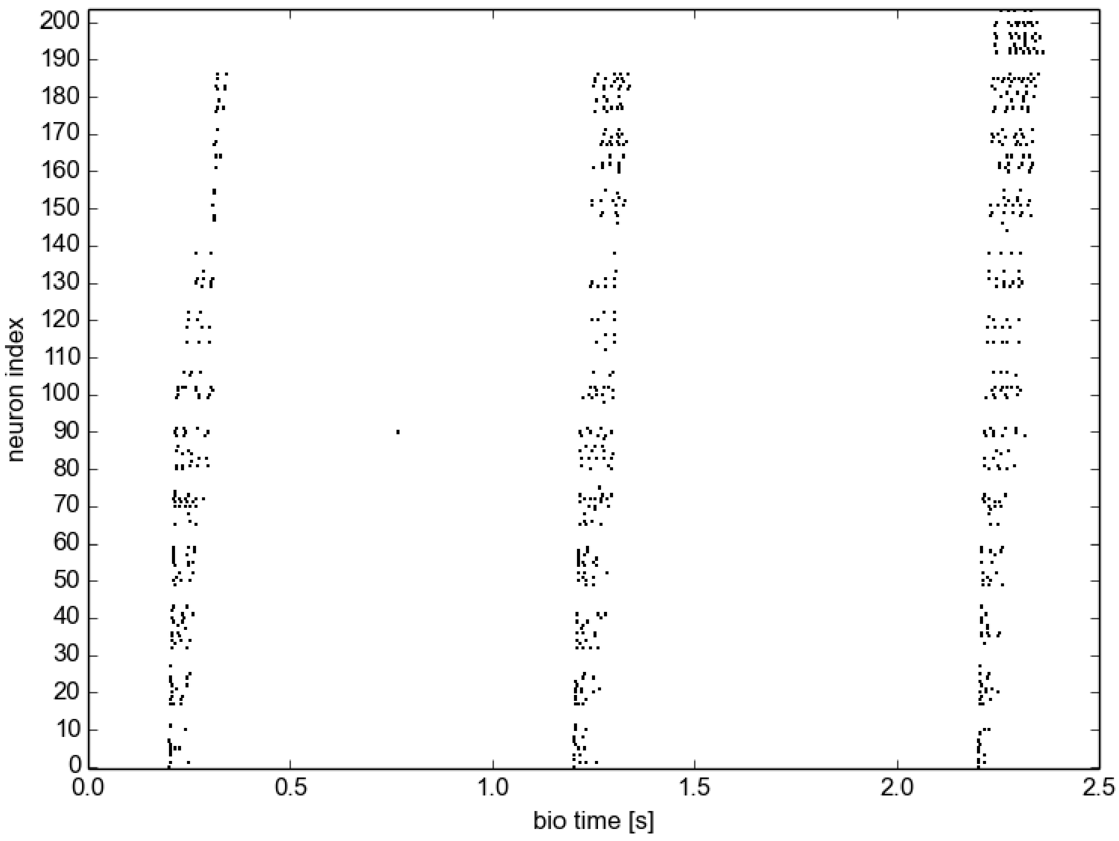


Fig. 10.1: Demo running on a HICANNv2 Wafer module.

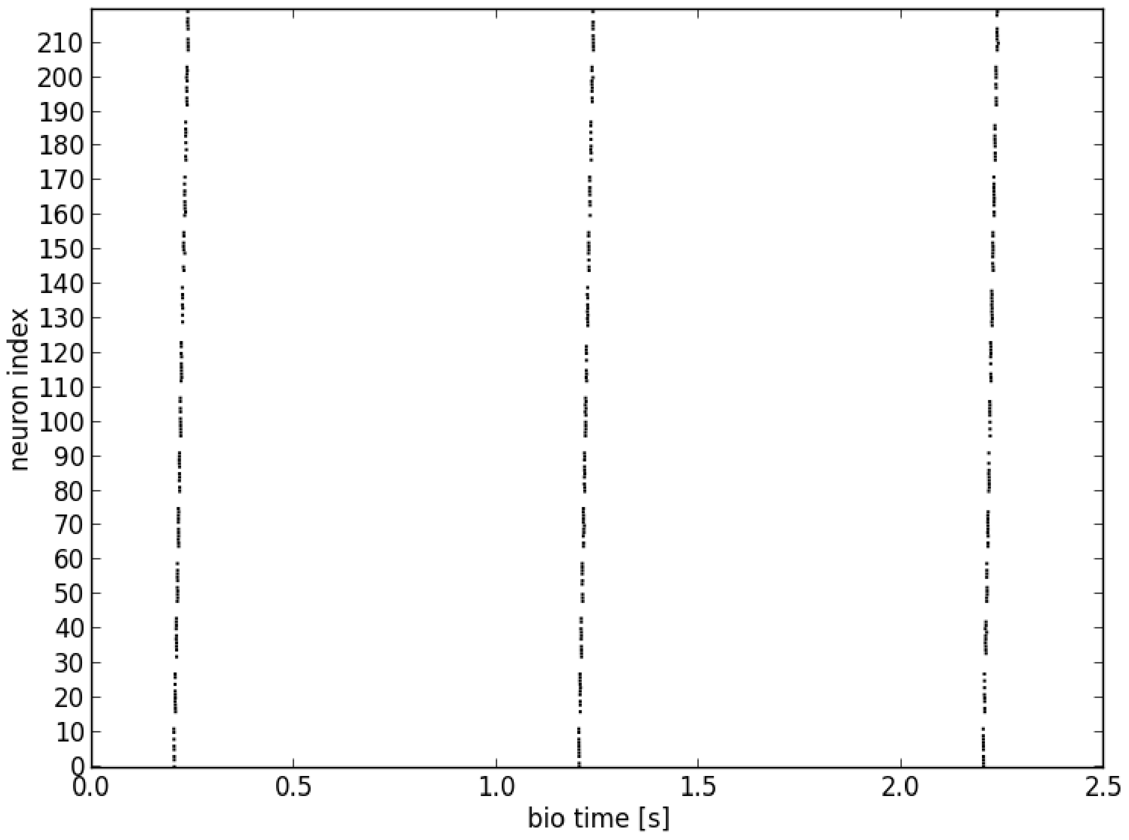


Fig. 10.2: Demo running on the ESS.

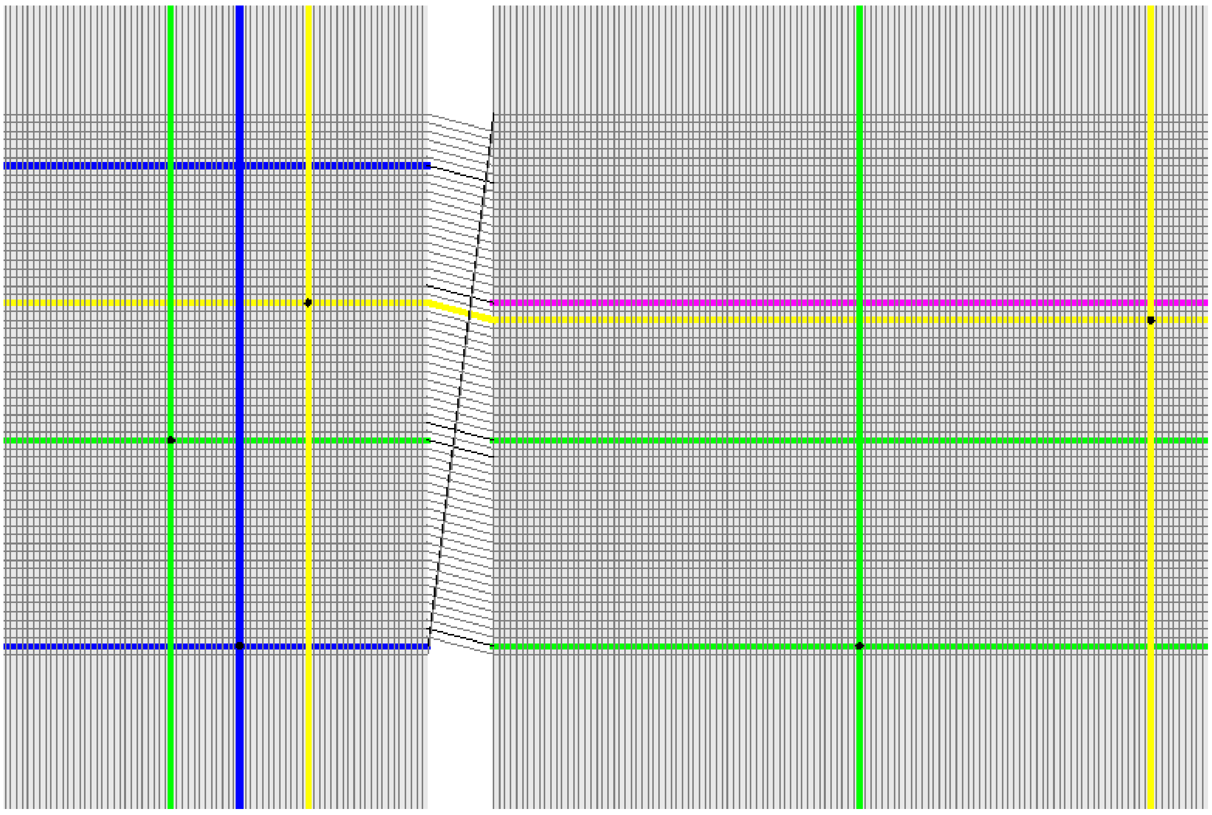


Fig. 10.3: Routing visualization of the roqt visualization tool.

CHAPTER ELEVEN

PYTHON CLIENT REFERENCE

class nmpi.**Client**(*username=None*, *password=None*, *job_service='https://nmpi.hbpneuromorphic.eu/api/v2/'*, *quotas_service='https://quotas.hbpneuromorphic.eu'*, *token=None*, *verify=True*)

Client for interacting with the Neuromorphic Computing Platform of the Human Brain Project.

This includes submitting jobs, tracking job status and retrieving the results of completed jobs.

***Arguments*:**

username, password credentials for accessing the platform. Not needed in Jupyter notebooks within the HBP Collaboratory.

job_service the base URL of the platform Job Service. Generally the default value should be used.

quotas_service the base URL of the platform Quotas Service. Generally the default value should be used.

token when you authenticate with username and password, you will receive a token which can be used in place of the password until it expires.

verify in case of problems with SSL certificate verification, you can set this to False, but this is not recommended.

user_info

Information about the current user, as retrieved from the Collaboratory

submit_job(*source*, *platform*, *collab_id*, *config=None*, *inputs=None*, *command='run.py {system}'*, *tags=None*, *wait=False*)

Submit a job to the platform.

***Arguments*:**

source the Python script to be run, the URL of a public version control repository containing Python code, a zip file containing Python code, or a local directory containing Python code.

platform the neuromorphic hardware system to be used. Either "BrainScaleS" or "SpiNNaker".

collab_id the ID of the collab to which the job belongs

config (optional) a dictionary containing configuration information for the hardware platform. See the Platform Guidebook for more details.

inputs a list of URLs for datafiles needed as inputs to the simulation.

command (optional) the path to the main Python script relative to the root of the repository or zip file. Defaults to "run.py {system}".

tags (optional) a list of tags (strings) describing the job.

wait (default False) if True, do not return until the job has completed, if False, return immediately with the job id.

***Returns*:** The job id as a relative URI unless *wait=True*, in which case returns the job as a dictionary.

***Notes*:** If the *source* argument is a directory containing Python code, the directory contents will be uploaded to Collab storage into a folder named according to the property *client.collab_source_folder* (default: "source_code")

`job_status` (*job_id*)
Return the current status of the job with ID *job_id* (integer or URI).

`get_job` (*job_id*, *with_log=True*)
Return full details of the job with ID *job_id* (integer or URI).

`remove_completed_job` (*job_id*)
Remove a job from the interface.

The job is hidden rather than being permanently deleted.

`remove_queued_job` (*job_id*)
Remove a job from the interface.

The job is hidden rather than being permanently deleted.

`queued_jobs` (*verbose=False*)
Return the list of jobs belonging to the current user in the queue.

***Arguments*:**

verbose if False, return just the job URIs, if True, return full details.

`completed_jobs` (*collab_id*, *verbose=False*)
Return the list of completed jobs in the given collab.

***Arguments*:**

verbose if False, return just the job URIs, if True, return full details.

`download_data` (*job*, *local_dir='.'*, *include_input_data=False*)
Download output data files produced by a given job to a local directory.

***Arguments*:**

job a full job description (dict), as returned by *get_job()*.

local_dir path to a directory into which files shall be saved.

include_input_data also download input data files.

`copy_data_to_storage` (*job_id*, *destination='collab'*)
Copy the data produced by the job with id *job_id* to Collaboratory storage or to the HPAC Platform. Note that copying data to an HPAC site requires that you have an account for that site.

Example:

To copy data to the JURECA machine:

```
client.copy_data_to_storage(90712, "JURECA")
```

To copy data to Collab storage:

```
client.copy_data_to_storage(90712, "collab")
```

`create_data_item` (*url*)
Register a data item with the platform.

`create_resource_request` (*title*, *collab_id*, *abstract*, *description=None*, *submit=False*)
Create a new resource request. By default, it will not be submitted.

`edit_resource_request` (*request_id*, *title=None*, *abstract=None*, *description=None*, *submit=False*)
Edit and/or submit an unsubmitted resource request

`list_resource_requests`(*collab_id*, *status=None*)
Return a list of resource requests for the Neuromorphic Platform

`list_quotas`(*collab_id*)
Return a list of quotas for running jobs on the Neuromorphic Platform

`my_collabs`()
Return a list of collabs of which the user is a member.

BIBLIOGRAPHY

[Bi2001] Bi et al. (2001). Synaptic modification by correlated activity: Hebb's postulate revisited. Annu. Rev. Neurosci. 24, 139–66.

[Coombs1955] Coombs et al. (1955). Excitatory synaptic action in motoneurones. The Journal of Physiology 130 (2), 374–395.

[Gruebl2007Phd] Grübl, A. (2007). VLSI Implementation of a Spiking Neural Network. PhD thesis, Heidelberg University. HD-KIP 07-10.

[Indiveri2011] Indiveri et al. (2011). Neuromorphic silicon neuron circuits. Front. Neurosci. 5 (73).

[Kremkow2010] Kremkow et al. (2010). Gating of signal propagation in spiking neural networks by balanced and correlated excitation and inhibition. J. Neurosci. 30 (47), 15760–15768.

[Morrison2008] Morrison et al. (2008). Phenomenological models of synaptic plasticity based on spike-timing. Biol. Cybern. 98, 459–478.

[Petkov2012] Petkov, V. (2012). Toward Belief Propagation on Neuromorphic Hardware. Diploma thesis, Heidelberg University. HD-KIP 12-23.

[Pfeil2012STDP] Pfeil et al. (2012). Is a 4-bit synaptic weight resolution enough? – constraints on enabling spike-timing dependent plasticity in neuromorphic hardware. Front. Neurosci. 6:90.

[Pfeil2013] Pfeil et al. (2013). Six networks on a universal neuromorphic computing substrate. Front. Neurosci. 7 (11).

[Pfeil2013STDP] Pfeil et al. (2013). Neuromorphic learning towards nano second precision. In Neural Networks (IJCNN), The 2013 International Joint Conference on, pp. 1–5. IEEE Press.

[Pfeil2015] Pfeil et al. (2013). The effect of heterogeneity on decorrelation mechanisms in spiking neural networks: a neuromorphic-hardware study. Submitted.

[Pfeil2015Phd] Pfeil (2015). Exploring the potential of brain-inspired computing. Doctoral thesis, Heidelberg University.

[Schemmel2007] Schemmel et al. (2007). Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In Proceedings of the 2007 International Symposium on Circuits and Systems (ISCAS), New Orleans, pp. 3367–3370. IEEE Press.

[Schemmel2006] Schemmel et al. (2006). Implementing synaptic plasticity in a VLSI spiking neural network model. In Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN), Vancouver, pp. 1–6. IEEE Press.

[ScholarpediaShortTermPlasticity] Misha Tsodyks and Si Wu (2013) Short-term synaptic plasticity. Scholarpedia, 8(10):3153.

[ScholarpediaSTDP] Jesper Sjöström and Wulfram Gerstner (2010) Spike-timing dependent plasticity. Scholarpedia, 5(2):1362.

[Tsodyks1997] Tsodyks et al. (1997). The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. Proc. Natl. Acad. Sci. USA 94, 719–723.

[Bruederle2009PhD] Brüderle, D. (2009). Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System. PhD thesis, Heidelberg University. HD-KIP 09-30.

INDEX